Improving Combinatorial Optimization

by

Geoffrey G Chu

Submitted in total fulfillment
of the requirements
of the degree of Doctor of Philosophy

October 2011

Department of Computer Science and Software Engineering
The University of Melbourne
Abstract

Combinatorial Optimization is an important area of computer science that has many theoretical and practical applications. In this thesis, we present important contributions to several different areas of combinatorial optimization, including nogood learning, symmetry breaking, dominance, relaxations and parallelization.

We develop a new nogood learning technique based on constraint projection that allows us to exploit subproblem dominances that arise when two different search paths lead to subproblems which are identical on the remaining unlabeled variables. On appropriate problems, this nogood learning technique provides orders of magnitude speedup compared to a base solver which does not learn nogoods.

We present a new symmetry breaking technique called SBDS-1UIP, which is an extension of Symmetry Breaking During Search (SBDS). SBDS-1UIP uses symmetric versions of the 1UIP nogoods derived by Lazy Clause Generation solvers to prune symmetric parts of the search space. We show that SBDS-1UIP can exploit at least as many symmetries as SBDS, and that it is strictly more powerful on some problems, allowing us to exploit types of symmetries that no previous general symmetry breaking technique is capable of exploiting.

We present two new general methods for exploiting almost symmetries (symmetries which are broken by a small number of constraints). The first is to treat almost symmetries as conditional symmetries and exploit them via conditional symmetry breaking constraints. The second is to modify SDBS-1UIP to handle almost symmetries. Both techniques are capable of producing exponential speedups on appropriate problems.

We examine three reasonably well known problems: the Minimization of Open
Stacks Problem, the Talent Scheduling Problem (CSPLib prob039), and the Maximum Density Still Life Problem (CSPLib prob032). By applying various powerful techniques such as nogood learning, dynamic programming, dominance and relaxations, we are able to solve these problems many orders of magnitude faster than the previous state of the art.

We identify cache contention as a serious bottleneck that severely limit the amount of speedup achievable when parallelizing SAT and LCG solvers on multi-core systems. We alter the data structures used in the state of the art SAT solver MiniSat [39] to be more cache aware, leading to a sequential SAT solver that is some 80% faster on average and a parallel SAT solver that is some 140% faster on average.

We examine an important issue in parallel search that has not been properly addressed in the literature. Many work stealing schemes used for load balancing completely ignore the branching heuristic, and instead try to maximize the granularity of the work stolen. This can result in many of the processors searching in unfruitful parts of the search tree. We analyze this problem and develop a new work stealing scheme called confidence based work stealing, which partitions the work based on how strong the branching heuristic is at each node. The new parallel algorithm produced near linear or super linear speedup on all the problems we tested.
This is to certify that:

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface

(ii) due acknowledgment has been made in the text to all other material used

(iii) the thesis is fewer than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices

__________________________
Geoffrey G Chu
Preface

This PhD thesis is based on research conducted at the Department of Computer Science and Software Engineering, University of Melbourne, from January 2008 to July 2011. The research was carried out in collaboration with my supervisor Peter J. Stuckey, and co-authors Maria Garcia de la Banda, Chris Mears, and Christian Schulte. Parts of this thesis has previously appeared in the following publications: [54, 28, 29, 27, 26, 25, 24].
Acknowledgment

I would like to thank my supervisor Peter J. Stuckey, who has given me invaluable advice and support during my candidature. It has been a great privilege to work with someone of such high caliber, and I have thoroughly enjoyed our many interesting discussions. I would also like to thank the members of my advisory committee Mark Wallace and Aaron Harwood, as well as my co-authors Maria Garcia de la Banda, Chris Mears, and Christian Schulte. Most importantly, I would like to thank my family, without which none of this would be possible.

Geoffrey Chu

University of Melbourne
July 2011
Contents

Abstract iii

Preface vii

Acknowledgment ix

1 Introduction 1
  1.1 Contributions ......................................................... 6

2 Background 11
  2.1 Constraint Satisfaction Problem ................................. 12
  2.2 Constraint Programming ........................................... 13
  2.3 Boolean Satisfiability ............................................. 19

I Nogood Learning 23
  I.1 Introduction .......................................................... 25
  I.2 Deriving Nogoods through Search ................................. 25
  I.3 Deriving Nogoods through Resolution ........................... 26
  I.4 Contributions ....................................................... 29

3 Automatic Caching via Constraint Projection 31
  3.1 Introduction .......................................................... 32
  3.2 Subproblem Dominance and Equivalence ........................ 34
  3.3 Manual Keys for Caching ........................................... 36
IV Parallelization

IV.1 Contributions .............................................................. 231

9 Cache Conscious Data Structures for SAT ................................ 233

9.1 Introduction .................................................................. 234
9.2 Boolean Constraint Propagation ..................................... 235
9.3 Improving the Cache Behavior ......................................... 238
9.4 Experiments .................................................................. 248
9.5 Related Work ................................................................ 254
9.6 Conclusion .................................................................... 257

10 Confidence Based Work Stealing ......................................... 259

10.1 Introduction .................................................................. 260
10.2 Analysis of Work Allocation .......................................... 263
10.3 Adaptive Work Stealing .................................................. 266
10.4 Dynamically Updating Solution Density Estimates .......... 266
10.5 Confidence Model ............................................................ 268
10.6 Algorithm .................................................................... 268
10.7 Experiments .................................................................. 273
10.8 Conclusion .................................................................... 278

11 Conclusion ........................................................................ 279

Bibliography ........................................................................ 283

A Appendix .......................................................................... 303

1.1 119 industrial SAT instances ........................................... 303
1.2 24 medium sized industrial instances .............................. 303
1.3 22 hard industrial instances ............................................ 304
1.4 200 industrial instances from SAT Race 2008 ............... 305
## List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Steps taken by $solv(C, D)$ algorithm in Example 2.2.1.</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Steps taken by $learn(v, S)$ algorithm in Example 2.3.1.</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Experimental results: Knapsack, MOSP and Black Hole.</td>
<td>78</td>
</tr>
<tr>
<td>3.2</td>
<td>Experimental results: BACP and Radiation</td>
<td>79</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparison of three SBDS implementations in CHUFFED, static symmetry breaking in CHUFFED, and LDSB in Eclipse, on the Concert Hall Scheduling problem</td>
<td>104</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of three SBDS implementations in CHUFFED, static symmetry breaking in CHUFFED, and LDSB in Eclipse, on the Graph Coloring problems</td>
<td>105</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison between ignoring almost symmetries (none), exploiting almost symmetries with conditional symmetry (cond-sym), SBDS-1UIP (sbds-1uip), decomposition using lex (remodel-lex) and decomposition using SBDS-1UIP (remodel-sbds)</td>
<td>122</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparison of upper bound heuristic, versus complete search on some difficult problems. Times in milliseconds</td>
<td>146</td>
</tr>
<tr>
<td>6.2</td>
<td>Results on the open problems from the Constraint Modeling Challenge 2005, comparing versus the the winner of the challenge [53]. Times in milliseconds.</td>
<td>152</td>
</tr>
<tr>
<td>6.3</td>
<td>Comparing customer search versus [53] on harder random instances. Search is to find the best solution found by [53] with node limit $2^{25}$.</td>
<td>153</td>
</tr>
</tbody>
</table>
6.4 (a) Comparing the effects of each optimization in turn, and (b) comparing the effects of relaxation. ........................................ 154

7.1 Arithmetic mean solving time (ms) for structured problems of size \( n \), and relative slowdown if the optimization is turned off. ............... 190

7.2 Arithmetic mean subproblems solved for structured problems of size \( n \), and relative increase if the optimization is turned off. ............... 191

7.3 Arithmetic mean solving time (ms) for random problems with \( m \) actors and \( n \) scenes. .................................................. 193

7.4 Arithmetic mean subproblems solved for random problems with \( m \) actors and \( n \) scenes. .................................................. 193

7.5 Comparison with the approach of [145] on the examples from that paper. 196

8.1 Possible patterns around dead cells, showing where they donate their tokens and any wastage. ........................................ 203

8.2 Contributions to the tokens of a live cell from its South neighbor. ... 204

8.3 Possible patterns around a live cell showing token benefactors and any wastage. .................................................. 205

8.4 Optimum number of live cells in the Maximum Density Still Life Problem found by complete search. ................................. 210

8.5 Ratio of wastage to edge cells in an optimal edge for different edge width, and number of patterns that appear in the first three rows. ... 213

8.6 Lower bounds on wastage and corresponding upper bounds on live cells as calculated by bounded dynamic programming on a problem relaxation. Instances where the bound on the relaxed problem differed from the original problem are indicated with an asterisk. ............... 218

xvi
9.1 Cache conscious optimizations for sequential SAT solving. Comparison of various combinations of optimizations: WL — Watched list modification (9.3.2), PC(n) — Clause modification with length n clause head, n = 16 if labeled as just PC (9.3.3), FL — Store first literal in watched list (9.3.4), BC — Handle binary clauses specially (9.3.5), TC — Handle ternary clauses specially (9.3.6) PA — Pack assigns (9.3.7). The results show the (geometric mean) runtime for the suite, the speedup compared to MiniSat 2.0, I/Loop stands for average number of instructions per inner loop of the propagation engine, L1M/Loop and L2M/Loop stands for the average number of L1 and L2 misses per inner loop of the propagation engine.

9.2 Cache conscious optimizations for parallel SAT solving. We compare a parallelized version of MiniSat 2.0, PMiniSat, using the original data structures and the combination of optimizations detailed in Table 9.1. The results are the geometric mean of 20 runs.

9.3 Parallel memory stress test.

10.1 Experimental results for optimization problems with simple confidence model. The results show: wall clock runtime in seconds (Runtime), speedup relative to the sequential version (Speedup), and runtime efficiency (RunE) which is Speedup/8, CPU utilization (CPU%), communication overhead (Comm%), number of steals (Steals), total number of nodes explored (Nodes), the algorithmic efficiency (AlgE) the total number of nodes explored in the parallel version versus the sequential version, the number of nodes explored to find the optimal solution (Onodes), and the solution finding efficiency (SFE) the total number of nodes explored in the parallel version to find the optimal versus the sequential version. Values for Runtime, CPU%, Comm%, Steals, Nodes, and Onodes are the geometric mean of the instances solved by all 4 versions.
10.2 Experimental results for satisfaction problems with simple confidence model .................................................. 276
10.3 Experimental results showing Nodes and algorithmic efficiency (AlgE) using accurate confidence values, where we follow the confidence value to degree $\alpha$. .................................................. 277
List of Figures

2.1 A simple graph coloring problem. ................................................. 13
2.2 Basic search algorithm for propagation based constraint solvers. ..... 15
2.3 Search tree in Example 2.2.1. .................................................. 16
2.4 Basic branch and bound algorithm for propagation based constraint
solvers. ................................................................. 17
2.5 Naive algorithm for the propagation engine. ............................... 18
2.6 Implication graph for Example 2.3.1. Decision literals are double boxed. 22
3.1 Search algorithm taking advantage of subproblem dominance. ....... 37
3.2 Two starting sequences for the Black Hole problem that lead to an
identical remaining subproblem. ............................................. 37
3.3 Three examples of partially colored graphs. ............................... 38
3.4 Sufficient conditions on propagation to ensure no key is required for
the binary constraints above. Note the bool2int() function converts a
Boolean b to a 0-1 integer y. ........................................ 52
3.5 Different fixed sub-schedules for cumulative leading to the same key. 58
3.6 Different fixed sub-schedules for cumulative leading to the same key
when using decomposition based keys. .................................. 61
4.1 A graph coloring problem where we can exploit additional symmetries. 96
5.1 A graph coloring problem where we can exploit additional symmetries. 118
6.1 (a) An example $c(p)$ function: $c_i \in c(p_j)$ if the row for $c_i$ in column $p_j$ has an X. (b) An example schedule: $c_i$ is active when product $p_j$ is scheduled if the row for $c_i$ in column $p_j$ has an X or a .

6.2 (a) A schedule corresponding to customer order $[c_5, c_4, c_3, c_2, c_1]$. (b) A schedule corresponding to customer order $[c_3, c_1, c_2, c_3, c_4]$. (c) The customer graph (ignoring self edges) with edges labeled by the products that generate them.

7.1 (a) An Example $a(s)$ Function: $a_i \in a(s_j)$ if the Row for $a_i$ in Column $s_j$ has an X. (b) An Example Schedule: $a_i$ is on location when scene $s_j$ is Scheduled if the Row for $a_i$ in Column $s_j$ has an X or a .

7.2 Pseudo-code for best-first call-based dynamic programming algorithm. $\text{schedule}(Q)$ returns the minimum cost required for scheduling the set of scenes $Q$.

7.3 An optimal order for the problem of Example 7.1.1

7.4 The problem of Example 7.1.1 after preprocessing.

7.5 An optimal schedule for the scenes $Q = \{s_1, s_2, s_4, s_7, s_8, s_9\}$ Assuming $\{s_3, s_5, s_6, s_{10}, s_{11}, s_{12}\}$ have already been scheduled

7.6 Pseudo-code for bounded best-first call-based dynamic programming algorithm. $\text{bnd}_\text{schedule}(Q, U)$ returns the minimum cost required for scheduling the set of scenes $Q$ if it is less than or equal to $U$. Otherwise it returns a lower bound on the minimal cost.

7.7 The schedule defined by the heuristic upper bound algorithm.

7.8 An original set of remaining scenes, assuming $a_1, a_2, a_3, a_4$ are on location.

7.9 Pseudo-code for bounded best-first call-based dynamic programming algorithm. $\text{bnd}_\text{de}_\text{schedule}(Q, B, E)$ returns the minimum cost required for scheduling the set of scenes $Q$ if it is less than or equal to $U$. Otherwise it returns a lower bound on the minimal cost.

7.10 Two equivalent subproblems.

7.11 An optimal schedule for the film “Mob Story”

8.1 Optimal solutions for $n = 21$ and $n = 22$.

xx
8.2 Optimal edge patterns for a width 2 and width 3 boundary relaxation. Wastage is highlighted with a star. .......................................................... 213
8.3 Closed form equations for the wastage lower bound for \( n \geq 61 \). ... 218
8.4 Closed form equations for the live cell upper bound for \( n \geq 61 \). ... 218
8.5 The two best ways to fill a corner (up to symmetry). Wastage is highlighted with a star. Note that there are two units of wastage in the cell in the 4th column and 4th row in the second pattern. ..... 219
8.6 Dynamic relaxation lookahead for still life search. ......................... 220
8.7 Optimal solution for \( n = 100 \). ............................................. 222
8.8 A periodic optimal solution where the center sections can be tiled indefinitely to produce optimal solutions for \( n = 100 + k \times 54 \). .......... 224

9.1 Boolean constraint propagation of the decision literal \( d \) given a current assignment \( A \). ................................................................. 237
9.2 Data structure used in MiniSat 2.0 ........................................... 238
9.3 The histogram over the 119 tested instances of the percentage of short watched lists (length \( \leq 16 \)) encountered during the inner loop for Boolean constraint propagation. ................................................. 241
9.4 Watched list data structure ..................................................... 242
9.5 The histogram over the 119 tested instances of the percentage of clauses where \( \leq k \) literals were examined in the inner loop for Boolean constraint propagation. The histogram shows the results for \( k = 1 \) (only first literal examined), \( k = 4 \) and \( k = 8 \). .................. 243
9.6 Clause data structures ............................................................ 245
9.7 Watched list element data structure ................................. 247
9.8 Runtime comparisons on 151 SAT Race instances .................. 250

10.1 Example with reasonably high confidence ......................... 270
10.2 Example with low confidence ........................................... 272
Chapter 1

Introduction

Combinatorial Optimization is an important area of computer science that has many theoretical and practical applications. Many real world problems in the areas of production, scheduling, resource allocation, facility location and layout, etc, can be expressed as combinatorial optimization problems. In a combinatorial optimization problem, we have a finite set of options, and we wish to find the one which is optimal with respect to some objective function, e.g., cost, time. The ability to find good or optimal solutions to real world combinatorial optimization problems can often translate into significant time, resource or monetary savings for industry.

Combinatorial optimization problems are usually modeled in terms of variables, constraints and objective function. A simple example of a combinatorial optimization problem is the 0-1 Knapsack Problem. Suppose we have a knapsack with a certain weight capacity $W$, and a set of $n$ items, each of which has a particular weight $w_i$ and value $v_i$, and we wish to maximize the value of the items that can fit into the knapsack. We can model this problem as follows. We define variables $x_i \in \{0, 1\}$, to represent whether the $i$th item is put into the knapsack ($x_i = 1$ means put in). Since the total weight cannot exceed the capacity of the knapsack, we have the constraint: $\sum_{i=1}^{n} w_i x_i \leq W$. The objective to be maximized is given by: $\sum_{i=1}^{n} v_i x_i$.

There are different techniques for solving combinatorial optimization problems, depending on what kind of variables and constraints are involved. Linear Programming (LP) can be used to solve problems which can be modeled using continuous variables,
linear constraints and linear objective function. Such problems can be solved very effectively using the simplex method [110], or the interior point method [81] and belong in the complexity class P. However, in many problems, variables represent quantities which are discrete, e.g., whether an item should be placed in the knapsack or not. Mixed Integer Programming (MIP) can be used to solve linear problems where one or more variables are restricted to be integers. The restriction to integer variables makes the problem NP-hard in general [83]. MIP solvers typically solve such problems using branch and bound. At each node, a linear relaxation of the problem is solved using LP methods in order to get a bound on the objective value. If this bound shows that the subproblem cannot have any solutions better than the best solution found so far, then the subproblem can be pruned. See [34] for more details on MIP solving. However, not all problems can be modeled effectively using only linear constraints. In such cases, other methods may be more appropriate.

Constraint Programming (CP) is a relatively new, and very effective technology for solving combinatorial optimization problems. Constraint programming allows the usage of many kinds of constraints. This expressiveness allows many problems to be modeled much more naturally than in LP. For example, constraints such as disequality (e.g., $x_1 \neq x_2$), multiplication (e.g., $x_1 = x_2 \times x_3$), reification (e.g., $b \leftrightarrow x_1 \geq x_2$), Boolean constraints (e.g., $b_1 = b_2 \land b_3$), element constraints ($x_1 = a[x_2]$ where $a[i]$ is an array of values), occur commonly in problems but are very difficult to express as linear constraints. Constraint Programming is able to handle such constraints effectively, and excels at handling complex constraints over large sets of finite domain variables (see [10] for a large list of standard CP constraints).

The distinguishing feature of Constraint Programming is the use of inference during search to restrict the search space and to detect the failure of subproblems earlier. At each node, an unfixed variable is chosen and the solver branches upon the value of this variable in order to create several more constrained subproblems. As branching decisions are made, the constraints in the problem are analyzed in order to make further inferences. If some constraint is violated by the partial assignment, then there are no solutions within the subtree and the solver can backtrack. Alternatively, the solver may infer that certain variable/value pairs can no longer be taken without violating
some constraint, in which case these values may be pruned to restrict the search space within that subtree. This kind of inference can dramatically reduce the amount of search that a CP solver has to do compared to a naive enumeration approach, and is one of the core strengths of Constraint Programming. A tremendous amount of work has been done to try to improve the amount of inference we can make for various constraints and to make these inferences as fast as possible (e.g., [127, 128, 149, 122, 152]).

Another important feature of Constraint Programming is programmable search. The order in which variables are branched upon, and the order in which their values are tried, can have a dramatic effect on the run time of the algorithm [17, 121, 66, 163]. Many CP systems (e.g., Eclipse, Gecode, CHUFFED) allow standard or custom search strategies to be declared as part of the problem model.

Although tremendous progress has been made in Constraint Programming in the last 20 years, there exists many opportunities for further improvement. In this thesis, we present important contributions to a number of areas in Constraint Programming, including nogood learning, symmetry breaking, dominance, relaxations, and parallelization. The thesis is divided into four parts, each addressing a different area of constraint programming. We give a brief introduction to each of these areas before describing our contributions.

**Nogood Learning** Nogood learning techniques are used to speedup constraint solving by learning implied constraints called *nogoods*. Such nogoods express sets of conditions under which there can be no solutions to the problem, and can be propagated to prune parts of the search space. There are various ways to derive nogoods. The simplest way is to derive them through exhaustive search. Whenever a subtree reached via a set of decisions has been exhaustively searched and has no solutions, the set of decisions we took to get to the subtree forms a nogood, which we call the decision nogood. Such decision nogoods are easy to derive but are also very weak in terms of pruning strength. It has been suggested that such nogoods can be used in CP solvers that perform restarts in order to prevent them from exploring previously explored parts of the search tree [94]. Another way to derive nogoods is via constraint resolution. Such nogoods are used in SAT solvers [142], and in Lazy Clause
CHAPTER 1. INTRODUCTION

Generation solvers [113, 45]. Whenever the solver finds a conflict, it is possible to find the subset of constraints responsible and to resolve them together to derive a new nogood. Such a nogood expresses the reason for the conflict, and can prevent the solver from making assignments that fail for the same reason again. They are very powerful and can provide orders of magnitude speedup on a wide range of problems.

Symmetry Breaking Symmetry breaking techniques are used to speedup constraint solving in problems with symmetries. Many constraint problems contain large amounts of symmetry. In such problems, examining one assignment is often enough to tell us everything we need to know about its symmetric counterparts (whether it is a solution, what its objective value is, etc). These redundancies in the search can exponentially increase the run time. One way to prevent this is to apply some form of symmetry breaking to prevent the CP solver from exploring symmetric parts of the search space. The most popular symmetry breaking techniques are lexicographical symmetry breaking constraints [118, 32, 98, 120, 50], Symmetry Breaking During Search (SBDS) [62], and Symmetry Breaking by Dominance Detection (SBDD) [43]. Lexicographical symmetry breaking constraints can be statically added to the problem to constrain the solver to look only for solutions which are lexicographically least within its equivalence class. SBDS and SBDD are dynamic symmetry breaking techniques which, after each partial assignment has been explored, adds constraints to disallow all of its symmetric counterparts from being solutions.

Relaxation Relaxation techniques seek to relax a problem $P$ into a related but easier problem $P'$ such that solving $P'$ yields useful information on how to solve $P$. This information may include candidate solutions or bounds on the objective. There are many ways to relax a problem. The most popular are linear relaxations [137, 156] and Lagrangian relaxations [46, 111, 95]. In linear relaxations, discrete variables are relaxed into continuous variables, and constraints are relaxed into linear constraints. The relaxed problem can be solved very effectively using linear programming techniques such as the simplex method in order to find bounds or candidate solutions for the original problem. In Lagrangian relaxations, the problem is relaxed by removing
certain constraints and adding corresponding “penalty” terms to the objective function which give a non-zero value when the removed constraint is violated. Solving the relaxed problem can yield bounds or candidate solutions for the original problem.

Dominance Dominance is a powerful concept that underlies many of the other major techniques used in CP. Symmetry breaking, relaxations, and many forms of nogood learning are applications of dominance. When we have pairs of objects (valuations, subproblems, problems) where one is known to be better than, or at least as good as, the other with respect to what we are trying to optimize, e.g., one satisfies the constraints while the other does not, one has a better objective value than the other, then it is sufficient to examine the one that is better, and we can prune the worse one without examining it. Finding and exploiting such dominance relations \[87, 78, 2, 76\] can give us exponential speedups on appropriate problems.

Parallelization Parallelization is an obvious way to try to speedup constraint solving, especially in view of the recent proliferation of multi-core systems. There are a number of ways to parallelize CP solvers. The simplest and most popular is search parallelization, where different parts of the search tree are searched by different processors (e.g., [114, 138, 105]). Another possibility is to parallelize the propagation engine by having different processors execute different propagators (e.g., [132, 167]). The basic search algorithm for constraint solving is almost trivially parallelizable. As such, previous work on parallel search has concentrated mainly on load balancing [130, 138], or on developing an appropriate architecture for parallel search [138, 105].
1.1 Contributions

Part I - Nogood Learning In Chapter 3, we present a new nogood learning technique based on constraint projection. In many combinatorial optimization problems, different search paths lead to subproblems which are actually identical on the remaining unlabeled variables. Nogoods derived via constraint projection can be used to exploit such subproblem dominances. On appropriate problems, our new nogood learning technique provides orders of magnitude speedup compared to a base solver which does not learn nogoods. Our technique is competitive with Lazy Clause Generation, which is the current state of the art in nogood learning. The nogoods derived through constraint projection are different in power to the nogoods derived by Lazy Clause Generation through constraint resolution, and can be exponentially stronger on some problems.

Part II - Symmetry Breaking In Chapter 4, we present a new symmetry breaking technique called SBDS-1UIP, which is an extension of SBDS. The original SBDS technique can be thought of as using symmetric versions of decision nogoods to prune symmetric parts of the search space. In our extension SBDS-1UIP, we use symmetric versions of the 1UIP nogoods derived by Lazy Clause Generation solvers instead. We show that SBDS-1UIP can exploit at least as many symmetries as SBDS, and that it is strictly more powerful on some problems. This is a very interesting result, as SBDS is already a complete method which guarantees that at most one solution in each equivalence class is found by the solver. However, SBDS-1UIP can be strictly more powerful than that. We show that SBDS-1UIP can exploit types of symmetries that no previous general symmetry breaking technique is capable of exploiting.

In Chapter 5, we present two new methods for exploiting almost symmetries. Many problems arising from real world applications do not exhibit full problem symmetry, but instead exhibit almost symmetries, where the symmetries are slightly broken. Such problems often still have large amounts of symmetries that can be exploited for speedup. However, traditional symmetry breaking methods like lexicographical symmetry breaking constraints, SBDS or SBDD cannot directly handle such partially
1.1. CONTRIBUTIONS

broken symmetries. There are several theoretical works in this area [65, 67], but only one general method which has actually been implemented [101]. In this work, the problem is remodeled to separate the subset of constraints which break the symmetry from the rest, so that the traditional symmetry breaking methods can be applied to the ones that satisfy the symmetry. We present two new general methods for exploiting almost symmetries. The first is to treat almost symmetries as conditional symmetries and exploit them by posting conditional symmetry breaking constraints. The second is to modify SDBS-UIP to handle almost symmetries. Both techniques are capable of producing orders of magnitude speedups on appropriate problems.

Part III - Relaxation and Dominance  In Chapter 6, we present a new solver for the Minimization of Open Stacks Problem (MOSP) [162]. The MOSP was the subject of the 2005 Modeling Challenge. It has appeared in many different guises in the literature, e.g., graph pathwidth, gate matrix layout (see [97] for a list of 12 equivalent problems). The MOSP is known to be NP-hard [97]. The previous state of the art in solving the MOSP [53] utilizes a branch and bound algorithm and top down dynamic programming in order to exploit the massive amounts of subproblem equivalences inherent in the problem. This solver was several orders of magnitude faster than all the other entrants in the 2005 Modeling Challenge. Our new solver is many orders of magnitude faster yet again. We identify 4 different dominance relations, which when exploited, provides some 5-6 orders of magnitude speedup. We also find an effective relaxation technique that allows us to speedup the proof of optimality on the hardest instances by a further 3-4 orders of magnitude.

In Chapter 7, we present a new solver for the Talent Scheduling Problem [23]. In the Talent Scheduling Problem, we have a set of actors and a set of scenes. We wish to find an ordering of the scenes such that the cost of hiring the actors are minimized. Each actor is involved in some subset of scenes and must be on location from the first to the last of their scenes, getting paid for each day they are on location. The Talent Scheduling Problem problem has a lot of subproblem equivalences which can be exploited by caching [145], dynamic programming, or similar techniques. We present a new solver based on dynamic programming that is some 2-3 orders of magnitude faster.
than the previous state of the art. We show that using bounded dynamic programming gives 1-2 orders of magnitude speedup compared to a naive dynamic programming approach. Exploiting dominance relations give a further $\sim 2$ times speedup, and utilizing a lower bound calculation based on relaxations gives a further 1 order of magnitude speedup.

In Chapter 8, we completely solve the Maximum Density Still Life Problem [19] for all $n$. The Maximum Density Still Life Problem is based on Conway’s Game of Life. The aim is to find the maximum number of live cells that can appear in an $n \times n$ region in a still life according to the rules of the game. It is an extremely difficult combinatorial optimization problem with a raw search space of $O(2^{n^2})$. Previous search methods using IP [18] and CP [19] could only solve up to $n = 9$, while a CP/IP hybrid method with symmetry breaking [19] could solve up to $n = 15$. An attempt using bucket elimination [90] reduced the time complexity to $O(n^2 2^{3n})$ but increased the space complexity to $O(n 2^n)$. This method could solve up to $n = 14$ before it ran out of memory. A subsequent improvement that combined bucket elimination with search [91], used less memory and was able to solve up to $n = 20$. We show that by using a combination of remodeling, relaxations, bounded dynamic programming and customized search guided by deep mathematical insights, we can completely solve the Maximum Density Still Life problem for all $n$. As a comparison, the previous state of the art method [91] would take in the order of $10^{100}$ years and $10^{30}$ bytes of memory to solve $n = 100$. Our new algorithm can solve $n = 100$ in just a few hours using less than 2 Gb of memory.

Part IV - Parallelization In Chapter 9, we identify cache contention as a serious bottleneck that severely limit the amount of speedup achievable by parallel SAT and Lazy Clause Generation solvers on multi-core systems. We alter the data structures used in the state of the art SAT solver MiniSat [39] to be more cache aware, leading to a sequential SAT solver that is some 80% faster on average and a parallel SAT solver that is some 140% faster on average.
1.1. CONTRIBUTIONS

In Chapter 10, we examine the interaction between branching heuristics and dynamic load balancing schemes. This important theoretical issue has not been properly addressed in the literature. Many analyses of parallel search and load balancing schemes implicitly or explicitly assume that the total amount of work is fixed (e.g. [89]). This assumption is not true for satisfiable problems or for optimization problems, as finding a (good) solution quickly can dramatically reduce the total amount of work required. Much work in CP is concerned with developing strong branching heuristics which can shape and order the branches of the search tree so that (good) solutions are found early on in the search. However, many commonly used parallelization schemes, e.g., splitting the search tree as close to the root as possible [138], focus on maximizing the granularity of the work partitions and completely ignore the branching heuristic. This can result in many of the processors searching in unfruitful parts of the search tree. We analyze this problem and develop a new work stealing scheme called confidence based work stealing, which partitions the work based on how strong the branching heuristic is at each node. The new parallel algorithm produced near linear or super linear speedup on all the problems we tested.

The thesis is organized as follows. In Chapter 2, we introduce the background and notation for the thesis. The remainder of the thesis is divided into 4 parts, each addressing a different area of Constraint Programming. At the beginning of each part, more detailed technical background for that area of CP is given. The four parts cover the areas of: nogood learning (Chapter 3), symmetry breaking (Chapter 4 and 5), relaxation and dominance (Chapter 6, 7 and 8), and parallelization (Chapter 9 and 10). Finally, in Chapter 11, we conclude.
Chapter 2

Background
In this chapter, we provide the basic background and notation required for the remainder of this thesis. We describe Constraint Satisfaction Problems (CSP), Constraint Programming (CP), and Boolean Satisfiability Problems (SAT). More detailed background on nogood learning, symmetry breaking, dominance, relaxations and parallelization techniques will be presented at the beginning of the respective parts of the thesis.

2.1 Constraint Satisfaction Problem

Let ≡ denote syntactical identity, ⇒ denote logical implication and ⇔ denote logical equivalence. A variable \( v \) is a mathematical quantity capable of assuming any value from a set of values called the domain of \( v \). A valuation \( \theta \) over a set of variables \( V \) assigns each \( v \in V \) to a value in its domain. A constraint \( c \) is a logical expression over some set of variables \( V \). If \( \theta \) assigns the variables in \( V \) such that the logical expression \( c \) evaluates to \( \text{true} \), then we say that \( \theta \) satisfies constraint \( c \). Otherwise, we say that \( \theta \) violates constraint \( c \).

A Constraint Satisfaction Problem (CSP) is a tuple \( P \equiv (V, D, C) \), where \( V \) is a set of variables, \( D \) is a set of domains, and \( C \) is a set of constraints. If a valuation \( \theta \) satisfies all the constraints in \( C \), we call it a solution of \( P \). The aim of a constraint satisfaction problem \( P \) is typically to find one solution, all solutions, or to prove that none exist. In a Constraint Optimization Problem, in addition to \( V, D, \) and \( C \), we also have an objective function \( f \) which we wish to minimize or maximize. Typically, we wish to find a solution with the minimal or maximal value of \( f \) among all the solutions of \( P \).

Example 2.1.1. Suppose we want to model the simple graph coloring problem in Fig 2.1, where we want to color the nodes of the graph using only 3 colors so that nodes sharing an edge are of different colors. We can define variables \( V \equiv \{x_1, \ldots, x_6\} \) to represent which color each node is colored to, domains \( D \equiv \{x_i \in \{1, 2, 3\} \mid i = 1, \ldots, 6\} \), and constraints: \( C \equiv \{x_1 \neq x_5, x_1 \neq x_6, x_2 \neq x_3, x_2 \neq x_4, x_3 \neq x_4, x_4 \neq x_5, x_4 \neq x_6, x_5 \neq x_6\} \).
2.2. CONSTRAINT PROGRAMMING

Constraint Programming (CP) is a very effective method for solving combinatorial problems. A typical CP system consists of two parts: 1) a modeling interface where

It is often useful to consider the domains enforced by a CSP as unary constraints (constraints over a single variable). This allows us to manipulate them as logical expressions, which makes the statement and proofs of many results more succinct. We give further definitions and redefine CSP’s slightly.

Each variable $v$ has an initial domain determined by its type, e.g., a Boolean variable $v$ has an initial domain $\{true, false\}$, an integer variable has an initial domain $\mathbb{Z}$, a continuous variable has an initial domain $\mathbb{R}$, etc. A valuation $\theta$ assigns each $v \in V$ to a value in its initial domain. We denote the set of all valuations over $V$ as $\Theta_V$. We denote the subset of $\Theta_V$ that satisfies $c$ as $solns(c)$. Clearly, a constraint is uniquely determined by $solns(c)$. Given an object $O$ (e.g., a constraint, a valuation), let $vars(O)$ denote the set of variables associated with it.

A constraint satisfaction problem is a tuple $P \equiv (V, D, C)$, where $V$ is a set of variables, $D$ is a set of unary constraints, and $C$ is a set of constraints. If a valuation $\theta \in \Theta_V$ satisfies all the constraints in $C$ and $D$, then we call it a solution of $P$. We denote the set of solutions of $P$ as $solns(P)$. We call a problem a finite domain problem if the domain $D$ constrains each variable to take values from a finite set of values.

2.2 Constraint Programming

Constraint Programming (CP) is a very effective method for solving combinatorial problems. A typical CP system consists of two parts: 1) a modeling interface where
CHAPTER 2. BACKGROUND

A CSP can be stated declaratively by the user, and 2) a solver which solves the CSP. There are many kinds of modeling interfaces, including solver specific modeling languages and API's, as well as more generic modeling languages like Zinc [52] or MiniZinc [112]. In this Thesis, we will model using either direct mathematical notation, or the simple modeling language MiniZinc [112].

Modern CP solvers are typically based on constraint propagation. They solve CSP’s by interleaving search with inference. The general algorithm is as follows. We begin with the original problem at the root of the search tree. At each node in the search tree, we propagate the constraints to try to infer variable/value pairs which cannot be taken in any solution to the problem. Such pairs are removed from the current domain. If some variable’s domain becomes empty, then the problem has no solution and the solver backtracks. If all the variables are assigned and no constraint is violated, then a solution has been found and the solver can terminate. If inference is unable to detect either of the above two cases, then the solver further divides the problem into a number of more constrained subproblems and searches each of those in turn. We give some additional definitions before giving the pseudo code for this algorithm.

In an abuse of notation, if a symbol $A$ refers to a set of constraints $\{c_1, \ldots, c_n\}$, we will often also use the symbol $A$ to refer to the constraint $c_1 \land \ldots \land c_n$. This allows us to avoid repetitive use of conjunction symbols. It should be clear from the context which meaning we take. If we apply set operators to $A$ like $\in$, $\cup$, $\cap$, we are treating $A$ as a set of constraints. If we apply logical operators to $A$ like $\land$, $\lor$, $\Rightarrow$, we are treating it as a constraint. A domain $D$ is a false domain iff $D \Rightarrow false$, i.e., one or more variables are constrained by $D$ to the point that it cannot take any value. Let $fixed(D)$ denote the set of variables fixed by a domain $D$, i.e., the subset of variables which are constrained by $D$ so that they can only take one value.

The basic algorithm for solving a CSP is shown in Figure 2.2. The function $solv(D, C)$ returns whether the constraint satisfaction problem $P \equiv (V, D, C)$ is satisfiable. $solv$ is implemented with the help of two functions $prop(C, D)$ and $branch(C, D)$. The function $prop(C, D)$ performs the propagation. It takes as input a set of constraints $C$ and a domain $D$ and returns a domain $D'$ s.t. $D' \Rightarrow D$ and
2.2. CONSTRAINT PROGRAMMING

C \land D \leftrightarrow C \land D', i.e., it finds and removes forbidden variable/value pairs to produce a smaller domain D'. The prop function must propagate strongly enough that if D' fixes all the variables, then either D' gives a solution of C, or D' is a false domain. The function branch(C, D) takes as input a set of constraints C and a domain D and returns a set of constraints B \equiv \{b_1, \ldots, b_k\} s.t. C \land D \Rightarrow b_1 \lor \ldots \lor b_k. The branch function is used to split the problem into several more constrained subproblems and controls the search strategy. We call b_i the decision constraints. Typically, the b_i are simple unary constraints like x = i, x \neq i, x \geq i, or x \leq i.

\begin{verbatim}
solv(D, C)
  1  D' = prop(C, D)
  2  if (D' \Rightarrow false) return false
  3  if (fixed(D') = V) return true
  4  B = branch(C, D')
  5  for (b \in B)
  6    if (solv(D', C \cup \{b\})) return true
  7  return false
\end{verbatim}

Figure 2.2: Basic search algorithm for propagation based constraint solvers.

Example 2.2.1. Consider the simple graph coloring problem above (Figure 2.1). Suppose we branch on the values of the variables in lexicographical order. The steps taken by the search algorithm is shown in Table 2.1. Each line corresponds to a node in the search tree and shows: 1) the depth of the node, 2) the decision that has just been made, 3) the inferences that can be made from that decision, 4) the state of the solver after propagation has been performed (unknown, failure or solution), and 5) the set of decisions to be used to split up the subproblem. Figure 2.3 gives the corresponding search tree. After 9 nodes and 3 failures, the solution x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 1, x_5 = 2, x_6 = 3 is found.

It is easy to modify the algorithm to return the first solution or to return all solutions. To return solutions, we simply add some code in line 3 of Figure 2.2 to store/print the solution before returning. To get all solutions, we return false in line
3 instead of true when we find a solution. It is also easy to modify this algorithm to solve optimization problems via the branch and bound algorithm [93]. Suppose we want to minimize some objective function \( f \). We define a global variable \( best \) which stores the objective value of the best known solution. At every node, in addition to the original constraints \( C \), we also have an additional constraint \( f < best \) which constrains the solver to find a solution better than the best found so far. The basic branch and bound algorithm \( bab(D,C) \) is shown in Figure 2.4. The optimal value is stored in \( best \) at the end of the algorithm.

The performance of these algorithms are highly dependent on exactly how the
functions \textit{prop} and \textit{branch} are implemented, and a vast amount of research in CP is aimed at improving these two functions.

The function \textit{prop}(C,D) is commonly called the propagation engine. There are two primary concerns with regards to the design of the propagation engine: how fast it is, and how much inference it can provide. The first is important because the vast majority of solving time in most CP solvers is spent in the propagation engine. The second is important as it affects how quickly (in terms of search depth) we can determine failure and hence the size of the search tree.

The propagation engine is typically implemented by propagating each constraint individually, until a common fixed point is reached. A propagator \( p_c \) for constraint \( c \) is a function mapping domains to domains s.t. \( c \land D \iff c \land p_c(D) \), i.e., it performs inference based on \( c \) and the current domain \( D \), pruning away any variable/value pairs that cannot be taken given \( c \) and \( D \). We say that \( D \) is a \textit{fixed point} of propagator \( p_c \) if \( p_c(D) = D \). A naive way to implement \textit{prop}(C,D) is shown in Figure 2.5.

This naive algorithm for the propagation engine is not very efficient, as there may be many calls to \( p_c(D') \) which result in no change to \( D' \). Modern solvers utilize a variety of techniques to reduce the number of times propagators are called. The most effective ones are: watching domain change events to lazily wake up propagators, and
\[ prop(C, D) \]
\[ D' = D \]
\[ \text{do} \]
\[ \text{changed} = \text{false} \]
\[ \text{for} \ (c \in C') \]
\[ D'' = p_c(D') \]
\[ \text{if} \ (D'' \neq D') \text{ changed} = \text{true} \]
\[ D' = D'' \]
\[ \text{while} \ (\text{changed}) \]
\[ \text{return} \ D' \]

Figure 2.5: Naive algorithm for the propagation engine.

ordering propagators by priority so that expensive propagators are called less often. See [139] for an in depth analysis of techniques for optimizing propagation engines.

The algorithms for the individual propagators are also of critical importance to the effectiveness of the solver. Enormous amounts of work have been done to try to increase the inference power of propagators and to improve their speed (e.g., [127, 128, 149, 122, 152]). Stronger inference typically costs more time. At one end of the scale, we have the bare minimum amount of inference that a propagator must produce in order for the CP solver to be correct. A propagator \( p_c \) for constraint \( c \) has to be at least checking, i.e., if \( D \) fixes all variables in \( vars(c) \), then \( D \) either satisfies \( c \), or \( p_c \) returns a false domain. At the other end of the scale, we have the maximally strong propagators that makes all possible inferences given the current domain and the constraint. A propagator \( p_c \) for constraint \( c \) is domain consistent if: \( \forall D, D' \) s.t. \( c \land D \Rightarrow D' \), \( p_c(D) \Rightarrow D' \), i.e., it is at least as strong as any correct propagator for \( c \). Finding the right trade off between inference power and speed is an important and difficult part of solving constraint problems effectively.

The function \( branch \) determines the search strategy by deciding how each problem is to be subdivided and the order in which the subproblems are to be examined. Many CP systems (Eclipse, Gecode, CHUFFED, etc) allow standard or custom search strategies to be defined as part of the problem model. Search strategies are often divided into two parts: a variable selector which picks the next variable to be branched
2.3. BOOLEAN SATISFIABILITY

on, and a value selector which determines which values to try next. Some common variable selectors include: lexicographical ordering, smallest domain [66], highest degree [37], domain size divided by degree [14], weighted degree [20], domain size divided by weighted degree [20], etc. Some common value selectors include: smallest, largest, middle out, smaller half, larger half, etc. Many problems also benefit from custom search strategies (e.g., [28, 29, 54]). Search Strategies can have a huge effect on the run time of the search algorithm [17, 121, 66, 163]. A good search strategy should make branching decisions which allow failure to be inferred quickly, reducing the depth and size of the search tree. It should also order the branches of the search tree so that branches more likely to lead to (good) solutions are explored first.

2.3 Boolean Satisfiability

The Boolean Satisfiability Problem (SAT) is a special case of the Constraint Satisfaction Problem. While CP allows arbitrary domains and constraints, SAT only allows Boolean variables and clausal constraints. A SAT clause is a constraint of the form: \( l_1 \lor \ldots \lor l_n \), where each \( l_i \) is either a Boolean variable \( x \) or its negation \( \neg x \). We call \( x \) and \( \neg x \) literals. The specialization to Boolean variables and clauses allows for several powerful optimizations which we describe in this section. Note that

\[
(l_1 \lor \ldots \lor l_n) \equiv (\neg l_1 \land \ldots \land \neg l_k \rightarrow l_{k+1} \lor \ldots \lor l_n) \equiv (\neg l_1 \land \ldots \land \neg l_n \rightarrow \text{false}),
\]

and we will refer to constraints of any of these forms as clauses.

Most modern SAT solvers are based on the DPLL algorithm [35]. The DPLL algorithm is a special case of the more general CP algorithm given in Figure 2.2, where \( prop \) enforces domain consistency on all the clauses in the problem, and \( branch \) picks some unfixed Boolean variable \( v \) and returns \( \{v, \neg v\} \). Due to the specialization to Boolean clauses, the constraints can be propagated to domain consistency very efficiently using the two watched literal scheme [109]. Many modern SAT solvers also utilize a very powerful dynamic search strategy called Variable State Independent Decaying Sum (VSIDS) [109]. In VSIDS, each Boolean variable has an activity count which is increased each time the variable is involved in a conflict. This activity count decays exponentially with time, and the unfixed variable with the highest activity is
picked to be branched on at each search node.

Modern SAT solvers also implement a powerful technique called clause learning [142]. The aim of clause learning is to derive additional clausal constraints which are implied by the original problem. Such learnt clauses can be added to the constraint store without changing the set of solutions to the problem, and can be used to make additional inferences, hopefully reducing the amount of search required.

In clause learning, the SAT solver constructs a directed, acyclic implication graph $G$ as follows. When a literal $m$ becomes true, either through a decision or through propagation, we add a node to the implication graph $G$ representing $m$. If $m$ was inferred through propagation, then there must be some clause $l_1 \land \ldots \land l_n \rightarrow m$ which produced this inference. We add the directed edges $(l_i, m)$ to the implication graph $G$.

We call $l_1 \land \ldots \land l_n \rightarrow m$ the explanation clause for the inference $m$. The explanation clause for an inference $m$ is referenced by an array $\text{expl}[m]$. Also, there is an array $\text{level}[m]$ which returns a pair $(l, p)$ where $l$ is the search depth at which $m$ became true, and $p$ is the number of literals which were set to true at depth $l$ before $m$ was set to true. We define a lexicographical ordering between such pairs so that $(l_1, p_1) < (l_2, p_2)$ iff $(l_1 < l_2 \lor (l_1 = l_2 \land p_1 < p_2))$. Before $m$ can be inferred via the explanation clause, we must have $l_1, \ldots, l_n$ all true. Hence $\text{level}[l_i] < \text{level}[m]$ which ensures that the implication graph is acyclic.

A conflict occurs when propagation causes the domain of some variable $v$ to become empty. At this point, the implication graph must have a node representing $v$ and a node representing $\neg v$. We call these two nodes the conflict nodes. Using the implication graph, we can now derive a new learnt clause.

First, we pick a cut in the implication graph with the following properties: all the decision nodes are on one side of the cut (call it the reason side $R$), the conflict nodes are on the other side (call it the conflict side $S$), and all edges in the cut-set are directed from $R$ to $S$. Many such cuts exists. For example, we can take any literal $m$ made true during the last decision level and define $R = \{l \mid l \in G, \text{level}[l] \leq \text{level}[m]\}$ and $S = \{l \mid l \in G, \text{level}[l] > \text{level}[m]\}$.

The resolution rule states that: if $l_1 \land \ldots \land l_n \rightarrow a$ and $a \land k_1 \land \ldots \land k_p \rightarrow \text{false}$,
then \( l_1 \land \ldots \land l_n \land k_1 \land \ldots \land k_p \rightarrow \text{false} \). We use \( \text{resolve}(c_1, c_2) \) to denote resolution between clauses \( c_1 \) and \( c_2 \). Let \( \text{lits}(L) = \{l_1, \ldots, l_n\} \) if \( L \equiv (l_1 \land \ldots \land l_n \rightarrow \text{false}) \). \( \text{learn}(v, S) \) is an algorithm for deriving a learnt clause.

\[
\text{learn}(v, S)
\]

\[
L = (v \land \lnot v \rightarrow \text{false})
\]

while \( (\text{lits}(L) \cap S \neq \emptyset) \)

\[
a = \operatorname{argmax}_{x \in \text{lits}(L) \cap S} (\text{level}[x])
\]

\[
L = \text{resolve}(L, \text{expl}[a])
\]

return \( L \)

We begin with \( L = (v \land \lnot v \rightarrow \text{false}) \) which is implied by \( v \)'s domain constraint. While \( L \) has any literals which are on the conflict side \( S \) of the cut, we pick the one with the highest level and eliminate it from \( L \) by resolving \( L \) with \( \text{expl}[a] \). Resolution is possible because \( \text{expl}[a] \) is some clause \( l_1 \land \ldots \land l_n \rightarrow a \) and \( L \) is some clause \( a \land k_1 \land \ldots \land k_p \rightarrow \text{false} \). The result of resolving two clauses which are implied by the problem is another clause which is implied by the problem. Hence the new \( L \) is implied by the problem and by induction, the algorithm is correct. The algorithm always terminates because \( G \) is acyclic.

Different choices of the cut give different learnt clauses \( L \), which may have vastly different pruning strengths. A common choice of cut in state of the art SAT solvers is the First Unique Implication Point (1UIP) cut. A literal \( m \) is a Unique Implication Point (UIP) if every path from the last decision node to the conflict nodes in \( G \) pass through \( m \). The 1UIP is the UIP with the highest \( \text{level}[m] \) (first one we meet as we backtrack). We define the 1UIP cut as \( R = \{l \mid l \in G, \text{level}[l] \leq \text{level}[m]\} \) and \( S = \{l \mid l \in G, \text{level}[l] > \text{level}[m]\} \), where \( m \) is the 1UIP.

**Example 2.3.1.** Suppose we tried to solve our simple graph coloring problem using a SAT solver. We first have to remodel the problem into Boolean variables and clauses. We define Boolean variables \( b_{i,k} \), \( i = 1, \ldots, 6, \ k = 1, 2, 3 \) where \( b_{i,k} \leftrightarrow x_i = k \). For clarity, we refer to \( b_{i,k} \) as \( [x_i = k] \) and \( \lnot b_{i,k} \) as \( [x_i \neq k] \). For each \( x_i \), we add \( [x_i = 1] \lor [x_i = 2] \lor [x_i = 3] \), \( [x_i \neq 1] \lor [x_i \neq 2] \), \( [x_i \neq 2] \lor [x_i \neq 3] \), \( [x_i \neq 3] \lor [x_i \neq 4] \), \( [x_i \neq 4] \lor [x_i \neq 5] \), \( [x_i \neq 5] \lor [x_i \neq 6] \), \( [x_i \neq 6] \lor [x_i \neq 1] \), \( [x_i \neq 1] \lor [x_i \neq 2] \lor [x_i \neq 3] \), \( [x_i \neq 2] \lor [x_i \neq 3] \lor [x_i \neq 4] \), \( [x_i \neq 3] \lor [x_i \neq 4] \lor [x_i \neq 5] \), \( [x_i \neq 4] \lor [x_i \neq 5] \lor [x_i \neq 6] \), \( [x_i \neq 5] \lor [x_i \neq 6] \lor [x_i \neq 1] \), \( [x_i \neq 6] \lor [x_i \neq 1] \lor [x_i \neq 2] \), \( [x_i \neq 1] \lor [x_i \neq 2] \lor [x_i \neq 3] \), \( [x_i \neq 2] \lor [x_i \neq 3] \lor [x_i \neq 4] \), \( [x_i \neq 3] \lor [x_i \neq 4] \lor [x_i \neq 5] \), \( [x_i \neq 4] \lor [x_i \neq 5] \lor [x_i \neq 6] \), \( [x_i \neq 5] \lor [x_i \neq 6] \lor [x_i \neq 1] \), \( [x_i \neq 6] \lor [x_i \neq 1] \lor [x_i \neq 2] \).
infer the 1UIP learnt clause until we end up with the 1UIP cut is given by:

\[ \{ [x_6 = 2], [x_6 \neq 2] \} \]

This learnt clause can provide additional inference. For example, when we try \( \neg x_1 \), we can now immediately infer \( \neg x_4 \). When we then try \( x_2 = 2 \), we can infer \( \neg x_4 \neq 2 \), \( x_4 = 1 \), \( x_3 = 3 \), leading us to a solution in one less conflict.

Figure 2.6: Implication graph for Example 2.3.1. Decision literals are double boxed.

Table 2.2: Steps taken by learn(v, S) algorithm in Example 2.3.1.

<table>
<thead>
<tr>
<th>( L )</th>
<th>( a )</th>
<th>expl[( a )]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [x_6 = 2] \land [x_6 \neq 2] \rightarrow false )</td>
<td>( [x_6 = 2] )</td>
<td>( [x_6 = 2] \rightarrow [x_6 \neq 2] )</td>
</tr>
<tr>
<td>( [x_5 = 2] \land [x_6 = 2] \rightarrow false )</td>
<td>( [x_6 = 2] )</td>
<td>( [x_6 \neq 1] \land [x_6 \neq 3] \rightarrow [x_6 = 2] )</td>
</tr>
<tr>
<td>( [x_6 \neq 1] \land [x_6 \neq 3] \land [x_5 = 2] \rightarrow false )</td>
<td>( [x_5 = 2] )</td>
<td>( [x_5 \neq 1] \land [x_5 \neq 3] \rightarrow [x_5 = 2] )</td>
</tr>
<tr>
<td>( [x_6 \neq 1] \land [x_6 \neq 3] \land [x_5 \neq 1] \land [x_5 \neq 3] \rightarrow false )</td>
<td>( [x_5 \neq 3] )</td>
<td>( [x_5 \neq 3] \rightarrow [x_5 \neq 3] )</td>
</tr>
<tr>
<td>( [x_4 = 3] \land [x_6 \neq 1] \land [x_5 \neq 1] \land [x_5 \neq 3] \rightarrow false )</td>
<td>( [x_4 = 3] )</td>
<td>( [x_4 = 3] \rightarrow [x_5 \neq 3] )</td>
</tr>
<tr>
<td>( [x_4 = 3] \land [x_6 \neq 1] \land [x_5 \neq 1] \rightarrow false )</td>
<td>( [x_4 = 3] )</td>
<td>( [x_4 = 3] \rightarrow [x_5 \neq 3] )</td>
</tr>
</tbody>
</table>

\( [x_i \neq 3] \lor [x_i \neq 1] \). For each CP constraint \( x_i \neq x_j \) we had before, we simply have the clauses \( [x_i \neq k] \lor [x_j \neq k] \) for \( k = 1, 2, 3 \). This is sufficient to model the problem as a SAT problem.

Suppose we run the same search as before until the first conflict. Figure 2.6 shows the implication graph at that point. \( [x_6 = 2] \) is the conflict variable. \( [x_4 = 3] \) is the 1UIP, so the 1UIP cut is given by: \( S = \{ [x_5 \neq 3], [x_6 \neq 3], [x_5 = 2], [x_6 = 2], [x_6 \neq 2] \} \). The algorithm learn runs as shown in Table 2.2. We begin with the conflict clause \( [x_6 = 2] \land [x_6 \neq 2] \rightarrow false \) and proceed to eliminate literals from \( S \) using clause resolution until we end up with the 1UIP learnt clause \( [x_4 = 3] \land [x_6 \neq 1] \land [x_5 \neq 1] \rightarrow false \). This learnt clause can provide additional inference. For example, when we try \( x_1 = 1 \), we can now immediately infer \( x_4 \neq 3 \). When we then try \( x_2 = 2 \), we can infer \( x_4 \neq 2 \), \( x_4 = 1 \), \( x_3 = 3 \), leading us to a solution in one less conflict.
Part I

Nogood Learning
I.1 Introduction

Nogood learning techniques are used to speed up constraint solving by learning implied constraints called nogoods which can be used to prune parts of the search space. The learnt clauses described in Section 2.3 are simply nogoods for the specialized domain of SAT. Nogoods state conditions under which the problem can have no solutions. More formally:

Definition I.1.1. A nogood \( n \) for a problem \( P \equiv (V, D, C) \), is a constraint of form: \( n \equiv \neg(l_1 \land \ldots \land l_k) \), where \( l_i \) are arbitrary constraints, such that \( C \land D \Rightarrow n \), i.e., the problem constraints imply that the constraints \( l_i \) cannot all be satisfied in any solution.

Once we have derived a nogood, we can use it to prune the search space. If the current domain in a subproblem make all the \( l_i \)'s in the nogood true, then that subproblem has no solutions and can be pruned. If the domain make all but one of the \( l_i \)'s true, then the remaining \( l_i \) must be false for any solution within this subproblem. Typically, the \( l_i \)'s are very simple unary constraints such as \( x = v \), \( x \neq v \), \( x \geq v \), \( x \leq v \). However, as we shall see in Chapter 3, more complex constraints can also be used as terms in nogoods. We will sometimes write a nogood in the equivalent form \( n \equiv l_1 \land \ldots \land l_{k-1} \rightarrow \neg l_k \), to emphasize that we are using \( l_1, \ldots, l_{k-1} \) true to infer \( l_k \) false. There are several ways to derive nogoods, which we describe in the following sections.

I.2 Deriving Nogoods through Search

We can derive nogoods through exhaustive search. Let \( P \equiv (V, D, C) \) be a problem. Let \( P' \equiv (V, D, C \cup \{d_1, \ldots, d_k\}) \) be the subproblem of \( P \) reached after the set of decisions \( d_1, \ldots, d_k \). If \( P' \) is exhaustively searched and no solutions are found, then clearly, \( n \equiv \neg(d_1 \land \ldots \land d_k) \) is a nogood of \( P \). We call \( n \) the decision nogood of \( P' \).

Decision nogoods are easy to derive, but are also very weak, because they are only capable of pruning valuations that have already been examined. In a simple CP solver that does exhaustive search without any restarts or backjumping, decision nogoods
never prune anything, since we never visit the same valuation twice. However, there are search strategies, e.g., randomized search [63] or dynamic search [126], where the search may occasionally be restarted. It has been suggested that decision nogoods can be learned right before each restart in order to prevent the solver from examining the same parts of the search tree again [94].

Nogoods can be strengthened by removing unnecessary terms from them. For example, given a nogood \( n \equiv \neg(d_1 \land \ldots \land d_k) \), if some subset \( \{d_{a_1}, \ldots, d_{a_m}\} \) of the \( d_i \)'s are sufficient for failure, i.e., \( C \land D \Rightarrow \neg(d_{a_1} \land \ldots \land d_{a_m}) \), then \( n' \equiv \neg(d_{a_1} \land \ldots \land d_{a_m}) \) is strictly stronger than \( n \) in terms of pruning strength.

In [94], it is shown that in a solver that uses only binary branching, if the first branch is called the positive decision and the second is called the negative decision, then the negation of the conjunction of the set of positive decisions gives a valid nogood. While this optimization can improve the propagation strength of decision nogoods, it does not increase its theoretical pruning power, as they are still only capable of pruning valuations that have already been examined.

In [36], the author proposes to strengthen the decision nogoods by finding subsets of \( \{d_1, \ldots, d_{k-1}\} \) which are sufficient to conflict with the last decision \( d_k \), i.e., some subset \( \{d_{a_1}, \ldots, d_{a_m}\} \) s.t. \( C \land D \Rightarrow \neg(d_{a_1} \land \ldots \land d_{a_m} \land d_k) \). A simple algorithm proposed is to remove from \( \{d_1, \ldots, d_{k-1}\} \) any \( d_i \) where \( \text{vars}(d_i) \) is not connected to \( \text{vars}(d_k) \) in the constraint graph, as such decisions cannot possibly be contributing to the inconsistency with \( d_k \). While this is reasonably efficient, the subsets so found are not minimal (in the sense that no smaller subset is sufficient to conflict with \( d_k \)), and there are no cheap ways to calculate minimal subsets. However, such strengthening represent a real increase in their pruning power, making them capable of pruning valuations that have never been examined before.

### I.3 Deriving Nogoods through Resolution

Conflicts directed nogood learning is a class of methods that derive nogoods using constraint resolution. When a conflict is found during search, the set of constraints involved in the conflict are found and are resolved together in some way to derive
I.3. DERIVING NOGOODS THROUGH RESOLUTION

a new nogood. The nogood explains why the failure occurred, and can prevent the search from failing again in the same way. Nogoods derived through these methods are typically significantly more powerful than the decision nogoods described in the previous section [165]. While decision nogoods can only prevent the solver from exploring a valuation that has previously been explored, nogoods derived through resolution can prevent the solver from exploring valuations that have never been explored before. They can produce exponential speedup on many problems and can change the asymptotic complexity of the search. We first give some definitions.

**Definition I.3.1.** Given current domain $D$, suppose the propagator for constraint $c$ makes an inference $m$, i.e., $c \land D \Rightarrow m$. An explanation for this inference is a nogood: $\text{expl}(m) \equiv l_1 \land \ldots \land l_k \Rightarrow m$ s.t. $c \Rightarrow \text{expl}(m)$ and $D \Rightarrow l_1 \land \ldots \land l_k$.

The nogood $\text{expl}(m)$ explains why $m$ has to hold given $c$ and the current domain $D$. We can consider $\text{expl}(m)$ as the fragment of the constraint $c$ from which we inferred that $m$ has to hold.

**Example I.3.2.** Suppose we have constraint $x \leq y$ and current domain $x \in \{3, 4, 5\}$. A propagator may infer that $y \geq 3$, and a valid explanation can be $x \geq 3 \Rightarrow y \geq 3$.

Suppose we have constraint $\text{alldiff}(x, y, z)$ and current domain $x, y, z \in \{1, 2\}$. A propagator may infer $false$ and a valid explanation can be $x \geq 1 \land x \leq 2 \land y \geq 1 \land y \leq 2 \land z \geq 1 \land z \leq 2 \Rightarrow false$.

Given two nogoods containing complementary terms, we can resolve them together to derive a new nogood.

**Lemma I.3.3.** Given a problem $P$, if $n_1 \equiv \neg (a \land l_1 \land \ldots \land l_p)$ and $n_2 \equiv \neg (\neg a \land m_1 \land \ldots \land m_q)$ are both nogoods of $P$, then so is $n_3 \equiv \neg (l_1 \land \ldots \land l_p \land m_1 \land \ldots \land m_q)$.

**Proof.** $C \land D \Rightarrow n_1 \land n_2 \Rightarrow n_3$. □

We call expressions of the form $x = v$ equality literals, expressions of the form $x \neq v$ disequality literals, and expressions of the form $x \geq v$ or $x \leq v$ inequality literals.

We now give a brief history of the development of conflict directed nogood learning.
One of the earliest conflict directed nogood learning schemes is described in [136]. In this system, constraints are only checked for failure w.r.t. the variables that are fixed, i.e., no propagation other than failure is allowed. At each node, the branching heuristic picks a variable and branches on its value. Suppose for current domain $D$, each variable $x_i \in \text{fixed}(D)$ is set to $x_i = v_i$ for some constant $v_i$. If the propagator for some constraint $c$ detects failure, and $V'$ is the set of variables in $\text{vars}(c)$ that are fixed, then we can trivially derive the explanation for failure $\wedge_{x_i \in V'} x_i = v_i \rightarrow \text{false}$. If $x_k = v_k$ was the last decision we made before this conflict, we store this explanation as $\text{expl}(x_k \neq v_k)$.

At any non-leaf node, when all the values of the branching variable have been tried and failed, it is possible to resolve together the nogoods to derive a new one. Suppose $x_k$ was the branching variable for the non-leaf node and $D_k$ was its domain. From the domain of $x_k$, we have the nogood $\neg(\wedge_{v \in D_k} x_k \neq v)$. We can resolve this nogood with each of $\text{expl}(x_k \neq v), v \in D_k$ to get a new nogood $n$. $n$ is added to the solver as a normal constraint. $n$ is currently violated. We backtrack to the earliest decision level where $n$ is violated and use $n$ to fail that node. The algorithm then continues. Unfortunately, this rather primitive form of conflict directed nogood learning can only handle the simplest forms of propagations, i.e., failure from fixed variables, and thus is not useful for general CP solving.

A major advance in conflict directed nogood learning was made in the context of SAT solving [142]. The nogood learning scheme for SAT was described in detail in Chapter 2. One major improvement was that due to the restriction to Boolean variables and clauses, explanations can be trivially derived even when the clauses are propagated to domain consistency rather than only checked for failure (the explanation is simply the clause itself). Another important improvement was the development of a different resolution scheme. In SAT, the implication graph formed by the explanation nogoods could be cut in a variety of ways, and the explanation nogoods on one side of the cut is resolved to form a nogood. Different cuts lead to different nogoods of differing strength. In particular, the 1UIP cut, which aims to derive a nogood that is as local to the conflict as possible, has been found to be very effective at reducing search. SAT nogood learning produces orders of magnitude speedup on structured
problems and greatly advanced the industrial use of SAT solving.

More progress was made in CP nogood learning with the introduction of generalized nogoods or g-nogoods [84], where disequality literals were allowed to appear in nogoods. Previous work on nogoods had mostly only used equality literals (called s-nogoods in [84]). G-nogoods are significantly more expressive than s-nogoods, as each g-nogood can potentially represent an exponential number of s-nogoods. This increase in expressiveness allowed arbitrary inferences to be explained, and also increased the pruning power of the nogoods derived. Arbitrary inferences can be explained as follows. Suppose we had initial domain $D$, current domain $D'$, and the value $v$ was removed from variable $x$ by constraint $c$, then we can explain this using:

$$\land_{y \in (vars(c) \setminus \{x\})} \land_{w \in (D_y \setminus D'_y)} y \neq w \rightarrow x \neq v.$$ 

The g-nogoods explaining the propagations form an implication graph just as in the case of SAT, and we can derive a nogood from each conflict in an analogous manner.

Lazy Clause Generation (LCG) [113, 45] represents the current state of the art in CP nogood learning. LCG further extends the types of literals that can be included in nogoods with inequality literals. This extension is particularly important for propagators that use variable bounds for inference. Each LCG nogood can represent a linear number of g-nogoods. This increase in expressiveness makes the LCG nogoods yet more powerful. Modern LCG solvers like CHUFFED can beat state of the art non-learning solvers like Gecode by very significant margins on a wide variety of problems. See [106] for example.

### I.4 Contributions

In Chapter 3, we describe a new method for deriving nogoods based on constraint projections. This differs significantly from most previous methods which rely on exhaustive search or constraint resolution. In many combinatorial search problems, it is possible for multiple search paths to lead to subproblems that are actually equivalent, in the sense that every solution of one corresponds to a solution of the other. Such equivalences arise because the projections of the subproblems onto the set of unfixed variables often yield the same problem. We show how we can exploit such
redundancies by using constraint projections to derive nogoods. Such nogoods can give us exponential speedups on appropriate problems. They differ in power and expressiveness compared to LCG nogoods, and can be exponentially stronger on some problems.
Chapter 3

Automatic Caching via Constraint Projection
3.1 Introduction

In many combinatorial search problems, different search paths can lead to subproblems which are actually identical on the remaining unlabeled variables. Consider the simple problem $C \equiv \{\text{alldiff}(x_1, x_2, x_3, x_4), x_1 + x_2 + x_3 + x_4 \geq 20, x_1 + 2x_2 \leq 10\}$. Consider the two partial assignments $x_1 = 1, x_2 = 2$, and $x_1 = 2, x_2 = 1$. They both lead to the same problem on $x_3, x_4$, i.e., $\text{alldiff}(1, 2, x_3, x_4), x_3 + x_4 \geq 17$. Clearly, examining one search path is enough for us to prune the other. Such subproblem equivalences are very common in certain types of problems, leading to a large amount of redundancy in the search. There are a number of methods to prevent this, e.g., caching [145], dynamic programming [11], nogood learning [113, 45], as well as methods that rely on certain problem structures, e.g., decomposition [86, 100] and variable elimination [90]. We briefly describe these methods.

Caching can be used to store results from solving subproblems (e.g., failure, solutions, bounds), so that they can be reused when an equivalent subproblem is encountered later in the search. For caching to be effective, the lookup operation must be efficient. A popular way to implement this operation is to map each subproblem to a *key* which is then used to store and access the information for that subproblem. While caching can be very effective, it can also be very laborious and error prone. It requires the careful manual construction of a key for each model and search strategy, and the modification of the solver to use these keys. The large amount of work required make it quite difficult to use caching as a general method.

Dynamic programming can also exploit subproblem equivalences. However, many constraint problems cannot easily be expressed as dynamic programs, e.g., it may not be recursive, or it may have non-recursive global constraints or side constraints. Also, unlike constraint programming, dynamic programming does not use inference, thus many opportunities for pruning will be lost, leading to a much larger search space.

Nogood learning is another possibility for exploiting subproblem equivalences. Nogood learning techniques like Lazy Clause Generation (LCG) [113, 45] derive nogoods using constraint resolution. Such nogoods explain the reasons why a subproblem failed, and can potentially prune off equivalent subproblems later in the search.
3.1. INTRODUCTION

However, there is no guarantee that it will. Whether it does depends on a number of factors such as how propagators explain their propagations, the order in which inferences were made, and which particular cut in the implication graph was used to derive the nogood.

There are certain structural properties of the constraint graph that make a problem more likely to have subproblem equivalences. For example, problems which can be decomposed by fixing a small number of variables will often have subproblem equivalences when one side of the decomposition has been labeled. We can prevent such equivalent subproblems by caching on the variables in the cut [86, 100]. Subproblem equivalences also often arise in problems where the constraint graph has low pathwidth (see [85] for a definition of pathwidth). These can be exploited by methods such as variable elimination [90].

In this chapter we present a new method called automatic caching via constraint projection. Our method differs from the above in that it can automatically detect and exploit caching opportunities in arbitrary optimization problems, and does not rely on problem structure such as low pathwidth or decomposability. The principal insight of our work is to define a key based on constraint projection that can be efficiently computed during search and can be used to exploit a relatively general notion of re-usability which we refer to as subproblem dominance. We cache failures only, thus our method is a form of nogood learning, where the terms of the nogoods consist of constraint projections. We experimentally demonstrate the effectiveness of our approach, which has been implemented in the competitive CP solver Chuffed. We also provide interesting insights into the relationships between subproblem dominance, dynamic programming, symmetry breaking and nogood learning.

The rest of the chapter is organized as follows. Section 3.2 defines subproblem dominance and subproblem equivalence, and shows how we can make use of these notions to improve search by caching. Section 3.3 gives some examples of manual keys to support caching of specific models. Section 3.4 shows how we can define keys for arbitrary problems on a per constraint basis and taking into account the propagation algorithm of the constraint. Section 3.5 shows how to construct keys and efficient representations of keys for common primitive and global constraints. Section 3.6
illustrates the keys generated by constraint projection on a number of example problems. Section 3.7 discusses earlier work on caching in constraint programming and the relationship with related approaches such as dynamic programming and symmetry elimination. Section 3.8 provides an experimental evaluation showing that automatic caching can be very effective when subproblem equivalence exists. Finally, Section 3.9 provides our conclusions.

3.2 Subproblem Dominance and Equivalence

We restrict our analysis to search algorithms which make decisions on single variables, i.e., decisions of the form \( x = v, x \neq v, x \geq v, x \leq v, \) or \( x \in S \) for some set \( S \). This restriction rules out certain kinds of search, but most commonly used search algorithms are still covered. We express subproblems as \( P' \equiv (V, D', C) \) where \( C \) are the original constraints, and \( D' \) is the current domain at propagation fixpoint. This is possible since all decisions are unary constraints and can be thought of as domain reductions (changing \( D \), but leaving \( C \) unchanged).

We use \( \pi_U(c) \) to denote the projection of a constraint \( c \) onto a subset of its variables \( U \). That is, \( \pi_U(c) \) is a constraint over \( U \) such that \( \theta \in \text{solns}(\pi_U(c)) \) iff \( \theta \) can be extended to a solution of \( c \) by some labelling of the variables in \( \text{vars}(c) \setminus U \). For example, the projection of \( \text{alldiff}([x_1, x_2, x_3, x_4]) \) onto the subset of variables \( U \equiv \{x_3, x_4\} \) is: \( \pi_U(\text{alldiff}([x_1, x_2, x_3, x_4])) \iff \text{alldiff}([x_3, x_4]) \). The projection of \( (x_1 = 1 \land x_1 + x_2 + x_3 \geq 10) \) onto \( U \equiv \{x_2, x_3\} \) is: \( \pi_U(x_1 = 1 \land x_1 + x_2 + x_3 \geq 10) \iff x_2 + x_3 \geq 9 \). Typically, we will use projection to project out variables which are already fixed by the constraint (as in the second example above). For domain constraints \( D \equiv \{x_i \in S_i \mid x \in V\} \), we use subscripts to denote projection onto \( V' \subseteq V \), i.e., \( D_{V'} \equiv \{x_i \in S_i \mid x \in V'\} \).

We define subproblem dominance and subproblem equivalence as follows:

**Definition 3.2.1.** Let \( P \equiv (V, D, C) \) and \( P' \equiv (V, D', C) \) be two different subproblems arising during search. Let \( F \) be the fixed variables in \( D \), i.e., \( F = \text{fixed}(D) \), and \( U \) be the unfixed variables in \( D \), i.e., \( U = \text{vars}(C) \setminus F \). Let \( F' \) be the fixed variables in
3.2. SUBPROBLEM DOMINANCE AND EQUIVALENCE

$D'$, i.e., $F' = \text{fixed}(D')$, and $U'$ be the unfixed variables in $D'$, i.e., $U' = \text{vars}(C) \setminus F'$. $P$ dominates $P'$ if:

- $F = F'$
- $\pi_U(C \land D') \Rightarrow \pi_U(C \land D)$

$P$ dominates $P'$ if the same set of variables are fixed in each, and the projection of $P'$ onto $U$ implies the projection of $P$ onto $U$, i.e., $P'$ constrains the unfixed variables at least as strongly as $P$. Note that dominance may hold even if the fixed variables in $P$ and $P'$ are fixed to different values. If $P$ dominates $P'$, then each solution of $P'$ corresponds to a solution of $P$, as formalized by the following theorem.

**Definition 3.2.2.** Let $fx(D)$ be a valuation over the set of variables $\text{fixed}(D)$, which maps each variable to the value it is fixed to in $D$. Given a problem $P \equiv (V, D, C)$, let $fx(P) \equiv fx(D)$.

**Theorem 3.2.3.** Suppose $P$ dominates $P'$. Then $\forall \theta \in \text{solns}(P'), (\theta_U \cup fx(P)) \in \text{solns}(P)$. \qed

**Proof.** Suppose $\theta \in \text{solns}(P')$. By the definition of projection, $\theta_U \in \text{solns}(\pi_U(C \land D'))$. By the definition of subproblem dominance, we have $\theta_U \in \text{solns}(\pi_U(C \land D))$. Since $D$ fixes all variables not in $U$, we then have $\theta_U \cup fx(D) \in \text{solns}(C \land D)$, so $\theta_U \cup fx(P) \in \text{solns}(P)$. \qed

**Corollary 3.2.4.** If $P$ dominates $P'$ and $P$ has no solutions, then $P'$ has no solutions.

**Example 3.2.5.** Consider the problem $P_{\text{init}} \equiv (V, D_{\text{init}}, C)$, where $V \equiv \{x_1, \ldots, x_5\}$, $D_{\text{init}} \equiv \{x_1 \in \{1, 2, 3\}, x_2 \in \{1, 2, 3, 4\}, x_3 \in \{2, 3, 4\}, x_4 \in \{3, 4, 5\}, x_5 \in \{3, 4, 5\}\}$ and $C \equiv \{x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20\}$. Consider two subproblems $P \equiv (V, D, C)$ and $P' \equiv (V, D', C)$ where $D_F \equiv \{x_1 = 3, x_2 = 1\}$, and $D'_F \equiv \{x_1 = 1, x_2 = 3\}$. Then $U \equiv \{x_3, x_4, x_5\}$ and $D_U \equiv D'_U \equiv \{x_3 \in \{2, 3, 4\}, x_4 \in \{3, 4, 5\}, x_5 \in \{3, 4, 5\}\}$. $P$ dominates $P'$ since $\pi_U(C \land D') \Leftrightarrow (x_3 + x_4 + 2x_5 \leq 13 \land D'_U) \Rightarrow (x_3 + x_4 + 2x_5 \leq 15 \land D_U) \Leftrightarrow \pi_U(C \land D)$. \qed
For the remainder of the chapter we will often make the implicit assumption that for subproblem $P$, $P \equiv (V, D, C)$ where $C$ is the set of constraints common to all subproblems considered and that, given domain $D$, $F = \text{fixed}(D)$ and $U = \text{vars}(C) \setminus F$.

**Definition 3.2.6.** $P$ and $P'$ are equivalent iff $P$ dominates $P'$ and $P'$ dominates $P$.

**Proposition 3.2.7.** If $P$ and $P'$ are equivalent then $\forall \theta \in \Theta_U, (\theta \cup f_x(P)) \in \text{solns}(P)$ iff $(\theta \cup f_x(P')) \in \text{solns}(P')$. □

**Example 3.2.8.** Consider the problem $P_{\text{init}} \equiv (V, D_{\text{init}}, C)$, where $V \equiv \{x_1, \ldots, x_5\}$,
$D_{\text{init}} \equiv \{x_1 \in \{1,2,3\}, x_2 \in \{1,2,3,4\}, x_3 \in \{2,3,4\}, x_4 \in \{3,4,5\}, x_5 \in \{3,4,5\}\}$
and $C \equiv \{\text{alldiff}([x_1,x_2,x_3,x_4,x_5])\}$. Consider two subproblems $P \equiv (V, D, C)$ and $P' \equiv (V, D', C)$ where $D_F \equiv \{x_1 = 1, x_2 = 2\}$, and $D'_F \equiv \{x_1 = 2, x_2 = 1\}$. Both subproblems project to $\text{alldiff}([1,2,x_3,x_4,x_5]) \wedge \{x_3 \in \{2,3,4\}, x_4 \in \{3,4,5\}, x_5 \in \{3,4,5\}\}$ and are therefore equivalent. □

Detecting subproblem dominances allows the search to avoid exploring subproblems known to have no solutions. We alter the basic algorithm for $\text{solv}(V, D, C)$ shown in Figure 2.2 by adding the two lines marked with asterisks in Figure 3.1. The first line checks to see if the current subproblem is dominated by something in the cache $N$. If so, we can immediately fail it. The second line adds a newly failed subproblem to the cache $N$.

It is also correct to use this modified algorithm for $\text{solv}(V, D, C)$ in the branch and bound algorithm shown in Figure 2.4. This is because the constraint $f < b$, which constrains the solver to find a solution better than the best solution found so far, strictly increases in strength as the search goes on, and any subproblem that fails under one set of constraints must also fail under a strictly stronger set of constraints.

### 3.3 Manual Keys for Caching

The principal difficulty in implementing the caching algorithm is to make the lookup and test for subproblem dominance sufficiently efficient. We want a key to represent
3.3. MANUAL KEYS FOR CACHING

\[ \text{solv}(D, C) \]
\[ D' = \text{prop}(C, D) \]
\[ \text{if } (D' \Rightarrow \text{false}) \text{ return false} \]
\[ \text{if } (\text{fixed}(D') = V) \text{ return true} \]
\* \[ \text{if } (\exists P \in \text{Cache} \text{ where } P \text{ dominates } (D', C)) \text{ return false} \]
\[ B = \text{branch}(C, D') \]
\[ \text{for } (b \in B) \]
\[ \quad \text{if } (\text{solv}(D', C \land b)) \text{ return true} \]
\* \[ N = N \cup \{(D', C)\} \]
\[ \text{return false} \]

Figure 3.1: Search algorithm taking advantage of subproblem dominance.

Figure 3.2: Two starting sequences for the Black Hole problem that lead to an identical remaining subproblem.

subproblems which allows dominance to be detected efficiently (preferably in \(O(1)\) time). Naively, one may think that \(D\) could be used as a key to characterize subproblem \(P \equiv (V, D, C)\). However, while \(D\) certainly uniquely identifies the subproblem, it is useless as a key since \(D\) is different for each subproblem in the search tree and will never produce a match. Thus we need to find a more powerful key; one that can be identical for equivalent subproblems reached through different sets of decisions. We now present hand crafted keys for three problems.

Example 3.3.1. Consider the Black Hole Problem [60], which seeks to find a solution to the Black Hole patience game. In this game the 52 playing cards (of standard 52-card deck) are laid out in 17 piles of 3, with the ace of spades starting in a “black hole”. Each turn, a card at the top of one of the piles can be played into the black hole if it is \(\pm 1\) from the card that was played previously, with king wrapping back around to ace. The aim is to play all 52 cards. This was one of two examples used
to illustrate CP with caching in [145]. Suppose the first $k$ cards have been played. The remaining subproblem only depends on the set of unplayed cards, and the value of the last card played. Subproblem equivalences arise because, for any particular assignment, there may be many permutations of the first $k - 1$ assignments which also satisfy the constraints, and all of these will lead to an equivalent subproblem, as exemplified in Figure 3.2. We can exploit this by defining the key as $(l, S)$, where $l$ is the value of the last card played, and $S$ is the set of unplayed cards. This key allows us to detect the equivalence of all of these subproblems and avoid the redundant search.

\[ \square \]

Example 3.3.2. Consider the Graph Coloring Problem, of coloring a graph with some set of colors (represented by integers) so that no two adjacent nodes have the same color. Given a partial (coloring) labeling of the nodes, define the frontier as the set of labeled nodes which are adjacent to an unlabeled node. The problem of coloring the remaining uncolored nodes is characterized by the set of labeled nodes, and the values of the labeled nodes on the frontier. It does not depend on the values of the non-frontier labeled nodes, as their value can have no further effect on the unassigned nodes. Subproblem equivalences arise because there could be many different ways to label the non-frontier labeled nodes while producing the same frontier, and all of these lead to equivalent subproblems. Whether this occurs frequently depends on the structure of the graph and the labeling strategy. We can define the key as
3.3. MANUAL KEYS FOR CACHING

\((L, [(x, d_x) | x \in F])\), where \(L\) is the set of labeled nodes, \(F\) is the set of frontier nodes, and \(d_x\) is the value assigned to node \(x\). This key allows us to detect the equivalence of these subproblems and avoid the redundant search. The partially colored graph in Figure 3.3(a) has a frontier (the second column) labeled 3, 4, 3, which is identical to the frontier for the graph in Figure 3.3(b).

\[\square\]

Example 3.3.3. The Minimization of Open Stacks Problem (MOSP) [162] can be described as follows. A factory manufactures a number of different products in batches, i.e., all copies of a given product need to be finished before a different product is manufactured, so there are never two batches of the same product. Each customer of the factory places an order requiring one or more different products. Once one product in a customer’s order starts being manufactured, a stack is opened for that customer to store all products in the order. Once all the products for a particular customer have been manufactured, the order can be sent and the stack is freed for use by another order. The aim is to determine the sequence in which the products should be manufactured to minimize the maximum number of open stacks, i.e., the maximum number of customers whose orders are simultaneously active. We give a simplified model for the MOSP based on [28], in the modeling language MiniZinc [112].

\[
\begin{align*}
\text{int: } & n; & & \text{\% number of customers} \\
\text{\% which pairs of customers share at least 1 product} \\
\text{array[1..n,1..n] of 0..1: } & W; \\
\text{array[1..n] of var 1..n: } & s; & & \text{\% customer start time} \\
\text{array[1..n] of var 1..n: } & e; & & \text{\% customer end time} \\
\text{array[1..n] of var 1..n: } & x; & & \text{\% customer end order} \\
\text{var 0..n: } & \text{objective; } & & \text{\% number of stacks} \\
\end{align*}
\]

class inverse(e,x);
constraint forall (i in 1..n) (minimum(s[i], [e[j] | j in 1..n where W[i,j]]));
constraint forall (t in 1..n) (sum (i in 1..n) (bool2int((s[i] <= t) \ (e[i] >= t))) <= objective);

solve :: int_search(x, input_order, indomain_min, complete)
    minimize objective;

Each MOSP instance is defined by \( n \), the number of customers, and a Boolean function \( W(i, j) \) which tells us whether customers \( i \) and \( j \) share a product. We have variables, \( s[i] \in \{1, \ldots, n\} \) representing when each customer’s stack is opened, and \( e[i] \in \{1, \ldots, n\} \) representing when each customer’s stack is closed. We channel the customer end times with the customer end order \( x \) with inverse. Each customer’s stack must be opened before the stack of any of the customers with which they share a product closes, because the product they share must have begun production. This gives rise to the minimum constraints. Lastly, the number of stacks which are open during each time period is less than or equal to the total number of stacks we need, giving rise to the linear inequality constraints. We label the \( x[i] \) in order, i.e., we pick which customer closes first, then which customer closes second, etc.

This problem contains many subproblem equivalences. A solver exploiting these equivalences achieves several orders of magnitude speedup [53]. The subproblem equivalences arise as follows: after labeling \( x[1], \ldots, x[m] \), if the subproblem has not yet failed, then the remaining subproblem does not depend on what order the first \( m \) customer’s stacks were closed, but only on the set of customers whose stacks were closed.

Let us consider a problem where \( n = 6 \), customers 1, 3 and 4 want product 1, customers 2 and 6 want product 2, and customers 3 and 5 want product 3. This is represented by the data file:

\[
\begin{array}{c}
n = 6;
W = [1, 0, 1, 1, 0, 0 \\
    | 0, 1, 0, 0, 0, 1 \\
    | 1, 0, 1, 1, 1, 0 \\
    | 1, 0, 1, 1, 0, 0
\end{array}
\]
| 0, 0, 1, 0, 1, 0 |
| 0, 1, 0, 0, 0, 1 |

where a 1 in position \((i,j)\) represents that customers \(i\) and \(j\) have a product in common.

Suppose we are currently looking for a solution with \(\leq 4\) stacks. Consider the following two subproblems \(P \equiv (V,D,C)\) where \(D_F \equiv \{x[1] = 4, x[2] = 3, x[3] = 5\}\), and \(P' \equiv (V,D',C)\) where \(D'_F \equiv \{x[1] = 5, x[2] = 4, x[3] = 3\}\). Neither \(D\) nor \(D'\) violates the constraints, and they both schedule the same three customers first. The remaining subproblem on \(x[4], x[5], x[6]\) is identical for the two subproblems, so \(P\) and \(P'\) are equivalent. If \(P\) has no solutions, then \(P'\) also has no solutions. In general, any subproblem which assigned \(\{x[1], x[2], x[3]\}\) to the set \(\{4, 3, 5\}\) will also have no solutions. This subproblem equivalence can be exploited with a very simple key. In this example, we could use the key \(\{x[1], x[2], x[3]\} = \{4, 3, 5\}\), or even more simply, \(\{4, 3, 5\}\). Any partial assignment on \(\{x[1], x[2], x[3]\}\) which produces the same key necessarily leads to an equivalent subproblem.

\[\square\]

### 3.4 Automatic Keys for Caching

Manually generating keys is reasonably straightforward for problems that are simple and highly structured. However, it can be very difficult for more complex problems. In this section, we examine how keys can be automatically constructed for arbitrary constraint problems.

By definition, if \(P\) dominates \(P'\) then the projection of \(P'\) onto \(U\) logically implies the projection of \(P\) onto \(U\). If we could automatically construct keys that characterize the projections of subproblems, we can use such keys for automatic dominance detection. We can do this by exploiting the fact that the piecewise dominance of the projections of all constraints in \(C\) is a sufficient condition for subproblem dominance.

This is formalized in the following theorem:

**Theorem 3.4.1.** Let \(P \equiv (V,D,C)\) and \(P' \equiv (V,D',C)\) be subproblems arising during search, where \(\text{fixed}(D) = \text{fixed}(D') = F\) and \(U = \text{vars}(C) \setminus F\). If \(\forall c \in \text{vars}(C) \setminus F\)
CHAPTER 3. AUTOMATIC CACHING VIA CONSTRAINT PROJECTION

\[ \pi_U(C \land D') \Rightarrow \pi_U(c \land D_F) \text{ and } D'_U \Rightarrow D_U, \text{ then } P \text{ dominates } P'. \]

**Proof.**

\[
\begin{align*}
\pi_U(C \land D') & \Leftrightarrow \pi_U(C \land D'_F \land D'_U) \\
& \Leftrightarrow \pi_U(\land_{c \in C}(c \land D'_F)) \land D'_U \\
\text{(⋆)} & \Leftrightarrow \land_{c \in C}(\pi_U(c \land D'_F)) \land D'_U \\
& \Rightarrow \land_{c \in C}(\pi_U(c \land D_F)) \land D_U \\
\text{(⋆)} & \Leftrightarrow \pi_U(\land_{c \in C}(c \land D_F)) \land D_U \\
& \Leftrightarrow \pi_U(C \land D_F \land D_U) \\
& \Leftrightarrow \pi_U(C \land D)
\end{align*}
\]

The third and fifth (marked) equivalences hold because all variables being projected out in every \(c \land D'_F\) and \(c \land D_F\) are already fixed. \(\square\)

The importance of Theorem 3.4.1 lies in the fact that it allows us to treat the projection of each constraint in \(C\) separately. Thus, if we can characterize the projection of each constraint in \(C\), we can concatenate these, together with \(D_U\) and \(F\), to form a projection key which completely characterizes the subproblem.

### 3.4.1 Projection Keys

Naively, we can define a function \(\text{key}(c, D) \equiv \pi_U(c \land D_F)\), where \(F = \text{fixed}(D)\) and \(U = \text{vars}(D) \setminus F\), to characterize the projection of the constraint \(c\) onto the unfixed variables \(U\). There are several problems with this. Firstly, it means that we need one key per constraint, which could make the key for the whole subproblem very large. Secondly, the projection of complex constraints is not always easy to describe or represent, thus \(\pi_U(c \land D_F)\) may have no reasonably sized representation. We therefore use a more complex definition of \(\text{key}(c, D_F)\) that allows for more flexibility and several important optimizations.

**Definition 3.4.2.** Let \(w(c, D_F)\) be the weakest domain on \(U\) that the propagator \(p_c\) for constraint \(c\) can return given \(D_F\), i.e., for any \(D'\) s.t. \(D' \Rightarrow D_F\) and \(D' \Rightarrow D_{\text{init}}\) we have \(p_c(D') \Rightarrow w(c, D_F)\).
3.4. AUTOMATIC KEYS FOR CACHING

Definition 3.4.3. The projection key \( \text{key}(c, D_F) \) of a constraint \( c \) for domain \( D \) is a set of constraints over \( \text{vars}(c) \), such that:

\[
\pi_U(c \land \text{key}(c, D_F)) \land w(c, D_F) \iff \pi_U(c \land D_F).
\]

In general, there are many ways to correctly define \( \text{key}(c, D_F) \). However, we are interested in defining \( \text{key}(c, D_F) \) such that we can detect as much dominance as possible. Note that whether a particular definition of \( \text{key}(c, D_F) \) is correct depends on the consistency level of the propagator used for constraint \( c \). There are two trivially correct definitions for \( \text{key}(c, D_F) \): \( \text{key}(c, D_F) \equiv D_{F \cap \text{vars}(c)} \) is correct and can always be represented, but it does not allow us to detect any dominances; \( \text{key}(c, D_F) \equiv \pi_U(c \land D_F) \) is correct and allows us to detect many dominances, but it cannot always be represented in a reasonable size.

Example 3.4.4. Let \( V \equiv \{x_1, \ldots, x_6\} \) and \( D_{\text{init}} \equiv \{x_i \in \{1, \ldots, 6\} \mid i = 1, \ldots, 6\} \). We show four definitions of \( \text{key}(c, D_F) \) for \( c \equiv \text{alldiff}([x_1, x_2, x_3, x_4, x_5, x_6]) \) and show that they do not all detect the same amount of dominance. Suppose we have some \( D \) where \( D_F \equiv \{x_1 = 3, x_2 = 4, x_3 = 5\} \). Define \( \text{key}_1(c, D_F) \equiv D_{F \cap \text{vars}(c)} \equiv x_1 = 3 \land x_2 = 4 \land x_3 = 5 \). Define \( \text{key}_2(c, D_F) \equiv \land_{x \in (F \cap \text{vars}(c))} x \in \{v \mid \exists x_i \in F, x_i = v\} \equiv x_1 \in \{3, 4, 5\} \land x_2 \in \{3, 4, 5\} \land x_3 \in \{3, 4, 5\} \). Define \( \text{key}_3(c, D_F) \equiv \pi_U(c \land D_F) \equiv \text{alldiff}([3, 4, 5, x_4, x_5, x_6]) \). Define \( \text{key}_4(c, D) \equiv \land_{x \in (\text{vars}(c) \setminus F)} x \notin \{v \mid \exists x_i \in F, x_i = v\} \equiv x_4 \notin \{3, 4, 5\} \land x_5 \notin \{3, 4, 5\} \land x_6 \notin \{3, 4, 5\} \). \( \text{key}_1 \) and \( \text{key}_3 \) are trivially correct. \( \text{key}_2 \) is correct because as long as the fixed vars take the same set of values, the constraints on the remaining unfixed vars will be the same. \( \text{key}_4 \) is correct since it captures the entire effect of \( D_F \) on the unfixed variables \( U \).

Now, consider a different subproblem \( D_F' \equiv \{x_1 = 4, x_2 = 5, x_3 = 3\} \). \( \text{key}_1(c, D_F') \equiv x_1 = 4 \land x_2 = 5 \land x_3 = 3 \) does not match the key of the first subproblem. \( \text{key}_2(c, D_F') \equiv x_1 \in \{3, 4, 5\} \land x_2 \in \{3, 4, 5\} \land x_3 \in \{3, 4, 5\} \) matches and shows that the \( \text{alldiff} \) projects to the same constraint in both subproblems. \( \text{key}_3(c, D_F') \equiv \text{alldiff}([4, 5, 3, x_4, x_5, x_6]) \) does not match syntactically, but is semantically equivalent to \( \text{key}_4(c, D_F) \). \( \text{key}_4(c, D_F') \equiv x_4 \notin \{3, 4, 5\} \land x_5 \notin \{3, 4, 5\} \land x_6 \notin \{3, 4, 5\} \) also matches. \( \square \)
We now illustrate how to use projection keys for checking subproblem dominance. Note that dominance checking, and thus projection key calculation, is only ever performed when the solver is at propagation fixpoint.

**Theorem 3.4.5.** Let \( P \equiv (V, D, C) \) and \( P' \equiv (V, D', C) \) be subproblems arising during search from problem \( P_{\text{init}} \equiv (V, D_{\text{init}}, C) \). If \( \text{fixed}(D') = \text{fixed}(D) \), and \( \forall c \in C, \text{key}(c, D'_F) \Rightarrow \text{key}(c, D_F) \), and \( D'_U \Rightarrow D_U \), then \( P \) dominates \( P' \).

**Proof.**

\[
\pi_U(C \land D') \\
\Leftrightarrow \pi_U(C \land D'_F \land D'_U) \\
(* \Leftrightarrow \land_{c \in C}(\pi_U(c \land D'_F)) \land D'_U \\
\Leftarrow \land_{c \in C}(\pi_U(c \land \text{key}(c, D'_F)) \land w(c, D'_F)) \land D'_U \\
\Rightarrow \land_{c \in C}(\pi_U(c \land \text{key}(c, D'_F))) \land D'_U \\
\Leftarrow \land_{c \in C}(\pi_U(c \land \text{key}(c, D_F))) \land D_U \\
\Rightarrow \land_{c \in C}(\pi_U(c \land \text{key}(c, D_F)) \land w(c, D_F)) \land D_U \\
\Rightarrow \land_{c \in C}(\pi_U(c \land D_F)) \land D_U \\
(* \Rightarrow \pi_U(C \land D_F \land D_U) \\
\Rightarrow \pi_U(C \land D)
\]

The second and eighth (marked) equivalences hold because, again, all variables being projected out in each \( c \land D'_F \) and \( c \land D_F \) are already fixed. The fourth and sixth equivalences hold because when we are at propagation fixpoint, \( D_U \Rightarrow w(c, D_F) \) for any \( c \in C \). \( \square \)

**Definition 3.4.6.** Define the subproblem projection key, \( \text{skey}(C, D) \) for subproblem \( (C, D) \) as follows: \( \text{skey}(C, D) \equiv (F, [\text{key}(c, D_F) \mid c \in C], D_U) \). \( \text{skey}(C, D) \) is a tuple of objects, the first of which is a set of variables, and the rest of which are constraints.

Extend the implication operator \( \Rightarrow \) w.r.t. subproblem projection keys as follows: \( \text{skey}(C, D) \Rightarrow \text{skey}(C, D') \) iff \( F' = F \) and \( \forall c \in C, \text{key}(c, D'_F) \Rightarrow \text{key}(c, D_F) \), and \( D'_U \Rightarrow D_U \).

Theorem 3.4.5 states that if \( \text{skey}(C, D') \Rightarrow \text{skey}(C, D) \), then \( P \) dominates \( P' \). Clearly, if \( \text{skey}(C, D') \Leftrightarrow \text{skey}(C, D) \), then \( P \) and \( P' \) are equivalent.
3.4. AUTOMATIC KEYS FOR CACHING

Example 3.4.7. Consider the subproblem $P \equiv (V,D,C)$ where $V \equiv \{x_1, \ldots, x_6\}$, $D \equiv \{x_1 = 3, x_2 = 4, x_3 = 5, x_4 \in \{0,1,2\}, x_5 \in \{0,1,2\}, x_6 \in \{1,2,6\}\}$ and $C \equiv \{\text{alldiff}([x_1, x_2, x_3, x_4, x_5, x_6]), x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20\}$. Then, $F = \{x_1, x_2, x_3\}$ and $U = \{x_4, x_5, x_6\}$. A projection key for the subproblem is given by: $\text{skey}(C,D) \equiv (\{x_1, x_2, x_3\}, [(x_1, x_2, x_3) = (3,4,5), x_4 + 2x_5 \leq 4], \{x_4 \in \{0,1,2\}, x_5 \in \{0,1,2\}, x_6 \in \{1,2,6\}\})$, if we choose $\pi_U(c \land D_F)$ to define the key for the linear constraint $c \equiv x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20$. □

Example 3.4.8. Consider $P \equiv (V,D,C)$ of Example 3.4.7 and another subproblem $P' \equiv (V,D',C)$ where $D' \equiv \{x_1 = 4, x_2 = 5, x_3 = 3, x_4 \in \{0,1,2\}, x_5 \in \{0,1\}, x_6 \in \{1,2,6\}\}$. We have that $\text{fixed}(D') = \text{fixed}(D) = \{x_1, x_2, x_3\}$ and the keys for the alldiff are identical. Also, the projection of the linear inequality is $x_4 + 2x_5 \leq 3$. This is stronger than its projection in $\text{key}(P)$ as: $x_4 + 2x_5 \leq 3 \Rightarrow x_4 + 2x_5 \leq 4$. Similarly, $D'_U \Rightarrow D_U$. Hence, $P$ dominates $P'$. □

3.4.2 Special Cases

The more complex definition of $\text{key}(c,D_F)$, introduced above, allows us to define $\text{key}(c,D_F) \equiv \text{true}$ in several very important cases. This is good because the weaker a set of constraints $\text{key}(c,D_F)$ is, and the more independent it is of $D_F$, the better a key it is at detecting dominances. Thus, when applicable, $\text{key}(c,D_F) \equiv \text{true}$ is essentially the best possible key for dominance detection. $\text{key}(c,D_F) \equiv \text{true}$ also has the additional benefit that it does not contribute to our key representation (described in the next section), and so we save space as well.

Suppose $F \cap \text{vars}(c) = \emptyset$. Then $\pi_U(c \land D_F) \equiv \pi_U c$ and therefore, $\text{key}(c,D_F) \equiv \text{true}$ is trivially correct. This means that for any $D_F$, if $c$ is inactive (none of its variables are fixed by $D_F$), we can define $\text{key}(c,D_F) \equiv \text{true}$.

Suppose $w(c,D_F) \Rightarrow \pi_U(c \land D_F)$. Then $\text{key}(c,D_F) \equiv \text{true}$ is trivially correct. This means that for any $D_F$, if $c$’s propagator is strong enough that the propagation from $D_F$ makes $c$ satisfied, we can define $\text{key}(c,D_F) \equiv \text{true}$. The vast majority of the constraints satisfied by $D$ fall under this category.
Example 3.4.9. Consider the subproblem $P \equiv (V, D, C)$ where $V \equiv \{x_1, \ldots, x_4\}$, $D \equiv \{x_1 = 3, x_2 = 4, x_3 \in \{0, 1, 2, 3, 5, 6\}, x_4 \in \{0, 1, 2, 3, 4, 5, 6\}\}$ and $C \equiv \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_4\}$. Then, $F = \{x_1, x_2\}$ and $U = \{x_3, x_4\}$. A correct projection key for this subproblem is given by: 

$$skey(C, D) \equiv \left(\{x_1, x_2\}, [true, true, true], \{x_3 \in \{0, 1, 2, 3, 5, 6\}, x_4 \in \{0, 1, 2, 3, 4, 5, 6\}\}\right)$$. The projection key for $x_1 \neq x_2$ is true since it is satisfied, similarly the projection key for $x_2 \neq x_3$ is true since it is satisfied (even though $x_3$ is not fixed), and the projection key for $x_3 \neq x_4$ is true since none of its variables are fixed. □

3.4.3 Using Projection Keys

In this section, we discuss how we can cache subproblems using projection keys, and how dominance checks can be carried out. Recall that each subproblem $P \equiv (V, D, C)$ is characterized by $skey(C, D) \equiv (F, [key(c, D_F) \mid c \in C], D_U)$. We are free to take any low level representation of $skey(C, D)$, as long as it is uniquely identified. In particular, we can take advantage of globally known information like the constraint type of each $c$, their arguments, any constants involved, $F$, and $D_{init}$, to dramatically reduce the size of the representation.

We define representations $rep(c, D_F)$ for each $key(c, D_F)$ on a per constraint basis. We require that there be a bijection between tuple $(F, c, rep(c, D_F))$ and the possible values of $key(c, D_F)$. Note that different values of $key(c, D_F)$ can map to the same value of $rep(c, D_F)$, as long as $F$ and $c$ are sufficient to distinguish them. See Section 3.5 for many examples of $key(c, D_F)$ and $rep(c, D_F)$.

We also need to define a representation $rep(D_v)$ for each $D_v$ where $v \in U$. In practice we simply use the existing solver representation of variable domains, e.g., a bit string, a set of ranges, etc. Finally, we need to define a representation $rep(F)$ to represent $F$, for which we use a bit string.

We can define the representation $rep(C, D)$ of a subproblem in one of two ways:

- The dense representation which always includes a representation for each constraint appearing in $C$, and every unfixed variable $U$ appearing in the problem:

$$rep_d(C, D) \equiv (rep(F), [rep(c, D_F) \mid c \in C], [rep(D_v) \mid v \in U])$$
• the sparse representation which omits constraints whose key is true and omits unfixed variables whose domain is the same as their initial domain:

\[ rep_s(C, D) \equiv (rep(F), [(id(c), rep(c, D_F)) \mid c \in C, \text{key}(c, D_F) \neq \text{true}],
\]

\[ [(id(v), rep(D(v)) \mid v \in U, D(v) \neq D_{\text{init}}(v))] \]

where \( id(v) \) and \( id(c) \) are unique id’s for variables and constraints respectively.

Clearly, both representations uniquely identify \( (F, [\text{key}(c, D_F) \mid c \in C], D_U) \).

For problems with lots of subproblem equivalences, it is generally the case that \( D(v) = D_{\text{init}}(v) \) and \( \text{key}(c, D_F) \equiv \text{true} \) for a very substantial portion of the variables and constraints. Thus, in the implementation we use \( rep_s(C, D) \) which can take advantage of the sparsity. Note that this means that we only need to define \( rep(c, D_F) \) for \( D_F \) where \( \text{key}(c, D_F) \neq \text{true} \).

Clearly, if \( rep(C, D) = rep(C, D') \), then \( skey(C, D) \leftrightarrow skey(C, D') \) and \( P \) and \( P' \) are equivalent. If all we want to do is to detect subproblem equivalence, we can accomplish this very efficiently by using a hash table to check for \( rep(C, D) = rep(C, D') \), where the projection key is used as both the hash table key and value.

We can thus alter the algorithm in Figure 3.1 by replacing line 5 by:

\[ \text{if } ((rep(C, D') \mapsto rep(C, D')) \in \text{Hashtable}) \text{ return false} \]

and replacing line 9 by:

\[ \text{Hashtable := Hashtable } \cup \{ \text{rep}(C, D') \mapsto \text{rep}(C, D') \} \]

Example 3.4.10. Consider the problem \( P \equiv (V, D, C) \) of Example 3.4.7. We can store its projection key \( \{x_1, x_2, x_3\} = \{3, 4, 5\} \land x_4 + 2x_5 \leq 4 \land D_U \) as follows: We store the fixed variables \( \{x_1, x_2, x_3\} \) for the subproblem since these must be identical for the equivalence check in any case. We store \( \{3, 4, 5\} \) for the alldiff constraint, and the fixed value 4 for the linear constraint, which give us enough information given the fixed variables to define the key. The remaining part of the key are domains. Thus, the projection key can be stored as \( (\{x_1, x_2, x_3\}, \{3, 4, 5\}, 4, \{0, 1, 2\}, \{0, 1, 2\}, \{1, 2, 6\}) \) using the dense representation, and \( (\{x_1, x_2, x_3\}, (c_1, \{3, 4, 5\}), (c_2, 4), (x_4, \{0, 1, 2\}), (x_5, \{0, 1, 2\}), (x_6, \{1, 2, 6\})) \) using the sparse representation. \( \square \)
To exploit subproblem dominances, and not simply equivalences, we need a method to detect when \( s\text{key}(C, D') \Rightarrow s\text{key}(C, D) \). Some constraints naturally produce projections which can dominate one another, while others do not. Consider a linear constraint \( x_1 + x_2 + x_3 \leq 5 \). If \( x_1 = 1 \), this projects to \( x_2 + x_3 \leq 4 \). If \( x_1 = 2 \), this projects to \( x_2 + x_3 \leq 3 \), which is clearly a strictly stronger constraint. Thus, we can see that linear constraints tend to produce dominating projections. On the other hand, this is not the case for other kinds of constraints such as \textit{alldiff}. No projection of the \textit{alldiff} constraint ever strictly dominates another. e.g., \textit{alldiff}(1, 2, x_3, x_4, x_5) does not dominate and is not dominated by \textit{alldiff}(1, 2, 3, x_4, x_5). Thus, we can divide up the constraints into two groups: ones which can generate (strictly) dominating projections, and ones which cannot. Note that domain constraints \( D_v \) fall in the first group as they can strictly dominate another \( D'_v \).

We can detect subproblem dominances as follows. We first divide the subproblem representation into two parts. The \textit{equivalence part} \( \text{rep}_e(C, D) \), which consists of \( \text{rep}(F) \) and \( \text{rep}(c, D_F) \) for each \( c \in C \) that cannot generate dominating projections, and the \textit{dominance part} \( \text{rep}_d(C, D) \), which consists of \( \text{rep}(c, D_F) \) for each \( c \in C \) that can generate dominating projections, together with \( \text{rep}(D_v) \).

For dominance detection, we use only the equivalence part \( \text{rep}_e(c, D_F) \) as the hash table key, and the whole representation \( \text{rep}(C, D) \) as the value. Now, looking up matches in the hash table returns a set of subproblems with the same \( \text{rep}_e(C, D) \), but possibly different \( \text{rep}_d(C, D) \). We then simply compare the dominance parts of each to see if \( P \) is dominated by one of these. We alter the algorithm in Figure 3.1 as follows. Line 5 is replaced by:

\[
\text{if} \ (\exists (\text{rep}_e(C, D) \mapsto \text{rep}(C, D)) \in \text{Hashtable} \text{ s.t. } \text{rep}_e(C, D) = \text{rep}_e(C, D') \\
\quad \text{and } \text{rep}_d(C, D) \text{ dominates } \text{rep}_d(C, D')) \text{ return false}
\]

and line 9 is replaced by:

\[
\text{Hashtable} := \text{Hashtable} \cup (\text{rep}_e(C, D') \mapsto \text{rep}(C, D'))
\]

In the above, we put all the \( \text{rep}(c, D_F) \) and \( \text{rep}(D_v) \) that can generate dominating projections into the dominance part. However, we can actually decide on an individual
basis whether to put them in the equivalence or dominance part, as long as we are consistent throughout the search. There is a trade off between speed and dominance detection strength. The more things we include in the equivalence part, the fewer matches we get and the quicker the dominance check, but the less dominance we can detect.

Example 3.4.11. Consider the subproblem $P \equiv (V, D, C)$ from Example 3.4.7. Its projection key is $(\{x_1, x_2, x_3\}, \{3, 4, 5\}, 4, \{0, 1, 2\}, \{0, 1\}, \{0, 1\})$. The key for \textit{alldiff} is in the equivalence part, and we need an exact match to determine dominance. The key for the linear constraint is in the dominance part, and we need to compare the value of the constant term to determine dominance. The keys for the domain constraints are in the dominance part, and we need to check for subset/superset property to determine dominance. For the subproblem $P'$ from Example 3.4.8 we have the projection key $(\{x_1, x_2, x_3\}, \{3, 4, 5\}, 3, \{0, 1, 2\}, \{0, 1\}, \{1, 2\})$. The first two arguments of the tuple are hashed to form the hash table key for $P'$. It will match the hash table key for $P$, so we will retrieve the projection key for $P$. The first two arguments of $P$ and $P'$'s projection keys match so we are OK. We then compare the third arguments to see that $3 < 4$, so that is also OK. Finally, we check that each of the last three arguments of $P'$'s key is a subset of the corresponding arguments in $P$'s key, which is true. Hence, we determine that $P$ dominates $P'$. □

Note that, for efficiency, our implementation checks $D'_U \Rightarrow D_U$ by using identity $(D'_U \equiv D_U)$ so the domains can be part of the hash value. This means that the problem $P'$ of Example 3.4.8 will not be detected as dominated in our implementation, since the domain of $x_5$ is different.

3.4.4 Branch and Bound Optimization

The presentation so far has concentrated on satisfaction problems. We show that the theory can be extended to branch and bound optimization search. Without loss of generality, let us consider a minimization problem. Branch and bound optimization typically proceeds as follows. We have some objective function $f$, e.g., $f \equiv \sum_{i=1}^{n} a_i x_i$ or $f \equiv \max_{i=1}^{n} x_i$, and one of the constraints in the problem is of the form $c_{obj} \equiv f < b$,
where \( b \) is the objective value of the best solution found so far. Suppose that during search, a better solution with \( f = b' \) is found, then the search is constrained so that only solutions better than \( b' \) can be found, i.e., \( c_{\text{obj}} \) is strengthened to \( c'_{\text{obj}} \equiv f < b' \). Search continues until the search space is exhausted, and no better solution exists.

Our previous theorems were proved based on the assumption that \( C \) remains the same in \( P \) and \( P' \). However, this is not true in branch and bound optimization, as \( c_{\text{obj}} \) is strengthened to \( c'_{\text{obj}} \). We can easily extend Theorem 3.4.5 to the case where all constraints in \( C \) monotonically increase in strength as search progresses.

**Lemma 3.4.12.** Suppose \( c' \Rightarrow c \), \( \text{key}(c', D'_F) \Rightarrow \text{key}(c, D_F) \), and \( D'_U \Rightarrow D_U \). Then \( \pi_U(c' \land D'_F) \land D'_U \Rightarrow \pi_U(c \land D_F) \land D_U \).

**Proof.**

\[
\begin{align*}
\pi_U(c' \land D'_F) \land D'_U & \iff \pi_U(c' \land \text{key}(c', D'_F)) \land D'_U \\
& \Rightarrow \pi_U(c \land \text{key}(c, D_F)) \land D_U \\
& \iff \pi_U(c \land D_F) \land D_U
\end{align*}
\]

\( \square \)

**Theorem 3.4.13.** Let \( P \equiv (V, D, C) \) and \( P' \equiv (V, D', C') \) be subproblems arising during search. Suppose there is a bijective mapping \( m : C \rightarrow C' \) s.t. \( m(c) \Rightarrow c \), i.e., every constraint \( c \in C \) has a corresponding constraint \( m(c) \in C' \) which is at least as strong. If \( F' = F \), and \( \forall c \in C, \text{key}(m(c), D'_F) \Rightarrow \text{key}(c, D_F) \) and \( D'_U \Rightarrow D_U \), then \( P \) dominates \( P' \).

\( \square \)

The proof is analogous to that of Theorem 3.4.5, except that we apply Lemma 3.4.12 at the appropriate place. Clearly, branch and bound optimization search is covered by Theorem 3.4.13 and we can use projection keys to detect dominance as we did for satisfaction problems.

### 3.4.5 Redundant constraints

Problems are often modeled with redundant constraints which enhance propagation and reduce the search space. Such redundant constraints can actually be ignored in
3.5. CONSTRUCTING PROJECTION KEYS

the subproblem projection. This is formalized by the following.

**Lemma 3.4.14.** Suppose we have $P \equiv (V, D, C)$ and $P' \equiv (V, D, C')$ where $C' \subseteq C$ s.t. $C \Leftrightarrow C'$. Then the projection of $P$ and $P'$ onto $U$ w.r.t. any $D$ is equivalent, i.e., $\pi_U(C \land D) \Leftrightarrow \pi_U(C' \land D)$.

*Proof.* Since $C \Leftrightarrow C'$, $C \land D \Leftrightarrow C' \land D$, thus $\pi_U(C \land D) \Leftrightarrow \pi_U(C' \land D)$ for any $D$. □

**Theorem 3.4.15.** Let $P \equiv (V, D, C)$ and $P' \equiv (V, D', C)$ be subproblems arising during search. Suppose $C' \subseteq C$ s.t. $C \Leftrightarrow C'$. If $\text{fixed}(D') = \text{fixed}(D)$, and $\forall c \in C', \text{key}(c, D') \Rightarrow \text{key}(c, D)$, and $D_F \Rightarrow D_U$, then $P$ dominates $P'$.

The proof is analogous to that of Theorem 3.4.5, except that we apply Lemma 3.4.14 at the beginning and end. This means that instead of creating projecting keys from the full set of constraints $C$, it is sufficient to choose a subset $C' \subseteq C$ which fully define the problem.

### 3.5 Constructing Projection Keys

In this section we discuss how to create projection keys for many common constraints. Constructing the function $\text{key}(c, D_F)$ is a form of constraint abduction, which can be a very complex task. However, we only need to perform constraint abduction on a per constraint basis. Recall that we can always define $\text{key}(c, D_F)$ as $D_{F \cap \text{vars}(c)}$ or $\pi_U(c \land D_F)$. However, we want a definition for $\text{key}(c, D_F)$ that detects as much dominance as possible and is representable in a reasonable size. First we try to see if $\text{key}(c, D_F) \equiv \text{true}$ is a valid definition, as this detects the most dominance. Recall that if $F \cap \text{vars}(c) = \emptyset$ or $w(c, D_F) \Rightarrow \pi_U(c \land D_F)$, then we can always define $\text{key}(c, D_F) \equiv \text{true}$. We only need to consider more complex definitions of $\text{key}(c, D_F)$ for $D_F$ for which neither of these conditions obviously hold, i.e., we only consider $D_F$ where at least some variables in $\text{vars}(c)$ are fixed, and where $D_F$ is not enough to satisfy $c$. Next we try $\text{key}(c, D_F) \equiv \pi_U(c \land D_F)$, which often detects a fair amount of dominance, but is not always representable. Thirdly, we try to find something weaker but representable. If all else fails, we use $\text{key}(c, D_F) \equiv D_{F \cap \text{vars}(c)}$, which detects no dominance, but is always representable.
Constraint $c$ | Sufficient set of propagation rules
---|---
$x = y$ | $x = d \iff y = d$
$x \neq y$ | $x = d \implies y \neq d, y = d \implies x \neq d$
$x \geq y$ | $x = d \implies y \leq d, y = d \implies x \geq d$
$x \vee y$ | $\neg x \implies y, \neg y \implies x$
$(x = d) \implies b$ | $x = d \implies b, \neg b \implies x \neq d$
$(x = d) \impliedby b$ | $x \neq d \implies \neg b, b \implies x = d$
$(x \geq d) \implies b$ | $x \geq d \implies b, \neg b \implies x < d$
$(x \geq d) \impliedby b$ | $x < d \implies \neg b, b \implies x \geq d$
$(x = d') \implies (y = d)$ | $x = d' \implies y = d, y \neq d \implies x \neq d'$
y = bool2int($b$) | $b \iff y = 1, \neg b \iff y = 0$
y = abs($x$) | $x = d \implies y = \text{abs}(x), y = d \implies x \in \{d, -d\}$

Figure 3.4: Sufficient conditions on propagation to ensure no key is required for the binary constraints above. Note the bool2int() function converts a Boolean $b$ to a 0-1 integer $y$.

3.5.1 Simple Constraints

Here we give the function $key(c, D_F)$ and the representation $rep(c, D_F)$ for some simple constraints. Any assumptions about the minimum consistency level of the propagator are stated as necessary.

**Binary Constraints** Almost all commonly used propagators for binary constraints have the property that once one variable is fixed, propagation on the other variable causes the constraint to become satisfied. For such constraints, either $F \cap \text{vars}(c) = \emptyset$ or $w(c, D_F) \implies \pi_U(c \land D_F)$. Thus, we can define $key(c, D_F) \equiv \text{true}$ for any $D_F$ which means that such constraints never need to contribute to the projection key. We give a non-exhaustive list of some of these constraints, along with a sufficient set of propagation rules for the above condition to hold in Figure 3.4.

**Linear** Let $c$ be a linear constraint of the form $c \equiv \sum_{i=1}^{n} a_i x_i \leq a_0$. Let $S = \{i \mid x_i \in F\}$ and $d_i$ be the value that $x_i$ is fixed to in $D_F$, for $i \in S$. Let $a'_0 = a_0 - \sum_{i \in S} a_i d_i$. We can define $key(c, D_F) \equiv \sum_{i \not\in S} a_i x_i \leq a'_0$. The variables $x_i$ and the constants $a_i$ are implicitly known from $c$, and $S$ is implicitly known from $F$ and $\text{vars}(c)$. Thus, the representation only needs to store the single number $a'_0$. We can
3.5. CONSTRUCTING PROJECTION KEYS

define \( \text{rep}(c, D_F) = \{a'_0\} \). The \( \geq \) and \( = \) case of the linear constraint are analogous.

**Reified Linear** Let \( c \) be a reified linear constraint of the form \( c \equiv b \Rightarrow \sum_{i=1}^{n} a_ix_i \leq a_0 \), that is if \( b \) is true the linear constraint holds and if the linear constraint does not hold then \( b \) is false. Let \( S = \{i \mid x_i \in F\} \), and \( d_i \) be the value that \( x_i \) is fixed to in \( D_F \), for \( i \in S \). Let \( a'_0 = a_0 - \sum_{i \in S} a_id_i \). There are two cases. If \( b = true \), then we can define \( \text{key}(c, D_F) \equiv \sum_{i \notin S} a_ix_i \leq a'_0 \) and \( \text{rep}(c, D_F) = \{a'_0, true\} \). If \( b = false \) the constraint is satisfied and \( \text{key}(c, D_F) = true \). Otherwise if \( b \notin F \), then we can define \( \text{key}(c, D_F) \equiv b \rightarrow \sum_{i \notin S} a_ix_i \leq a'_0 \) and \( \text{rep}(c, D_F) = \{a'_0, false\} \). Note that the representation is thus simply the representation of the non-reified linear, plus an additional Boolean value. Other kinds of reified constraints can be handled in a similar manner.

**Clause** Let \( c \) be a clausal constraint of the form \( c \equiv \bigvee_{i=1}^{n} l_i \) where \( l_i \) are Boolean literals (either \( b_i \) or \( \neg b_i \)). Let \( S = \{i \mid b_i \in F\} \). Suppose \( \exists i \in S \) s.t. \( l_i = true \) (that is \( l_i \equiv b_i \) and \( D(b_i) = true \) or \( l_i \equiv \neg b_i \) and \( D(b_i) = false \)). Then \( w(c, D_F) \Rightarrow \pi_U(c \land D_F) \) and we can define \( \text{key}(c, D_F) \equiv true \) and \( \text{rep}(c, D_F) = \{true\} \). Suppose \( \nexists i \in S \) s.t. \( l_i = true \). Then we can define \( \text{key}(c, D_F) \equiv \bigvee_{i=1,i \not\in S} l_i \). Once again, \( S \) is implicitly known from \( F \) and \( vars(c) \), so in fact, we don’t need to store any additional information. We can define \( \text{rep}(c, D_F) = \{true\} \) in the first case and \( \text{rep}(c, D_F) = \{false\} \) in the second.

**Arithmetic constraints** Most arithmetic constraints like \( c \equiv z = x \times y, c \equiv z = x \mod y \), do not project well. Thus, we can only use as the last resort \( \text{key}(c, D_F) \equiv D_{F \cap \text{var}(c)} \) and \( \text{rep}(c, D_F) = [D_v \mid v \in (F \cap \text{var}(c))] \).

3.5.2 Global Constraints

Constraint programming models often make use of global constraints to define important subproblems of the model. On the face of it defining a projection key for a complex global constraint may seem a daunting task. But in many cases it is quite straightforward.
For many global constraints, we can use the following theorem to work out a correct definition of \( \text{key}(c, D_F) \):

**Theorem 3.5.1.** Let \( c \iff \bigwedge_{i=1}^n c_i \) be a decomposition of \( c \) that introduces no new variables. If \( w(c, D_F) \Rightarrow w(c_i, D_F) \) for all \( i \), then we can define \( \text{key}(c, D_F) \equiv \bigwedge_{i=1}^n \text{key}(c_i, D_F) \).

**Proof.**

\[
\begin{align*}
\pi_U(c \land \text{key}(c, D_F)) \land w(c, D_F) \\
\iff \pi_U(\bigwedge_{i=1}^n (c_i \land \text{key}(c_i, D_F))) \land \bigwedge_{i=1}^n w(c_i, D_F) \land w(c, D_F) \\
\iff \bigwedge_{i=1}^n (\pi_U(c_i \land \text{key}(c_i, D_F)) \land w(c_i, D_F)) \land w(c, D_F) \\
\iff \bigwedge_{i=1}^n \pi_U(c_i \land D_F) \land w(c, D_F) \\
\iff \pi_U(\bigwedge_{i=1}^n (c_i \land D_F)) \land w(c, D_F) \\
\iff \pi_U(c \land D_F)
\end{align*}
\]

The reverse direction of the last equivalence holds because for any correct propagator, we must have \( \pi_U(c \land D_F) \Rightarrow w(c, D_F) \) if \( D \) is a fixpoint of \( p_c \).

Thus, if our global constraint \( c \) can be decomposed into simpler constraints for which we already have correctly defined \( \text{key}(c_i, D_F) \), we can simply take their conjunction as the definition for \( \text{key}(c, D_F) \). Note that the solver does not need to use the constraint decomposition for propagation. The decomposition is introduced purely to define \( \text{key}(c, D_F) \). For this reason it is more effective to take a simple, weak decomposition than to take a complicated and strong decomposition with many redundant constraints. Simpler decompositions with fewer constraints produce fewer keys to conjunct, and there is a higher chance that our global propagator has stronger propagation than the decomposition.

Here we give the function \( \text{key}(c, D_F) \) and the representation \( \text{rep}(c, D_F) \) for some global constraints. Any assumptions about the minimum consistency level of the propagator are stated as necessary.
3.5. CONSTRUCTING PROJECTION KEYS

**Alldiff**  The *alldiff* constraint is defined by (and decomposes to):

\[ c \equiv \text{alldiff}(x_1, \ldots, x_n) \iff \bigwedge_{i=1}^{n} \bigwedge_{j=1, j \neq i}^{n} x_i \neq x_j. \]

If our propagator enforces at least \( x_i = v \rightarrow x_j \neq v \) for all \( i \neq j \), then we can apply Theorem 3.5.1 with this decomposition. Since all of the constraints in the decomposition are binary, \( \text{key}(c, D_F) \equiv true \) for all of them. So we can define \( \text{key}(c, D_F) \equiv true \) for all \( D_F \) for the *alldiff* constraint, which means we don’t need to define \( \text{rep}(c, D_F) \).

**And**  Let \( c \) be a global conjunction constraint of the form \( c \equiv y = \bigwedge_{i=1}^{n} x_i \). We can decompose \( c \) into clausal constraints: \( c \iff \bigwedge_{i=1}^{n} (\neg y \lor x_i) \land (y \lor \bigvee_{i=1}^{n} \neg x_i) \). Suppose our *and* propagator is domain consistent. Then we can apply Theorem 3.5.1. So we can define \( \text{key}(c, D_F) \equiv \bigwedge_{i=1}^{n} \text{key}(\neg y \lor x_i, D_F) \land \text{key}(y \lor \bigvee_{i=1}^{n} \neg x_i, D_F) \). Now \( \text{key}(c, D_F) \equiv true \) for all the binary clauses in the decomposition, and so this simplifies to \( \text{key}(c, D_F) \equiv \text{key}(y \lor \bigvee_{i=1}^{n} \neg x_i, D_F) \). Thus, the key and representation is exactly the same as that for a clause, which we have already covered in the previous subsection.

**Element**  Let \( c \) be an element constraint of the form \( c \equiv \text{element}(x, [a_1, \ldots, a_n], y) \) where \( a_i \) are constants. \( \text{element} \) is defined by (and decomposes to):

\[ \text{element}(x, [a_1, \ldots, a_n], y) \iff \bigwedge_{i=1}^{n} (x = i \rightarrow y = a_i). \]

Suppose our propagator enforces at least \( x = i \rightarrow y = a_i \) and \( y \neq a_i \rightarrow x \neq i \). Then, we can apply Theorem 3.5.1 with this decomposition. Since all of the constraints in the decomposition are binary, \( \text{key}(c, D_F) \equiv true \) for all of them. Hence just as for *alldiff*, we don’t need to define \( \text{rep}(c, D_F) \).

**Inverse**  The *inverse* constraint is defined by (and decomposes to):

\[ c \equiv \text{inverse}(x_1, \ldots, x_n, y_1, \ldots, y_n) \iff \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} x_i = j \leftrightarrow y_j = i. \]
If our propagator enforces at least \( x_i = j \rightarrow x_j = i \) and \( x_i \neq j \rightarrow x_j \neq i \), then we can apply Theorem 3.5.1 with this decomposition. Since all of the constraints in the decomposition are binary, again we don’t need to define \( \text{rep}(c, D_F) \).

**Minimum**  
Let \( c \) be a minimum constraint of the form \( c \equiv y = \min_{i=1}^n x_i \). Let \( S = \{i \mid x_i \in F\} \) and let \( d_i \) be the value that \( x_i \) is fixed to in \( D_F \) for \( i \in S \). Let \( d = \min_{i \in S} d_i \). Assume we have a special constant \( \infty \), whose value cannot be taken by any integer variable in the system. There are three cases. If \( y \notin F \) and \( S \neq \emptyset \), then we can define \( \text{key}(c, D_F) \equiv y = \min(d, \min_{i \in S} x_i) \) and \( \text{rep}(c, D_F) = (\infty, d) \). If \( y \in F \) and \( S \neq \emptyset \), then we can define \( \text{key}(c, D_F) \equiv e = \min(d, \min_{i \in S} x_i) \) and \( \text{rep}(c, D_F) = (e, d) \) where \( e \) is the fixed value of \( y \) (\( y = e \in D_F \)). Finally, if \( y \in F \) and \( S = \emptyset \), then we can define \( \text{key}(c, D_F) \equiv e = \min_{i=1}^n x_i \) and \( \text{rep}(c, D_F) = (e, \infty) \) where \( e \) is defined as before. The **maximum** constraint is analogous to **minimum**.

**Global Cardinality Constraint**  
The global cardinality constraint \( \text{gcc} \) \([128]\) requires that the number of variables in \( \{x_1, \ldots, x_n\} \) which take the value \( j \) be \( n_j \). We consider the case where \( n_j \) are constants. It is defined by:

\[
c \equiv \text{gcc}([x_1, \ldots, x_n], [n_1, \ldots, n_k]) \iff \land_{i=1}^k (\sum_{j=1}^n \text{bool2int}(x_j = i) = n_i).
\]

Let \( S = \{i \mid x_i \in F\} \). Let \( n_j' = n_j - \sum_{i \in S} \text{bool2int}(x_i = j) \) for \( j = 1, \ldots, k \). Then we can define \( \text{key}(c, D_F) \equiv \land_{j=1}^k (\sum_{i \in S} \text{bool2int}(x_i = j) = n_j') \). \( S \) is known implicitly from \( \text{vars}(c) \) and \( F \), so we can simply define \( \text{rep}(c, D_F) = \{n_1', \ldots, n_k'\} \). The **global cardinality low up** constraint which requires the number of values taking value \( j \) to be between lower bound \( l_j \) and upper bound \( u_j \), defined by

\[
\text{gcclu}([x_1, \ldots, x_n], [l_1, \ldots, l_k], [u_1, \ldots, u_k]) \iff \land_{i=1}^k (l_i \leq \sum_{j=1}^n \text{bool2int}(x_j = i) \leq u_i),
\]

is analogous.
3.5. CONSTRUCTING PROJECTION KEYS

Cumulative Constraint The cumulative constraint [152] requires that \( n \) tasks with start time \( s_i \), duration \( d_i \) and resource utilization \( r_i \) are scheduled such that at no time more than \( b \) resources are used. It can be defined as:

\[
c ≡ cumulative([s_1, \ldots, s_n], [d_1, \ldots, d_n], [r_1, \ldots, r_n], b)
≡ \bigwedge_t \sum_{i=1}^n bool2int((s_i \leq t) \land (s_i > t - d_i)) \times r_i \leq b,
\]

where \( t \) ranges from \( \min_{i=1}^n \min(D_{\text{init}}(s_i)) \) to \( \max_{i=1}^n \max(D_{\text{init}}(s_i) + d_i) \) which are all the time periods when tasks can be scheduled. We consider only the version where \( s_i \) are variables and \( r_i, d_i \) and \( b \) are constants. We present a handcrafted key \( key(c, D_F) \) which is intuitively obvious, but which is not the most powerful possible. Ideally, we would like to define \( key(c, D_F) ⟷ π_U(c \land D_F) \). However, \( π_U(c \land D_F) \) cannot really be represented in any simple way.

Intuitively, if we fix the start times of the exact same set of tasks, in such a way that the time profile of the resources they consumed is exactly identical, then, the constraint on the remaining tasks must be the same. We define the time profile of consumed resources as follows: Let \( S = \{i \mid s_i \in F\} \). Let \( p(t) = \sum_{i \in S} bool2int(D_F(s_i) \leq t \land D_F(s_i) > t - d_i) \times r_i \) be the amount of resources consumed by the fixed tasks at time \( t \). \( p(t) \) is a set of constants determined by \( D_F \). Now we only have to examine the profile for times that unfixed tasks can still actually make use of. Let \( t_{\text{min}} \) be the minimum start time of an unfixed task in \( w(c, D_F) \), and \( t_{\text{max}} \) be the maximum end time of an unfixed task in \( w(c, D_F) \). Now, we define

\[ key(c, D_F) \equiv \land_{t \in t_{\text{min}}..t_{\text{max}}} p(t) = \sum_{i \in S} bool2int(s_i \leq t \land s_i > t - d_i) \times r_i, \]

which is a constraint over \( F \cap vars(c) \). Clearly, \( π_U(c \land key(c, D_F)) \land w(c, D_F) \Leftrightarrow π_U(c \land D_F) \). Now, any two \( D_F \) and \( D_F' \) which has the same set of fixed tasks and produce the same \( t_{\text{min}} \) and \( t_{\text{max}} \) and \( p(t) \) over this range of times will have matching keys.

Example 3.5.2. Consider a cumulative constraint with fixed tasks \( A, B, C, D, E, F, G, H \) illustrated in Figure 3.5(a) with task durations \( d = [4, 5, 2, 1, 3, 2, 2, 1] \) and resource requirements \( r = [1, 3, 2, 2, 1, 1, 1] \) and resource bound \( b = 4 \). The start times illustrated in Figure 3.5(a) are \( s = [0, 0, 5, 5, 6, 6, 8, 7] \). Assume that no other unfixed
Figure 3.5: Different fixed sub-schedules for cumulative leading to the same key.

task (not shown) can start before time 7. Then the key is represented \( p(t) \) for \( t = 7 \) onwards, that is \( p(7) = 3, p(8) = 2, p(9) = 1, p(t) = 0, t > 9 \). Now consider the cumulative constraint with the same fixed tasks as illustrated in Figure 3.5(b), where again no other task can start before time 7. The key is identical even though the fixed schedules and even profiles of the two subproblems are different. □

### 3.5.3 More Global Constraints

For some global constraints, the best projection keys can only be expressed as constraints over the internal variables, \( I \), of the global propagator. We cannot rigorously define \( key(c, D_F) \) as constraints over such internal variables \( I \) unless they are explicitly defined. Such internal variables are often defined by the decompositions of the global constraint.

Suppose \( c \leftrightarrow \pi_{vars(c)}(\wedge_{i=1}^n c_i) \) is a decomposition that introduces new variables \( I \). We extend \( P \equiv (V, D, C) \) to \( P' \equiv (V, D', C') \) as follows: Define \( D' = D \cup \{ v \in D_{init(v)} \mid v \in I \} \). Define \( c' \) to be a constraint over \( vars(c) \cup I \) s.t. \( c' \leftrightarrow \wedge_{i=1}^n c_i \). Define \( C' = C \setminus c \cup c' \). Then, \( P \) and \( P' \) are related as follows: \( C \land D \leftrightarrow \pi_{vars(C)}(C' \land D') \).

Clearly, \( P \) is satisfiable iff \( P' \) is, and there is a surjective mapping from the solutions of \( P' \) to \( P \). Thus we can work with the problem \( P' \) instead.

Let \( F' = F \cup fixed(I) \) and \( U' = vars(C') \setminus F' \). Then \( D'_{U'} = D_u \cup D'_{U \cap I} \). Now we can apply Theorem 3.5.1 to \( P' \). This tells us that if \( w(c', D'_{F'}) \Rightarrow w(c_i, D'_{F'}) \) for all \( i \), then we can define \( key(c', D'_{F'}) \equiv \wedge_{i=1}^n key(c_i, D'_{F'}) \). Thus the projection key for \( P' \), as compared to \( P \), changes in the following ways: \( F \) becomes \( F \cup fixed(I), D_u \)
becomes \( D_U \cup D'_U \cap I \), and key\((c, D_F)\) which we did not know how to define before, becomes key\((c', D'_F)\) \(\equiv \land_{i=1}^{n} key(c_i, D'_F)\).

Although we had to explicitly define \(c'\) and \(I\) in order to apply Theorem 3.5.1, \(c'\) and \(I\) only have to be logically defined. They do not have to be implemented as real constraints or real variables in the system. For example, suppose the global propagator for \(c\) has some propagator-specific internal data structures, such that the domains for the variables in \(I\) are uniquely determined by them. Then \(I\) exist logically in the propagator and the propagator is already implementing a constraint over \(\text{vars}(c) \cup I\). Also, all extra parts of the key required by the decomposition, i.e., fixed\((I)\), \(D'_U \cap I\) and key\((c', D'_F)\) \(\equiv \land_{i=1}^{n} key(c_i, D'_F)\), depend only on \(c_i\) and \(I\). So if \(I\) is kept internally in the propagator, the propagator can produce all of the extra key parts we need. Thus, we can completely encapsulate the decomposition in the propagator without the rest of the system ever needing to know anything about \(I\), or that we extended from \(P\) to \(P'\). To express this, we can simply define

\[
\text{key}(c, D_F) \equiv \land_{i=1}^{n} \text{key}(c_i, D'_F) \land D'_U \cap I \quad \text{and} \quad \text{rep}(c, D_F) = \text{fixed}(I) \oplus \text{rep}(D'_U \cap I) + [\text{id}(c_i), \text{rep}(c_i, D'_F)] | \text{key}(c_i, D'_F) \neq \text{true},
\]

and use the normal key construction algorithm as applied to \(P\).

**Table** The table constraint is defined by (and decomposes to):

\[
\text{table}([x_1, \ldots, x_n], [a_{i,j} | i = 1..k, j = 1..n]) \Leftrightarrow \pi_{\text{vars}(c)}(\land_{i=1}^{k} \land_{j=1}^{n} r_i \rightarrow x_j = a_{i,j} \land \lor_{i=1}^{k} r_i),
\]

where \(r_i\) are introduced Boolean variables. Suppose our propagator internally keeps track of the domains of the \(r_i\) (which tuples are still possible) and enforces at least \(r_i \rightarrow x_j = a_{i,j}, x_j \neq a_{i,j} \rightarrow \neg r_i\). Then we can extend the problem to include the \(r_i\) and apply Theorem 3.5.1. All of the constraints in the decomposition are binary and has key\((c_i, D_F)\) \(\equiv \text{true}\) except for the clause \(\lor_{i=1}^{k} r_i\). Thus we can define key\((c, D_F)\) \(\equiv \text{key}(\lor_{i=1}^{k} r_i, D_F) \land D'_U \cap I\) and rep\((c, D_F) = \text{fixed}(I) \oplus \text{rep}(D'_U \cap I) \oplus \{\text{rep}(\lor_{i=1}^{k} r_i, D_F)\}\). Now, since the variables in \(I\) are all Boolean, \(\text{rep}(D'_U \cap I)\) is empty. We represent the projection for the clause with just a single Boolean value indicating whether it is satisfied by \(D_F\) or not. So \(\text{rep}(c, D_F) = \text{fixed}(I) \oplus \{\text{true}/\text{false}\}\).
Cumulative  Now we define a projection key for the cumulative constraint via decomposition. The cumulative constraint can be decomposed in the following way:

\[
\text{cumulative}([s_1, \ldots, s_n], [d_1, \ldots, d_n], [r_1, \ldots, r_n], b) \equiv \bigwedge_t\big((s_{o,i,t} \leftrightarrow s_i \leq t) \land (e_{o,i,t} \leftrightarrow s_i > t - d_i) \land (o_{i,t} \leftrightarrow s_{o,i,t} \land e_{o,i,t}) \land \sum_{i=1}^{n} \text{bool2int}(o_{i,t}) \times r_i \leq b,\]

where \( t \) is appropriately limited by the ranges of \( s_i \) and \( d_i \).

Suppose our propagator internally keeps track of the domains of the introduced variables \( I = \{s_{o,i,t}\} \cup \{e_{o,i,t}\} \cup \{o_{i,t}\} \) and enforces the constraints in the decomposition on them. Then we can extend the problem to include \( I \) and apply Theorem 3.5.1. All the reified constraints are binary and have \( key(c, D_F) \equiv true \). However, the Boolean constraints and the linear constraints may be active. We know how to define keys and representations for those from Section 3.5.1. Thus we simply conjunct them and \( D \cup I \) together to define \( key(c, D_F) \) for the cumulative constraint. \( \text{rep}(c, D_F) \) consists of the concatenation of \( F \cap I \) along with the representations of the active constraints in the decomposition, and the representation of \( D \cup I \).

Now, let us consider what this projection key means semantically. The projection key of linear constraints is \( key(c, D_F) \equiv \sum_{i \in S} a_i x_i \leq a'_0 \). Since we have one linear constraint for each time \( t \), these keys tell us exactly how much resource is available at each time \( t \) for the set of tasks which can still fit there. This is very similar to the resource profile used in the handcrafted key from Section 3.5.2. However, there is one major difference. It is possible for the linear constraint for some time \( t \) to become satisfied through propagation, in which case \( key(c, D_F) \equiv true \) and that part of the profile no longer matters. In the handcrafted key, we manually took into account such times if they were at the beginning or at the end of the schedule. Here, the projection key does it for all such time \( t \).

Example 3.5.3. Consider the cumulative example from Example 3.5.2 once more but this time where \( B, E, F, G, H \) are fixed as shown in Figure 3.6(a). The earliest start time for remaining tasks \( A, C, D \) is 4 (for task \( A \)), so clearly the previously defined key
for this partial schedule is different from that of the state shown in Figure 3.6(b), since the profiles at time 5 are different. In the new key however, in both cases the start times for A, C and D are exactly 4., 7., and \{5\} \cup 7.. The linear constraint at \(t = 4\) is satisfied since only \(o_{A,4}\) is unfixed, and task A fits in the remaining 1 unit. The linear constraint at \(t = 5\) is satisfied since only \(o_{A,5}\) and \(o_{D,5}\) are unfixed, and both A and D can simultaneously fit in the remaining 3 or 4 units in Figure 3.6(a) and Figure 3.6(b) respectively. The keys for the linear constraint for \(t \geq 7\) are clearly equivalent. Hence the decomposition based key will detect these situations as equivalent with respect to \(\text{cumulative}\).

\[\square\]

As can be seen, using decomposition allows us to define projection keys for even complicated constraints in a straightforward manner. Further, such keys are often fairly simple and powerful. In fact, they are often better at detecting dominance than the intuitively obvious keys that a human being may design.

### 3.6 Examples of Automatic Caching

In this section, we apply our algorithm to several problems to illustrate what the projection keys look like and what kinds of subproblem dominances they can detect. We also compare our automatically generated keys with handcrafted keys that a human may design. In most cases, our automatically generated projection keys contain significant amounts of redundant information. Thus, a handcrafted key will almost always be smaller in size. However, there are several advantages to automatically generating projection keys. Firstly, subproblem dominances are often very hard to
identify and describe. Secondly, altering a solver to exploit subproblems dominances for a particular problem via handcrafted keys can require a tremendous amount of work. Thirdly, if the problem is ever altered, or side constraints added, etc, the handcrafted key will have to be changed, its correctness will have to be reproven, and the system will have to be altered. All of this is very tedious and error prone. On the other hand, a system which implements caching via projection keys can handle all this automatically. We now go through the three example problems presented previously.

### 3.6.1 Black Hole

The Black Hole Problem can be modeled as follows:

```plaintext
array[1..17, 1..3] of int: layout; % layout: pile and layer
array[1..416, 1..2] of int: neighbours; % next card is +/- 1

array[1..52] of var 1..52: x; % Position of card
array[1..52] of var 1..52: y :: is_output; % Card at position

constraint y[1] == 1; % Ace of spades starts in blackhole
constraint inverse(x,y);
constraint forall (i in 1..17, j in 1..2) (x[layout[i,j]] < x[layout[i,j+1]]);
constraint forall (i in 1..51) (table([y[i], y[i+1]], neighbours));

solve satisfy;
```

We take card value \( v \) to mean the \((v - 1) \% 13 + 1\)th number of the \((v - 3)/13 + 1\)th suit (spades, hearts, clubs, diamonds), e.g., 1 is the ace of spades, 14 is the ace of hearts, 27 is the ace of clubs, 40 is the ace of diamonds. The array `layout` gives the layout of the cards in the 17 piles. The array `neighbours` lists pairs of cards which are \( \pm 1 \) w.r.t to each other, and is used in the `table` constraints to enforce the \( \pm 1 \) condition.
3.6. EXAMPLES OF AUTOMATIC CACHING

For simplicity of explanation we assume that the table constraint’s propagator is only checking, i.e., when all its variables are fixed, it checks whether that set of values is in its table and fails if it is not.

Consider two subproblems $P \equiv (V, D, C)$ where $D_F \equiv x_1 = 1, x_2 = 41, x_3 = 42, x_4 = 28, x_5 = 16, x_6 = 4, x_7 = 29, x_8 = 2$ and $P' \equiv (V, D', C)$ where $D_F \equiv x_1 = 1, x_2 = 28, x_3 = 42, x_4 = 41, x_5 = 16, x_6 = 4, x_7 = 29, x_8 = 2$, illustrated in Figure 3.2. Assume that both partial assignments satisfy the constraints. According to our manual analysis, since the last cards are the same and the set of unplayed cards are the same, $P$ and $P'$ should lead to equivalent subproblems.

Let $S$ denote the set of played cards, i.e., $\{1, 2, 4, 16, 28, 29, 41, 42\}$, which is the same for both $P$ and $P'$. In $P$, the inverse constraint will enforce $x_i \neq v$ for $i \geq 9, v \in S$, and $y_i \geq 9$ for $i \notin S$. The table constraints perform no propagations. The inequality constraints perform some propagations. For example, if card $i$ is under card $j$ and neither have been played, then since $x_i > x_j$, $x_j \geq 9 \rightarrow x_i \geq 10$. Now, the inverse constraint and the binary inequalities all have $key(c, D_F) \equiv true$. A table constraint which is only checking, has $key(c, D_F) \equiv true$ if it is satisfied or inactive, and has $key(c, D_F) \equiv D_F \cap \text{vars}(c)$ otherwise. The table constraints involving only $x_i$ where $i \leq 8$ are all satisfied and has $key(c, D_F) \equiv true$. The ones involving only $x_i$ where $i \geq 9$ are all inactive and has $key(c, D_F) \equiv true$. Thus the only one with a non-trivial key is the one involving $x_8$ and $x_9$, which has $key(c, D_F) \equiv x_8 = 2$. Thus the projection key characterizes the subproblem by $F$, $D_U$, and the last card played $x_8 = 2$. Since $F$ and $D_U$ are uniquely determined by the set of unplayed cards, our projection key detects the same amount of equivalence as the handcrafted key. However, it is much larger in size.

3.6.2 Graph Coloring

In Section 3.3, we gave a handcrafted key for the graph coloring problem. It turns out that this key, which is intuitively obvious, is not the strongest key possible for detecting subproblem equivalences. Projection gives a better key. Consider the simple graph coloring problem shown in Figure 3.3.
Consider $P \equiv (V, D, C)$ where $D_F \equiv x_1 = 1, x_2 = 1, x_3 = 3, x_4 = 4, x_5 = 3$ illustrated in Figure 3.3(a). After propagation, we have $F = \{x_1, x_2, x_3, x_4, x_5\}$ and $D_U \equiv \{x_6 \in \{1,2\}, x_7 \in \{1,2\}, x_8 \in \{1,2,3,4\}\}$. Since all constraints are binary inequalities, we have $key(c, D_F) \equiv true$. Thus, the projection key completely characterizes the subproblem by $F$ and $D_U$. Consider $P' \equiv (V, D', C)$ where $D'_F \equiv \{x_1 = 2, x_2 = 4, x_3 = 4, x_4 = 3, x_5 = 3\}$ illustrated in Figure 3.3(c). After propagation, we have $F' = \{x_1, x_2, x_3, x_4, x_5\}$ and $D'_U \equiv \{x_6 \in \{1,2\}, x_7 \in \{1,2\}, x_8 \in \{1,2,3,4\}\}$. Clearly, $F = F'$ and $D_U = D'_U$ and our projection key shows that these two subproblems are equivalent. However, our handcrafted key would have characterized $P$ with $(\{x_1, x_2, x_3, x_4, x_5\}, [(x_3, 3), (x_4, 4), (x_5, 3)])$ and $P'$ with $(\{x_1, x_2, x_3, x_4, x_5\}, [(x_3, 4), (x_4, 3), (x_5, 3)])$, which are clearly different. Thus, on this problem, the projection key is stronger than the intuitively obvious handcrafted key. The size of the keys are comparable as well.

### 3.6.3 Minimization of Open Stacks Problem

The model for this problem has been presented in Section 3.1. The linear constraints $\sum_{i=1}^n \text{bool2int}((s[i] \leq t) \land (e[i] \geq t)) \leq \text{objective}$ are flattened into primitive constraints before being sent to the solver, since solvers typically only handle primitive constraints natively. Explicitly, those linear inequalities would be replaced by:

```plaintext
array[1..n,1..n] of var bool: so;
array[1..n,1..n] of var bool: eo;
array[1..n,1..n] of var bool: o;

constraint forall (t in 1..n) (  
    forall (i in 1..n) (  
        so[i,t] <-> (s[i] <= t) /
        eo[i,t] <-> (e[i] >= t) /
        o[i,t] <-> (so[i,t] /
                   eo[i,t])
    ) /

    sum (i in 1..n) (bool2int(o[i,t])) <= objective)
```
3.6. EXAMPLES OF AUTOMATIC CACHING

Consider the problem data considered in Example 3.3.3. Suppose we are currently solving the problem with \( \text{objective} \leq 3 \). Let us consider the following two subproblems \( P \equiv (C, D) \) where \( D_F \equiv x[1] = 4, x[2] = 3, x[3] = 5 \), and \( P' \equiv (V, D', C) \) where \( D'_F \equiv x[1] = 5, x[2] = 4, x[3] = 3 \). We know from manual analysis that these two should produce equivalent subproblems. Let us look at the automatically generated projection keys.

In \( P \), the inverse constraint gives \( e[4] = 1, e[3] = 2, e[5] = 3 \). The minimum constraints give \( s[1] = 1, s[2] \in \{4, 5, 6\}, s[3] = 1, s[4] = 1, s[5] = 2, s[6] \in \{4, 5, 6\} \). The binary reification constraints and the clausal constraints will now fix many of the \( so, eo \) and \( o \). Everything with \( t \leq 3 \) becomes fixed. Everything related to customers 3, 4, 5 become fixed. All the \( so[i, t], t \geq 4 \) where customer \( i \)'s stack is currently open (i.e., customer 1) become fixed. All the \( eo[i, t], t = 4 \) becomes fixed. All the \( o[i, 4] \) where customer \( i \)'s stack is currently open become fixed. All the rest are unfixed. In \( D_U, s[2] \in \{4, 5, 6\}, s[6] \in \{4, 5, 6\}, e[1], e[2], e[6] \in \{4, 5, 6\}, x[4], x[5], x[6] \in \{4, 5, 6\} \), and all other unfixed variables are at their initial domains.

Now let's consider the constraint projections. The inverse\((e, x)\) constraint and the binary reification constraints \( so[i, t] \leftrightarrow (s[i] \leq t) \) and \( eo[i, t] \leftrightarrow (e[i] \geq t) \) all have \( \text{key}(c, D_F) \equiv \text{true} \). Of the minimum constraints, the ones regarding \( s[i], i = 1, 3, 4, 5 \) are all satisfied and have \( \text{key}(c, D_F) \equiv \text{true} \). The remaining two have no fixed variables and also have \( \text{key}(c, D_F) \equiv \text{true} \). The linear constraints for \( t = 1, 2, 3 \) are all satisfied and have \( \text{key}(c, D_F) \equiv \text{true} \). The linear constraint for \( t = 4 \), projects to \( o[2, 4] + o[6, 4] \leq 2 \), which is satisfied, so \( \text{key}(c, D_F) \equiv \text{true} \). The linear constraints for \( t = 5, 6 \) project to \( o[1, t] + o[2, t] + o[6, t] \leq 3 \) which is also satisfied, and so \( \text{key}(c, D_F) \equiv \text{true} \). Of the clausal constraints \( o[i, t] \leftrightarrow (so[i, t] \land eo[i, t]) \), all the ones with \( t = 1, 2, 3 \) are satisfied and have \( \text{key}(c, D_F) \equiv \text{true} \). All the ones with \( i = 3, 4, 5 \) are satisfied and have \( \text{key}(c, D_F) \equiv \text{true} \). The ones with \( o[i, t], i = 2, 6, t = 4, 5, 6 \) have no fixed variables and have \( \text{key}(c, D_F) \equiv \text{true} \). The one with \( o[1, 4] \) is satisfied and has \( \text{key}(c, D_F) \equiv \text{true} \). The ones with \( o[i, t], i = 1, t = 5, 6 \) are active, and have \( \text{key}(c, D_F) \equiv \neg eo[i, t] \lor o[i, t] \). As can be seen, almost all the constraints have \( \text{key}(c, D_F) \equiv \text{true} \). This is typical of problems with lots of subproblem equivalences.
Many constraints will either be inactive, or will propagate in such a way that they are no longer relevant for the remaining subproblem.

Now, let us consider $P'$. Propagation on the inverse constraint gives $e[5] = 1, e[4] = 2, e[3] = 3$. The minimum constraints give $s[1] = 2, s[2] \in \{4, 5, 6\}, s[3] = 1, s[4] = 2, s[5] = 1, s[6] \in \{4, 5, 6\}$. Once again, the binary reification constraints and the clausal constraints will now fix many of the $so, eo$ and $o$. We can see, from the previous description, that $F, D_U$ and $key(c, D_F)$ only depends on: the set of closed customers, and the set of currently open customers. These things are identical for $P$ and $P'$, thus the projection keys of these two subproblems match. The key point here is that although $e[3], e[4], e[5], s[1], s[3], s[4], s[5]$ take on different values in $P$ and $P'$, their effect on the variables with $t \geq 4$ are identical. And since the remaining non-satisfied constraints in each subproblem only involve variables with $t \geq 4$, the subproblems end up being equivalent.

In general, if the set of closed customers is $S$, then the set of currently open customers is $\{i \mid i \notin S, \exists j \in S.W(i, j)\}$, which clearly only depend on $S$. Thus any two subproblems with the same set of closed customers will have the same projection key. This means that our automatically generated projection key detects exactly the same amount of subproblem equivalence as the handcrafted key presented in Section 3.3. However, the projection key is clearly much larger. On the other hand, this subproblem equivalence is fairly hard to identify manually. The MOSP was the subject of the 2005 Modeling Challenge [146], and of the 13 entrants into the competition, only 3 identified this subproblem equivalence. Thus such subproblem equivalences are hard to identify even for expert problem modelers.

### 3.7 Related Work

Problem specific approaches to dominance detection/subproblem equivalence are widespread in combinatorial optimization (see e.g., [51, 145]). There is also a significant body of work on caching that rely on problem decomposition by fixing variables (e.g [86, 100]). This work effectively looks for equivalent projected problems, but since they do not take into account the semantics of the constraints, they effectively
use $D_{F \cap \text{vars}(c)}$ for every constraint $c$ as the projection key, which finds strictly fewer equivalent subproblems than our approach to keys. The success of these approaches in finding equivalent subproblems relies on decomposing the projected subproblem into disjoint parts. We could extend our approach to also split the projected problem into connected components but this typically does not occur in the problems of interest to us. Interestingly, [86] uses symmetry detection to detect symmetric (rather than equivalent) subproblems, but the method used does not appear to scale.

3.7.1 Dynamic Programming

Dynamic programming (DP) [11] is a powerful approach for solving optimization problems whose optimal solutions are derivable from the optimal solutions of its subproblems. It relies on formulating an optimization as recursive equations relating the answers to optimization problems of the same form. When applicable, it is often near unbeatable by other optimization approaches.

Constraint programming (CP) with caching is similar to DP, but provides several additional capabilities. For example, arbitrary side constraints not easily expressible as recursions in DP can easily be expressed in CP, and dominance can be expressed and exploited much more naturally in CP.

Consider the 0-1 Knapsack problem, a well known NP-hard problem that is easy to formulate using recursive equations suitable for DP. The DP formulation defines $knp(j, w)$ as the maximum profit achievable using the first $j$ items with a knapsack of size $w$. The recursive equation is:

$$
knp(j, w) = \begin{cases} 
0 & j = 0 \lor w \leq 0 \\
\max(knp(j-1, w), knp(j-1, w-w_j) + p_j) & \text{otherwise}
\end{cases}
$$

The DP solution is $O(nW)$ since values for $knp(j, w)$ are cached and only computed once.

The 0-1 knapsack problem is terrible in its natural CP formulation, it requires $O(2^n)$ search to prove the optimal solution. CP with caching will store as the key: the set of fixed variables $F$, an integer $w$ representing the remaining weight limit,
and an integer \( p \) the remaining profit required. We can see that with a fixed search ordering, \( F \) can only take on \( n \) values, the representation for the projection key of the weight constraint can only take on \( W \) values, and the representation for the projection key of the profit constraint can only take on \( U \) values where \( U \) is the optimal profit. Thus the maximum number of distinct entries in the hash table, and hence the worse case complexity of the algorithm, is \( O(nWU) \), which is much better than \( O(2^n) \), but not quite as good as the DP solution.

The solutions are in fact quite different: the DP approach stores the optimal profit for each set of unfixed variables and remaining weight limit, while the CP approach stores the fixed variables and remaining weight plus remaining profit required. The CP approach in fact implements a form of DP with bounding [117]. In particular, the CP approach can detect subproblem dominance. A problem with remaining weight \( w' \) and remaining profit required \( p' \) is dominated by a problem with remaining weight \( w \geq w' \) and remaining profit \( p \leq p' \). The DP solution must examine both subproblems since the remaining weights are different.

In practice the number of remaining profits arising for the same set of fixed variables and remaining weight is \( O(1) \) and hence the practical number of subproblems visited by the CP approach is \( O(nW) \).

Note that while adding a side constraint like \( x_3 \geq x_8 \) destroys the DP approach (or at least forces it to be carefully reformulated), the CP approach with automatic caching works seamlessly.

### 3.7.2 Symmetry Breaking

Symmetry breaking aims to speed up execution by not exploring search nodes known to be symmetric to nodes already explored. Once the search is finished, all solutions can be obtained by applying each symmetry to each solution. In particular, Symmetry Breaking by Dominance Detection (SBDD) [43] works by performing a “dominance check” at each search node and, if the node is found to be dominated, not exploring the node.
At first glance, SBDD may appear to be very similar to automatic caching. However, the two techniques actually exploit different kinds of dominances. SBDD detects and exploits symmetric subproblems, whereas automatic caching exploits equivalent and dominated subproblems. In our terminology, we could describe SBDD as follows.

Let $\sigma$ be a mapping from valuations of $P$ to valuations of $P$. $\sigma$ is a symmetry if: $\sigma(\theta) \in \text{solns}(P)$ iff $\theta \in \text{solns}(P)$. We can extend $\sigma$ to a mapping from constraints to constraints as follows: $\theta \in \text{solns}(\sigma(c))$ iff $\sigma(\theta) \in \text{solns}(c)$. We also extend $\sigma$ to map variables to variables.

**Definition 3.7.1.** Let $P \equiv (V, D, C)$ and $P' \equiv (V, D', C)$ be two different subproblems arising during search. Let $F = \text{fixed}(D)$, $U = \text{vars}(C) \setminus F$, $F' = \text{fixed}(D')$, $U' = \text{vars}(C) \setminus F'$. $P$ symmetrically dominates $P'$ w.r.t. symmetry $\sigma$ iff

- $\text{fixed}(D) = \text{fixed}(\sigma(D'))$
- $\sigma(\pi_U(C \land D')) \Rightarrow \pi_U(C \land D)$

It follows from the definition of entailment and symmetry that if $P$ symmetrically dominates $P'$ and $P$ has no solutions, then $P'$ has no solutions either. SBDD works by detecting $\sigma(D'_{F'}) = D_F$ where $D_F$ is a previous partial assignment that led to failure. Since $\sigma(D'_{F'}) = D_F$ implies $\sigma(\pi_U(C \land D')) \Rightarrow \pi_U(C \land D)$, SBDD can correctly prune $P'$. Automatic caching on the other hand, looks for subproblem dominance of the form $\pi_U(C \land D') \Rightarrow \pi_U(C \land D)$, where $D'_F \neq D_F$, which is a different set of dominances.

Note that it is perfectly feasible, and correct, to use both automatic caching and SBDD simultaneously. At a particular node, automatic caching will prune the node if it detects that the subproblem is dominated by a previously failed subproblem, while SBDD will prune the node if it detects that it is dominated by a previously failed symmetric subproblem. We show with the following example that neither method subsumes the other.

**Example 3.7.2.** Consider $P \equiv (V, D, C)$ where $V \equiv \{x_1, \ldots, x_5\}$, $D \equiv \{x_1 \in \{1..3\}, x_2 \in \{1..4\}, x_3 \in \{2..5\}, x_4 \in \{1..5\}, x_5 \in \{1..5\}\}$ and $C \equiv \{\text{alldiff}([x_1, x_2, x_3, x_4, x_5]), x_1 + 4 \leq x_4 + x_5\}$. There is a variable symmetry between $x_4$ and $x_5$. 
Consider the subproblems $P$ and $P'$ where $D_F \equiv \{x_1 = 1, x_2 = 2\}$, and $D'_F \equiv \{x_1 = 2, x_2 = 1\}$. In both cases, the `alldiff` propagates to give $x_3 \in \{3, 4, 5\}, x_4 \in \{3, 4, 5\}, x_5 \in \{3, 4, 5\}$. The linear constraint becomes satisfied in both and has $key(c, D_F) \equiv true$. Thus their projection keys match and automatic caching detects that $P$ and $P'$ are equivalent. On the other hand, there is no symmetry which maps $x_1 = 1, x_2 = 2$ to $x_1 = 2, x_2 = 1$, so SBDD can do nothing here.

Consider the subproblems $P$ and $P'$ where $D_F \equiv \{x_1 = 2, x_4 = 3\}$, and $D'_F \equiv \{x_1 = 2, x_5 = 3\}$. In $P$, propagation gives $x_2 \in \{1, 4\}, x_3 \in \{4, 5\}, x_5 \in \{4, 5\}$. In $P'$, propagation gives $x_2 \in \{1, 4\}, x_3 \in \{4, 5\}, x_4 \in \{4, 5\}$. SBDD would detect that these two subproblems are symmetric, because the partial assignments $x_1 = 2, x_4 = 3$ and $x_1 = 2, x_5 = 3$ map to each other under the symmetry $x_4 \leftrightarrow x_5$. On the other hand, automatic caching determines that they are not equivalent since $F \neq F'$, and can do nothing here.

It is also possible that both techniques detect a subproblem dominance. Consider the subproblems $P$ and $P'$ where $D_F \equiv \{x_4 = 3, x_5 = 4\}$, and $D'_F \equiv \{x_4 = 4, x_5 = 3\}$. Both propagate to $x_1 \in \{1, 2\}, x_2 \in \{1, 2\}, x_3 \in \{2, 5\}$. The linear constraint becomes satisfied in both and has $key(c, D_F) \equiv true$. Thus their projection keys match and automatic caching detects that $P$ and $P'$ are equivalent. SBDD detects that they are also symmetric since $x_4 = 3, x_5 = 4$ and $x_4 = 4, x_5 = 3$ map to each other under $x_4 \leftrightarrow x_5$. □

It is possible to extend our caching approach to exploit whole problem symmetries.

**Theorem 3.7.3.** Let $P \equiv (V, D, C)$ and $P' \equiv (V, D', C')$ be subproblems arising during search. Let $F = fixed(D), U = \text{vars}(C) \setminus F, F' = fixed(D'), U' = \text{vars}(C) \setminus F'$. Suppose $\sigma$ is a symmetry s.t. $\forall c \in C, \sigma(c) \in C$. If $\sigma(F') = F$, and $\forall c \in C, \sigma(key(c, D'_F)) \Rightarrow key(\sigma(c), D_F)$ and $\sigma(D'_U) \Rightarrow D_U$, then $P$ symmetrically dominates $P'$ w.r.t. symmetry $\sigma$. □

Theorem 3.7.3 tells us that once again, we can detect subproblem symmetric dominance by doing a piecewise comparison of constraint projection keys.

Our characterization of subproblem symmetry in terms of projections is strictly more general than the characterization using partial assignments. This is because
3.7. RELATED WORK

\[ \sigma(D'_F) = D_F \] implies \[ \sigma(\pi_U(C \land D')) \Rightarrow \pi_U(C \land D) \], but not vice versa. Therefore it is possible that \( P \) and \( P' \) are symmetric subproblems even when \( \sigma(D'_F) \neq D_F \). For example, this can occur when conditional symmetries exist. Since SBDD only detects \( \sigma(D'_F) = D_F \), it only exploits a subset of the possible subproblem symmetries. Automatic caching, which directly works on subproblem projections, has the potential to exploit a strictly wider range of subproblem symmetries than SBDD. Extending automatic caching to efficiently exploit symmetries is a clear avenue of future work.

3.7.3 Nogood learning

Nogood learning approaches in constraint programming attempt to learn nogoods from failures and record these as new constraints in the program. Automatic caching can be considered one such technique, since we record sets of constraint projections which cause failure as nogoods. Another very successful methods is Lazy Clause Generation (LCG) [113, 45]. LCG use nogoods consisting of equality literals \((x = v)\), disequality literals \((x \neq v)\) and inequality literals \((x \geq v, x \leq v)\) to record the reasons for failures, and use SAT techniques to efficiently manage the nogoods. LCG also exploits subproblem equivalences/dominances, and is the main competitor to automatic caching.

In terms of pruning power, neither technique strictly dominates the other. LCG has two advantages. Firstly, in LCG, only constraints which are directly involved in the conflict are used to produce the nogood, whereas in automatic caching, all active and non-satisfied constraints are used. This sometimes allows LCG to produce more powerful nogoods than automatic caching. Secondly, the nogoods derived by LCG can be propagated, rather than merely be checked for failure, so they can prune more. These two factors sometimes allow a much greater reduction in search space. However this extra power comes with a high overhead. In LCG, every nogood that is derived is added as a clausal propagator. This takes \( O(n) \) to propagate where \( n \) is the number of nogoods kept. In automatic caching however, the failure check is done in \( O(1) \).

Automatic caching also has an advantage in terms of the kinds of nogoods they
can express. The language of nogoods used by LCG, i.e., atomic constraints of the form $v = d$ and $v \leq d$, limits the kinds of nogoods that can be expressed, whereas the constraint projections used in automatic caching can sometimes express nogoods which are more powerful. Consider the subproblem $x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20 \land C$, with $D \equiv \{x_1 = 1 \land x_2 = 2 \land x_3 = 3\}$. If this subproblem fails, the projection key stores that $x_4 + 2x_5 \leq 12 \land$ other keys leads to failure. LCG would express this as $x_1 \geq 1 \land x_2 \geq 2 \land x_3 \geq 3 \land$ other keys leads to failure, since there are no literals to represent partial sums. The nogood from automatic caching is strictly stronger. For example, $D \equiv \{x_1 = 2 \land x_2 = 3 \land x_3 = 0\}$ would fire the nogood from automatic caching, but not the one from LCG. This difference is apparent in the experimental results for 0-1 Knapsack.

Another advantage of automatic caching over LCG is in the ease of implementation. Automatic caching can be added to a CP solver by adding to each propagator code to build a representation of the propagator for inclusion in the key, as well as adding a generic caching mechanism to the search. In Section 3.5 we show how to do this for many constraints, and give generic approaches to building efficient representation for global constraints via decomposition. LCG requires much more fundamental changes to a CP solver to record an implication graph of inferences made. In particular each global constraint needs to be extended to explain its inferences, and this is far from straightforward.

3.8 Experiments

We use the state of the art CP solver Chuffed in our experiments. Chuffed can be run as a naive CP solver (denoted as Chuffed), as a CP solver with caching (denoted as ChuffedC), and as a Lazy Clause Generation solver (denoted as ChuffedL). We also compare against Gecode 3.2.2 [56] – widely recognized as one of the fastest constraint programming systems (to illustrate we are not optimizing a slow system). We use the MurmurHash 2.0 hash function [6]. We use models written in the modeling language MiniZinc [112]. This facilitates a fair comparison between the solvers, as all solvers use the same model and search strategy. Note that caching and lazy clause
3.8. **EXPERIMENTS**

generation do not interfere with the search strategies used here, as all they can do is fail subtrees earlier. Thus, **Chuffed** with caching or lazy clause generation always finds the same solution as the naive version **Chuffed** and Gecode, and any speedup observed comes from a reduced search.

The experiments were conducted on Xeon Pro 2.4GHz processors with a 900 second timeout. Tables 3.1 and 3.2 presents the number of variables and constraints as reported by **Chuffed**, the times for each solver in seconds as well as the number of (leaf) fails. For the automatic caching solver we also report the number of cache hits occurring in the search and the average key size (in bytes). We discuss the results for each problem below. All the MiniZinc models and instances are available at [www.cs.mu.oz.au/~pjs/autocache/](http://www.cs.mu.oz.au/~pjs/autocache/).

### 3.8.1 Knapsack

0-1 knapsack is ideal for caching. There are two linear constraints: one for the weight and one for the objective, and the domains are all binary. The representation of the key thus simply requires the set of fixed variables and two integers. The non-caching solvers timeout as $n$ increases, as their time complexity is $O(2^n)$. This is a worst case for lazy clause generation since the nogoods generated are not reusable at all.

**ChuffedC**, on the other hand, is easily able to solve much larger instances (see Table 3.1). The node to $nW$ ratio (not shown) stays fairly constant as $n$ increases (varying between 0.86 and 1.06), showing that it indeed has search (node) complexity $O(nW)$. The time to $nW$ ratio grows as $O(n)$ though, since we are using a general CP solver where the linear constraints take $O(n)$ to propagate at each node, while DP requires constant work per node. Hence, automatic caching is not as efficient as pure DP.

### 3.8.2 MOSP

The MOSP model used in the experiments is similar to that presented in Section 3.3, but has some additional conditional dominance breaking constraints [28] that make...
CHAPTER 3. AUTOMATIC CACHING VIA CONSTRAINT PROJECTION

the (non-caching) search much faster. We use random instances from [28]. Automatic caching gives up to two orders of magnitude speedup. The speedup grows exponentially with problem size. LCG also exploits a similar amount of equivalence/dominance and achieves a similar speedup.

3.8.3 Black Hole

The Black Hole model used in the experiments is similar to that presented in Section 3.6.1, but has additional conditional symmetry breaking constraints [61]. We generated random instances and used only the hard ones for this experiment. Automatic caching gives a modest speedup of around 2-3. The speedup is relatively low on this problem because the conditional symmetry breaking constraints have already removed many equivalent subproblems, and the caching is only exploiting the ones which are left. Note that the manual caching reported in [145] achieves speedups in the same range (on hard instances). LCG exploits even more dominance than automatic caching. But this extra dominance exploitation comes at such a high cost that it ends up being much slower overall.

3.8.4 BACP

In the Balanced Academic Curriculum Problem (BACP), we form a curriculum by assigning a set of courses to a set of periods, with certain restrictions on how many courses and how much “course load” can be assigned to each period. We also have prerequisite constraints between courses. The BACP can be viewed as a bin packing problem with a lot of additional side constraints. The subproblem symmetry is the same as that in bin packing. Suppose that a set of courses have been assigned to the first $k$ periods and has “filled” them. The remaining subproblem only depends on the set of unassigned courses, and not on how the earlier courses were assigned to the first $k$ periods. Any permutation of those $k$ period assignments that satisfy the constraints lead to the same subproblem. We use the model of [71], but with some additional redundant constraints that considerably reduce the search required to find an optimal solution for caching and non-caching solvers. Note that by Theorem 3.4.15,
the redundant linear constraints do not have to be considered in the projection key. The 3 instances curriculum\textsubscript{8/10/12} given in CSPLib can be solved to optimality in just a few milliseconds. We generate random instances with 50 courses, 10 periods, and course credit ranging between 1 and 10. Almost all are solvable in milliseconds so we pick out only the non-trivial ones for the experiment. We also include the 3 standard instances from CSPLib. Automatic caching can give orders of magnitude speedup. Both automatic caching and lazy clause generation are capable of exploiting the subproblem equivalence, giving orders of magnitude speedup. In this case, lazy clause generation exploits more equivalence/dominance and is faster.

### 3.8.5 Radiation Therapy

In the Radiation Therapy problem [7], the aim is to decompose an integral intensity matrix describing the radiation dose to be delivered to each area, into a set of patterns to be delivered by a radiation source, while minimizing the amount of time the source has to be switched on, as well as the number of patterns used (setup time of machine). The subproblem equivalence arises because there are equivalent methods to obtain the same cell coverages, e.g., radiating one cell with two intensity 1 patterns is the same as radiating it with one intensity 2 pattern, etc. We use random instances generated as in [7]. Once again automatic caching can give orders of magnitude speedup. Both automatic caching and lazy clause generation produce orders of magnitude speedup, though lazy clause generation once again exploits more equivalence/dominance and is faster.

### 3.8.6 Memory Consumption

The memory consumption of our caching scheme is linear in the number of nodes searched. The size of each key is dependent on the structure of the problem and can range from a few hundred bytes to thousands of bytes. One can read the total cache size from the tables as Key size multiplied by number of fails. On a modern computer, this means we can usually search several hundreds of thousands of nodes before running out of memory. There are simple schemes to reduce the memory usage,
which we plan to investigate in the future. For example, much like in SAT learning, we can keep an “activity” score for each entry to keep track of how often they are used. Inactive entries can then periodically be pruned to free up memory.

3.8.7 Limitations

Automatic caching is not useful for every problem. Although many structured problems do exhibit subproblem dominances, there are also many problems where subproblem dominances do not occur at all. When automatic caching is used on such problems, it will incur a constant factor overhead and produce no benefit whatsoever. The number of cache hits provides a good measure of whether the caching is useful or not. If the proportion of nodes which are being pruned by the caching is low or zero, then the caching is probably not useful. Ideally, the user can simply turn off automatic caching when this occurs. Alternatively, it may be possible to have the solver dynamically check whether the caching is working and to turn it off when it is not. This is an avenue of future research.

3.9 Conclusion

In this chapter, we developed the theory behind subproblem dominance, and showed that they can be detected by comparing projection keys. We described a new technique called automatic caching via constraint projection, where constraint projection is used to derive nogoods from each failed subproblem. These nogoods are propagated using a hash table so that failure can be detected in $O(1)$ time. Our experiments showed that our new technique can produce orders of magnitude speedup compared to the base solver on problems containing subproblem dominances. Comparison with lazy clause generation, which is the current state of the art in nogood learning, shows that the nogoods derived via projection are competitive with those that LCG derives via constraint resolution. The nogoods derived via projection can be exponentially stronger in some problems, e.g., ones with long linear constraints, but can also be
weaker on other problems. The speedups from our technique tends to grow exponentially with problem size.

Automatic caching via constraint projection is quite robust. It can find and exploit subproblem dominances even in models that are not “pure”, e.g., MOSP with dominance and conditional symmetry breaking constraints, Black Hole with conditional symmetry breaking constraints, and BACP which can be seen as bin packing with lots of side constraints and some redundant constraints. It is also easier to extend a CP system to include automatic caching via constraint projection than it is to extend it to implement lazy clause generation. In particular we explained how to easily define projection keys for arbitrarily complex global constraints.
<table>
<thead>
<tr>
<th>Instance</th>
<th>vs Core</th>
<th>Cache</th>
<th>Chuffed</th>
<th>Gecode</th>
<th>Time Fails</th>
<th>Cache hits</th>
<th>Key size</th>
</tr>
</thead>
<tbody>
<tr>
<td>knapsack-50</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
</tr>
<tr>
<td>knapsack-30</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
</tr>
<tr>
<td>knapsack-40</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
</tr>
<tr>
<td>knapsack-60</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
</tr>
<tr>
<td>knapsack-70</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
</tr>
<tr>
<td>knapsack-100</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
<td>–</td>
<td>900</td>
</tr>
</tbody>
</table>

Table 3.1: Experimental Results: Knapsack, MOSP and Black Hole
<table>
<thead>
<tr>
<th>Instance</th>
<th>Vars.</th>
<th>Cons.</th>
<th>Gecode</th>
<th>CHUFFED</th>
<th>CHUFFEDL</th>
<th>CHUFFEDC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Fails</td>
<td>Time</td>
<td>Fails</td>
<td>Time</td>
<td>Fails</td>
</tr>
<tr>
<td>curriculum_8</td>
<td>0.01</td>
<td>0</td>
<td>0.01</td>
<td>0</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>curriculum_10</td>
<td>0.01</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>curriculum_12</td>
<td>0.01</td>
<td>0</td>
<td>0.01</td>
<td>0</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>bcp-medium-1</td>
<td>1121</td>
<td>2654</td>
<td>29.47</td>
<td>404779</td>
<td>5.95</td>
<td>26764</td>
</tr>
<tr>
<td>bcp-medium-2</td>
<td>1122</td>
<td>2650</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>0.01</td>
<td>41</td>
</tr>
<tr>
<td>bcp-medium-3</td>
<td>1121</td>
<td>2648</td>
<td>46.24</td>
<td>8330028</td>
<td>0.03</td>
<td>105</td>
</tr>
<tr>
<td>bcp-medium-4</td>
<td>1119</td>
<td>2644</td>
<td>5.76</td>
<td>101650</td>
<td>0.69</td>
<td>5441</td>
</tr>
<tr>
<td>bcp-medium-5</td>
<td>1119</td>
<td>2641</td>
<td>54.17</td>
<td>896859</td>
<td>0.30</td>
<td>1783</td>
</tr>
<tr>
<td>bcp-hard-1</td>
<td>1121</td>
<td>2655</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>0.01</td>
<td>5</td>
</tr>
<tr>
<td>bcp-hard-2</td>
<td>1118</td>
<td>2651</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>0.01</td>
<td>8</td>
</tr>
<tr>
<td>radiation-6-9-1</td>
<td>877</td>
<td>942</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>1.39</td>
<td>4290</td>
</tr>
<tr>
<td>radiation-6-9-2</td>
<td>877</td>
<td>942</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>2.08</td>
<td>7835</td>
</tr>
<tr>
<td>radiation-7-8-1</td>
<td>1076</td>
<td>1168</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>0.31</td>
<td>1361</td>
</tr>
<tr>
<td>radiation-7-8-2</td>
<td>1076</td>
<td>1168</td>
<td>188.12</td>
<td>3940867</td>
<td>0.22</td>
<td>808</td>
</tr>
<tr>
<td>radiation-7-9-1</td>
<td>1210</td>
<td>1301</td>
<td>312.09</td>
<td>5519065</td>
<td>0.78</td>
<td>3341</td>
</tr>
<tr>
<td>radiation-7-9-2</td>
<td>1210</td>
<td>1301</td>
<td>140.44</td>
<td>2316137</td>
<td>3.13</td>
<td>18207</td>
</tr>
<tr>
<td>radiation-8-9-1</td>
<td>1597</td>
<td>1718</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>1.89</td>
<td>5232</td>
</tr>
<tr>
<td>radiation-8-9-2</td>
<td>1597</td>
<td>1718</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>2.04</td>
<td>6518</td>
</tr>
<tr>
<td>radiation-8-10-1</td>
<td>1774</td>
<td>1894</td>
<td>9.73</td>
<td>101062</td>
<td>8.95</td>
<td>201703</td>
</tr>
<tr>
<td>radiation-8-10-2</td>
<td>1774</td>
<td>1894</td>
<td>&gt;900</td>
<td>&gt;900</td>
<td>5.75</td>
<td>10322</td>
</tr>
</tbody>
</table>
CHAPTER 3. AUTOMATIC CACHING VIA CONSTRAINT PROJECTION
Part II

Symmetry Breaking
Many constraint problems contain large amounts of symmetry. In such problems, examining one valuation is often enough to tell us everything we need to know about its symmetric counterparts (whether it is a solution, what its objective value is, etc). These redundancies in the search can exponentially increase the run time. There are two main ways to deal with this. We can either try to eliminate the symmetry by remodeling the problem (e.g., [61, 7, 143]), or we can apply some form of symmetry breaking which prevents the CP solver from exploring symmetric parts of the search space. In this part of the thesis, we concentrate on symmetry breaking techniques, which can give significant speedups on appropriate problems [48, 118, 47, 124, 119]. The most popular of these are: lexicographical symmetry breaking constraints [118, 32, 98, 120, 50], Symmetry Breaking During Search (SBDS) [62] and Symmetry Breaking by Dominance Detection (SBDD) [43].

**II.1 Definitions**

Symmetries have been expressed in many different ways depending on which subclass of symmetries the authors are trying to describe [48, 118, 104, 31]. In the most general form, symmetries can be defined as follows:

**Definition II.1.1.** A symmetry of a problem $P \equiv (V,D,C)$ is a bijective function $\sigma : \Theta_V \rightarrow \Theta_V$ mapping valuations to valuations such that: $\theta \in \text{solns}(P)$ iff $\sigma(\theta) \in \text{solns}(P)$, i.e., it maps solutions to solutions and non-solutions to non-solutions.

**Definition II.1.2.** A variable symmetry is a symmetry which permutes the set of variables in the problem. Let $\rho : V \rightarrow V$ be a permutation of the variables in the problem. Let $\sigma$ be a mapping defined by: $\sigma(\theta)(x) = \theta(\rho(x))$. If $\sigma$ is a symmetry, then it is called a variable symmetry.

**Example II.1.3.** Consider the problem $P \equiv (V,D,C)$, $V \equiv \{x_1, x_2, x_3\}$, $D \equiv \{x_i \in \{1,\ldots,5\} \mid i = 1, 2, 3\}$, $C \equiv \{\text{alldiff}(x_1, x_2, x_3), x_1 + x_2 + x_3 \geq 10\}$. $x_1$, $x_2$ and $x_3$ are fully interchangeable since they have the same domains, and they are interchangeable in all the constraints in the problem. For any permutation $\rho$ of $\{x_1, x_2, x_3\}$, the mapping $\sigma_\rho$ defined as: $\sigma_\rho(\theta)(x) = \theta(\rho(x))$, is a variable symmetry of $P$. 
**Definition II.1.4.** A value symmetry is a symmetry which permutes the values among a subset of variables in the problem. Let \( V' \subseteq V \) be a subset of the variables with identical domains \( T \). Let \( \rho : T \to T \) be a permutation of the values in \( T \). Let \( \sigma \) be a mapping defined by: \( \sigma(\theta)(x) = \rho(\theta(x)) \) for \( x \in V' \) and \( \sigma(\theta)(x) = \theta(x) \) otherwise. If \( \sigma \) is a symmetry, then it is called a value symmetry.

**Example II.1.5.** Consider the problem \( P \equiv (V, D, C) \), \( V \equiv \{x_1, x_2, x_3\}, D \equiv \{x_i \in \{1, \ldots, 5\} \mid i = 1, 2, 3\}, C \equiv \{\text{alldiff}(x_1, x_2, x_3), x_1 \neq x_2, x_2 \neq 2, x_3 \neq 4\} \). The values \( \{1, 3, 5\} \) on \( \{x_1, x_2, x_3\} \) are fully interchangeable, since they are interchangeable in all four of the constraints in the problem. For any permutation \( \rho \) of \( \{1, 2, 3, 4, 5\} \) such that \( \rho(2) = 2 \) and \( \rho(4) = 4 \), the mapping \( \sigma_\rho \) defined as: \( \sigma_\rho(\theta)(x) = \rho(\theta(x)) \), is a value symmetry of \( P \).

**Definition II.1.6.** A variable sequence symmetry is a symmetry which permutes sequences of variables in the problem. Let \( Q \equiv [[x_{1,1}, \ldots, x_{1,n}], \ldots, [x_{n,1}, \ldots, x_{n,m}]] \) be an \( n \times m \) matrix of variables where each variable is distinct. Let \( \rho \) be a permutation of \( \{1, \ldots, n\} \). Let \( \sigma \) be a mapping defined by: \( \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, m\}, \sigma(\theta)(x_{i,j}) = \theta(x_{\rho(i),j}) \) and \( \sigma(\theta)(x) = \theta(x) \) for \( x \notin Q \). If \( \sigma \) is a symmetry, then it is called a variable sequence symmetry.

**Example II.1.7.** Consider the problem \( P \equiv (V, D, C) \), \( V \equiv \{x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2}, x_{3,1}, x_{3,2}\}, D \equiv \{x_{i,j} \in \{1, \ldots, 5\}\}, C \equiv \{x_{1,1} + x_{1,2} = 4, x_{2,1} + x_{2,2} = 4, x_{3,1} + x_{3,2} = 4, x_{1,1} + x_{2,1} + x_{3,1} = 5, x_{1,2} + x_{2,2} + x_{3,2} = 7\} \). We can view the \( x_{i,j} \) as a 3 \( \times \) 2 matrix. The constraints state that all three rows sum to 4, the first column sums to 5, and the second column sums to 7. The rows are all interchangeable, e.g., simultaneously swapping \( x_{1,1} \) with \( x_{2,1} \) and \( x_{1,2} \) with \( x_{2,2} \) preserves solutions. Given any permutation \( \rho \) of \( \{1, 2, 3\} \), the mapping \( \sigma_\rho \) defined by: \( \sigma_\rho(\theta)(x_{i,j}) = \theta(x_{\rho(i),j}) \) is a variable sequence symmetry of \( P \).

We are often interested in considering sets of symmetries which form a mathematical group under function composition.

**Definition II.1.8.** A symmetry group of problem \( P \) is a set \( S \) of symmetries of \( P \), s.t. \( S \) is a group under the operation of function composition, i.e., \( \forall \sigma_1, \sigma_2 \in S, \sigma_1 \cdot \sigma_2 \in S, \forall \sigma \in S, \sigma^{-1} \in S \), and the identity mapping \( I \) is in \( S \).
Example II.1.9. In Example II.1.3, if \( R \) is the set of permutations of \( \{x_1, x_2, x_3\} \), then the set of symmetries \( \{\sigma_\rho \mid \rho \in R\} \) forms a symmetry group.

Definition II.1.10. The orbit of a valuation \( \theta \) under the symmetry group \( S \) is the set: \( \{\sigma(\theta) \mid \sigma \in S\} \).

Clearly, each orbit under \( S \) consists either entirely of solutions, or entirely of non-solutions.

Definition II.1.11. A symmetry breaking technique is complete w.r.t. symmetry group \( S \) if one and only one valuation within each orbit under \( S \) is not pruned by the technique.

II.2 Lexicographical Symmetry Breaking

Lexicographical symmetry breaking constraints break the symmetry by restricting the search to find only solutions which are lexicographically least within their orbit.

Definition II.2.1. A lexicographical ordering \(<_{\text{lex}}\) on the valuations of problem \( P \) is defined as follows. We first define a total ordering \(<_v\) on the variables in \( P \). Without loss of generality, let this ordering be: \( x_1 <_v x_2 <_v \ldots <_v x_n \). We then define the total ordering \(<_{\text{lex}}\) based on \(<_v\) as: \( \theta_1 <_{\text{lex}} \theta_2 \) iff \( \exists k \in \{1, \ldots, n\} \) s.t. \( \forall i < k, \, \theta_1(x_i) = \theta_2(x_i) \) and \( \theta_1(x_k) < \theta_2(x_k) \).

Definition II.2.2. A lexicographical symmetry breaking constraint for symmetry group \( S \) and lexicographical ordering \(<_{\text{lex}}\) is a constraint \( \text{sb}(S, <_{\text{lex}}) \) such that: \( \forall \theta \in \Theta_V, \, \min_{\text{lex}}\{\sigma(\theta) \mid \sigma \in S\} \in \text{solns}(\text{sb}(S, <_{\text{lex}})) \), i.e., within each orbit, the lex-least valuation must not be pruned by \( \text{sb}(S, <_{\text{lex}}) \). It is complete if: \( \theta \in \text{solns}(\text{sb}(S, <_{\text{lex}})) \) iff \( \theta = \min_{\text{lex}}\{\sigma(\theta) \mid \sigma \in S\} \), i.e., within each orbit, only the lex-least valuation is not pruned by \( \text{sb}(S, <_{\text{lex}}) \).

\( \text{sb}(S, <_{\text{lex}}) \) can be implemented as one global constraint, or more commonly, decomposed into a set of smaller constraints. For most simple symmetry groups, the complete lexicographical symmetry breaking constraint is easy to find and can be propagated efficiently.
Example II.2.3. Suppose problem $P$ has a variable symmetry on $\{x_1, \ldots, x_n\}$. Let $S$ be the symmetry group corresponding to this variable symmetry. Let $<_\text{lex}$ be the lexicographical ordering based on the variable ordering $x_1 < v \ldots < v x_n$. Then the complete lexicographical symmetry breaking constraint can be expressed as: $sb(S,<_\text{lex}) \equiv \land_{i=1}^{n-1} x_i \leq x_{i+1}$. This constraint can be propagated to domain consistency in $O(n)$.

Suppose problem $P$ has a value symmetry on the values $\{1, \ldots, k\}$ over the variables $\{x_1, \ldots, x_n\}$. Let $S$ be the symmetry group corresponding to this value symmetry. Let $<_\text{lex}$ be the lexicographical ordering based on the variable ordering $x_1 < v \ldots < v x_n$. Suppose we take the minimum of an empty set to be $\infty$ and take $\infty < \infty$ to be true. Then the complete lexicographical symmetry breaking constraint can be expressed as: $sb(S,<_\text{lex}) \equiv \land_{i=1}^{k-1} \min\{i \mid x_i = v\} < \min\{i \mid x_i = v + 1\}$. This constraint can be propagated to domain consistency in $O(n)$.

Suppose problem $P$ has a variable sequence symmetry on the variable sequences $\{[x_{1,1}, \ldots, x_{1,n}], \ldots, [x_{m,1}, \ldots, x_{m,n}]\}$. Let $S$ be the symmetry group corresponding to this variable sequence symmetry. Let $<_\text{lex}$ be the lexicographical ordering based on the variable ordering $x_{1,1} < v \ldots < v x_{m,1} < v \ldots < v x_{1,n} < v \ldots < v x_{m,n}$. Then the complete lexicographical symmetry breaking constraint can be expressed as: $sb(S,<_\text{lex}) \equiv \land_{i=1}^{n-1} \text{lexlesseq}([x_{i,1}, \ldots, x_{i,n}], [x_{i+1,1}, \ldots, x_{i+1,n}])$, where $\text{lexlesseq}([a_1, \ldots, a_m], [b_1, \ldots, b_m]) \equiv (a_1 < b_1) \lor (a_1 = b_1 \land a_2 < b_2) \lor \ldots \lor (a_1 = b_1 \land \ldots \land a_{m-1} = b_{m-1} \land a_m \leq b_m)$. This constraint can be propagated to domain consistency in $O(n^2)$.

For more complicated symmetry groups, the complete lexicographical symmetry breaking constraint may be hard to express and may be very costly to propagate to domain consistency. In such cases, it is typically better to use an incomplete lexicographical symmetry breaking constraint which can be propagated in a reasonable time [50, 47].

Example II.2.4. Suppose problem $P$ had variables $\{x_{i,j} \mid i, j = 1, \ldots, n\}$ and has both a row and column symmetry on the $x_{i,j}$. Let $S_1$ denote the symmetry group consisting of row symmetries, $S_2$ denote the symmetry group consisting of column
symmetries, and $S$ denote the group generated by $S_1$ and $S_2$. The complete lexicographical symmetry breaking constraint for $S$ has no simple closed form, and is NP-hard to propagate to domain consistency [13]. This is clearly not practical. Instead, we can post the complete lexicographical symmetry breaking constraints for $S_1$ and $S_2$, i.e., break all row symmetries and break all column symmetries individually. This is incomplete for $S$, but only costs $O(n^2)$ to propagate, and still breaks enough of the symmetry to be worthwhile [47].

Lexicographical symmetry breaking constraints are simple to implement and can work very well under many circumstances. However, they are not always the best choice of symmetry breaking, as the lexicographical ordering may conflict with the search strategy. For example, suppose we chose $\prec_{lex}$ such that the first solution that the search encounters is not lex-least within its orbit. Then enforcing $sb(S, \prec_{lex})$ would prune off this solution, forcing the search to find another solution. In the worse case, adding the lexicographical symmetry breaking constraint may actually slow down the search. It is often possible to choose a $\prec_{lex}$ which is consistent with the search strategy so that there is no conflict. However, if that is not possible, e.g., because we are using a dynamic search strategy, then other symmetry breaking techniques may be more appropriate.

II.3 Dynamic Symmetry Breaking

Dynamic symmetry breaking techniques like SBDS and SBDD never conflict with the search strategy. Rather than preferring the lexicographically least solution in each orbit, dynamic techniques always prefer the first solution found in each orbit. Although SBDS, SBDD, and other similar techniques [8, 102] are often described in other terms, they can all be thought of as using symmetric versions of nogoods to prune symmetric parts of the search space. We first need to define how constraints map under symmetries.

Definition II.3.1. Given a symmetry $\sigma$, we can extend $\sigma$ to map constraints to constraints as follows: given a constraint $c$, $\sigma(c)$ is a constraint over $V$ such that: $\theta$
satisfies \( \sigma(c) \) iff \( \sigma(\theta) \) satisfies \( c \).

Example II.3.2. Let \( P \equiv (V, D, C) \) where \( V \equiv \{x_1, x_2, x_3\} \). Let \( \sigma : \Theta_V \rightarrow \Theta_V \) be the symmetry which swaps \( x_1 \) and \( x_2 \). It is easy to see that for example: \( \sigma(x_1 = 1) \equiv x_2 = 1, \sigma(x_1 + 2x_2 + 3x_3) \equiv x_2 + 2x_1 + 3x_3 \), etc.

It is easy to see that the following identities hold:

- \( \sigma(a \land b) \Leftrightarrow \sigma(a) \land \sigma(b) \)
- \( \sigma(a \lor b) \Leftrightarrow \sigma(a) \lor \sigma(b) \)
- \( \sigma(\neg(a)) \Leftrightarrow \neg(\sigma(a)) \)
- \( a \Rightarrow b \) iff \( \sigma(a) \Rightarrow \sigma(b) \)

In this notation, we can express symmetries as follows:

Lemma II.3.3. Given a problem \( P \equiv (V, D, C) \), a bijective function \( \sigma : \Theta_V \rightarrow \Theta_V \) is a symmetry of \( P \) iff: \( C \land D \Leftrightarrow \sigma(C \land D) \).

Recall from Definition I.1.1 that a nogood of \( P \equiv (V, D, C) \) is a constraint such that: \( C \land D \Rightarrow n \). We have the following result:

Theorem II.3.4. Given a problem \( P \equiv (V, D, C) \) and a symmetry \( \sigma \) of \( P \), if \( n \) is a nogood of \( P \), then so is \( \sigma(n) \).

Proof. \( C \land D \Leftrightarrow \sigma(C \land D) \Rightarrow \sigma(n) \). So \( \sigma(n) \) is also a nogood of \( P \). □

Theorem II.3.4 tells us that if we derive any nogood \( n \), then all the symmetric versions of \( n \) are also valid and can be used to prune other parts of the search space.

Example II.3.5. Consider the simple problem \( P \equiv (V, D, C) \), \( V \equiv \{x_1, x_2, x_3\} \), \( D \equiv \{x_i \in \{1, \ldots, 5\} \mid i = 1, 2, 3\} \), \( C \equiv \{\text{alldiff}(x_1, x_2, x_3), x_1 + x_2 + x_3 \geq 10\} \). There is a variable symmetry on \( \{x_1, x_2, x_3\} \). The partial assignment \( x_1 = 1, x_2 = 2 \) fails, giving a nogood \( \neg(x_1 = 1 \land x_2 = 2) \). Any symmetric version of this nogood, e.g., \( \neg(x_2 = 1 \land x_1 = 2) \), \( \neg(x_2 = 1 \land x_3 = 2) \), are also valid nogoods.
SBDS, SBDD, and other similar techniques are all simply applications of Theorem II.3.4. They differ in what kind of nogoods $n$ are used, which $\sigma$ are applied, and how $\sigma(n)$ is derived and propagated. Both SBDS and SBDD use decision nogoods. When some subproblem $P' \equiv (V, D, C \cup \{d_1, \ldots, d_k\})$ has been searched and failed, the decision nogood is $n \equiv \neg(d_1 \land \ldots \land d_n)$. By Theorem II.3.4, for all $\sigma \in S$, $\sigma(n)$ is also a valid nogood for this problem.

SBDS proceeds as follows. Let $P''$ be the parent subproblem of $P'$. We check each of the $\sigma(n)$ to see if they are locally satisfied within $P''$. If not, then we post the nogood $\sigma(n)$ as a local constraint within $P''$. It is proved in [62] that this is sufficiently to break all symmetries completely. Posting $\sigma(n)$ only as a local constraint, even though it holds globally, is an optimization to reduce work. The decision nogood derived when $P''$ is exhaustively searched is $n' \equiv \neg(d_1 \land \ldots \land d_{k-1})$, and we have $\forall \sigma \in S, \sigma(n') \Rightarrow \sigma(n)$. Hence SBDS will post strictly stronger nogoods $\sigma(n')$ when $P''$ is finished, and so it is sufficient to post the $\sigma(n)$ locally within $P''$.

SBDD is similar, however, instead of posting $\sigma(n)$ as propagators, it stores only $n$ in a nogood store $N$. Later on, at each other subproblem $P'''$, the nogood store $N$ is examined to see if $\exists n \in N, \exists \sigma \in S$ s.t. $\sigma(n)$ can propagate in $P'''$. Another way to think of this is simply that SBDD implements all of $\sigma(n)$ as one giant global propagator whereas SBDS posts them all as separate propagators. SBDD has the advantage that the global propagator may have significantly better space and time complexity than posting the $\sigma(n)$ separately, but it also takes more work to implement.

II.4 Contributions

In Chapter 4, we describe an extension to SBDS called SBDS-1UIP, where we use symmetric versions of the 1UIP nogoods derived by LCG solvers instead of symmetric versions of decision nogoods. We show that this is significantly more powerful than the original SBDS method, allowing us to exploit kinds of redundancies that no previous general technique is capable of exploiting. In particular, SBDS-1UIP’s power is beyond complete, as it is capable of using the failure of a subproblem in one orbit to prune a subproblem in a different orbit, and thus does not have to examine at least
one valuation per orbit as previous techniques do. This extra pruning power can give us exponential speedups compared to the traditional symmetry breaking methods on appropriate problems.

In Chapter 5, we consider problems which do not exhibit full problem symmetry, but which exhibit *almost symmetries*, i.e., symmetries which are slightly broken. Such problems often still have large amounts of symmetries that can be exploited for speedup. However, traditional symmetry breaking methods like lexicographical symmetry breaking constraints, SDBS or SBDD cannot directly handle such partially broken symmetries. We present two new general techniques for exploiting almost symmetries. The first is to treat almost symmetries as conditional symmetries and exploit them via conditional symmetry breaking constraints. The second is to modify SDBS-1UIP to handle almost symmetries. Both techniques are capable of producing exponential speedups on appropriate problems.
Chapter 4

Symmetry Breaking During Search and Lazy Clause Generation
4.1 Introduction

As we described in Section II.3, Symmetry Breaking During Search (SBDS) [62] is a technique which utilizes symmetric versions of decision nogoods to prune symmetric parts of the search space. It is an application of Theorem II.3.4. However, Theorem II.3.4 actually applies to any kind of nogood, not just decision nogoods, and there is no reason why we should be restricted to using only decision nogoods. Boolean Satisfiability solvers [109] and Lazy Clause Generation solvers (LCG) [113, 45] are capable of deriving nogoods called 1UIP nogoods, which have empirically been found to produce significantly more pruning than decision nogoods in non-symmetric problems [165]. Clearly, it would be interesting to see if the extra pruning power of 1UIP nogoods carry over to the symmetric case as well. In this chapter, we show how to modify SBDS to use 1UIP nogoods. We call the original version SBDS-DEC, and the modified version SBDS-1UIP.

There are several issues which we have to deal with in order to implement SBDS-1UIP. SBDS-DEC only deals with simple decision nogoods involving equality literals on the decision variables. However, the 1UIP nogoods derived by LCG may have disequality and inequality literals as well, and may also have literals involving intermediate variables that were introduced by the solver. It is not immediately apparent how these more general nogoods map under the symmetries. Our contributions in this chapter are as follows:

- We show that SBDS can be extended to work with more general nogoods, i.e., nogoods with equality, disequality and inequality literals.
- We show how to handle intermediate variables introduced by the solver.
- We prove that SBDS-1UIP is at least as powerful as SBDS-DEC.
- We show that SBDS-1UIP can exploit symmetries that cannot be exploited by SBDS-DEC or by any other previous method.
4.2 SBDS-1UIP

Adapting SBDS to use 1UIP nogoods is simple. When a subproblem \( P' \) fails, the LCG solver will derive some 1UIP nogood \( n \equiv \neg(l_1 \land \ldots \land l_k) \). Theorem II.3.4 tells us that every symmetric version of this nogood \( \sigma(n) \equiv \neg(\sigma(l_1) \land \ldots \land \sigma(l_k)) \) is also valid.

Upon backtracking to the parent subproblem \( P'' \), we examine \( \sigma(n) \) for \( \sigma \in S \) where \( S \) is the symmetry group for the problem. If some term \( \sigma(l_i) \) within the nogood is locally false, then the symmetry \( \sigma \) is locally broken in \( P'' \) and we throw away \( \sigma(n) \).

If none of the \( \sigma(l_i) \) are false, then the symmetry \( \sigma \) still holds locally in \( P'' \), and we can post \( \sigma(n) \) as a constraint.

In contrast to SBDS-DEC, in SBDS-1UIP, it can be useful to post \( \sigma(n) \) as global rather than local constraints. In SBDS-DEC, the decision nogood derived for the parent subproblem \( P'' \) always subsumed the nogood derived for subproblem \( P' \) so we only had to post \( \sigma(n) \) locally. However, the 1UIP nogood at \( P'' \) does not generally subsume the one at \( P' \), so posting the \( \sigma(n) \) globally can potentially give additional pruning.

Example 4.2.1. Consider a simple problem \( P = (V, D, C) \), where \( V \equiv \{x_1, \ldots, x_5\} \), \( D \equiv \{x_i \in \{1, \ldots, 8\} | i = 1, \ldots, 8\} \), \( C \equiv \{\text{alldiff}([x_1, x_2, x_3, x_4, x_5]), \sum_{i=1}^{5} x_i \leq 12\} \). This problem has a variable symmetry between all 5 variables. Suppose we first try \( x_1 = 1 \). After propagation, we have \( x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_5 \geq 2 \). Suppose we now try \( x_2 = 2 \). The \text{alldiff} constraint forces \( x_3 \neq 2, x_4 \neq 2, x_5 \neq 2 \), and the domain constraints force \( x_3 \geq 3, x_4 \geq 3, x_5 \geq 3 \). The linear constraint then forces \( x_3 \leq 3, x_4 \leq 3, x_5 \leq 3 \), but then \( x_3, x_4, x_5 \) are all fixed to 3 and the \text{alldiff} fails.

On backtracking the LCG solver will infer the 1UIP nogood \( x_3 \geq 2 \land x_4 \geq 2 \land x_5 \geq 2 \rightarrow x_2 \neq 2 \), which immediately propagates to prune 2 from the domain of \( x_2 \). We can apply any of the variable symmetries to this nogood to derive another valid nogood. For example, swapping \( x_2 \) with \( x_3 \) gives us the nogood \( x_2 \geq 2 \land x_4 \geq 2 \land x_5 \geq 2 \rightarrow x_3 \neq 2 \). This symmetric nogood can immediately propagate to prune 2 from the domain of \( x_3 \) as well. Similarly, we can also infer \( x_4 \neq 2, x_5 \neq 2 \).

In the above simple example, the symmetric 1UIP nogoods perform the same pruning as the symmetric decision nogoods. However, we will show in Section 4.4...
that there are problems where they provide different amounts of pruning.

## 4.3 1UIP vs Decision Nogood

The power of SBDS-1UIP is difficult to analyze, because it depends on how strong the propagators are and how they explain their propagations. However, we can show that under some fairly reasonable assumptions about the solver’s propagation strength, SBDS-1UIP is at least as powerful as SBDS-DEC. The properties we need are **monotonicity**, and **propagation symmetry**. Let \( \text{prop}(C, D) \) characterize the propagation engine of our solver, i.e., \( \text{prop} \) takes the constraints \( C \) and the current domain \( D \) and returns a new domain \( D' \).

### Definition 4.3.1

The propagation engine \( \text{prop}(C, D) \) has **monotonicity** if: if \( D' \Rightarrow D \), then \( \text{prop}(C, D') \Rightarrow \text{prop}(C, D) \), i.e., given a smaller domain as input, it returns a domain at least as small.

### Definition 4.3.2

The propagation engine \( \text{prop}(C, D) \) has **propagation symmetry** w.r.t. symmetry group \( S \) if: \( \forall \sigma \in S, \sigma(\text{prop}(C, D)) \Leftrightarrow \text{prop}(C, \sigma(D)) \), i.e., given a symmetric domain as input, it returns a symmetric domain as output.

A sufficient condition for monotonicity is that all propagators are monotonic. A sufficient condition for propagation symmetry is that each symmetry maps propagators to symmetric propagators of the same type/strength. Global symmetric monotonicity is therefore very common, as most propagators are monotonic, and the vast majority of symmetries that are usually exploited are propagator symmetric.

### Theorem 4.3.3

Suppose monotonicity and propagation symmetry hold. Let \( n_{\text{dec}} \equiv \neg(d_1 \land \ldots \land d_k) \) be the decision nogood derived from the failure of the subproblem \( P' \equiv (V, D, C \cup \{d_1, \ldots, d_k\}) \). Let \( n_{\text{1uip}} \equiv \neg(l_1 \land \ldots \land l_m) \) be the decision nogood derived from the same subproblem \( P' \). Then for any \( \sigma \in S \), if a domain \( D' \) that is at fixpoint violates \( \sigma(n_{\text{dec}}) \), then it also violates \( \sigma(n_{\text{1uip}}) \).

**Proof.** Suppose we have a domain \( D' \) such that \( D' \) violates \( \sigma(n_{\text{dec}}) \), i.e., all the \( \sigma(d_i) \) are true in \( D' \). Recall that in the subproblem \( P' \), the decisions \( d_1, \ldots, d_k \) were enough
for the propagation engine to infer each of $l_1, \ldots, l_m$. By monotonicity and propagation symmetry, since $\sigma(d_i)$ currently all hold in $D'$, and $D'$ is at fixpoint, $\sigma(l_i)$ must all hold in $D'$. Hence, $D'$ also violates $\sigma(n_{1ui})$. □

Theorem 4.3.3 tells us that the symmetric versions of the 1UIP nogoods are at least as powerful as the symmetry versions of the decision nogoods in terms of their ability to fail subproblems. In the next section, we show that SBDS-1UIP can be strictly stronger than SBDS-DEC for some problems.

4.4 Beyond Complete Methods

SBDS-DEC is a complete method which guarantees that once we have visited a partial assignment, we will never visit any symmetric version of it. It may seem surprising that we can do better than this, but it is possible. Fundamentally, SBDS, SBDD and lexicographical constraints all use the failure of a valuation $\theta$ to infer the failure of its symmetric counterparts $\sigma(\theta)$. This means that they have to visit at least one member of each orbit. However, this is not true for SBDS-1UIP. The 1UIP nogoods used in SBDS-1UIP are powerful enough that we may not actually need to visit any member of some orbits.

Example 4.4.1. Consider the graph coloring problem shown in Figure 4.1, where we are trying to 5 color the graph. We represent the colors of the nodes with the variables $V \equiv \{x_1, \ldots, x_{10}\}$, and domain $D \equiv \{x_i \in \{1, \ldots, 5\} | i = 1, \ldots, 10\}$. For each edge in the graph between node $i$ and node $j$, we post the constraint $x_i \neq x_j$. Clearly, all 5 colors are interchangeable, so there is a value symmetry on all the variables in $V$.

Suppose we make the decisions $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5$. Propagation produces the domains shown in Figure 4.1. Suppose we try $x_6 = 1$ next. Then propagation gives $x_7 \in \{2, 3\}, x_8 \in \{2, 3\}, x_9 \in \{2, 3\}$. Now, suppose we try $x_7 = 2$. This forces $x_8 = 3, x_9 = 3$, which conflicts. The 1UIP nogood from this conflict is $x_8 \neq 1 \land x_8 \neq 4 \land x_8 \neq 5 \land x_9 \neq 1 \land x_9 \neq 4 \land x_9 \neq 5 \rightarrow x_7 \neq 2$. After propagating this nogood, we have $x_7 = 3$, which after further propagation, once again conflicts. At this point, we backtrack to before $x_6$ was labeled and derive the
nogood $x_7 \neq 4 \land x_7 \neq 5 \land x_8 \neq 4 \land x_8 \neq 5 \land x_9 \neq 4 \land x_9 \neq 5 \rightarrow x_6 \neq 1$, which immediately prunes 1 from the domain of $x_6$.

Now, let’s examine what SBDS-1UIP can do at this point. It is clear that if we swap the values 1 and 2 or swap the values 1 and 3 in this nogood, the LHS remains unchanged while the RHS changes. Therefore, we can post these two symmetric 1UIP nogoods and immediately get the inferences $x_6 \neq 2$ and $x_6 \neq 3$. On the other hand, SBDS-DEC cannot do anything. The decision nogood is $x_1 = 1 \land x_2 = 2 \land x_3 = 3 \land x_4 = 4 \land x_5 = 5 \rightarrow x_6 \neq 1$, and it is easy to see since all 5 values are used on the LHS, no matter which value symmetry we use on it, the LHS will have a set of literals incompatible with the current assignment and thus the symmetric nogoods cannot prune anything.

Let us consider whether any of the other symmetry breaking techniques can exploit this symmetry. SBDD is equal in power to SBDS, so it also cannot exploit this symmetry. Lexicographical symmetry breaking constraints cannot exploit this symmetry, since all 5 values have already been used in $x_1, \ldots, x_5$ and the constraint is already satisfied. Even conditional symmetry breaking constraints [61] cannot exploit this, as the subproblem does not have value symmetries between 1 and 2 or between

![Figure 4.1: A graph coloring problem where we can exploit additional symmetries.](image)
4.4. **BEYOND COMPLETE METHODS**

1 and 3 due to the domain of $x_{10}$.

We note something very interesting. The 1UIP nogood is derived after the failure of subproblem $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5, x_6 = 1$. The symmetric version of this 1UIP nogood can prune the subproblem $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5, x_6 = 2$. However, the partial assignments leading to these two subproblems do not actually lie in the same orbit! The valuations in the second subproblem were pruned despite the fact that we have never examined any of their symmetric versions before!

This rather surprising fact is due to the fact that 1UIP nogoods are much more powerful than decision nogoods. Decision nogoods are derived through exhaustive search, and can only prune valuations that we have already examined. Therefore, the symmetric decision nogoods can only prune something for which we have already examined a symmetric version. However, 1UIP nogoods are derived through constraint resolution, and generally prune more valuations than were contained in the failed subproblem from which it was derived. Since 1UIP nogoods can prune valuations that we have never examined before, symmetric versions of 1UIP nogoods can prune valuations for which we have never examined a symmetric version before. Hence, SBDS-1UIP can be strictly stronger than complete methods like SBDS-DEC, SBDD or lexicographical symmetry breaking constraints.

1UIP nogoods are constructed by finding a sufficient set of conditions for the conflict to reoccur, such that the conditions are as “local” to the conflict as possible. This enables SBDS-1UIP to exploit symmetry that exists in the sub-component of a subproblem which is the actual cause of failure. In the subproblem above, the sub-component that caused the failure are the variables $x_6, x_7, x_8, x_9$, their current domains in the subproblem, and the constraints linking them. The fact that $x_1, \ldots, x_5$ were labeled to values that broke all the symmetries globally did not matter to SBDS-1UIP. It is capable of realizing that symmetries still held locally within $x_6, x_7, x_8, x_9$, and can be exploited for additional pruning. It is because LCG solvers give us such precise information about which variables are involved in conflicts that we can exploit this kind of symmetry. We show in our experiments in Section 4.6 that these additional kinds of redundancies can be exploited for more speedup.
4.5 Applying Symmetries to 1UIP Nogoods

SBDS-1UIP is much more powerful than SBDS-DEC. However, having to manipulate 1UIP nogoods raises several problems. In particular, unlike decision nogoods which only involve equality literals on the decision variables, 1UIP nogoods can contain virtually any literal in the problem, i.e., they may include disequality literals, inequality literals, and also literals involving intermediate variables which are introduced by the solver. Intermediate variables can be a rather serious problem, as symmetries are often defined only in terms of the decision variables, and may not properly describe how intermediate variables map to each other.

The original SBDS-DEC method [62] is only capable of handling symmetries which map equality literals to equality literals, i.e., \( \sigma \) such that for any variable \( x \) and value \( v \), \( \sigma(x = v) \Leftrightarrow x' = v' \) for some variable \( x' \) and value \( v' \). This includes most commonly exploited symmetries such as variable symmetries, value symmetries, variable sequence symmetries, value sequence symmetries and the simpler kinds of variable/value symmetries. We extend the SBDS-1UIP method to handle these kinds of symmetries only. We assume that the symmetry declaration gives us enough information to know how equality literals map to each other. We now show how disequality literals, inequality literals and literals on intermediate variables can be handled.

4.5.1 Disequality and Inequality literals

One of the strengths of LCG is the use of disequality and inequality literals in addition to equality literals in explanations and nogoods. This makes many explanations and nogoods much shorter and is essential for explaining bounds propagation. LCG solvers derive 1UIP nogoods of form \( n \equiv \neg(l_1 \land \ldots \land l_k) \), where each \( l_i \) is an equality, disequality or inequality literal. In order for the symmetric 1UIP nogood \( \sigma(n) \equiv \neg(\sigma(l_1) \land \ldots \land \sigma(l_k)) \) to be of a form that an LCG solver can handle, we need to express each of the \( \sigma(l_k) \) as the conjunction of one or more equality, disequality or inequality literals.

Disequality literals can be handled simply by applying negation to equality literals. If \( \sigma(x = v) \Leftrightarrow x' = v' \), then it follows that \( \sigma(x \neq v) \Leftrightarrow x' \neq v' \). Inequality literals
are harder to handle. They map simply under variable symmetries but not so simply under value symmetries. For example, let \( \sigma \) be the variable symmetry where we swap \( x_1 \) and \( x_2 \). Clearly, \( \sigma(x_1 \geq d) \equiv x_2 \geq d, \sigma(x_2 \geq d) \equiv x_1 \geq d, \sigma(x_1 \leq d) \equiv x_2 \leq d, \sigma(x_2 \leq d) \equiv x_1 \leq d. \) However, suppose \( \sigma \) is the value symmetry on \( x_1, \ldots, x_k \) where we swap the values \( v_1 \) and \( v_2 \). Without loss of generality, assume \( v_1 < v_2 \). If \( v > v_2 \) or \( v \leq v_1 \), then \( x_i \geq v \) simply maps to itself and there is no problem. But if \( v_1 < v \leq v_2 \), then \( \sigma(x_i \geq v) \) is not an inequality literal as it does not allow a contiguous range of values (it allows \( x_i = v_1 \) and \( x_i \geq v \), but disallows \( x_i = v_2 \)).

We can handle this case by converting \( x_i \geq v \) into a conjunction of disequality literals before applying the symmetry. If \( l \) is a lower bound for \( x_i \)'s original domain, then \( x_i \geq v \Leftrightarrow x_i \neq l \land \ldots \land x_i \neq v - 1, \) and \( \sigma(x_i \geq v) \equiv \sigma(x_i \neq l) \land \ldots \land \sigma(x_i \neq v - 1), \) and we already know how to map disequality literals.

**Example 4.5.1.** Suppose \( x \) has an original domain of \( \{1, 2, 3, 4, 5\} \). If \( \sigma \) is a symmetry swapping the values 2 and 4, then \( \sigma(x \geq 3) \Leftrightarrow \sigma(x \neq 1) \land \sigma(x \neq 2) \Leftrightarrow x \neq 1 \land x \neq 4. \)

Each inequality literal may potentially map to quite a large number of disequality literals. This can significantly increase the length of \( \sigma(n) \) compared to \( n \). However, the impact on performance is not great. Nogoods in LCG solvers are propagated using the two watched literal scheme commonly used in SAT solvers. The cost of this scheme is not linear in the length of the nogood, and increasing the length of the nogood typically does not necessarily increase its propagation cost by much. The only real cost then is the memory to store the longer nogood, but memory is rarely a problem on modern computers.

### 4.5.2 Intermediate variables

An important problem for manipulating 1UIP nogoods is the fact that intermediate variables may be introduced in the course of converting a high level model to the low level variables and constraints implemented by the solver. For example, a high level model written in the modeling language MiniZinc is first flattened into primitive constraints, with intermediate variables introduced as necessary, and then given to the solver, which may then introduce its own variables, e.g., in global propagators.
100

CHAPTER 4. SBDS AND LCG

implemented by decomposition. 1UIP nogoods often contain literals from such intermediate variables. However, if the symmetry declaration was made only in the high
level model, it may not specify how literals on such introduced intermediate variables
map to each other, and the solver will be unable to derive the symmetric versions of
the 1UIP nogoods.
There are two main ways to handle this. Ideally, the user manually extends the
symmetry declaration to include any intermediate variables introduced by flattening
or by the solver. This is the best solution, but requires the user to do work. Alternatively, we can modify the solver’s conflict analysis algorithm so that any literal which
is not covered by the symmetry declaration cannot appear in the nogoods it derives.
Example 4.5.2. Consider the Concert Hall problem. In the Concert Hall problem, we
have k identical halls, and n orders for renting halls. Each order has a start time,
end time and a price. We can choose to accept each order or not. If we accept the
order, we must schedule it in one of the k halls. Two accepted orders which overlap
in time cannot be scheduled in the same hall. The aim is to maximize the profit we
can make. A MiniZinc model for the Concert Hall problem is as follows:
int: n;

% number of offers

int: k;

% number of halls

set of int: Offers = 1..n;
array [Offers] of int: start;
array [Offers] of int: end;
array [Offers] of int: price;
% does the pair of offers overlap
array [Offers,Offers] of bool: o = array2d(Offers, Offers,
[start[i] <= end[j] /\ start[j] <= end[i] | i in Offers, j in Offers]);
array [Offers] of var 1..k+1: x;

% hall to take, k+1 means reject offer

var int: total;

% total money we can make

constraint forall (i, j in Offers where i < j /\ o[i,j])
(x[i] = k+1 \/ x[j] = k+1 \/ x[i] != x[j]);


4.5. APPLYING SYMMETRIES TO 1UIP NOGOODS

constraint sum (i in Offers) 
  (bool2int(! (x[i] = k+1)) * price[i]) >= total;

solve maximise total;

The problem has a value symmetry on all the $x[i]$ variables between values 1, \ldots, k (but not $k+1$). Also, if order $i$ and order $j$ has the same start time, end time and price, then there is a variable symmetry between $x[i]$ and $x[j]$.

When this model is flattened into primitive constraints, some intermediate variables will be introduced. The expressions: $x[i] = k + 1$, will be given corresponding variables:

array [Offers] of var bool: t;
constraint forall (i in Offers) (t[i] <-> x[i] = k+1);

The expressions: $x[i] \neq x[j]$, will be given corresponding variables:

array [Offers,Offers] of var bool: r;
constraint forall (i, j in Offers where i < j /\ o[i,j])
  (r[i,j] <-> x[i] != x[j]);

The expression: $\text{bool2int}(! (x[i] = k + 1))$, will be given corresponding variables:

array [Offers] of var 0..1: b;
constraint forall (i in Offers) (b[i] = bool2int(!t[i]));

Suppose we want to extend our symmetry declarations to cover these intermediate variables. Consider the value symmetry between values 1, \ldots, k. Clearly, the $t[i]$ are idempotent under these symmetries, since the value $k+1$ is unaffected. The $r[i,j]$ are also idempotent under these symmetries although it is less obvious. The $b[i]$ which only depend on the $t[i]$ are similarly idempotent under these symmetries. So in this case, we did not really need to do much work to extend the symmetry. Consider a variable symmetry between $x[i]$ and $x[j]$. Clearly, if we swap $x[i]$ with $x[j]$, then we must also swap $t[i]$ with $t[j]$, and $b[i]$ with $b[j]$. Also, $r[i,k]$ must be swapped with $r[j,k]$ for each $k \neq i,j$. So we have to extend the symmetry declaration to specify these additional mappings.
If getting the user to extend the symmetry declaration is impractical, then we use the second method where we modify the solver to prevent any literals on intermediate variables from appearing in nogoods. We first declare to the solver the set of variables which are actually covered by the symmetry declarations. All other variables are considered intermediate variables. We require that search only make decisions on the variables covered by the symmetry declaration. As described in Chapter 2, the conflict analysis algorithm works as follows. We start with the conflicting nogood. While the nogood has at least two literals from the last decision level, we eliminate the one inferred last by resolving the nogood with the explanation for that literal. We modify this algorithm by adding an additional step. If the nogood contains any literal from an intermediate variable, we also eliminate it by resolving it with the explanation for that literal. This is guaranteed to be possible, because no decisions are made on intermediate variables, so any literal on an intermediate variable cannot be the decision literal and therefore must have an explanation. The conflict analysis will now terminate with a nogood which is free of any literals on the intermediate variables. Since all the literals in the nogood are from variables covered by the symmetry declarations, we know how to calculate the symmetric versions of the nogood.

Example 4.5.3. Suppose the unmodified conflict analysis algorithm derived the nogood \( n \equiv \neg(b[1] = 1 \land x[2] = 1 \land x[3] = 2). \) This nogood has literals on the intermediate variable \( b[1], \) so we cannot map it under the symmetries. With the modified conflict analysis algorithm, we would attempt to eliminate the literals on \( b[1] \) as well. The explanation for \( b[1] = 1 \) is: \( t[1] = false \rightarrow b[1] = 1. \) The explanation for \( t[1] = false \) is: \( x[1] \neq k + 1 \rightarrow t[1] = false, \) so the nogood turns into \( n \equiv \neg(x[1] \neq k + 1 \land x[2] = 1 \land x[3] = 2). \) Now, all literals are covered by the symmetry declaration and we can apply any of our symmetries to it.

4.6 Experiments

We now provide experimental evidence that SBDS-1UIP can be much stronger than SBDS-DEC on some problems, and that symmetry breaking can work well with lazy clause generation. The two problems we will examine are the Concert Hall Scheduling
problem and the Graph Coloring problem. We take the benchmarks used in [102].

We implemented SBDS in Chuffed, which is a state of the art lazy clause solver. We run Chuffed with three different versions of SBDS. The first version is \texttt{dec}, where we use symmetric versions of decision nogoods. The second is \texttt{1UIP}, where we use symmetric versions of 1UIP nogoods. The third version we call \texttt{crippled}, where we use symmetric versions of 1UIP nogoods, but only those nogoods derived from symmetries where \texttt{dec} could also exploit the symmetry. We compare against Chuffed with no symmetry breaking (none) and with standard lexicographical symmetry breaking constraints (static). Finally, we compare against an implementation of SBDS in [102], which is called Lightweight Dynamic Symmetry Breaking (LDSB). LDSB is implemented on the Eclipse constraint programming platform and was the fastest implementations of dynamic symmetry breaking on the two problems we examine, beating GAP-SBDS [58] and GAP-SBDD [59] by significant margins.

All versions of Chuffed are run on Xeon Pro 2.4GHz processors. The results for LDSB were run on a Core i7 920 2.67 GHz processor. We group the instances by size, so that the times displayed are the average run times for the instances of each size. A timeout of 600 seconds was used. Instances which timeout are counted as 600 seconds.

The results are shown in Tables 4.1 and 4.2. Eclipse LDSB fails to solve many instances before timeout, and \texttt{dec} fails to solve a few instances. \texttt{1UIP}, \texttt{crippled} and \texttt{static} all solve every instance in the benchmarks. In fact, this set of instances, which is of an appropriate size for normal CP solvers, is a bit too easy for lazy clause solvers such as Chuffed, as is apparent from the run times.

Comparison between \texttt{dec} and \texttt{1UIP} shows that SBDS-1UIP is superior to SBDS-DEC. Comparison between \texttt{crippled} and \texttt{1UIP} shows that the additional symmetries that we can only exploit with SBDS-1UIP indeed gives us reduced search and additional speedup. Comparison with \texttt{static} shows that dynamic symmetry breaking can be superior to static symmetry breaking on appropriate problems. The comparison with LDSB shows that lazy clause solvers can be much faster than normal CP solvers,
Table 4.1: Comparison of three SBDS implementations in CHUFFED, static symmetry breaking in CHUFFED, and LDSB in Eclipse, on the Concert Hall Scheduling problem

<table>
<thead>
<tr>
<th>Size</th>
<th>none</th>
<th>1UIP</th>
<th>crippled</th>
<th>dec</th>
<th>static</th>
<th>LDSB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Fails</td>
<td>Time</td>
<td>Fails</td>
<td>Time</td>
<td>Fails</td>
</tr>
<tr>
<td>20</td>
<td>259.8</td>
<td>686018</td>
<td>0.04</td>
<td>84</td>
<td>0.05</td>
<td>130</td>
</tr>
<tr>
<td>22</td>
<td>381.5</td>
<td>749462</td>
<td>0.07</td>
<td>181</td>
<td>0.08</td>
<td>299</td>
</tr>
<tr>
<td>24</td>
<td>576.9</td>
<td>1438509</td>
<td>0.10</td>
<td>275</td>
<td>0.11</td>
<td>316</td>
</tr>
<tr>
<td>26</td>
<td>483.4</td>
<td>1189930</td>
<td>0.10</td>
<td>282</td>
<td>0.19</td>
<td>677</td>
</tr>
<tr>
<td>28</td>
<td>530.7</td>
<td>1282797</td>
<td>0.68</td>
<td>1611</td>
<td>1.12</td>
<td>2613</td>
</tr>
<tr>
<td>30</td>
<td>581.3</td>
<td>1251980</td>
<td>0.27</td>
<td>761</td>
<td>0.53</td>
<td>2042</td>
</tr>
<tr>
<td>32</td>
<td>542.4</td>
<td>936019</td>
<td>0.40</td>
<td>1522</td>
<td>1.01</td>
<td>4845</td>
</tr>
<tr>
<td>34</td>
<td>600.0</td>
<td>1039051</td>
<td>1.10</td>
<td>2636</td>
<td>3.22</td>
<td>8761</td>
</tr>
<tr>
<td>36</td>
<td>600.0</td>
<td>1223864</td>
<td>1.40</td>
<td>3156</td>
<td>5.02</td>
<td>13606</td>
</tr>
<tr>
<td>38</td>
<td>600.0</td>
<td>1027778</td>
<td>1.91</td>
<td>5053</td>
<td>12.56</td>
<td>26556</td>
</tr>
<tr>
<td>40</td>
<td>600.0</td>
<td>1447604</td>
<td>2.96</td>
<td>6648</td>
<td>10.27</td>
<td>27028</td>
</tr>
</tbody>
</table>

and that they retain this advantage when integrated with symmetry breaking methods. It also show by proxy that SBDS-1UIP is superior to GAP-SBDS or GAP-SBDD on these problems.

The total speed difference between 1UIP and dec is up to 2 orders of magnitude for the Concert Hall problems and up to 4 orders of magnitude for the Graph Coloring problems. Most of this speedup can be explained by the dramatic reduction in search space, which is apparent from the node counts in the results table. The redundancies exploited by lazy clause solvers are different from those redundancies caused by symmetries, and it is very clear here that by exploiting both at the same time with SBDS-1UIP, we get much higher speedups than possible with either of them separately.

It should also be noted that CHUFFED with static symmetry breaking constraints is also reasonably fast. While symmetry breaking constraints cannot exploit the extra redundancies that SBDS-1UIP can, it does have very low overhead and integrates well with lazy clause. Not every problem with symmetry has extra redundancies that SBDS-1UIP can exploit. Often the amount of symmetry actually exploited by the different techniques will be similar, thus SBDS-1UIP will not always gain any additional speedup compared to other techniques.
4.7 Conclusion

In this chapter, we showed how SBDS can be modified to use the 1UIP nogoods derived by LCG solvers instead of the usual decision nogoods. We proved that SBDS-1UIP is at least as strong as SBDS-DEC, and showed that it can be strictly stronger on some problems. In particular, it is capable of exploiting symmetries in the subcomponent of a subproblem which actually caused the failure. This kind of redundancy is very difficult to exploit and we are not aware of any other general symmetry breaking method that can exploit it. To implement SBDS-1UIP, we had to solve a number of new issues, such as how to map disequality literals, inequality literals, and literals from intermediate variables under the symmetries. We demonstrated experimentally
that SBDS-1UIP can be orders of magnitude faster than the original SBDS on appropriate problems.
Chapter 5

Exploiting Almost Symmetries
CHAPTER 5. EXPLOITING ALMOST SYMMETRIES

5.1 Introduction

While symmetries occur frequently in constraint problems, it is more common that constraint problems arising from real world situations contain almost symmetries, i.e., symmetries which are broken by a small number of constraints. Such almost symmetries are not well behaved mathematically, and it is difficult to adapt the traditional symmetry breaking techniques like lexicographical constraints, SBDS or SBDD to exploit them.

Almost symmetries are closely related to conditional symmetries [61]. In fact, they can be considered two sides of the same coin. There are two kinds of slightly broken symmetry: 1) ones where the symmetry can be restored by adding a small number of constraints, and 2) ones where the symmetry can be restored by removing a small number of constraints. The first kind is called a conditional symmetry. The second kind is called an almost symmetry. Although conditional symmetries and almost symmetries are conceptually very similar, the methods for exploiting them are significantly different.

Conditional symmetries are relatively well understood and can be exploited very effectively using conditional symmetry breaking constraints [61]. Almost symmetries on the other hand, are not well studied and are much harder to exploit. Previous work on almost symmetries have concentrated on exploiting them by first transforming the original problem \( P \) into a related problem \( P' \) where the symmetries are restored, and then applying the traditional symmetry breaking techniques to \( P' \) [38, 65, 67, 101]. If constraints are removed to restore the symmetry, then the problem \( P' \) is less constrained than \( P \) and may have a larger search space. A solution of \( P' \) may or may not be a solution of \( P \) and extra work has to be done to deal with such situations.

We present two new and general ways to exploit almost symmetries. Both methods work natively on \( P \) without needing to restore the symmetry. The first method is based on treating almost symmetries as conditional symmetries. We show how to systematically post conditional symmetry breaking constraints to exploit the almost symmetries and discuss their correctness and completeness. The second method is based on modifying SBDS-1UIP, described in the previous chapter, to directly handle
almost symmetries. Both techniques are capable of producing exponential speedups on appropriate problems.

## 5.2 Definitions

An almost symmetry is a symmetry which holds over most of the constraints in $C$, but is broken by some small subset $H \subseteq C$. Intuitively, if $H$ is “small” relative to $C$, then we expect a lot of symmetry left to exploit. Whereas if $H$ is “large” or equal to $C$, then there is very little symmetry left to exploit. More formally:

**Definition 5.2.1.** An almost symmetry of a problem $P \equiv (V, D, C)$ is a bijective function $\sigma : \Theta_V \rightarrow \Theta_V$ mapping valuations to valuations such that: $\exists H \subseteq C$ s.t. $\sigma$ is a symmetry of problem $P' \equiv (V, D, C \setminus H)$, where $H$ is “small” relative to $C$. We call $H$ a breaking set of $\sigma$ w.r.t. $P$.

**Definition 5.2.2.** An almost symmetry group of problem $P \equiv (V, D, C)$ is a set $S$ of almost symmetries of $P$ such that: $\exists H \subseteq C$, s.t. $S$ is a symmetry group of $P' \equiv (V, D, C \setminus H)$, where $H$ is “small” relative to $C$. We call $H$ a breaking set of $S$ w.r.t. $P$.

**Example 5.2.3.** Consider a simple problem $P$ where $C \equiv x_1 + x_2 + x_3 \geq 3 \land x_1 \neq 1 \land x_3 \neq 2$. We can consider each of the pairwise variable swaps as almost symmetries. The mapping swapping $x_1$ and $x_2$ is an almost symmetry with breaking set $H \equiv \{x_1 \neq 1\}$. The mapping swapping $x_1$ and $x_3$ is an almost symmetry with $H \equiv \{x_1 \neq 1, x_3 \neq 2\}$. The mapping swapping $x_2$ and $x_3$ is an almost symmetry with $H \equiv \{x_3 \neq 2\}$. The set of all variable permutations forms an almost symmetry group with $H \equiv \{x_1 \neq 1, x_3 \neq 2\}$.

Breaking sets are not necessarily unique. However, there is always a unique, **minimal** breaking set.

**Definition 5.2.4.** A breaking set $H$ is minimal if no proper subset of it is a breaking set.
Lemma 5.2.5. Each almost symmetry \( \sigma \) or almost symmetry group \( S \) of \( P \equiv (V, D, C) \) has a unique minimal breaking set.

Proof. We prove that the breaking set property is closed under set intersection. Let \( H_1, H_2 \) be breaking sets for almost symmetry \( \sigma \). Let \( H_3 \equiv H_1 \cap H_2 \). Let \( C_i \equiv C \setminus H_i \) for \( i = 1, 2, 3 \). Then \( C_3 \equiv C_1 \cap C_2 \). Since \( H_1 \) and \( H_2 \) are breaking sets, we have \( C_1 \wedge D \Leftrightarrow \sigma(C_1 \wedge D) \) and \( C_2 \wedge D \Leftrightarrow \sigma(C_2 \wedge D) \). Then: \( (C_3 \wedge D) \Leftrightarrow (C_1 \wedge D) \wedge (C_2 \wedge D) \Leftrightarrow \sigma(C_1 \wedge D) \wedge \sigma(C_2 \wedge D) \Leftrightarrow \sigma(C_3 \wedge D) \). So \( H_3 \) is also a breaking set of \( \sigma \). Since the breaking set property is closed under set intersection, there exists a unique minimal breaking set for \( \sigma \). The proof for almost symmetry group \( S \) is analogous. \( \square \)

We denote this unique minimal breaking set by \( H(\sigma) \) and \( H(S) \) for almost symmetry \( \sigma \) and almost symmetry group \( S \) respectively. Symmetries are a special case of almost symmetries where \( H(\sigma) \equiv \emptyset \) or \( H(S) \equiv \emptyset \). All results we derive below apply to symmetries, almost symmetries, or any combination of them in the same problem.

5.3 Almost Symmetry as Conditional Symmetry

Conditional symmetry breaking constraints were first introduced in [61]. The idea is that when certain additional conditions (constraints) are satisfied, the problem may gain additional symmetries which can be exploited.

Definition 5.3.1. A conditional symmetry of problem \( P \equiv (V, D, C) \) is a bijective function \( \sigma : \Theta_V \to \Theta_V \) mapping valuations to valuations such that: \( \exists G, \text{ s.t. } \sigma \) is a symmetry of problem \( P' \equiv (V, D, C \cup G) \).

Definition 5.3.2. A conditional symmetry group of problem \( P \equiv (V, D, C) \), is a set \( S \) of bijective mappings such that: \( \exists G(S), \text{ s.t. } S \) is a symmetry group of \( P' \equiv (V, D, C \cup G(S)) \).

Example 5.3.3. Consider a simple problem \( P \) where \( C \equiv x_1 + x_2 + x_3 \geq 3 \land x_1 \neq 1 \land x_2 \neq 1 \). The mapping swapping \( x_1 \) and \( x_2 \) is an unconditional symmetry. The mapping swapping \( x_1 \) and \( x_3 \) is a conditional symmetry with \( G \equiv \{x_3 \neq 1\} \). The mapping swapping \( x_2 \) and \( x_3 \) is a conditional symmetry with \( G \equiv \{x_3 \neq 1\} \). The set of all variable permutations forms a conditional symmetry group with \( H \equiv \{x_3 \neq 1\} \).
Note that the definition of a conditional symmetry and almost symmetry is not mutually exclusive. It’s entirely possible that a mapping $\sigma$ can have its symmetry restored by both adding some constraints and removing some constraints.

**Theorem 5.3.4.** Given a conditional symmetry group $S$, and any lexicographical ordering $<_{lex}$, it is correct to post the conditional symmetry breaking constraint: $\text{csb}(S, <_{lex}) \equiv G(S) \rightarrow \text{sb}(S, <_{lex})$, where $\text{sb}(S, <_{lex})$ is the normal lexicographical symmetry breaking constraint for $S$.

**Proof.** We prove that $\text{csb}(S, <_{lex})$ cannot prune all solutions. Suppose $\theta \in \text{solns}(P)$ and $\theta$ is pruned by $\text{csb}(S, <_{lex})$. We show that the lex-least valuation $\theta'$ in the orbit of $\theta$ under $S$ is also a solution of $P$ and will not be pruned. Let $P' \equiv (V, D, C \cup G(S))$. Since $\theta$ was pruned by $\text{csb}(S, <_{lex})$, it must satisfy $G(S)$. So $\theta \in \text{solns}(P')$. Since $S$ is a symmetry group of $P'$, $\theta' \in \text{solns}(P')$, which means $\theta' \in \text{solns}(P)$. Also, $\theta'$ is lex-least within its orbit, so it satisfies $\text{sb}(S, <_{lex})$ and cannot be pruned by $\text{csb}(S, <_{lex})$. □

### 5.3.1 Breaking a Single Almost Symmetry

We can exploit almost symmetries by considering them as conditional symmetries.

**Lemma 5.3.5.** Given an almost symmetry group $S$ with breaking set $H(S)$, $S$ can also be considered a conditional symmetry group with $G(S) \equiv \{\sigma(c) \mid c \in H(S), \sigma \in S\}$.

**Proof.** We need to show that $S$ is a symmetry group of $P' \equiv (V, D, C \cup G(S))$. This is true iff: $\forall c \in C \cup G(S), \forall \sigma \in S, C \cup G(S) \Rightarrow \sigma(c)$. We have two cases. Suppose $c \in C \setminus H(S)$. By the definition of an almost symmetry group, $\forall \sigma \in S, C \setminus H(S) \Rightarrow \sigma(c)$, so certainly $\forall \sigma \in S, C \cup G(S) \Rightarrow \sigma(c)$. Suppose $c \notin C \setminus H(S)$, then $c$ must be in $H(S)$ or $G(S)$. However, $H(S) \subseteq G(S)$ by construction, so $c \in G(S)$. Also, $\forall \sigma \in S, \sigma(c) \in G(S)$ since $S$ is a group, so certainly $\forall \sigma \in S, C \cup G(S) \Rightarrow \sigma(c)$. □

**Corollary 5.3.6.** Given an almost symmetry group $S$, it is correct to post the conditional symmetry breaking constraint: $\text{csb}(S, <_{lex}) \equiv \land_{c \in H(S)} \land_{\sigma \in S} \sigma(c) \rightarrow \text{sb}(S, <_{lex})$. 
This csb($S, <_{\text{lex}}$) can be implemented as $|H(S)| \times |S|$ reified constraints, one reified lexicographical symmetry breaking constraint, and a single clausal constraint to link them all. Note that $H(S)$ is generally different for different almost symmetry subgroups $S$, and we can post a conditional symmetry breaking constraint for each almost symmetry subgroup we wish to exploit.

There are several things we can do to simplify/strengthen this constraint. csb($S, <_{\text{lex}}$) can be rearranged into the form of a nogood: $\neg(a_1 \land \ldots \land a_k)$, where the $a_i$ are either $\sigma(c)$ for some $\sigma \in S, c \in H(S)$, or is $\neg sb(S, <_{\text{lex}}$). We can simplify the nogood in the following ways. If $C \implies \neg a_i$ for some $a_i$, then the conditional symmetry breaking constraint is redundant and we can throw it away. If $\exists j \text{ s.t. } C \land \land_{i \neq j} a_i \implies a_j$, then we can simplify the clause by removing $a_j$ from it. Note that there are some terms we can always simplify away. The identity map $I$ is always in $S$ and $c \implies I(c)$ for any $c \in C$, so we will remove these terms by default.

**Example 5.3.7.** Recall the Black Hole Problem, which we described in Example 3.3.1 and gave a MiniZinc model for in Section 3.6.1. We restate the problem for convenience. We seek to find a solution to the Black Hole patience game. In this game the 52 playing cards (of standard 52-card deck) are laid out in 17 piles of 3, with the ace of spades starting in a “black hole”. Each turn, a card at the top of one of the piles can be played into the black hole if it is $\pm 1$ from the card played previously, with king wrapping back around to ace. The aim is to play all 52 cards.

There is an almost symmetry between cards of the same number and different suits. Consider $2\spadesuit$ and $2\heartsuit$. Swapping these two cards ($2\spadesuit \leftrightarrow 2\heartsuit$) induces a variable swap between $x[2\spadesuit]$ and $x[2\heartsuit]$, and a value swap between $2\spadesuit$ and $2\heartsuit$ on all the $y$. It can be seen that this is a symmetry of the inverse and $\pm 1$ constraints, but is broken by some of the card precedence constraints. Thus we have an almost symmetry group $S = \{I, 2\spadesuit \leftrightarrow 2\heartsuit\}$.

In an abuse of notation, we will use $2\spadesuit$ to denote $x[2\spadesuit]$, etc. Suppose that in the initial layout, we had piles $5\diamondsuit < 2\spadesuit < 7\heartsuit$ and $3\spadesuit < 2\heartsuit < 9\spadesuit$. Then $H(S) \equiv \{5\diamondsuit < 2\spadesuit, 2\spadesuit < 7\heartsuit, 3\spadesuit < 2\heartsuit, 2\heartsuit < 9\spadesuit\}$ are the only constraints breaking the symmetry. We can construct a conditional symmetry breaking constraint according to Corollary 5.3.6. Suppose we have a lexicographical ordering $<_{\text{lex}}$ constructed from
the variable ordering $2\spadesuit < v 2\heartsuit < v 2\clubsuit < v 2\diamondsuit$. Then $sb(S, <_{\text{lex}}) \equiv 2\spadesuit < 2\heartsuit$. We now work out $\sigma(c)$ for each $\sigma \in S$, $c \in H(S)$. Recall that all terms with $\sigma = I$ can be simplified away. For $\sigma \equiv 2\spadesuit \leftrightarrow 2\heartsuit$, $\sigma(5\spadesuit < 2\spadesuit) = 5\spadesuit < 2\heartsuit$, and $\sigma(2\spadesuit < 7\heartsuit) = 2\heartsuit < 7\heartsuit$, $\sigma(3\diamondsuit < 2\heartsuit) = 3\diamondsuit < 2\heartsuit$, and $\sigma(2\heartsuit < 9\heartsuit) = 2\heartsuit < 9\heartsuit$. So $csb(S, <_{\text{lex}}) \equiv 5\spadesuit < 2\heartsuit \land 2\heartsuit < 7\heartsuit \land 3\diamondsuit < 2\heartsuit \land 2\spadesuit < 9\heartsuit \rightarrow 2\spadesuit < 2\heartsuit$ is a valid conditional symmetry breaking constraint. We can simplify this further by noting that: $3\diamondsuit < 2\heartsuit \land 2\heartsuit < 2\spadesuit \rightarrow 3\diamondsuit < 2\spadesuit$, and $2\heartsuit < 2\spadesuit \land 2\spadesuit < 7\heartsuit \rightarrow 2\heartsuit < 7\heartsuit$, which means we can remove the second and third condition in the conditional symmetry breaking constraint, giving us: $csb(S, <_{\text{lex}}) \equiv 5\spadesuit < 2\heartsuit \land 2\spadesuit < 9\heartsuit \rightarrow 2\spadesuit < 2\heartsuit$.

The conditional symmetry breaking constraints we derive here for the Black Hole Problem are identical to those proposed in [61]. However, those authors constructed conditional symmetry breaking constraints for specific problems in an ad-hoc and non-generalizable way, whereas here we describe a systematic way to exploit any almost symmetry with conditional symmetry breaking constraints.

### 5.3.2 Breaking Multiple Almost Symmetries

It is not always correct to post lexicographical symmetry breaking constraints for multiple symmetries, as they may not be compatible and may end up pruning away all solutions. A sufficient condition for them to be compatible in the full symmetry case is for all of the lexicographical symmetry breaking constraints to use the same lexicographical ordering $<_{\text{lex}}$. We show that a similar condition holds for conditional symmetry breaking constraints.

**Theorem 5.3.8.** Given constraint problem $P$ with conditional symmetry groups $S_1, \ldots, S_k$ and a lexicographical ordering $<_{\text{lex}}$, it is correct to post $csb(S_i, <_{\text{lex}})$ for each $S_i$.

**Proof.** Suppose $\theta_0$ is a solution of $P$ which is pruned by one of the $csb(S_i)$, say $csb(S_{k_0})$. Then $\theta_0 \notin \text{solns}(sb(S_{k_0}))$, so $\exists \sigma \in S_{k_0}$ s.t. $\sigma(\theta_0)$ is the lex-least solution in the orbit of $\theta_0$ under $S_{k_0}$. Let $\theta_1 = \sigma(\theta_0)$. We have that $\theta_1 \in \text{solns}(sb(S_{k_0}))$, $\theta_1 \in \text{solns}(C)$ and that $\theta_1 <_{\text{lex}} \theta_0$. If $\theta_1$ still violates one of the $csb(S_i)$, say $csb(S_{k_1})$, then we do
the same thing again to get another $\theta_2 = \sigma(\theta_1)$, where $\theta_2 \in \text{solns}(C)$, $\theta_2 <_{\text{lex}} \theta_1$. As long as $\theta_j$ violates some $\text{csb}(S_{k_j})$, we can always find another $\theta_{j+1}$ which is a solution of $P$, and which is strictly lex smaller. Since the sequence $\theta_1, \theta_2, \ldots$ is monotonically decreasing w.r.t. $<_{\text{lex}}$, and there are only finitely many solutions, it must terminate in a $\theta_j$ which is a solution of $P$ and satisfies all of the $\text{csb}(S_i)$. $\square$

From this point on, we will assume that every lexicographical symmetry breaking constraint and conditional symmetry breaking constraint uses the same lexicographical ordering $<_{\text{lex}}$. We will drop the $<_{\text{lex}}$ arguments and write them more simply as $sb(S)$ and $csb(S)$ respectively.

### 5.3.3 Completeness of Conditional Symmetry Breaking Constraints

Recall that a lexicographical symmetry breaking constraint for whole problem symmetry group $S$ is complete iff it prunes all but one solution within each orbit. The situation for almost symmetries is much more complicated. In an almost symmetry group $S$, the elements of $S$ are not whole problem symmetries. Therefore the orbit of a solution $\theta$ under $S$ may contain non-solutions as well as solutions. Such “mixed” orbits are not well behaved mathematically. It is possible to extend the definition of completeness to almost symmetry groups $S$, i.e., if some orbit $O$ has solutions, then exactly one of these solutions is not pruned. However, this definition of completeness is so strong that achieving completeness probably requires exponential time, largely nullifying any benefit we might gain from symmetry breaking. Instead, we seek incomplete methods which are efficient enough to give us speedup in practice. We show that conditional lexicographical symmetry breaking constraints can achieve a weaker form of completeness which we call pure orbit completeness.

**Definition 5.3.9.** Given an almost symmetry group $S$ for problem $P \equiv (V, D, C)$, a set of almost symmetry breaking constraints $B$ is pure orbit complete w.r.t. $S$ if:

Given any solution $\theta \in \text{solns}(P)$, if $\exists S' \subseteq S$ s.t. 1) $\forall \sigma \in S'$, $\sigma(\theta) \in \text{solns}(P)$, and 2) $\exists \sigma \in S'$ s.t. $\sigma(\theta) <_{\text{lex}} \theta$, then $\theta \notin \text{solns}(B)$, else $\theta \in \text{solns}(B)$. 
5.3. ALMOST SYMMETRY AS CONDITIONAL SYMMETRY

Pure orbit completeness requires a subgroup \( S' \subseteq S \) under which \( \theta \)'s orbit is pure (are all solutions) before we prune \( \theta \) with \( B \). It reduces to the normal definition of completeness if \( S \) is a whole problem symmetry group (with \( S' = S \)).

**Lemma 5.3.10.** Given almost symmetry group \( S \) for problem \( P \equiv (V, D, C) \), if \( sb(S) \) is complete w.r.t to \( S \), then \( csb(S) \) prunes a solution \( \theta \in solns(P) \) iff \( \forall \sigma \in S, \ \sigma(\theta) \in solns(P) \), and \( \exists \sigma \in S \) s.t. \( \sigma(\theta) \prec_{lex} \theta \).

*Proof.* Firstly, \( \forall \sigma \in S, \ \sigma(\theta) \in solns(C \land D) \Leftrightarrow \forall \sigma \in S, \ \theta \in solns(\sigma(C \land D)) \Leftrightarrow \theta \in solns(C \land D) \land \forall \sigma \in S, \ \forall c \in H(S), \ \theta \in solns(\sigma(c)) \), so \( \theta \) satisfies the conditional part of \( csb(S) \) iff the first condition holds. Secondly, \( \exists \sigma \in S \) s.t. \( \sigma(\theta) \prec_{lex} \theta \) iff \( \theta \notin sb(S) \), so \( \theta \) is pruned by the RHS of \( csb(S) \) iff the second condition holds. \( \square \)

**Theorem 5.3.11.** Given almost symmetry group \( S \) for problem \( P \equiv (V, D, C) \), if for every subgroup \( S_i \subseteq S \), \( sb(S_i) \) is complete w.r.t. \( S_i \), then \( B \equiv \{ csb(S_i) \mid S_i \subseteq S \} \) is pure orbit complete w.r.t. \( S \), and no stronger.

*Proof.* Given any solution \( \theta \in solns(P) \), if \( \exists S' \subseteq S \) s.t. \( \forall \sigma \in S', \ \sigma(\theta) \in solns(P) \), and \( \exists \sigma \in S' \) s.t. \( \sigma(\theta) \prec_{lex} \theta \), then by Lemma 5.3.10, \( \theta \) will be pruned by \( csb(S') \). Conversely, if no such \( S' \) exists, then by Lemma 5.3.10, \( \theta \) cannot be pruned by any of the \( csb(S') \). \( \square \)

Theorem 5.3.11 tells us that the maximum power of this method is exactly pure orbit completeness. Posting \( csb(S_i) \) for every almost symmetry subgroup \( S_i \subseteq S \) will achieve this. However, there are in general a super-exponential number of such subgroups. We consider whether we can achieve pure orbit completeness by posting only a subset of the \( csb(S_i) \).

**Theorem 5.3.12.** Given an almost symmetry group \( S \), if \( S \) has subgroups \( S_1, \ldots, S_k \) such that \( \land_{i=1}^k sb(S_i) \Rightarrow sb(S) \), then \( \land_{i=1}^k csb(S_i) \Rightarrow csb(S) \).

*Proof.* Since \( S_i \subseteq S \), we must have \( H(S_i) \subseteq H(S) \) for each \( S_i \).

\[
\land_{i=1}^k (\land_{c \in H(S_i)} \land_{\sigma \in S_i} \sigma(c) \rightarrow sb(S_i)) \\
\Rightarrow \land_{i=1}^k (\land_{c \in H(S)} \land_{\sigma \in S} \sigma(c) \rightarrow sb(S_i)) \\
\Leftrightarrow \land_{c \in H(S)} \land_{\sigma \in S} \sigma(c) \rightarrow \land_{i=1}^k sb(S_i) \\
\Rightarrow \land_{c \in H(S)} \land_{\sigma \in S} \sigma(c) \rightarrow sb(S) \]

\( \square \)
Theorem 5.3.12 tells us that some of the \( \text{csb}(S_i) \) are unnecessary, because they are subsumed by the \( \text{csb}(S_i) \) of their subgroups. Thus in order to achieve pure orbit completeness w.r.t. \( S \), it is often sufficient to post \( \text{csb}(S_i) \) for all simple (primitive) symmetry subgroups \( S_i \subseteq S \), since the \( \text{csb}(S_i) \) of composite symmetry subgroups are often subsumed by the \( \text{csb}(S_i) \) of its simple subgroups. However, the number of simple subgroups is still typically exponential.

**Example 5.3.13.** Consider the Black Hole Problem from above. Let \( S_1, S_2, S_3 \) be the symmetry groups consisting of permutations of \{2\(\heartsuit\), 2\(\heartsuit\), 2\(\clubsuit\)\}, \{2\(\heartsuit\), 2\(\heartsuit\)\} and \{2\(\heartsuit\), 2\(\heartsuit\)\} respectively. Let \( <_{\text{lex}} \) use the variable ordering: 2\(\heartsuit\), 2\(\heartsuit\), 2\(\clubsuit\). Then Theorem 5.3.12 tells us that \( \text{csb}(S_1) \) is subsumed by \( \text{csb}(S_2) \wedge \text{csb}(S_3) \).

We try to find an even smaller subset of \( \text{csb}(S_i) \) which is sufficient for pure orbit completeness.

**Theorem 5.3.14.** Given variable almost symmetry group \( S \equiv x_1 \equiv \ldots \equiv x_n \), the set of conditional symmetry breaking constraints \( B \equiv \bigwedge_{i<j} \text{csb}(x_i \equiv x_j) \) is pure orbit complete w.r.t. \( S \) if: \( \forall i < j, \forall c \in H(x_i \equiv x_j) \), \( |\text{vars}(c) \cap \text{vars}(S)| = 1 \).

**Proof.** Let \( S' \subseteq S \) be any subgroup of \( S \). Let \( \text{orb}(x_i) \equiv \{\sigma(x_i) \mid \sigma \in S' \} \) be the orbit of \( x_i \) generated by \( S' \). By the properties of a group, if \( x_j \in \text{orb}(x_i) \) then \( \text{orb}(x_i) = \text{orb}(x_j) \). The orbits partition \( \{x_1, \ldots, x_n\} \) into disjoint sets. Let \( \{x_1, \ldots, x_n\} = \text{orb}_{i_1} \cup \ldots \cup \text{orb}_{i_k} \). Suppose \( S_{i,j} \equiv x_i \equiv x_j \). We show that \( \bigwedge_{m=1}^k \bigwedge_{x_i,x_j \in \text{orb}_{i_m},x_i \neq x_j} \text{sb}(S_{i,j}) \Rightarrow \text{sb}(S') \), because any solution to the LHS cannot be improved lexicographically by permutations of variables within each \( \text{orb}_{i_m} \).

Firstly, \( \bigwedge_{m=1}^k \bigwedge_{x_i,x_j \in \text{orb}_{i_m},x_i \neq x_j} \text{sb}(S_{i,j}) \Rightarrow \text{sb}(S') \), because any solution to the LHS cannot be improved lexicographically by permutations of variables within each \( \text{orb}_{i_m} \).

Secondly, we show that the conditions for \( \text{csb}(S_{i,j}) \) is a subset of those for \( \text{csb}(S') \). Consider \( c \in H(S_{i,j}) \). By assumption, \( |\text{vars}(c) \cap \text{vars}(S)| = 1 \). Without loss of generality, suppose \( \text{vars}(c) \cap \text{vars}(S) = \{x_i\} \). Since \( x_j \in \text{orb}(x_i) \), \( \exists \sigma \in S' \) s.t. \( \sigma(x_i) = x_j \). Suppose \( S_{i,j} \equiv \{I, \sigma'\} \). Then \( \sigma'(c) \equiv \sigma(c) \). This is because they both map \( x_i \) to \( x_j \), and map every other variable in \( c \) maps to themselves since \( \text{vars}(c) \cap \text{vars}(S) = \{x_i\} \). Now, \( c \in H(S_{i,j}) \), so \( C \not\Rightarrow \sigma'(c) \equiv \sigma(c) \), hence \( c \in H(S') \).

Thus \( \forall c \in H(S_{i,j}) \), \( \forall \sigma \in S_{i,j} \), \( \sigma(c) \) is also a condition in \( \text{csb}(S) \). This, combined with the first result, gives \( \bigwedge_{m=1}^k \bigwedge_{x_i,x_j \in \text{orb}_{i_m},x_i \neq x_j} \text{csb}(S_{i,j}) \Rightarrow \text{csb}(S') \). \( \square \)
Theorem 5.3.14 tells us that under certain circumstances, posting $csb(S_i)$ for all pairwise variable symmetries is sufficient for pure orbit completeness. Similar theorems exist for value symmetries, variable sequence symmetries and value sequence symmetries, as well as for $S$ where $S$ is generated by multiple symmetry groups of the above four types.

Example 5.3.15. Consider the Black Hole Problem from above. Consider the symmetry group $S$ generated by $(2\heartsuit, 2\spadesuit, 2\diamondsuit) \leftrightarrow (2\spadesuit, 2\diamondsuit, 2\heartsuit)$, that is simultaneously replacing $2\heartsuit$ by $2\spadesuit$, $2\spadesuit$ by $2\diamondsuit$, and $2\diamondsuit$ by $2\heartsuit$. This group is simple and Theorem 5.3.12 does not apply. However, if $2\spadesuit, 2\diamondsuit, 2\heartsuit$ are in different piles, then the constraints in $H(2\diamondsuit \Rightarrow 2\heartsuit), H(2\diamondsuit \Rightarrow 2\spadesuit), H(2\spadesuit \Rightarrow 2\diamondsuit)$ each involve only one variable in the set $\{2\spadesuit, 2\diamondsuit, 2\heartsuit\}$. Thus Theorem 5.3.14 applies and tells us that $csb(S)$ is subsumed by $csb(2\spadesuit \Rightarrow 2\diamondsuit) \land csb(2\diamondsuit \Rightarrow 2\heartsuit) \land csb(2\heartsuit \Rightarrow 2\spadesuit)$.

We propose posting only $csb(S)$ for pairwise swaps within each variable, value, variable sequence or value sequence almost symmetry. This is incomplete in general, but in view of Theorem 5.3.14, this may be pure orbit complete or near pure orbit complete for a variety of problems. It also has the benefit that there are typically only a polynomial number of such subgroups, and since $|S|$ is small, they have far fewer conditions than those for larger $S$ and are much more likely to prune something during search.

5.4 Breaking Almost Symmetries with SBDS-1UIP

In the previous chapter, we described a more powerful version of Symmetry Breaking During Search (SBDS) called SBDS-1UIP. Instead of using symmetric versions of the decision nogoods as traditional SBDS does, SBDS-1UIP uses symmetric versions of the 1UIP nogoods derived by Lazy Clause Generation solvers. This allows for the exploitation of new types of symmetries. SBDS-1UIP can be modified to handle almost symmetries relatively easily. In an LCG solver, each nogood $n$ is derived from a certain subset of the constraints $m(n) \subseteq C$, i.e., $m(n) \Rightarrow n$. 
Theorem 5.4.1. Given almost symmetry $\sigma$ and nogood $n$ of problem $P \equiv (V, D, C)$, if $m(n)$ and $H(\sigma)$ are disjoint, then $\sigma(n)$ is a valid nogood.

Proof. $(C \setminus H(\sigma)) \iff \sigma(C \setminus H(\sigma)) \implies \sigma(m(n)) \implies \sigma(n) \square$

Theorem 5.4.1 tells us that in order to figure out whether we can apply almost symmetry $\sigma$ to nogood $n$ to get another valid nogood, we simply need to know $H(\sigma)$ and $m(n)$. $m(n)$ is easy to keep track of. In LCG, new nogoods $n$ are either generated by a constraint $c$, in which case $m(n) = \{c\}$, or by resolving two nogoods $n_1$ and $n_2$ together, in which case $m(n) = m(n_1) \cup m(n_2)$. We use a bit-vector for each nogood to keep track of its $m(n)$. Set union is then implemented as a bit operation. Note that we only need a bit for constraint $c$ if $c \in H(S_i)$ for some $S_i$ we are exploiting. Typically $|H(S_i)|$ is small, as otherwise, the symmetry is too broken for almost symmetry techniques to gain anything anyway. We use the variant of SBDS where we post $\sigma(n)$ iff it can immediately propagate. This is incomplete in general, but is a good trade off between speed and pruning. Completeness for almost symmetries is most likely of exponential complexity in general and is not practical.

Example 5.4.2. Consider the graph coloring problem in Fig 5.1 where we want to 4 color the graph. Normally, if there are two vertices such that they have the exact
same set of neighbors, then they are symmetric (a variable symmetry). However, often the set of neighbors are not identical, but only similar. Consider the variables $x_1$ and $x_2$. They have almost the same set of neighbors, so we can exploit it as an almost symmetry $\sigma$ where $\sigma$ swaps $x_1$ and $x_2$, and $H(\sigma) \equiv \{x_1 \neq x_3, x_2 \neq x_4\}$.

Suppose we try $x_5 = 1, x_1 = 2$. Propagation forces $x_6, x_7, x_8 \in \{3, 4\}$ which eventually fails and gives us the 1UIP nogood: $x_6 \neq 1 \land x_7 \neq 1 \land x_8 \neq 1 \rightarrow x_1 \neq 2$. The set of constraints used to derive this nogood is: $m(n) \equiv \{x_1 \neq x_6, x_1 \neq x_7, x_1 \neq x_8, x_5 \neq x_6, x_5 \neq x_7, x_5 \neq x_8, x_6 \neq x_7, x_6 \neq x_8, x_7 \neq x_8\}$. Clearly, $m(n) \cap H(\sigma) \equiv \emptyset$, thus Theorem 5.4.1 applies and the symmetric nogood $x_6 \neq 1 \land x_7 \neq 1 \land x_8 \neq 1 \rightarrow x_2 \neq 2$ is also valid, allowing us to immediately prune 2 from $x_2$’s domain. It is easy to see that $csb(\sigma) \equiv x_1 \neq x_4 \land x_2 \neq x_3 \rightarrow x_1 \leq x_2$ would not have made this inference, since the conditions do not hold.

5.5 Related Work

Although there have been several theoretical works on exploiting almost symmetries [65, 67], we are only aware of one working implementation of a general method for exploiting almost symmetries [101]. We describe this method in detail, and compare our new methods against it in the experiments.

One way to exploit almost symmetries is to decompose the problem into two parts $P_1$ and $P_2$, where $P_1$ contains $C \setminus H(S)$ and is fully symmetric, and $P_2$ contains $H(S)$. Full symmetry breaking can be applied to $P_1$. When a solution of $P_1$ is found, we try to extend each of its symmetric versions to a solution of $P_2$. In [101], the authors propose to implement this by remodeling the problem. They add to the model a “sym var”, whose domain consists of the elements of $S$. Labeling the sym var chooses which symmetric version of the solution of $P_1$ to try to extend to $P_2$.

More formally, given the global almost symmetry group $S$ for problem $P \equiv (V, D, C)$, we modify $P$ to $P'$ as follows. Let $V_B = vars(C \setminus H(S)), V_H = vars(H(S))$ and $V_1 = V_B \setminus V_H, V_2 = V_B \cap V_H$ and $V_3 = V_H \setminus V_B$. We add a set of shadow variables $V'_2$ which is simply a copy of each variable in $V_2$. We modify the constraints as follows. For each $c$ in $H(S)$, we replace any occurrence of a variable in $V_2$ with
its shadow copy in $V_2'$. At this point, we have fully decomposed the problem into two parts, $P_1 \equiv (V_1 \cup V_2, D_{V_1 \cup V_2}, C \setminus H(S))$ and $P_2 \equiv (V_2' \cup V_3, D_{V_2' \cup V_3}, H(S))$. $S$ is a full symmetry group on $P_1$ and we can use whatever symmetry breaking approach we desire for $S$ on $P_1$.

We now channel $P_1$ and $P_2$ into one model as follows. We add a sym var $sv$, whose domain is $S$. We post a channeling constraint $Q$ over $V_1 \cup V_2 \cup sv \cup V_2'$ such that: $\theta \in solns(Q)$ iff $\theta(sv) = \sigma \in S$, then $\theta_{V_2'} = \sigma(\theta_{V_1 \cup V_2})_{V_2}$, i.e., the channeling constraint forces $V_2'$ to take on the values that $V_2$ would take if we had applied $\sigma$ to this solution of $P_1$.

The search proceeds as follows. We first label all the variables in $P_1$. If we succeed, we then label the sym var $sv$, after which the channeling constraint $Q$ forces $V_2'$ to take on the symmetric values of $V_2$. Then we label the remaining variables in $P_2$. If we succeed, then the following holds: if $\theta \in solns(P')$ and $\theta_{SV} = \sigma$, then $\sigma(\theta_{V_1 \cup V_2}) \cup \theta_{V_3} \in solns(P)$.

**Example 5.5.1.** Consider a simple graph coloring problem with constraints: $C \equiv \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1, x_1 \neq 1, x_2 \neq 2\}$. Consider the almost symmetry group $S \equiv x_1 \leftrightarrow x_2 \leftrightarrow x_3$ (saying all three variables are interchangeable). We have $H(S) \equiv \{x_1 \neq 1, x_2 \neq 2\}$. We can construct $P'$ as follows. We add shadow variables $x_1', x_2'$. We modify $C$ to: $C' \equiv \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1, x_1' \neq 1, x_2' \neq 2\}$. We add lexicographical symmetry breaking constraint: $x_1 \leq x_2 \leq x_3$. We decompose the sym var $sv$ into a set of variables and constraints $sv_1, sv_2, sv_3 \in \{1, 2, 3\}$, $\text{alldiff}([sv_1, sv_2, sv_3])$ so that each solution to $sv_i$ corresponds to a permutation of the variables $x_1, x_2, x_3$. We add the channeling constraint $Q \equiv x_1' = x_{sv_1} \land x_2' = x_{sv_2}$. Consider a solution to $P'$, e.g., $x_1 = 1, x_2 = 2, x_3 = 3, sv_1 = 3, sv_2 = 1, sv_3 = 2, x_1' = 3, x_2' = 1$. Then the corresponding solution to $P$ is $x_1 = 3, x_2 = 1, x_3 = 2$ (by applying the permutation selected by the $sv_i$).

### 5.6 Experiments

We compare five methods: 1) ignore all almost symmetries, but exploit full symmetries if any exist using standard lexicographical symmetry breaking constraints (**none**),
2) use conditional symmetry breaking constraints to exploit both symmetries and almost symmetries (cond-sym), 3) use SBDS-1UIP to exploit both types of symmetries (SBDS-1UIP), 4+5) use the decomposition/remodeling method as described above to exploit both types of symmetries using either 4) lexicographic symmetry breaking (remodel-lex) or 5) SBDS-1UIP (remodel-sbds). We use the following four problems (all models and instances are available at www.csse.unimelb.edu.au/~pjs/almostsym/).

Black Hole Problem As discussed in Example 5.3.7, the Black hole Problem has a number of variable almost symmetries. We use 100 random instances.

Graph Coloring As discussed in Example 5.4.2, almost variable symmetries arise when pairs of vertices have almost the same set of neighbors. We construct instances as follows. We first randomly partition 50 vertices into 8 sets. For each pair of sets $S_i, S_j$, we decide with 0.5 probability whether they are densely connected or sparsely connected. If dense, then for each pair of vertices $v_1 \in S_i, v_2 \in S_j$, we add an edge between them with probability 0.99. If sparse, then for each pair of vertices $v_1 \in S_i, v_2 \in S_j$, we add an edge between them with probability 0.01. The end result is that pairs of vertices in the same set $v_1, v_2 \in S_i$ typically have very similar sets of neighbors, with the number of difference varying between about 0 to 3. Roughly 40% of the variable pairs are symmetric (0 difference) while the rest are almost symmetric (> 0 differences), thus the problem contains a mixture of variable symmetries and variable almost symmetries. There is also the standard value symmetry in the problem.

Quasi-Group Completion The Quasi-Group Completion Problem [64] is an instance of the Latin Squares Problem with side constraints of form $x_i = v$. While QCP typically has no symmetries, the Latin Square Problem has many row/column symmetries and value symmetries. We can model these underlying symmetries in QCP as almost symmetries. We try 100 instances of $n = 8$ with 25% of the squares prefilled and find all solutions of each instance.

Concert Hall Problem The Concert Hall Problem [92] normally has value symmetries (representing that all $k$ concert halls are identical). However, more realistically,
Table 5.1: Comparison between ignoring almost symmetries (none), exploiting almost symmetries with conditional symmetry (cond-sym), SBDS-1UIP (sbds-1uip), decomposition using lex (remodel-lex) and decomposition using SBDS-1UIP (remodel-sbds)

<table>
<thead>
<tr>
<th>Problem</th>
<th>none</th>
<th>cond-sym</th>
<th>sbds-1uip</th>
<th>remodel-lex</th>
<th>remodel-sbds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Fails</td>
<td>Time</td>
<td>Fails</td>
<td>Time</td>
</tr>
<tr>
<td>BlackHole</td>
<td>45.8</td>
<td>50427</td>
<td>1.4</td>
<td>4488</td>
<td>21.8</td>
</tr>
<tr>
<td>GraphColor</td>
<td>35.2</td>
<td>73133</td>
<td>28.2</td>
<td>80921</td>
<td>0.5</td>
</tr>
<tr>
<td>QCP</td>
<td>252.5</td>
<td>4392785</td>
<td>-</td>
<td>-</td>
<td>17.7</td>
</tr>
<tr>
<td>ConcertHall</td>
<td>257.1</td>
<td>600190</td>
<td>43.23</td>
<td>86786</td>
<td>224.7</td>
</tr>
</tbody>
</table>

halls may vary in size, equipment, location, etc. As a result, it is possible that for each concert only a certain subset of the halls are acceptable, corresponding to side constraints of form $x_i \neq v$. This creates almost value symmetries and almost variable symmetries. We use randomly generated instances following [92], with an additional 50 random side constraints of form $x_i \neq v$.

We use the state of the art solver CHUFFED, where SBDS-1UIP has previously been implemented. We modify this implementation to support almost symmetries as described in Section 5.4. To exploit almost symmetries using conditional symmetry breaking constraints, we post the additional constraints $csb(S)$ for each pairwise swap in each variable, value, variable sequence or value sequence symmetry, as proposed in Section 5.3. CHUFFED was run on Xeon Pro 2.4GHz processors with a timeout of 300 seconds. Table 5.1 shows average fails and times on each set of instances, with timeouts counting as 300 seconds.

It can be seen from Table 5.1 that all three methods can produce significant speedups on at least some of the problems compared to the baseline of none. SBDS-1UIP is fastest on Graph Coloring and QCP, while remodel-lex is fastest on Black Hole and remodel-sbds is fastest on Concert Hall. The different techniques are effective on different problems. Their effectiveness appears to be very sensitive to the structure of the problem and the types of constraints in them, so that one method that performs well on one problem may perform very poorly on a different problem.

Conditional symmetry breaking constraints tend to be effective if the partial assignments often happen to restore the broken symmetry, i.e., satisfy the symmetric
versions of $H(S)$. It also depends on how high in the search tree the condition becomes satisfied. We get good speedup for Black Hole and Concert Hall, and much less for Graph Coloring. For QCP, the symmetric versions of $H(S)$ for pairwise row swaps, pairwise column swaps, or pairwise value swaps never hold, meaning that during simplification, we would have thrown away all the $csb(S)$ as being redundant, so the method does nothing.

SBDS-1UIP tends to be effective if the $H(S)$ are not often directly involved in the conflict, e.g., because it is a weak constraint (Graph Coloring), or because it becomes satisfied by the partial assignment, or because it has already propagated at a higher decision level (QCP). This is the only method that could get any speedup on the QCP.

The decomposition method is effective if $H(S)$ is weak, so that not too much information is lost in the relaxation $C \setminus H(S)$. This is true for Black Hole and Concert Hall, thus we get very good speedups. On the other hand, if $H(S)$ contains strong constraints that prune many solutions of $C \setminus H(S)$, then the decomposition severely weakens the model, potentially significantly increasing the run time (Graph Coloring, QCP).

5.7 Conclusion

We have defined almost symmetries, and presented two new general techniques for exploiting them. We described how to systematically post conditional symmetry breaking constraints to exploit almost symmetries, and proved several theorems concerning their correctness and completeness. We also described how SBDS-1UIP can be modified to exploit almost symmetries. Our experimental evaluation showed that both techniques can give significant speedup on appropriate problems.
Part III

Relaxation and Dominance
III.1 Introduction to Relaxations

There are a variety of techniques in combinatorial optimization based on problem relaxations. Relaxations transform a problem $P$ into a related but easier problem $P'$, such that every solution in $P$ corresponds to a solution in $P'$. There are two main ways in which we might want to use relaxations.

In the first situation, we have some problem $P$ which is over-constrained or is simply too hard. We may be willing to settle for some approximate solution of $P$ that is “good enough” according to some criteria. We therefore relax the problem $P$ into an easier to satisfy problem $P'$ and solve $P'$ instead. For example, some of the constraints arising in real life problems may not absolutely have to be satisfied, but may merely express preferences of various strengths. We may be willing to violate some of the less important ones in order to get a solution, rather than to have no solution at all. Techniques like soft constraints [16, 49], valued constraints, etc, can be used to handle such problems.

In the second situation, we have a problem $P$ which we wish to solve exactly. We try to find a relaxed problem $P'$ such that $P'$ can be solved very quickly, and such that solving $P'$ gives us useful information on how to solve $P$. Such information may include candidate solutions, nogoods, bounds, etc. Techniques of this kind include linear relaxations [137, 156], Lagrangian relaxations [46, 111, 95], and other problem specific relaxations (e.g., [30, 28]). We describe such techniques in further detail below.

III.2 Definitions

We state the definitions and theorems for optimization problems only. The corresponding results for satisfaction problems can be found by setting all objective functions $f$ to $f(\theta) = 0$ for all $\theta$. The types of relaxation we are interested in can be defined as follows:

**Definition III.2.1.** Given a problem $P \equiv (V, D, C, f)$ where $f$ is to be minimized, a problem $P' \equiv (V', D', C', f')$ is a relaxation of $P$ under function $\phi : \Theta_V \rightarrow \Theta_{V'}$ if:
∀θ ∈ solns(P), φ(θ) ∈ solns(P') and f'(φ(θ)) ≤ f(θ), i.e., φ maps each solution of P to a solution of P' that has an objective value at least as good.

The following theorem follows trivially from the above definition.

**Theorem III.2.2.** If P' is a relaxation of P, then the following hold:

- If P' has no solutions, then P has no solutions
- If b is a lower bound on the objective value of P', then b is a lower bound on the objective value of P

We describe a relaxation as **tight** or **loose** depending on how different P' is compared to P. For satisfaction problems, a relaxation P' of P is **tight** if for most solutions θ' of P', there exists a solution θ of P such that φ(θ) = θ', i.e., the relaxation does not introduce many extra solutions. Otherwise, we say that it is **loose**. For optimization problems, a relaxation P' of P is **tight** if \( \min\{f(\theta) | \theta \in \text{solns}(P)\} - \min\{f'(\theta) | \theta \in \text{solns}(P')\} \) is small, i.e., the relaxation does not change the optimal value by much. Otherwise, we say that it is **loose**.

The tighter the relaxation is, the more useful the nogoods and bounds derived in P' will be. Also, if the relaxation is tight, then the solutions of P' are very good candidates for solutions of P, and we may be able to find solutions to P much faster by solving P' instead. Much of the difficulty in applying relaxations is to find relaxed problems P' which are tight, but which can also be solved very quickly.

**III.3 Types of Relaxations**

**III.3.1 Simple relaxations**

The easiest way to relax a problem is to simply weaken or remove some of its constraints. Given a problem \( P \equiv (V, D, C, f) \), we can relax it by replacing C with \( C' \) where \( C \Rightarrow C' \), or by replacing D with \( D' \) where \( D \Rightarrow D' \). We can also relax a problem by removing one or more of its variables and projecting the constraints onto the remaining variables \( U \subset V \). That is, we define \( P' \equiv (U, D_U, C', f') \) where
III.3. TYPES OF RELAXATIONS

$C' \equiv \{ \bar{c} \cup c | c \in C \}$, $f'(\theta) = \min\{f(\theta \cup \theta')|\theta' \in \Theta_{V \setminus U}\}$ and $\phi(\theta) = \theta_U$. Clearly, if $\theta$ is a solution of $P$, then $\theta_U$ is a solution of $P'$, and by construction, $f'(\theta_U) \leq f(\theta)$.

In Chapter 8, we use this type of relaxation to exponentially speedup the optimality proofs for the Maximum Density Still Life Problem.

### III.3.2 Lagrangian relaxations

A somewhat more complicated way to relax a problem is the Lagrangian relaxation [46, 111, 95]. In Lagrangian relaxations, we relax the problem $P \equiv (V, D, C, f)$ by moving one or more of the constraints into the objective function as “penalty” terms describing whether the constraint is violated. For each constraint $c$ that we want to move, we define a penalty function $r_c : \Theta_V \to \mathbb{R}$ mapping valuations to the reals such that if $\theta$ satisfies $c$, then $r_c(\theta) = 0$ (no penalty). If $G$ is the set of constraints to be moved, then the relaxed problem is $P' \equiv (V, D, C \setminus G, f')$, where $f' = f + \sum_{c \in G} r_c$, and $\phi$ is the identity function. Clearly, if $\theta$ is a solution of $P$, then it is also a solution of $P'$, and by definition, $\forall c \in G, r_c(\theta) = 0$, so $f(\theta) = f'(\theta)$.

Lagrangian relaxations can be useful when the problem $P$ has a set of “core” constraints which are actually quite easy to solve, but $P$ has been made harder by the presence of some “side” constraints. By moving these side constraints into the objective, we recover the easy to solve core, which can be quickly solved to give us a bound on the objective for the original problem. Lagrangian relaxations have been used effectively in many problems (e.g., [41, 70, 140, 12]).

### III.3.3 Linear relaxations

One of the most useful relaxations is the linear relaxation [137, 156]. The problem is relaxed into a Linear Programming (LP) problem, which can then be solved very efficiently using LP methods like the simplex method [110]. In a linear relaxation, we relax the problem $P \equiv (V, D, C)$ by removing the integrality constraints on the variables and linearizing all the constraints. That is, each integer variable $v \in V$ is replaced by a continuous variable $v'$ with the same upper and lower bounds, and each constraint $c \in C$ is replaced by a set of linear constraint $\{l_1, \ldots, l_m\}$ such that $c \Rightarrow l_i$
for each $l_i$.

There are many ways to utilize linear relaxations. For example, Mixed Integer Programming (MIP) solvers [34, 137] use linear relaxations at every subproblem in the search tree to derive bounds which may fail the subproblem. Various LP/CP hybrids have been proposed in order to take advantage of both the strength of CP propagation and the bounds provided by the linear relaxations [129, 69, 80]. Linear relaxations can also be used by individual propagators to make more powerful inferences or to make inferences faster, e.g., in global propagators for bin-packing [21], cumulative [75], all-different [73, 154], element [74, 148] and others.
III.4 Introduction to Dominance

Dominance is a powerful concept that underlies many of the major techniques discussed so far. Symmetry breaking, relaxations, and many forms of nogood learning are applications of dominance. The basic idea behind dominance is to find pairs of objects (valuations, subproblems, problems) where one is better than (or at least as good as) the other with respect to what we are trying to optimize, e.g., one satisfies the constraints while the other does not, one has a better objective value than the other. Finding and exploiting such dominance relations [87, 78, 2, 76] can give us exponential speedups on appropriate problems. In this section, we define dominance relations and describe how they can be exploited.

We state the definitions and theorems for optimization problems only. The corresponding results for satisfaction problems can be found by setting all objective functions $f$ to $f(\theta) = 0$ for all $\theta$. We first define dominance relations over valuations.

**Definition III.4.1.** Given a problem $P \equiv (V, D, C, f)$ where $f$ is to be minimized, a dominance relation $\preceq$ over $\Theta_V$ is a reflexive, transitive binary relation on $\Theta_V$ such that: if $\theta_1 \preceq \theta_2$ and $\theta_2 \in \text{solns}(P)$, then $\theta_1 \in \text{solns}(P)$ and $f(\theta_1) \leq f(\theta_2)$.

**Definition III.4.2.** If $\theta_1 \preceq \theta_2$, but $\theta_2 \not\preceq \theta_1$, then we write $\theta_1 \prec \theta_2$. If $\theta_1 \preceq \theta_2$, we say that $\theta_1$ dominates $\theta_2$ and that $\theta_2$ is dominated by $\theta_1$, where “dominates” means “at least as good as”. If $\theta_1 \prec \theta_2$, we say that $\theta_1$ strictly dominates $\theta_2$, where “strictly dominates” means “strictly better than”. There are two main ways to exploit a dominance relation, depending on whether a dominance is strict or not. We have the following two theorems.

**Theorem III.4.3.** Given a problem $P \equiv (V, D, C, f)$, and a dominance relation $\preceq$ over $\Theta_V$. If $\theta_1 \preceq \theta_2$, and we have searched $\theta_1$, then we can prune $\theta_2$ without changing the satisfiability or optimal value of $P$.

**Proof.** Follows trivially from Definition III.4.1.

**Theorem III.4.4.** Given a finite domain problem $P \equiv (V, D, C)$, and a dominance relation $\preceq$ over $\Theta_V$, we can prune all valuations $\theta$ such that $\exists \theta' \in \Theta_V$ s.t. $\theta' \prec \theta$, without changing the satisfiability or optimal value of $P$. 
Proof. Let \( \theta_0 \) be an optimal solution. If \( \theta_0 \) is pruned, then there exists some solution \( \theta_1 \) s.t. \( \theta_1 \prec \theta_0 \). Then \( \text{obj}(\theta_1) \leq \text{obj}(\theta_0) \), so \( \theta_1 \) is also an optimal solution. In general, if \( \theta_i \) is pruned, then there must exist some \( \theta_{i+1} \) s.t. \( \theta_{i+1} \prec \theta_i \) and \( \theta_{i+1} \) is also an optimal solution. It is impossible for the sequence \( \theta_0, \theta_1, \ldots \) to repeat, since if \( \theta_i \equiv \theta_j \) for some \( i < j \), then we must have \( \theta_{i+1} \prec \theta_i \) and \( \theta_i \prec \theta_{i+1} \), which violates the definition of \( \prec \). Since there are finitely many optimal solutions, the sequence \( \theta_0, \theta_1, \ldots \) must terminate in some \( \theta_i \) that is an optimal solution and is not pruned.

We will call the pruning allowed by Theorem III.4.3 and Theorem III.4.4 dominance breaking. It is easy to see that symmetry breaking is a special case of dominance breaking.

Definition III.4.5. We define the dominance breaking constraint \( \text{db}(\preceq) \) for dominance relation \( \preceq \) as: \( \text{solns}(\text{db}(\preceq)) \equiv \{ \theta \mid \theta \in \Theta_V, \nexists \theta' \in \Theta_V \text{ s.t. } \theta' \prec \theta \} \).

Theorem III.4.6. Let \( S \) be a symmetry group for problem \( P \). Let \( \text{sb}(S, \prec_{\text{lex}}) \) be a global lexicographical order on the valuations of \( P \). Let \( \text{sb}(S, \prec_{\text{lex}}) \) be a complete lexicographical symmetry breaking constraint as defined in Definition II.2.2. Let \( \preceq \) be a dominance relation defined as follows: \( \theta_1 \preceq \theta_2 \) iff \( \exists \sigma \in S \text{ s.t. } \sigma(\theta_1) = \theta_2 \) and \( \theta_1 \prec_{\text{lex}} \theta_2 \). Then \( \text{db}(\preceq) \Rightarrow \text{sb}(S, \prec_{\text{lex}}) \).

Proof. Both \( \text{sb}(S, \prec_{\text{lex}}) \) and \( \text{db}(\preceq) \) prune exactly those valuations that are not lex-least in their orbits. \( \square \)

Similarly, dynamic symmetry breaking is an application of Theorem III.4.3, where we define \( \theta \preceq \theta' \) iff \( \exists \sigma \in S \text{ s.t. } \sigma(\theta) = \theta' \).

Dominance relations can also be defined over subproblems.

Definition III.4.7. Given a problem \( P \equiv (V, D, C, f) \) where \( f \) is to be minimized, a dominance relation \( \preceq \) over the subproblems of \( P \) is a reflexive, transitive binary relation on the subproblems of \( P \) such that: if \( P_1 \preceq P_2 \), then \( \min\{ f(\theta) \mid \theta \in \text{solns}(P_1) \} \leq \min\{ f(\theta) \mid \theta \in \text{solns}(P_2) \} \).

Theorem III.4.8. If \( P_1 \preceq P_2 \), and we have examined \( P_1 \), then we can prune \( P_2 \) without changing the satisfiability or optimal value of \( P \).
Theorem III.4.9. Given a finite domain problem \( P \equiv (V, D, C) \), and a dominance relation \( \preceq \) over a set of disjoint subproblems of \( P \) (i.e., no two subproblems share any solutions), we can prune all subproblems \( P' \) such that \( \exists P'' \) s.t. \( P'' \prec P' \), without changing the satisfiability or optimal value of \( P \).

Dominance relations can be found in many problems, allowing us to prune many parts of the search space without having to examine it (e.g., [3, 159, 28, 54, 108, 116]). They can be very powerful in optimization problems, because they provide a completely different and complementary kind of pruning to the branch and bound paradigm. In the branch and bound paradigm, the only way to show that a subproblem \( P' \) is suboptimal is to prove a sufficiently strong bound on its objective value. Proving such bounds can be very expensive, especially if the model does not propagate strong bounds on the objective. In the worse case, further search is required, which can take an exponential amount of time. On the other hand, dominance relations can prune a subproblem \( P' \) without having to prove any bounds on its objective value at all. In many problems, there are parts of the search space that are obviously suboptimal, and which can be proved to be suboptimal using simple logical arguments. Once dominance relations expressing these are found and proved, the only cost in the algorithm is to check whether a subproblem is dominated, which is typically much lower than that required to prove a sufficiently strong bound to prune the subproblem.

### III.5 Contributions

In Chapter 6, we apply relaxation and dominance techniques to the Minimization of Open Stacks Problem [162] (MOSP). We describe 4 different dominance relations, which when exploited, gives around 5-6 orders of magnitude speedup on the hardest instances. We also apply a relaxation technique to the MOSP, where we relax the problem by merging customers together. The relaxation technique gives around 3-4 orders of magnitude speedup on the hardest instances.

In Chapter 7, we apply bounded dynamic programming, relaxation and dominance techniques on the Talent Scheduling Problem (prob039 in CSPLib). Bounded
dynamic programming is a variant of dynamic programming that is much more suitable for optimization problems. Applying it gives some 1-2 orders of magnitude speedup compared to a simple dynamic programming approach. We prove 2 dominance relations, which when exploited, gives a modest $\sim 2$ times speedup. We also apply relaxation and dominance techniques to derive lower bounds on cost for subproblems. This stronger lower bound significantly reduces the size of the search space, giving a further 1 order of magnitude speedup.

In Chapter 8, we examine the Maximum Density Still Life Problem (prob032 in CSPLib). The Maximum Density Still Life Problem is a very hard combinatorial optimization problem with a raw search space of $O(2^{n^2})$. The previous best approach is only able to solve up to $n = 20$. By using a combination of remodeling, relaxations, bounded dynamic programming and customized search, guided by deep mathematical insights into the problem, we are able to solve the problem for all $n$. As a comparison, our new approach can solve $n = 100$ in only a matter of hours, whereas the previous best approach would require in the order of $10^{100}$ years and $10^{30}$ bytes of memory to solve $n = 100$. It is possible to solve the Still Life problem for all $n$, because the Still Life Problem becomes well behaved mathematically for sufficiently large $n$ (around $n > 200$), and it is possible to solve all the large cases in constant time.
Chapter 6

Minimization of Open Stacks

Problem
6.1 Introduction

In this chapter, we describe a new exact solver for the Minimization of Open Stacks Problem (MOSP) [162]. By combining nogood learning and dominance breaking with a branch and bound strategy, our solver is able to solve hard instances of MOSP some 5-6 orders of magnitude faster than the previous state of the art. We also show how relaxation techniques can be used to further speed up the proof of optimality by up to another 3-4 orders of magnitude on the hardest instances.

The MOSP can be described as follows. A factory manufactures a number of different products in batches, i.e., all copies of a given product need to be finished before a different product is manufactured, so there are never two batches of the same product. Each customer of the factory places an order requiring one or more different products. Once one product in a customer’s order starts being manufactured, a stack is opened for that customer to store all the products in the order. Once all the products for a particular customer have been manufactured, the order can be sent and the stack is freed for use by another order. The aim is to determine the sequence in which products should be manufactured in order to minimize the maximum number of open stacks, i.e., the maximum number of customers whose orders are simultaneously active. The importance of this problem comes from the variety of real situations in which the problem (or an equivalent version of it) arises, such as cutting, packing, and manufacturing environments, or VLSI design. Indeed the problem appears in many different guises in the literature, including: graph pathwidth and gate matrix layout (see [97] for a list of 12 equivalent problems). The problem is known to be NP-hard [97].

We can formalize the problem as follows. Let \( P \) be a set of products, \( C \) a set of customers, and \( c(p) \) a function that returns the set of customers who have ordered product \( p \in P \). Since the products ordered by each customer \( c \in C \) are placed in a stack different from that of any other customer, we use \( c \) to denote both a client and its associated stack. We say that customer \( c \) is active (or that stack \( c \) is open) at time \( k \) in the manufacturing sequence if there is a product required by \( c \) that is manufactured before or at time \( k \), and a product required by \( c \) that is manufactured
6.1. INTRODUCTION

<table>
<thead>
<tr>
<th></th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
<th>$p_5$</th>
<th>$p_6$</th>
<th>$p_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>$c_2$</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>$c_3$</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>$c_4$</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>$c_5$</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th></th>
<th>$p_7$</th>
<th>$p_6$</th>
<th>$p_5$</th>
<th>$p_4$</th>
<th>$p_3$</th>
<th>$p_2$</th>
<th>$p_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>$c_2$</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>$c_3$</td>
<td>.</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>.</td>
</tr>
<tr>
<td>$c_4$</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>$c_5$</td>
<td>.</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

(b)

Figure 6.1: (a) An example $c(p)$ function: $c_i \in c(p_j)$ if the row for $c_i$ in column $p_j$ has an X. (b) An example schedule: $c_i$ is active when product $p_j$ is scheduled if the row for $c_i$ in column $p_j$ has an X or a –.

at or after time $k$. In other words, $c$ is active from the time the first product ordered by $c$ is manufactured until the last product ordered by $c$ is manufactured. The MOSP aims at finding a schedule for manufacturing the products in $P$ (i.e., a permutation of the products) that minimizes the maximum number of active customers (or of open stacks) at any time. We call a problem with $n$ customers and $m$ products an $n \times m$ problem.

Example 6.1.1. Consider a $5 \times 7$ MOSP defined by the set of customers $C \equiv \{ c_1, c_2, c_3, c_4, c_5 \}$, the set of products $P \equiv \{ p_1, p_2, p_3, p_4, p_5, p_6, p_7 \}$, and a $c(p)$ function determined by the matrix $M$ shown in Figure 6.1(a), where an X at position $M_{ij}$ indicates that client $c_i$ has ordered product $p_j$.

Consider the manufacturing schedule given by sequence $[p_7, p_6, p_5, p_4, p_3, p_2, p_1]$ and illustrated by the matrix $M$ shown in Figure 6.1(b), where client $c_i$ is active at position $M_{ij}$ if the position contains either an X ($p_j$ is in the stack) or an – ($c_i$ has an open stack waiting for some product scheduled after $p_j$). Then, the active customers at time 1 are $\{ c_1, c_4 \}$, at time 2 $\{ c_1, c_3, c_4 \}$, at time 3 $\{ c_1, c_3, c_4, c_5 \}$, at time 4 $\{ c_1, c_2, c_3, c_4, c_5 \}$, at time 5 $\{ c_1, c_2, c_3, c_4, c_5 \}$, at time 6 $\{ c_1, c_2, c_3 \}$, and at time 7 $\{ c_1, c_2 \}$. The maximum number of open stacks for this particular schedule is thus 5.

The MOSP was chosen as the subject of the first Constraint Modeling Challenge [146] posed in May 2005. Many different techniques were explored in the 13 entries to the challenge. The winning entry by Garcia de la Banda and Stuckey [53] used a dynamic programming approach in a branch and bound framework to solve
the MOSP. The natural search strategy of labeling the products in chronological order leads to many subproblem equivalences. These subproblem equivalences can be exploited using dynamic programming [53] or nogood learning [141]. Also, various lookaheads can be used to calculate upper bounds on the number of open stacks so that suboptimal branches can be pruned earlier. These techniques led to a solver that was orders of magnitude faster than any of the other entries in the 2005 MOSP challenge.

The search strategy used in this winning entry (branching on which product to produce next), is actually far from optimal. As was first discussed in [157] and shown in [158], branching on which customer stack to close next is never worse than branching on which product to produce next, and is usually much better, even when the number of customers is far greater than the number of products. This is the result of a simple dominance relation. In this chapter, we show that combining this search strategy with nogood recording produces a solver that is some 5-6 orders of magnitude faster than the winning entry to the Modeling Challenge on hard instances. We also derive several other dominance rules that provide a further 1-2 orders of magnitude improvement. One rule in particular largely subsumes the effect of the nogood recording. This allows us to reduce the memory usage from a potentially exponential amount to a constant $\sim 2$Mb for even the largest solvable instances. We also utilize relaxation techniques to speed up the proof of optimality for the hardest instances by a further 3-4 orders of magnitude. With all the improvements, our solver is able to solve all the open instances from the Modeling Challenge within 10 seconds!

6.2 Searching on Customers

We use a branch and bound search strategy based on branching on which customer to close next, rather than on which product to produce next. This customer based search strategy has been discussed in [157, 155] and is based on the following dominance relation:

**Proposition 6.2.1.** Given a product order $U \equiv [p_1, p_2, \ldots, p_n]$, define a customer close order $T \equiv [c_1, c_2, \ldots, c_m]$ as the order in which customer stacks can close given
6.2. SEARCHING ON CUSTOMERS

U. Construct another product ordering \( U' \) as follows: schedule the products required by \( c_1 \), then the products required by \( c_2 \), then the products required by \( c_3 \), etc. Then \( U' \) uses no more stacks than \( U \).

This dominance relation is proven in [157] and [155], and shows that it is sufficient to examine only product orderings where every product is required for the next customer’s stack to close. This dramatically reduces the number of product orderings that have to be examined. We can implement this by branching on which customer’s stack to close next, and then scheduling exactly those products which are needed.

Example 6.2.2. Consider the schedule \( U \) shown in Figure 6.1(b), the customers are closed in the order \( \{c_4, c_5\} \) when \( p_3 \) is scheduled, then \( \{c_3\} \) when \( p_2 \) is scheduled, then \( \{c_2, c_1\} \) when \( p_1 \) is scheduled. Consider closing the customers in the order \( T = [c_5, c_4, c_3, c_2, c_1] \) compatible with \( U \). This leads to a product ordering, for example, of \( U' = [p_3, p_5, p_4, p_6, p_7, p_2, p_1] \). The resulting scheduling is shown in Figure 6.2(a). It only requires 4 stacks (and all other schedules with this closing order will use the same maximum number of open stacks).

Rather than thinking in terms of products, it is simpler to think of the MOSP purely in terms of the customers. We define the customer graph as follows:

**Definition 6.2.3.** Let the customer graph \( G = (V, E) \) for an instance of the MOSP be: \( V = C \) and \( E = \{(c_1, c_2) \mid \exists p \in P, \{c_1, c_2\} \subseteq c(p)\} \), i.e., a graph in which nodes

\[
\begin{array}{cccccc}
{c_1} & {p_3} & {p_5} & {p_4} & {p_6} & {p_7} & {p_2} & {p_1} \\
{c_2} & . & . & X & . & X & . & X \\
{c_3} & . & X & X & . & X & . & X \\
{c_4} & X & X & X & . & X & . & X \\
{c_5} & X & X & . & . & . & . & . \\
\end{array}
\]

(a)

\[
\begin{array}{cccccc}
{c_1} & {p_3} & {p_5} & {p_4} & {p_6} & {p_7} & {p_2} & {p_1} \\
{c_2} & . & . & X & . & X & . & X \\
{c_3} & . & X & X & . & X & . & X \\
{c_4} & X & X & X & . & X & . & X \\
{c_5} & X & X & . & . & . & . & . \\
\end{array}
\]

(b)

(c)
represent customers, and two nodes are adjacent if their customers require at least one product in common.

Note that, by definition, each node is self-adjacent. The customer graph for the problem of Example 6.1.1 is shown in Figure 6.2(c). It is easy to see that any two instances of the MOSP with the same customer graph must have the same minimum number of open stacks. Thus an instance of the MOSP is uniquely specified by its customer graph. The sole effect of the products are to create edges in the customer graph. A MiniZinc model for the MOSP using the customer graph and customer search is given in Example 3.3.3. If a customer order satisfying these constraints are found, then a corresponding product ordering using the same number of stacks can trivially be constructed. We show that the subproblem equivalences exploited in [53] still apply to the customer search and can be exploited in the same way.

**Theorem 6.2.4.** Suppose we have some partial assignment \( x[1] = v_1, \ldots, x[m] = v_m \), and it does not violate the stack constraints from time 1 to time \( m \). Then there exists solutions mapping the set \( \{x[1], \ldots, x[m]\} \) to \( \{v_1, \ldots, v_m\} \) iff there exists solutions extending \( x[1] = v_1, \ldots, x[m] = v_m \).

**Proof.** The “if” direction of the proof is trivial, so we only prove the “only if” direction. Suppose there exists a solution \( \theta' \) mapping the set \( \{x[1], \ldots, x[m]\} \) to \( \{v_1, \ldots, v_m\} \), say \( \theta' \equiv x[1] = v_1', \ldots, x[m] = v_m', x[m+1] = v_{m+1}', \ldots, x[c] = v_c' \), where \( v_1', \ldots, v_m' \) is some permutation of \( v_1, \ldots, v_m \). We show that \( \theta \equiv x[1] = v_1, \ldots, x[m] = v_m, x[m+1] = v_{m+1}', \ldots, x[c] = v_c' \) is also a solution.

Firstly, by assumption, the partial assignment \( x[1] = v_1, \ldots, x[m] = v_m \) does not violate the stack constraints from time 1 to time \( m \). It is also easy to see that the values of \( x[m+1], \ldots, x[c] \) cannot cause the stack constraints from time 1 to time \( m \) to be violated, as their only effect is to force some stacks to open at time \( m + 1 \) or later via a \( s[i] \leq e[j] \) constraint where \( e[j] \geq m + 1 \). This cannot force any \( s[i] \leq t \) where \( t \leq m \) to become true, thus the stack constraints from time 1 to time \( m \) remain satisfied.

Secondly, the only way that the values of \( x[1] \) to \( x[m] \) can affect the stack constraints from time \( m + 1 \) to time \( c \) is by forcing some stacks to open before time \( m \).
via a $s[i] \leq e[j]$ constraint. However, the stack constraints from time $m + 1$ to time $c$ only care about the value of $s[i] \leq t$ where $t \geq m + 1$. So they only care whether the stack was opened before $m + 1$ or not. They do not care exactly when the stack was opened between time 1 and time $m$. Now, both $\theta$ and $\theta'$ schedules the same set of customers during time 1 to time $m$, thus the set of stacks opened before time $m$ is identical and has the same effect on the stack constraints from time $m + 1$ to time $c$. Then, since the values of $x[m + 1]$ to $x[c]$ are identical in $\theta$ and $\theta'$, and by assumption $\theta'$ satisfies the stack constraints from time $m + 1$ to time $c$, $\theta$ must also satisfy the stack constraints from time $m + 1$ to time $c$. Thus $\theta$ is also a solution. □

We exploit this result by deriving nogoods of the form $\neg(\{x[1], \ldots, x[m]\} = \{v_1, \ldots, v_m\})$, and propagating them in $O(1)$ using hash table lookups.

## 6.3 Improving the Search

In this section we show how to improve the customer search by exploiting several dominance relations. We first define some terminology.

**Definition 6.3.1.** Given a customer graph $G$ and a customer $c$, let $N(c)$ be the set of customers adjacent to $c$ in $G$.

In the next three definitions, let $S$ denote the set of customers whose stacks have been closed at a particular node in the search tree.

**Definition 6.3.2.** Let $O(S) \equiv \cup_{c \in S} N(c)$ be the set of stacks which were forced open as a consequence of the closure of the stacks of customers in $S$.

**Definition 6.3.3.** Given a customer $c \notin S$, let $o(c, S) \equiv N(c) \setminus O(S)$ be the new stacks which will open if $c$ is the next stack to close.

**Definition 6.3.4.** Let $\text{open}(c, S) = |o(c, S)|$ and $\text{close}(c, S) = |\{d \mid o(d, S) \subseteq o(c, S)\}|$, i.e., the number of new stacks that will open and close respectively if we close $c$’s stack next.
Definition 6.3.5. A partial assignment \( x[1] = v_1, \ldots, x[m] = v_m \) is \( k \)-playable if it only needs \( k \) stacks during the first \( m \) time periods.

We will represent partial assignments \( x[1] = v_1, \ldots, x[m] = v_m \) as sequences \([v_1, \ldots, v_m]\) for easier manipulation. We will also often use the same symbol to denote the sequence and its corresponding set.

6.3.1 Definite Moves

Under certain circumstances, one particular choice for the next customer will become so good that it is at least as good as any other choice. In this case, we can pick that choice and prune all other choices. We call this a definite move.

Theorem 6.3.6. Suppose \( S \) is some sequence, and \( q \notin S \) is a customer such that: \( S + [q] \) is \( k \)-playable, and \( \text{close}(q, S) \geq \text{open}(q, S) \). If \( S \) has an extension that uses \( \leq k \) stacks, then \( S + [q] \) also has an extension that uses \( \leq k \) stacks.

Proof. Suppose \( U \equiv S + [c_1, c_2, \ldots, c_m, q, c_{m+1}, \ldots, c_n] \) uses \( \leq k \) stacks. We claim that \( U' \equiv S + [q, c_1, \ldots, c_m, c_{m+1}, \ldots, c_n] \) also uses \( \leq k \) stacks. The two sequences differ only in the placement of \( q \). The number of stacks which are open at any time before, or any time after the set of customers \( \{c_1, c_2, \ldots, c_m, q\} \) are played is identical for \( U \) and \( U' \), since it only depending on the set of customers closed and not the order. The number of open stacks when \( q \) is played in \( U' \) is also \( \leq k \), since \( S + [q] \) is \( k \)-playable. Finally, the number of open stacks when \( S + [q, c_1, c_2, \ldots, c_i] \) is played in \( U' \) is always less than or equal to the number of open stacks when \( S + [c_1, c_2, \ldots, c_i] \) is played in \( U \), because we have at most \( \text{open}(q, S) \) extra stacks open, but at least \( \text{close}(q, S) \) extra stacks closed. So \( U' \) uses \( \leq k \) stacks at all times. \( \square \)

Corollary 6.3.7. Suppose in the branch and bound algorithm, we are currently seeking solutions that use \( \leq k \) stacks. If at some search node, \( S \) is the current sequence, and there exists \( q \notin S \) such that: \( S + [q] \) is \( k \)-playable and \( \text{close}(q, S) \geq \text{open}(q, S) \), then it is correct to prune all choices other than \( q \).
6.3. IMPROVING THE SEARCH

6.3.2 Better Moves

Similarly, there are situations where one choice is at least as good as another, and we can prune the one that is not as good. We call this a better move.

Theorem 6.3.8. Suppose $S$ is some sequence, and $q, r \notin S$ are customers such that: $S + [q]$ and $S + [r, q]$ are both $k$-playable, and $\text{close}(q, S) \geq \text{open}(q, S \cup \{r\})$. If $S + [r]$ has an extension that uses $\leq k$ stacks, then $S + [q]$ also has an extension that uses $\leq k$ stacks.

Proof. Suppose $U \equiv S + [r, c_1, c_2, \ldots, c_m, q, c_{m+1}, \ldots, c_n]$ uses $\leq k$ stacks. We have $\text{close}(q, S \cup \{r\}) \geq \text{close}(q, S) \geq \text{open}(q, S \cup \{r\})$, so after $r$ is played, $q$ becomes a definite move. By the same argument as the previous proof, $U' \equiv S + [r, q, c_1, \ldots, c_n]$ uses $\leq k$ stacks. Now, consider $U'' \equiv S + [q, r, c_1, \ldots, c_n]$, where we swap the positions of $q$ and $r$. Consider the stacks which are open when $r$ is played in $U'$ compared to when $r$ is played in $U$. The number of new open stacks is exactly $\text{open}(q, S \cup \{r\})$, and the number of new closed stacks is exactly $\text{close}(q, S)$. So playing $r$ in $U''$ uses no more stacks than playing $r$ in $U'$. Also, the number of open stacks when $q$ is played in $U''$ is $\leq k$ since $S + [q]$ is $k$-playable. Thus $U''$ uses $\leq k$ stacks at all times. □

Corollary 6.3.9. Suppose in the branch and bound algorithm, we are currently seeking solutions that use $\leq k$ stacks. If at some search node, $S$ is the current sequence, and there exists $q, r \notin S$ such that: $S + [q]$ and $S + [r, q]$ are both $k$-playable, and $\text{close}(q, S) \geq \text{open}(q, S \cup \{r\})$, then it is correct to prune $r$.

Although better move seems weaker than definite move, since it only prunes one branch at a time rather than prune all branches but one, it is actually a generalization. It is easy to see from the definitions that any definite move is also a better move with respect to all other moves. Our implementation of better move subsumes definite move so we will simply consider them as one improvement.

6.3.3 Old Move

Sometimes, a subproblem which we are about to explore has actually been fully explored before using an alternative search path. This occurs because of the subproblem
equivalences inherent in the MOSP.

**Theorem 6.3.10.** Suppose \( S \equiv [s_1, s_2, \ldots, s_n] \), and \( q \notin S \) is a customer such that: 
\( S' = [s_1, s_2, \ldots, s_m, q, s_{m+1}, \ldots, s_n] \) is \( k \)-playable. If \( U \equiv S + [q] \vdash R \) is a solution that uses \( \leq k \) stacks, then \( U' \equiv S' \vdash R \) is also a solution that uses \( \leq k \) stacks.

**Proof.** \( S' \) is \( k \)-playable by assumption so the number of open stacks at any time during \( S' \) is \( \leq k \). At any point after \( S' \), the number of open stacks are identical for \( U \) and \( U' \) since it only depends on the set of closed customers and not the order. Hence \( U' \) uses \( \leq k \) at all times. \( \square \)

**Corollary 6.3.11.** Suppose in the branch and bound algorithm, we are currently seeking solutions that use \( \leq k \) stacks. If at some search node, \( S \) is the current sequence, and there exists \( q \notin S \) such that: 
\( S' = [s_1, s_2, \ldots, s_m, q, s_{m+1}, \ldots, s_n] \) is \( k \)-playable, and we have previously exhaustively searched the subproblem \( S'' = [s_1, s_2, \ldots, s_m, q] \), then it is correct to prune \( q \).

This pruning scheme was mentioned in [157], but it was incorrectly stated there. The author of [157] failed to note that the condition that \( S' = [s_1, s_2, \ldots, s_m, q, s_{m+1}, \ldots, s_n] \) is \( k \)-playable is in fact crucial, because if it does not hold, then the subproblem rooted at \( S' \) would have been pruned via breaking the upper bound on the number of stacks, and none of its extensions would have been explored. Thus pruning \( q \) now would be incorrect.

Naively, it would appear to take \( O(|C|^3) \) time to check the old move condition at each node. However, it is possible to do so in \( O(|C|) \) time. At each node we keep a set \( Q(S) \) of all the old moves, i.e., the set of moves \( q \) such that we can find \( S' = [s_1, s_2, \ldots, s_m, q, s_{m+1}, \ldots, s_n] \) which is \( k \)-playable, and such that move \( q \) has already been searched at the node \( S'' = [s_1, \ldots, s_m] \). Note that by definition, when a move \( r \) has been searched at the current node, \( r \) will be added to \( Q(S) \). It is easy to calculate \( Q(S \vdash [s_{n+1}]) \) when we first reach that child node. First, \( Q(S \vdash [s_{n+1}]) \subseteq Q(S) \), since if \( S' = [s_1, s_2, \ldots, s_m, q, s_{m+1}, \ldots, s_n, s_{n+1}] \) is \( k \)-playable then by definition so is \( S' = [s_1, s_2, \ldots, s_m, q, s_{m+1}, \ldots, s_n] \). Second, to check if each \( q \in Q(S) \) is also in \( Q(S \vdash [s_{n+1}]) \), we simply have to check whether the last move
6.3. IMPROVING THE SEARCH

in $S' = [s_1, s_2, \ldots, s_m, q, s_{m+1}, \ldots, s_n, s_{n+1}]$ is $k$-playable, as all the previous moves are already known to be $k$-playable since $q \in Q(S)$. Checking the last move takes constant time so the total complexity is $O(|C|)$. There are some synergies between the better move improvement and the old move improvement. If $q \in Q(S)$ and $q$ is better than move $r$, then we can add $r$ to $Q(S)$ as well. This allows old move to prune sets that we have never even seen before.

6.3.4 Upper bound heuristic

In this section, we describe an upper bound heuristic which was found to be very effective on our instances. A good heuristic for finding an optimal solution is useful from a practical point of view if no proof of optimality is required. It is also a crucial component for the relaxation techniques described in the next subsection which can give several orders of magnitude speedup on the proof of optimality for hard problems.

In [53] the authors tried multiple branching heuristics in order to compute an upper bound, but only applied them in a greedy fashion, effectively searching only 1 leaf node for each. We can do much better by performing an incomplete search where we are willing to explore a larger number of nodes, but still much fewer than a complete search. Simple ways of doing this using our complete search engine include, sorting the choices according to some criteria, and only trying the first $m$ moves for some $m$. Or trying all the moves which are no worse than the best by some amount $e$, etc.

One heuristic that is extremely effective is to only consider the moves where we close a customer stack that is currently open, the intuition being that if a stack is not even open yet, there is no point trying to close it now. Although this seems intuitively reasonable, it is in fact not always optimal. In practice however, an incomplete search using this criteria is very fast, and finds the optimal solution almost all the time, and several orders of magnitude faster than the complete search for some hard instances. The reason for its strength comes from its ability to exploit a not quite perfect dominance relation. Almost all the time, subtrees where we close a stack that is not yet open is dominated by one where we close a currently open stack, and
Table 6.1: Comparison of upper bound heuristic, versus complete search on some difficult problems. Times in milliseconds

<table>
<thead>
<tr>
<th>Instance</th>
<th>Orig. time</th>
<th>Heur. time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-100-2</td>
<td>4136.3</td>
<td>39.8</td>
<td>104.0</td>
</tr>
<tr>
<td>100-100-4</td>
<td>4715.5</td>
<td>43.3</td>
<td>108.8</td>
</tr>
<tr>
<td>100-100-6</td>
<td>8.2</td>
<td>12.8</td>
<td>0.6</td>
</tr>
<tr>
<td>100-100-8</td>
<td>9.2</td>
<td>6.3</td>
<td>1.5</td>
</tr>
<tr>
<td>100-100-10</td>
<td>1.6</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>125-125-2</td>
<td>1159397.3</td>
<td>385.1</td>
<td>3010.3</td>
</tr>
<tr>
<td>125-125-4</td>
<td>2593105.1</td>
<td>398.9</td>
<td>6500.1</td>
</tr>
<tr>
<td>125-125-6</td>
<td>8975.9</td>
<td>424.9</td>
<td>21.1</td>
</tr>
<tr>
<td>125-125-8</td>
<td>187.8</td>
<td>146.1</td>
<td>1.3</td>
</tr>
<tr>
<td>125-125-10</td>
<td>22.2</td>
<td>8.3</td>
<td>2.7</td>
</tr>
</tbody>
</table>

thus we can exploit this to prune branches similarly to what we do in Section 6.3. The dominance is not always true however, so using such a pruning rule makes it an incomplete, heuristic search. See Table 6.1 for a brief comparison of the times required to find the optimal solution.

### 6.3.5 Relaxation

Relaxation techniques have been used on the MOSP before in the context of a local search method [115]. The idea was to try to relax the problem in such a way that the solution density is increased, so that better solutions can be found faster. However, these methods are of no help in proving optimality. In this section we show how relaxations can be used to speed up the proof of optimality.

As is seen in the experimental results in [53], the sparser instances of MOSP are substantially harder than denser instances of MOSP given the same number of customers and products. This can be explained by the fact that in sparser instances, each customer has far fewer neighbors in the customer graph, thus many more moves would fall under the upper bound limit at each node and both the depth and the branching factor of the search tree are dramatically increased compared to a dense instance of the same size.
However, the sparsity of these instances also leads to a potential optimization. Since the instance is sparse and the optimum is low (e.g., 20-50 for a $125 \times 125$ problem) it is possible that not all of the constraints are actually required to force the lower bound. It is possible that there is some small “unsatisfiable core” of customers which are producing the lower bound. If such an unsatisfiable core exists and can be identified, we can potentially remove a large number of customers from the problem and make the proof of optimality much quicker. It turns out that this is often possible.

First, we will show how we can relax the MOSP instance. Naively, we can simply delete one or more nodes from the customer graph and remove all edges involving those node. This represents the deletion of some variables and constraints and is of course a valid relaxation. However, we can do much better. We show that given an MOSP instance $P$ corresponding to customer graph $G$, if $G'$ is a contraction of $G$, then the MOSP instance $P'$ corresponding to $G'$ is a relaxation of $P$. Using this, we can get some compensation when we remove nodes, as we can retain some of the edges. This makes $P'$ stronger and allows more nodes to be removed without loosening the lower bound.

**Theorem 6.3.12.** Let $G'$ be a contraction of graph $G$. If $P$ is the MOSP corresponding to $G$ and $P'$ is the MOSP corresponding to $G'$, then $P'$ is a relaxation of $P$.

**Proof.** We prove the result for the case where $G'$ is derived from $G$ by merging a single pair of adjacent nodes. The general result follows by induction. Suppose $G$ had $n$ nodes $c_1, \ldots, c_n$. Without loss of generality, suppose $c_1$ and $c_2$ are adjacent and we merge them to create $G'$. Call the new merged node $c_2$. We show that every solution of $P$ that uses $k$ stacks corresponds to a solution of $P'$ that uses $\leq k$ stacks. Let $S$ be a solution of $P$. Without loss of generality, let it be $S \equiv [v_1, \ldots, v_a, 1, v_{a+1}, \ldots, v_b, 2, v_{b+1}, \ldots, v_{n-2}]$. We claim that $S' \equiv [v_1, \ldots, v_a, v_{a+1}, \ldots, v_b, 2, v_{b+1}, \ldots, v_{n-2}]$ is a solution of $P'$. We show that for each customer, the number of stacks open when the customer is played in $S'$ for problem $P'$ is no greater than the number of stacks open when the customer is played in $S$ for problem $P$. 

Let us compare the stacks which are open when $v_i$ is played for $i \leq a$. For any $j \geq 3$, $c_j$’s stack is open in $S'$ iff it is open in $S$. $c_2$’s stack is open in $S'$ iff $c_1$’s or $c_2$’s stack is open in $S$. $c_1$’s stack is never open in $S'$ since $c_1$ does not exist in $G'$. In all cases, the number of stacks open in $S'$ is less than or equal to the number of stacks open in $S$. Let us compare the stacks which are open when $v_i$ is played for $a < i \leq b$. For any $j \geq 3$, $c_j$’s stack is open in $S'$ iff it is open in $S$. $c_2$’s stack may or may not be open in $S'$, but, $c_2$’s stack is guaranteed to be open in $S$ because $c_1$ is adjacent to $c_2$, and $c_1$ had previously been played. So once again, the number of stacks open in $S'$ is less than or equal to the number of stacks open in $S$. For the remainder of the customers, the number of stacks that are open are identical for $S$ and $S'$. Hence, $S'$ is a solution that uses $\leq k$ stacks. This shows that $P'$ is a relaxation of $P$. □

Next we need to identify the nodes which can be removed/merged without loosening the lower bound on the problem. The main idea is that the longer a customer’s stack is open in the optimal solutions, the more likely it is that that customer is contributing to the lower bound, since removal of such a customer would mean that there is a high chance that one of the optimal solutions can reduce to one needing one fewer stack. Thus we want to avoid removing such customers. Instead we want to remove or merge customers whose stacks are usually open for a very short time. One naïve heuristic is to greedily remove nodes in the customer graph with the lowest degree. Fewer edges coming out of a node presumably means that the stack is open for a shorter period of time on average.

A much better heuristic comes from the following idea. Suppose there exist a node $c$ such that any neighbor of $c$ is also connected to most of the neighbors of $c$, then when $c$ is forced open by the closure of one of those neighbors, that neighbor would also have forced most of the neighbors of $c$ to open, and thus $c$ will be able to close soon afterwards and will only be open for a short time. The condition that most neighbors of $c$ are connected to most other neighbors of $c$ is in fact quite common for sparse instances due to the way that the customer graph is generated from the products (each product produces a clique in the graph). To be more precise, in our implementation, the customers are ranked according to:
6.3. IMPROVING THE SEARCH

\[ F(c) = \sum_{c' \in N(c)} \frac{|N(c) - N(c')|}{|N(c)|} \]  \hspace{1cm} (6.1)

This is a weighted average of the number of neighbors \( c' \) of \( c \) that are not connected to each neighbor of \( c \). The weights represent the fact that neighbors with fewer neighbors are more likely to close early and be the one that forces \( c \) to open. We merge the node \( c \) with the highest value of \( F(c) \) with the neighboring node \( c' \) with the highest value of \( |N(c) - N(c')| \), as that node stands to gain the highest number of edges.

Although we have a good heuristic for finding nodes to merge, it is quite possible to relax too much to the point that the relaxed problem has a solution lower than the true lower bound of the original problem, in which case it will be impossible to prove the true lower bound using this relaxed problem. Thus it is important that we have a quick way of finding out if we have relaxed too much. This is where the very fast and strong upper bound heuristic of the previous subsection is needed. Let \( |G| \) denote the number of vertices in a graph \( G \). We denote a complete search on the MOSP problem specified by customer graph \( G \) by \( \text{MOSP}(G) \) and an incomplete search heuristic for upper bounds as \( \text{ub}_\text{MOSP}(G) \). The overall relaxation algorithm is as follows:

\[ \text{relax\_MOSP}(G) \]
\[ \begin{align*}
ub &:= \text{ub\_MOSP}(G) & \text{\% ub is an upper bound} \\
G' &:= G \\
\text{while} \ (|G'| > ub) & \\
G' &:= \text{merge\_one\_pair}(G') & \text{\% relax problem} \\
\text{while} \ (G \neq G') & \\
relax\_ub &:= \text{ub\_MOSP}(G') \\
\text{if} \ (relax\_ub < ub) & \text{\% too relaxed to prove lb} \\
G' &:= \text{unmerge\_one\_pair}(G') & \text{\% unrelax problem} \\
\text{else} \\
lb &:= \text{MOSP}(G') & \text{\% compute lower bound}
\end{align*} \]
if \((lb < ub)\)  % too relaxed to prove lb
\[ G' := \text{unmerge\_one\_pair}(G') \]  % unrelax problem
else return \(ub\)  % \(lb = ub\)
return \(\text{MOSP}(G)\)  % relaxation failed!

As can be seen, the upper bound heuristic is necessary to find a good (optimal) solution quickly. It is also used to detect when we are too relaxed as quickly as possible so that we can unrelax. If the upper bound heuristic is sufficiently good, we will quickly be able to find a relaxation that removes as many customers as possible without being too relaxed. If the upper bound heuristic is weak however, we could waste a lot of time searching in a problem that is in fact too relaxed to ever give us the true lower bound. In practice, we have found that our upper bound heuristic is quite sufficient for the instances we tested it on.

There are a few optimizations we can make to this basic algorithm. Firstly, when an unmerge is performed, we can attempt to extend the last solution found to a solution of this less relaxed problem. If the solution extends, then it is still too relaxed and we need to unmerge again. This saves us having to actually look for a solution to this problem. Secondly, naively, when we perform an unrelax, we can simply unmerge the last pair of nodes that were merged. However, we can do better. One of the weaknesses of the current algorithm is that the nodes to be merged are chosen greedily using equation (6.1). If this happens to choose a bad relaxation that lowers the lower bound early on, then we will not be able to remove any more customers beyond that point. We can fix this to some extent by choosing which pair of nodes to unmerge when we unrelax. We do this by considering each of the problems that we get by unmerging each pair of the current merges. If the last solution found does not extend to a solution for one of these, then we choose that unmerge, as this unrelaxation gives us a chance to prove the true lower bound. If the last solution extends to a solution for all of them, we unmerge the last pair as per usual. This helps to get rid of early mistakes in merging and is useful on several of our instances.
6.4 Experiments

In this section we demonstrate the performance of our algorithm, and the effect of the improvements. The experiments were performed on a Xeon Pro 2.4GHz processor with 2Gb of memory. The code implementing the approaches were compiled using g++ with -O3 optimization.

6.4.1 Modeling Challenge instances

Very stringent correctness tests were performed in view of the large speedups achieved. All versions of our solver were run on the 5000+ instances used in the 2005 model challenge [146], as well as another 100,000 randomly generated instances of size 10×10 to 30×30 and various densities. The answers were compared with the solver submitted to the 2005 model challenge and all answers were identical.

We compare our solver with the previous state of the art MOSP solver, on which our solver is based. The results clearly show that our solver is orders of magnitude faster than the original version. Getting an exact speedup is difficult as almost all of the instances that the original version can solve are solved trivially by our solver in a few milliseconds, whereas instances that our solver finds somewhat challenging are completely unsolvable by the original version.

Our solver was able to solve all the open problems from the Modeling Challenge: SP2, SP3, and SP4. Table 6.2 compares these problems with the best results from the Challenge by [53]. The nodes and times (in milliseconds) for [53] are for finding the best solution they can. The times for our method are for the full solve including proof of optimality (using all improvements).

6.4.2 Harder Random instances

Of the 5000+ instances used in the 2005 challenge, only SP2, SP3 and SP4 take longer than a few milliseconds for our solver to solve. Thus we generate some difficult random instances for this experiment. First we specify the number of customers, number of products and the average number of customers per product. We then calculate a
density that will achieve the specified average number of customers per product. The customer vs product table is then randomly generated using the calculated density to determine when to put 1’s and 0’s. As a post condition, we throw away any instance where the customer graph can be decomposed into separate components. This is done because we want to compare on instances of a certain size, but if the customer graph decomposes, then the instance degenerates into a number of smaller and relatively trivial instances.

We generate instances of sizes 30×30, 40×40, 50×50, 75×75, 100×100, 125×125, 100×50, 50×100, and average number of customer per product values of 2, 4, 6, 8, 10. 5 instances for each set of values were generated for a total of 200 instances.

Ideally, we want to measure speedup by comparing total solve time. However, as mentioned before, the instances that our solver finds challenging are totally unsolvable by the original. Table 6.3 is split into two parts. Above the horizontal line are the instances where the original managed to prove optimality. Here, nodes, time (in milliseconds) and speedup are for the total solve. Below the line the original cannot prove optimality. Here, nodes, time and speedup are for the finding a solution that is at least as good as the solver of [53] could find. The column $\delta$Opt shows the average distance this solution is from the optimal. Note that our approach finds and proves the optimal in all cases although the statistics for this are not shown in the table. Time to find an equally good solution is not necessarily a good indication of the speedup achievable for the full solve, as other factors like branching heuristics come into play. However, the trend is quite clear. The original solver is run with

<table>
<thead>
<tr>
<th>Garcia de la Banda and Stuckey [53]</th>
<th>This thesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Nodes</td>
<td>Time</td>
</tr>
<tr>
<td>SP2</td>
<td>19</td>
</tr>
<tr>
<td>SP3</td>
<td>36</td>
</tr>
<tr>
<td>SP4</td>
<td>56</td>
</tr>
</tbody>
</table>
Table 6.3: Comparing customer search versus [53] on harder random instances. Search is to find the best solution found by [53] with node limit $2^{25}$.

<table>
<thead>
<tr>
<th>Instance</th>
<th>$\delta$</th>
<th>[53] Nodes</th>
<th>Time(ms)</th>
<th>This paper Nodes</th>
<th>Time(ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>30-30-2</td>
<td>0</td>
<td>14318</td>
<td>480</td>
<td>408</td>
<td>4.4</td>
<td>109</td>
</tr>
<tr>
<td>30-30-4</td>
<td>0</td>
<td>48232</td>
<td>1981</td>
<td>158</td>
<td>2.0</td>
<td>979</td>
</tr>
<tr>
<td>30-30-6</td>
<td>0</td>
<td>89084</td>
<td>2750</td>
<td>56</td>
<td>0.9</td>
<td>3136</td>
</tr>
<tr>
<td>30-30-8</td>
<td>0</td>
<td>83558</td>
<td>2010</td>
<td>18</td>
<td>0.4</td>
<td>5322</td>
</tr>
<tr>
<td>30-30-10</td>
<td>0</td>
<td>18662</td>
<td>506</td>
<td>8</td>
<td>0.2</td>
<td>2335</td>
</tr>
<tr>
<td>40-40-2</td>
<td>0</td>
<td>669384</td>
<td>192917</td>
<td>1472</td>
<td>14.9</td>
<td>12942</td>
</tr>
<tr>
<td>40-40-4</td>
<td>0</td>
<td>3819542</td>
<td>227087</td>
<td>556</td>
<td>6.1</td>
<td>36959</td>
</tr>
<tr>
<td>40-40-6</td>
<td>0</td>
<td>11343235</td>
<td>625892</td>
<td>217</td>
<td>2.9</td>
<td>218062</td>
</tr>
<tr>
<td>40-40-8</td>
<td>0</td>
<td>8379706</td>
<td>334392</td>
<td>49</td>
<td>0.8</td>
<td>403272</td>
</tr>
<tr>
<td>40-40-10</td>
<td>0</td>
<td>3040040</td>
<td>98194</td>
<td>20</td>
<td>0.4</td>
<td>229305</td>
</tr>
<tr>
<td>50-50-2</td>
<td>0</td>
<td>12356205</td>
<td>1300311</td>
<td>6344</td>
<td>65.8</td>
<td>19758</td>
</tr>
<tr>
<td>50-50-4</td>
<td>0.2</td>
<td>5412259</td>
<td>446409</td>
<td>219</td>
<td>2.7</td>
<td>164728</td>
</tr>
<tr>
<td>50-50-6</td>
<td>0.2</td>
<td>7949026</td>
<td>510831</td>
<td>45</td>
<td>0.8</td>
<td>636409</td>
</tr>
<tr>
<td>50-50-8</td>
<td>0.2</td>
<td>525741</td>
<td>28337</td>
<td>15</td>
<td>0.4</td>
<td>75274</td>
</tr>
<tr>
<td>50-50-10</td>
<td>0</td>
<td>16809</td>
<td>784</td>
<td>7</td>
<td>0.2</td>
<td>3411</td>
</tr>
<tr>
<td>75-75-2</td>
<td>0.8</td>
<td>2485310</td>
<td>420935</td>
<td>7030</td>
<td>76.8</td>
<td>5484</td>
</tr>
<tr>
<td>75-75-4</td>
<td>2.6</td>
<td>3507703</td>
<td>666784</td>
<td>63</td>
<td>1.4</td>
<td>486669</td>
</tr>
<tr>
<td>75-75-6</td>
<td>1.2</td>
<td>4412286</td>
<td>756632</td>
<td>59</td>
<td>1.4</td>
<td>548132</td>
</tr>
<tr>
<td>75-75-8</td>
<td>1.2</td>
<td>4121046</td>
<td>519778</td>
<td>19</td>
<td>0.6</td>
<td>843336</td>
</tr>
<tr>
<td>75-75-10</td>
<td>0.6</td>
<td>1198282</td>
<td>120807</td>
<td>15</td>
<td>0.5</td>
<td>244128</td>
</tr>
<tr>
<td>100-100-2</td>
<td>2.2</td>
<td>3008009</td>
<td>765131</td>
<td>481</td>
<td>9.4</td>
<td>81653</td>
</tr>
<tr>
<td>100-100-4</td>
<td>4.8</td>
<td>6777140</td>
<td>2017286</td>
<td>145</td>
<td>3.3</td>
<td>619257</td>
</tr>
<tr>
<td>100-100-6</td>
<td>4</td>
<td>1269071</td>
<td>347970</td>
<td>39</td>
<td>1.4</td>
<td>241145</td>
</tr>
<tr>
<td>100-100-8</td>
<td>4.4</td>
<td>1686045</td>
<td>414456</td>
<td>31</td>
<td>1.1</td>
<td>363468</td>
</tr>
<tr>
<td>100-100-10</td>
<td>1.6</td>
<td>4195494</td>
<td>789039</td>
<td>15</td>
<td>0.7</td>
<td>1097494</td>
</tr>
<tr>
<td>125-125-2</td>
<td>1.8</td>
<td>7418402</td>
<td>3276210</td>
<td>36672</td>
<td>436.8</td>
<td>7500</td>
</tr>
<tr>
<td>125-125-4</td>
<td>3.8</td>
<td>3412379</td>
<td>1559691</td>
<td>916</td>
<td>20.4</td>
<td>76286</td>
</tr>
<tr>
<td>125-125-6</td>
<td>5.9</td>
<td>6076996</td>
<td>2643707</td>
<td>57</td>
<td>2.5</td>
<td>114480</td>
</tr>
<tr>
<td>125-125-8</td>
<td>6.2</td>
<td>942290</td>
<td>321050</td>
<td>28</td>
<td>1.5</td>
<td>217007</td>
</tr>
<tr>
<td>125-125-10</td>
<td>3.4</td>
<td>170852</td>
<td>45798</td>
<td>24</td>
<td>1.3</td>
<td>35647</td>
</tr>
<tr>
<td>50-100-2</td>
<td>0.2</td>
<td>90076</td>
<td>9971</td>
<td>97</td>
<td>1.6</td>
<td>6290</td>
</tr>
<tr>
<td>50-100-4</td>
<td>1</td>
<td>1973322</td>
<td>139300</td>
<td>23</td>
<td>0.6</td>
<td>220776</td>
</tr>
<tr>
<td>50-100-6</td>
<td>0.6</td>
<td>1784</td>
<td>116</td>
<td>13</td>
<td>0.4</td>
<td>301</td>
</tr>
<tr>
<td>50-100-8</td>
<td>0</td>
<td>97</td>
<td>9</td>
<td>5</td>
<td>0.2</td>
<td>47</td>
</tr>
<tr>
<td>50-100-10</td>
<td>0</td>
<td>99</td>
<td>8</td>
<td>3</td>
<td>0.2</td>
<td>39</td>
</tr>
<tr>
<td>100-50-2</td>
<td>0</td>
<td>2393401</td>
<td>438220</td>
<td>11117</td>
<td>133.7</td>
<td>3279</td>
</tr>
<tr>
<td>100-50-4</td>
<td>0.4</td>
<td>1421006</td>
<td>3499389</td>
<td>183260</td>
<td>1592.3</td>
<td>2198</td>
</tr>
<tr>
<td>100-50-6</td>
<td>1.2</td>
<td>5326088</td>
<td>1395417</td>
<td>1569</td>
<td>21.4</td>
<td>65163</td>
</tr>
<tr>
<td>100-50-8</td>
<td>0.6</td>
<td>1522796</td>
<td>408908</td>
<td>3506</td>
<td>45.8</td>
<td>8932</td>
</tr>
<tr>
<td>100-50-10</td>
<td>1</td>
<td>3594743</td>
<td>906559</td>
<td>524</td>
<td>10.6</td>
<td>85710</td>
</tr>
</tbody>
</table>

its best options. Our MOSP algorithm is run with “better move”, “old move” and nogood recording turned on (but no relaxation). Both solvers have a node limit of $2^{25}$ iterations.

Note that because a single move can close multiple stacks, it is possible to completely solve an instance using fewer nodes than there are customers. This occurs frequently in the high density instances. Thus the extremely low node counts shown...
Table 6.4: (a) Comparing the effects of each optimization in turn, and (b) comparing the effects of relaxation.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Better</th>
<th>Old</th>
<th>Nogood</th>
<th>No relax(ms)</th>
<th>Relax(ms)</th>
<th>Removed</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-100-2</td>
<td>25.2</td>
<td>63.7</td>
<td>1.21</td>
<td>603120</td>
<td>3790</td>
<td>51.2</td>
<td>1639.6</td>
</tr>
<tr>
<td>100-100-4</td>
<td>8.94</td>
<td>5.95</td>
<td>1.01</td>
<td>296295</td>
<td>4798</td>
<td>20.8</td>
<td>55.5</td>
</tr>
<tr>
<td>100-100-6</td>
<td>1.90</td>
<td>1.71</td>
<td>0.91</td>
<td>10344</td>
<td>3909</td>
<td>8</td>
<td>2.6</td>
</tr>
<tr>
<td>100-100-8</td>
<td>1.54</td>
<td>1.30</td>
<td>0.66</td>
<td>551</td>
<td>712</td>
<td>2.4</td>
<td>0.8</td>
</tr>
<tr>
<td>100-100-10</td>
<td>2.96</td>
<td>2.75</td>
<td>1.00</td>
<td>46</td>
<td>94</td>
<td>0.6</td>
<td>0.5</td>
</tr>
<tr>
<td>125-125-2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>59642993</td>
<td>3284</td>
<td>62.2</td>
<td>18161.8</td>
</tr>
<tr>
<td>125-125-4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>29768563</td>
<td>251634</td>
<td>24.8</td>
<td>118.3</td>
</tr>
<tr>
<td>125-125-6</td>
<td>11.9</td>
<td>3.10</td>
<td>0.98</td>
<td>810678</td>
<td>167384</td>
<td>9</td>
<td>4.8</td>
</tr>
<tr>
<td>125-125-8</td>
<td>1.65</td>
<td>1.17</td>
<td>0.98</td>
<td>18781</td>
<td>11978</td>
<td>5</td>
<td>1.6</td>
</tr>
<tr>
<td>125-125-10</td>
<td>1.27</td>
<td>0.98</td>
<td>0.64</td>
<td>768</td>
<td>1041</td>
<td>3</td>
<td>0.7</td>
</tr>
</tbody>
</table>

There are no errors. The speedup is around 2-3 orders of magnitude for the smallest problems (30×30), and around 5-6 orders of magnitude for the hardest problems that the original version can solve (40×40). The speedup appears to grow exponentially with problem size. We cannot get any speedup numbers for the harder instances since the original cannot solve them. However, given the trend in the speedup, it would not be surprising if the speedup for a full solve on the hardest instances solvable by our solver (100×100) is in the realms of 10^{10} or more.

Next we examine the effect of each of our improvements individually by disabling them one at a time. The three improvements we test here are “better move”, “old move” and nogood recording. We use only the instances which are solvable without the improvements and non-trivial, i.e., the 100×100 instances and the easier 125×125 instances. For each improvement, we show the relative slowdown compared to the version with all three optimizations on.

As Table 6.4(a) shows, both “better move” and “old move” can produce up to 1 to 2 orders of magnitude speedup on the harder instances. The lower speedups are from instances that are already fairly easy and solvable in seconds. The results from disabling the nogood recording are very interesting. It is known from previous work, e.g., the DP approach of [53] and the CP approach of [141] that nogood recording or equivalent techniques produce several orders of magnitude speedup. However, these approaches require (in the worst case) exponential memory usage for the nogood table. It appears, however, that once we have the “old move” improvement, we can
6.4. EXPERIMENTS

actually turn off nogood recording without a significant loss of performance. In fact, some instances run faster. Thus our “old move” improvement largely subsumes the effect of the huge nogood tables used in the DP [53] or CP [141] approaches and reduces the memory usage from an exponential to a linear amount. The solver of [53] uses up all 2Gb of main memory in \(\sim 5\) min with nogood recording. However, our new solver using “old move” pruning uses a constant amount of memory < 2Mb even for 125\times125 problems.

6.4.3 Relaxation

In the following set of experiments, we demonstrate the effectiveness of our relaxation technique. For each of our largest instances, we show in Table 6.4(b) the total runtime (in milliseconds) without relaxation, with relaxation, and the number of customers that was successfully removed without changing the lower bound, as well as the speedup for relaxation. Both versions are run with the customer search strategy, “better move” and “old move” improvements.

As can be seen from the results in Table 6.4(b), relaxation is most effective for sparse instances where we can get up to 3-4 orders of magnitude improvement. There is a slight slowdown for several dense instances but that is because they are trivial to begin with (take < 1s). The sparser the instance, the more customers can be removed without changing the lower bound and the greater the speedup from the relaxation technique. For the hardest instances, 125-125-2, it is often possible to remove some 60-70 of the 125 customers without changing the bound. This reduces the proof of optimality that normally takes 10+ hours into mere seconds. The 125-125-4 instances are now comparatively harder, since we are only able to remove around 25 customers and get a speedup of \(\sim 100\). Relaxation is largely ineffective for the denser instances like 125-125-8,10. However, dense instances are naturally much easier to solve anyway, so we have speedup where it is needed the most.

Our relaxation techniques are also useful if we only wish to prove a good lower bound rather than the true lower bound. For example, if we only insist on proving a lower bound that is 5 less than the true optimum, then \(\sim 45\) customers can be removed
from the 125-125-4 instances and the bound can be proved in seconds. This is again several orders of magnitude speedup compared to using a normal complete search to prove such a bound. In comparison, although the HAC lower bound heuristic of [9] uses virtually no time, it gives extremely weak lower bounds for the 125-125-4 instances, which are some 30 stacks below the optimum and are of little use.

6.5 Conclusion

In this chapter, we showed how combining nogood learning with dominance breaking can yield an MOSP solver that is some 5-6 orders of magnitude faster than the previous state of the art. One dominance relation led naturally to a customer based reformulation of the problem, yielding some 3-4 orders of magnitude speedup. This customer based search is not new, but has somehow been largely ignored in recent literature. We also proved several new dominance relations, which we called “definite move”, “better move” and “old move”. Exploiting these produced a further 1-2 orders of magnitude speedup. The “old move” improvement in particular is able to subsume the effect of pruning using an extremely large nogood table. This allows us to reduce the memory usage of our solver from an amount exponential in the size of the problem to a constant ∼2Mb. Finally we showed how relaxation techniques can be used to speed up the proof of optimality of the hardest instances by another 3-4 orders of magnitude.
Chapter 7

Talent Scheduling Problem
7.1 Introduction

The talent scheduling problem [23] can be described as follows. A film producer needs to schedule the scenes of their movie at a given location. Each scene has a duration (the number of days it takes to shoot it) and requires some subset of the cast to be on location. Each cast member is required to be on location from the day the first scene they are in is shot, to the day the last scene they are in is shot, even if they are not required for some of the days in between. Each cast member is paid a certain daily salary for each day they are on location. The problem is to order the scenes in such a way as to minimize the salary cost of the shooting.

We can formalize the problem as follows. Let $S$ be a set of scenes, $A$ a set of actors, and $a(s)$ a function that returns the set of actors involved in scene $s \in S$. Let $d(s)$ be the duration in days of scene $s \in S$, and $c(a)$ be the cost per day for actor $a \in A$. We say that actor $a \in A$ is on location at the time the scene placed in position $k$, $1 \leq k \leq |S|$ in the schedule is being shot, if there is a scene requiring $a$ scheduled before or at position $k$, and also there is a scene requiring $a$ scheduled at or after position $k$. In other words, $a$ is on location from the time the first scene $a$ is in is shot, until the time the last scene $a$ is in is shot. The talent scheduling problem aims at finding a schedule for scenes $S$ (i.e., a permutation of the scenes) that minimizes the total salary cost.

The talent scheduling problem as described in the previous paragraph is certainly an idealized version of the real problem. Real shooting schedules must contend with actor availability, setup costs for scenes, and other constraints ignored in this thesis. In addition, actors can be flown back from location mid shoot to avoid paying their holding costs for extended periods. However, the talent cost, in real situations, is a prominent feature of the movie budget [23]. Hence, concentrating on this core problem is worthwhile. Furthermore, the underlying mathematical problem has many other uses, including archaeological site ordering, concert scheduling, VLSI design and graph layout. See Section 7.6 for more discussion on this.

Example 7.1.1. Consider the talent scheduling problem defined by the set of actors $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$, the set of scenes $S = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}\}$,
7.1. INTRODUCTION

Figure 7.1: (a) An Example $a(s)$ Function: $a_i \in a(s_j)$ if the Row for $a_i$ in Column $s_j$ has an X. (b) An Example Schedule: $a_i$ is on location when scene $s_j$ is scheduled if the Row for $a_i$ in Column $s_j$ has an X or a –.

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>X</th>
<th>X</th>
<th>.</th>
<th>.</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_2$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>5</td>
</tr>
<tr>
<td>$a_3$</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>$a_4$</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>10</td>
</tr>
<tr>
<td>$a_5$</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>$a_6$</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>7</td>
</tr>
</tbody>
</table>

$\begin{array}{cccccccccccc} s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} & s_{12} \\ \hline \end{array}$


$\begin{array}{cccccccccccc} c(a) \\ \hline \end{array}$

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>X</th>
<th>X</th>
<th>.</th>
<th>.</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>.</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_2$</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>5</td>
</tr>
<tr>
<td>$a_3$</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>$a_4$</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>10</td>
</tr>
<tr>
<td>$a_5$</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>$a_6$</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>7</td>
</tr>
</tbody>
</table>

$\begin{array}{cccccccccccc} \text{cost} & 35 & 39 & 78 & 43 & 129 & 43 & 33 & 66 & 29 & 64 & 25 & 20 \\ \hline \end{array}$

$\begin{array}{cccccccccccc} \text{extra} & 0 & 20 & 28 & 34 & 84 & 13 & 24 & 10 & 0 & 10 & 0 & 0 \\ \hline \end{array}$

The $a(s)$ function determined by the matrix $M$ shown in Figure 7.1(a), where an X at position $M_{ij}$ indicates that actor $a_i$ takes part in scene $s_j$. The daily cost per actor $c(a)$ is shown in the rightmost column, and the duration of each scene $d(s)$ is shown in the last row.

Consider the schedule obtained by shooting the scenes in order $s_1s_2s_3s_4s_5s_6s_7s_8s_9s_{10}s_{11}$. The consequences of this schedule in terms of actor’s presence and cost are illustrated by the matrix $M$ shown in Figure 7.1(b), where actor $a_i$ is on location at the $j^{th}$ shot scene if the position $M_{ij}$ contains either an X ($a_i$ is in the scene) or an – ($a_i$ is waiting). The cost of each scene is shown in the second last row, being the sum of the daily costs of all actors on location multiplied by the duration of the scene. The total cost for this schedule is 604. The extra cost for each scene is shown in the last row, being the sum of the daily costs of only those actors waiting on location, multiplied by the duration of the scene. The extra cost for this schedule is 223.
The Talent Scheduling problem was introduced in [23]. In its original form each of the scenes is actually a shooting day and, hence, the duration of each of the scenes is 1. A variation of the problem, called concert scheduling [1], considers the case where the cost for each player is identical. The scene scheduling problem is known to be NP-hard [23] even if each actor appears in only two scenes, all actor costs are identical and all durations are identical.

The main contributions of this chapter are:

- We define an effective dynamic programming solution to the problem
- We define and prove correct a number of optimizations for the dynamic programming solution, that increase the size of problems we can feasibly tackle
- We show how using bounded dynamic programming can substantially improve the solving of these problems
- We show how, by considering a more accurate notion of subproblem equivalence, we can substantially improve the solving

The final code can find optimal solutions to problems larger than previous methods.

In Section 7.2 we give our call-based best-first dynamic programming formulation for the talent scheduling problem, and consider ways it can be improved by preprocessing and modifying the search. Section 7.3 examines how to solve a bounded version of the problem, which can substantially improve performance, and how to compute upper and lower bounds for the problem. Section 7.4 investigates a better search strategy where we schedule scenes from both ends of the search and Section 7.5 presents an experimental evaluation of the different approaches. In Section 7.6 we discuss related work and in Section 7.7 we conclude.

7.2 Dynamic Programming Formulation

The talent scheduling problem is naturally expressible in a dynamic programming formulation. To do so we extend the function \( a(s) \) which returns the set of actors in scene \( s \in S \), to handle a set of scenes \( Q \subseteq S \). That is, we define \( a(Q) = \bigcup_{s \in Q} a(s) \)
as a function that returns the set of actors appearing in any set of scenes \( Q \subseteq S \). Similarly, we extend the cost function \( c(a) \) to sets of actors \( G \subseteq A \) in the obvious way: \( c(G) = \sum_{a \in G} c(a) \).

Let \( l(s, Q) \) denote the set of actors on location at the time scene \( s \) is scheduled assuming that the set of scenes \( Q \subset (S\setminus\{s\}) \) is scheduled after \( s \), and the set \( S\setminus Q\setminus\{s\} \) is scheduled before \( s \). Then

\[
l(s, Q) = a(s) \cup (a(Q) \cap a(S \setminus Q \setminus \{s\})),
\]

i.e., the on locations actors are those who appear in scene \( s \), plus those who appear in both a scene scheduled after \( s \) and one scheduled before \( s \). The problem is amenable to dynamic programming because \( l(s, Q) \) does not depend on any particular order of the scenes in \( Q \) or \( S \setminus Q \setminus \{s\} \). Let \( Q \subseteq S \) denote the set of scenes still to be scheduled, and let \( schedule(Q) \) be the minimum cost required to schedule the scenes in \( Q \). Dynamic programming can be used to define \( schedule(Q) \) as:

\[
schedule(Q) = \begin{cases} 
0 & Q = \emptyset \\
\min_{s \in Q}((d(s) \times c(l(s, Q \setminus \{s\}))) + schedule(Q \setminus \{s\})) & \text{otherwise}
\end{cases}
\]

which computes, for each scene \( s \), the cost of scheduling the scene \( s \) first \( d(s) \times c(l(s, Q \setminus \{s\})) \) plus the cost of scheduling the remaining scenes \( Q \setminus \{s\} \). Dynamic programming is effective for this problem because it reduces the raw search space from \(|S|!\) to \( 2^{|S|} \), since we only need to investigate costs for each subset of \( S \) (rather than for each permutation of \( S \)).

### 7.2.1 Basic Best-first Algorithm

The code in Figure 7.2 illustrates our best-first call-based dynamic programming algorithm, which improves over a naïve formulation by pruning children that cannot yield a smaller cost.

The algorithm starts by checking whether \( Q \) is empty, in which case the cost is 0. Otherwise, it checks whether the minimum cost for \( Q \) has already been computed
schedule($Q$)
  if ($Q = \emptyset$) return 0
  if ($\text{scost}[Q]$) return $\text{scost}[Q]$
  $min := +\infty$
  $T := Q$
  while ($T \neq \emptyset$)
    $s := \text{index min}_{s \in T} \text{cost}(s, Q \setminus \{s\}) + \text{lower}(Q \setminus \{s\})$
    $T := T \setminus \{s\}$
    if ($\text{cost}(s, Q \setminus \{s\}) + \text{lower}(Q \setminus \{s\}) \geq min$) break
    $sp := \text{cost}(s, Q \setminus \{s\}) + \text{schedule}(Q \setminus \{s\})$
    if ($sp < min$) $min := sp$
  $\text{scost}[Q] := min$
  return $min$

Figure 7.2: Pseudo-code for best-first call-based dynamic programming algorithm. $\text{schedule}(Q)$ returns the minimum cost required for scheduling the set of scenes $Q$ (and stored in $\text{scost}[Q]$), in which case it returns the previously stored result (code shown in light gray). We assume the $\text{scost}$ array is initialized with zero. If not, the algorithm selects the next scene $s$ to be scheduled (using a simple heuristic that will be discussed later) and computes in $sp$ the value $\text{cost}(s, Q \setminus \{s\}) + \text{schedule}(Q \setminus \{s\})$, where function $\text{cost}(s, B)$ returns the cost of scheduling scene $s$ before any scene in $B \subset S$ (and after any scene in $S \setminus B \setminus \{s\}$), calculated as:

$$\text{cost}(s, B) = d(s) \times c(l(s, B))$$

Note, however, that the algorithm avoids (thanks to the break) considering scenes whose lower bound is greater than or equal to the current minimum $min$, since they cannot improve on the current solution. As a result, the order in which the $Q$ scenes are selected can significantly affect the amount of work performed. In our algorithm, this order is determined by a simple heuristic that selects the scene $s$ with the smallest calculated lower bound if scheduled immediately $\text{cost}(s, Q \setminus \{s\}) + \text{lower}(Q \setminus \{s\})$, where function $\text{lower}(B)$ returns a lower bound on the cost of scheduling the scenes in $B \subset S$,
7.2. DYNAMIC PROGRAMMING FORMULATION

<table>
<thead>
<tr>
<th></th>
<th>s_5</th>
<th>s_2</th>
<th>s_7</th>
<th>s_1</th>
<th>s_6</th>
<th>s_8</th>
<th>s_4</th>
<th>s_9</th>
<th>s_3</th>
<th>s_{11}</th>
<th>s_{10}</th>
<th>s_{12}</th>
<th>c(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>20</td>
</tr>
<tr>
<td>a_2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>5</td>
</tr>
<tr>
<td>a_3</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>a_4</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>10</td>
</tr>
<tr>
<td>a_5</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>a_6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>7</td>
</tr>
<tr>
<td>cost</td>
<td>45</td>
<td>19</td>
<td>19</td>
<td>39</td>
<td>39</td>
<td>66</td>
<td>29</td>
<td>29</td>
<td>50</td>
<td>25</td>
<td>54</td>
<td>20</td>
<td>434</td>
</tr>
<tr>
<td>extra</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>53</td>
</tr>
</tbody>
</table>

Figure 7.3: An optimal order for the problem of Example 7.1.1

and it is simply calculated as:

\[
\text{lower}(B) = \sum_{s \in B} d(s) \times c(a(s))
\]

which is the sum of the costs for actors that appear in each scene. The index construct \( \min_{s \in Q} e(s) \) returns the \( s \) in \( Q \) that causes the expression \( e(s) \) to take its minimum value.

A call to function \( \text{schedule}(S) \) returns the minimum cost required to schedule the scenes in \( S \). Extracting the optimal schedule found from the array of stored answers \( scost[] \) is straightforward, and standard for dynamic programming.

**Example 7.2.1.** Consider the problem of Example 7.1.1. An optimal solution is shown in Figure 7.3. The total cost is 434, and the extra cost 53. □

### 7.2.2 Preprocessing

We can simplify the problem in the following two ways:

- Eliminating single scene actors: Any actor \( a' \) that appears only in one scene \( s \) can be removed from \( s \) (i.e., we can redefine set \( a(s) \) as \( a(s) \setminus \{a'\} \) and add its fixed cost \( d(s) \times c(a') \) to the overall cost. This is correct because the cost of \( a' \) is the same independently of where \( s \) is scheduled (since \( a' \) will never have to wait while on location).
• Concatenating duplicate scenes: Any two scenes $s_1$ and $s_2$ such that $a(s_1) = a(s_2)$ can be replaced by a single scene $s$ with duration $d(s) = d(s_1) + d(s_2)$. This is correct because there is always an optimal schedule in which $s_1$ and $s_2$ are scheduled together.

Since each simplification can generate new candidates for the other kind of simplification, we need to repeatedly apply them until no new simplification is possible.

The simplification that concatenates duplicate scenes has been applied before, but not formally proved to be correct. For example, the real scene scheduling data from [23] was used in [145] with this simplification applied.

Lemma 7.2.2. If there exists $s_1$ and $s_2$ in $S$ where $a(s_1) = a(s_2)$, then there is an optimal order with $s_1$ and $s_2$ scheduled together.

Proof. Let $\Pi$ denote a possibly empty sequence of scenes. In an abuse of notation, and when clear from context, we will sometimes use sequences as if they were sets. Without loss of generality, take the order $\Pi_1s_1\Pi_2s'\Pi_3s_2\Pi_4$ of the scenes in $S$, and consider the actors on location for scene $s_1$ to be $l(s_1, \Pi_2s'\Pi_3s_2\Pi_4) = A_1$ and for scene $s_2$ to be $l(s_2, \Pi_4) = A_2$. Now, either $c(A_1) \leq c(A_2)$ (which, since $a(s_1) = a(s_2)$ means that the cost of the actors waiting in $A_1$ is smaller or equal than that of the actors waiting in $A_2$) or $c(A_1) > c(A_2)$. We will show how, in the first case, choosing the new order (a) $\Pi_1s_1s_2\Pi_2s'\Pi_3\Pi_4$ can only decrease the cost for each scene. It is symmetric to show that, in the second case, choosing the new order (b) $\Pi_1\Pi_2s'\Pi_3s_1s_2\Pi_4$ can only decrease the cost for each scene.

Let’s examine the costs of $s_1$ and $s_2$. The cost of $s_1$ does not change from the original order to that of (a) since the set of scenes before and after $s_1$ remains unchanged (i.e., since by definition $l(s_1, s_2\Pi_2s'\Pi_3\Pi_4) = l(s_1, \Pi_2s'\Pi_3s_2\Pi_4)$). The cost of $s_2$ in (a)
is the cost of the actors in

\[ l(s_2, \Pi_2 s' \Pi_3 \Pi_4) = a(s_2) \cup (a(\Pi_2 s' \Pi_3 \Pi_4) \cap a(\Pi_1 s_1)) \]

By definition of \( l(s, Q) \)

\[ = a(s_1) \cup (a(\Pi_2 s' \Pi_3 \Pi_4) \cap a(\Pi_1 s_1)) \]

By hypothesis of \( a(s_1) = a(s_2) \)

\[ = a(s_1) \cup (a(\Pi_2 s' \Pi_3 \Pi_4) \cap a(\Pi_1)) \]

By definition of \( a(Q) \)

\[ = a(s_1) \cup (a(\Pi_2 s' \Pi_3 s_2 \Pi_4) \cap a(\Pi_1)) \]

By definition of \( a(Q) \) and by hypothesis of \( a(s_1) = a(s_2) \)

\[ = l(s_1, \Pi_2 s' \Pi_3 s_2 \Pi_4) \]

By definition of \( l(s, Q) \)

which is known to be \( A_1 \). Hence, the cost of \( s_2 \) can only decrease, since \( c(A_1) \leq c(A_2) \).

Let's consider the other scenes. First, it is clear that the products in \( \Pi_1 \) and \( \Pi_4 \) have the same on location actors since the set of scenes before and after remain unchanged. Second, let us consider the changes in the on locations actors for \( s' \), which can be seen as a general representative of scenes scheduled in between \( s_1 \) and \( s_2 \) in the original order. While in the original order the set of on location actors at the time \( s' \) is scheduled is \( l(s', \Pi_3 s_2 \Pi_4) = a(s') \cup (a(\Pi_1 s_1 \Pi_2) \cap a(\Pi_3 s_2 \Pi_4)) \), in the new order the set of on location actors is \( l(s', \Pi_3 \Pi_4) = a(s') \cup (a(\Pi_1 s_1 s_2 \Pi_2) \cap a(\Pi_3 \Pi_4)) \). Clearly (a) \( a(\Pi_3 \Pi_4) \subseteq a(\Pi_3 s_2 \Pi_4) \) and (b) since \( a(s_1) = a(s_2) \), we have that \( a(\Pi_1 s_1 s_2 \Pi_2) = a(\Pi_1 s_1 \Pi_2) \). Hence, by (a) and (b) we have that \( l(s', \Pi_3 \Pi_4) \subseteq l(s', \Pi_3 s_2 \Pi_4) \), which means the set of on location actors when \( s' \) is scheduled can only decrease and, hence, the cost of scheduling it does not increase. \( \square \)

*Example 7.2.3.* Consider the scene scheduling problem from Example 7.1.1. Since actor \( a_6 \) only appears in one task, we can remove this actor and add a total of \( 2 \times 7 = 14 \) to the cost of the resulting problem to get the cost of the original problem. We also have that \( a(s_3) = a(s_{11}) \) and, after the simplification above, \( a(s_{10}) = a(s_{12}) \). Hence, we can replace these pairs by single new scenes of the combined duration. The
166  

CHAPTER 7. TALENT SCHEDULING PROBLEM

<table>
<thead>
<tr>
<th></th>
<th>s_1</th>
<th>s_2</th>
<th>s_3</th>
<th>s_4</th>
<th>s_5</th>
<th>s_6</th>
<th>s_7</th>
<th>s_8</th>
<th>s_9</th>
<th>s_10</th>
<th>c(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>a_2</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>a_3</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>a_4</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>a_5</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>a_6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>d(s)</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.4: The problem of Example 7.1.1 after preprocessing.

<table>
<thead>
<tr>
<th></th>
<th>s_12</th>
<th>s_10</th>
<th>s_11</th>
<th>s_3</th>
<th>s_5</th>
<th>s_6</th>
<th>o(Q)</th>
<th>s_1</th>
<th>s_2</th>
<th>s_9</th>
<th>s_8</th>
<th>s_7</th>
<th>s_4</th>
<th>c(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>a_2</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>a_3</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>a_4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>a_5</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>a_6</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>d(s)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.5: An optimal schedule for the scenes Q = \{s_1, s_2, s_4, s_7, s_8, s_9\} Assuming \{s_3, s_5, s_6, s_10, s_11, s_12\} have already been scheduled

resulting preprocessed problem is show in Figure 7.4. □

7.2.3 Scheduling Actor Equivalent Scenes First

Let o(Q) = a(S \ Q) \cap a(Q) be the set of on location actors just before an element of Q is scheduled, i.e., those for whom some of their scenes have already been scheduled (appear in S \ Q), and some have not (appear in Q). We can reduce the amount of search performed by the code shown in Figure 7.2 (and thus improve its efficiency) by noticing that any scene whose actors are exactly the same as those on location now can always be scheduled first without affecting the optimality of the solution. In other words, for every s ∈ Q for which a(s) = o(Q), there must be an optimal solution to schedule(Q) that starts with s.

Example 7.2.4. Consider the scene scheduling problem of Example 7.1.1. Let us
assume that the set of scenes \( Q = \{s_1, s_2, s_4, s_7, s_8, s_9\} \) is scheduled after those in \( S \setminus Q = \{s_3, s_5, s_6, s_{10}, s_{11}, s_{12}\} \) have been scheduled. Then, the set of on location actors after \( S \setminus Q \) is scheduled is \( o(Q) = \{a_1, a_2, a_4\} \) and an optimal schedule can begin with \( s_1 \) since \( a(s_1) = o(Q) \). An optimal schedule of this form is shown in Figure 7.5. □

**Lemma 7.2.5.** If there exists \( s \in Q \) where \( a(s) = o(Q) \), then there is an optimal order for schedule \( (Q) \) beginning with \( s \).

**Proof.** Let \( \Pi \) denote a possibly empty sequence of scenes. As before, we will sometimes use sequences as if they were sets. Without loss of generality, take the order \( \Pi_1 \Pi_2 s' \Pi_3 s \Pi_4 \) of the scenes in \( S \) where \( \Pi_2 s' \Pi_3 s \Pi_4 \) is the sequence of scenes in \( Q \), and consider altering the order to \( \Pi_1 s \Pi_2 s' \Pi_3 \Pi_4 \). We show that the cost for each scene in \( Q \) can only decrease.

First, it is clear that the scenes in \( \Pi_4 \) have the same on location actors since the set of scenes before and after it remains unchanged. Second, let us consider the changes in the on locations actors for \( s' \), which can be seen as a general representative of scenes scheduled before \( s \) in the original order. While in the original order the set of on location actors at the time \( s' \) is scheduled is \( l(s', \Pi_3 s \Pi_4) = a(s') \cup (a(\Pi_1 \Pi_2) \cap a(\Pi_3 s \Pi_4)) \), in the new order the set of on location actors is \( l(s', \Pi_3 \Pi_4) = a(s') \cup (a(\Pi_1 s \Pi_2) \cap a(\Pi_3 \Pi_4)) \). Now, for every set of scenes \( Q' \subseteq Q' \) we know that \( a(Q'' \subseteq a(Q'), \) i.e., increasing the number of scenes can only increase the number of actors involved. Thus, we have that (a) \( a(\Pi_3 \Pi_4) \subseteq a(\Pi_3 s \Pi_4) \), and (b) since \( a(s) = o(Q) = (a(Q) \cap a(\Pi_1)) \) we have that \( a(s) \subseteq a(\Pi_1) \), and thus that \( a(\Pi_1 s \Pi_2) = a(\Pi_1 \Pi_2) \). Hence, by (a) and (b) we have that \( l(s', \Pi_3 \Pi_4) \subseteq l(s', \Pi_3 s \Pi_4) \), which means the set of on location actors for \( s' \) in the altered schedule can only decrease and, thus, its cost cannot increase. Finally, we also have to examine the cost for \( s \). Since \( a(s) = o(Q) \) we have that \( l(s, Q \setminus \{s\}) = a(s) \). That means there is no actor waiting if we schedule \( s \) now, which is the cheapest possible way to schedule \( s \). Hence, the costs of scheduling this scene here is no more expensive than in the original position. □

We can modify the pseudo code of Figure 7.2 to take advantage of Lemma 7.2.5 by adding the line
if (∃s ∈ Q.a(s) = o(Q)) return d(s) × c(l(s, Q \ {s})) + schedule(Q \ {s})

before the line min := +∞.

7.2.4 Pairwise Subsumption

When we have two scenes s₁ and s₂ where the actors in one scene (s₁) are a subset of the actors in the other (s₂) and the extra actors a(s₂) \ a(s₁) are already on location then we can guarantee a better schedule if we always schedule s₂ before s₁. Intuitively, this is because if s₁ is shot first the missing actors would be waiting on location for scene s₂ to be shot, while if s₂ is shot first some of those missing actors might not be needed on location anymore.

Lemma 7.2.6. If there exists {s₁, s₂} ⊆ Q, such that a(s₁) ⊆ a(s₂), a(S\Q)∪(s₁) ⊇ a(s₂), then for any order of Q where s₁ appears before s₂, there is a permutation of that order where s₂ appears before s₁ with equal or lower cost.

Proof. Let Π denote a possibly empty sequence of scenes. As before, we will sometimes use sequences as if they were sets. Without loss of generality, take the order Π₁Π₂s₁Π₃s’Π₄s₂Π₅ of scenes in S where Π₂s₁Π₃s’Π₄s₂Π₅ is the sequence of scenes in Q, and consider the actors on location for scene s₁ to be l(s₁, Π₃s’Π₄s₂Π₅) = A₁ and for s₂ to be l(s₂, Π₅) = A₂. Now either c(A₁) ≤ c(A₂) or c(A₁) > c(A₂).

Case c(A₁) ≤ c(A₂): We show that choosing Π₁Π₂s₂s₁Π₃s’Π₄Π₅ as new order can only decrease the cost for each scene. The cost of s₁ in the original schedule is the cost of the actors in l(s₁, Π₃s’Π₄s₂Π₅) which is computed as a(s₁) ∪ (a(Π₃s’Π₄s₂Π₅) ∩ a(Π₁Π₂)), while for the second schedule is the cost of the actors in l(s₁, Π₃s’Π₄Π₅) which is computed as a(s₁) ∪ (a(Π₃s’Π₄Π₅) ∩ a(Π₁Π₂s₁)). Since by hypothesis a(Π₁) ∪ a(s₁) ⊇ a(s₂) and by definition a(Π₃s’Π₄Π₅) ⊆ a(Π₃s’Π₄s₂Π₅), we have that l(s₁, Π₃s’Π₄Π₅) ⊆ l(s₁, Π₃s’Π₄s₂Π₅) and, hence, the cost of s₁ can only decrease.
Regarding $s_2$, the set of actors in the new order is

$$l(s_2, s_1\Pi_3 s'\Pi_4 \Pi_5) = a(s_2) \cup (a(s_1\Pi_3 s'\Pi_4 \Pi_5) \cap a(\Pi_1\Pi_2))$$

By definition of $l(s, Q)$

$$= (a(s_2) \cup a(s_1\Pi_3 s'\Pi_4 \Pi_5)) \cap (a(s_2) \cup a(\Pi_1\Pi_2))$$

Distributing $\cup$ over $\cap$

$$= (a(s_1) \cup a(s_2\Pi_3 s'\Pi_4 \Pi_5)) \cap (a(s_2) \cup a(\Pi_1\Pi_2))$$

By definition of $a(Q)$

$$\subseteq (a(s_1) \cup a(s_2\Pi_3 s'\Pi_4 \Pi_5)) \cap (a(s_1) \cup a(\Pi_1\Pi_2))$$

By hypothesis of $a(\Pi_1) \cup a(s_1) \supseteq a(s_2)$

$$= l(s_1, \Pi_3 s' s_2 \Pi_5)$$

By definition of $l(s, Q)$

which is known to be $A_1$. Hence, the cost of $s_2$ can only decrease in the new schedule.

Let's now consider the other scenes. First, it is clear that the products in $\Pi_1 \Pi_2$ and $\Pi_5$ have the same on location actors since the set of scenes before and after remain unchanged. Second, let us consider the changes in the on locations actors for $s'$, which can be seen as a general representative of scenes scheduled in between $s_1$ and $s_2$ in the original order. While in the original order the set of on location actors at the time $s'$ is scheduled is $l(s', \Pi_4 s_2 \Pi_5) = a(s') \cup (a(\Pi_1 s_2 s_1 \Pi_3) \cap a(\Pi_4 s_2 \Pi_5))$, in the new order the set of on location actors is $l(s', \Pi_4 \Pi_5) = a(s') \cup (a(\Pi_1 s_2 s_1 s_3) \cap a(\Pi_4 s_2 \Pi_5))$, Clearly (a) $a(\Pi_4 \Pi_5) \subseteq a(\Pi_4 s_2 \Pi_5)$ and (b) since by hypothesis $a(\Pi_1) \cup a(s_1) \supseteq a(s_2)$, we have that $a(\Pi_1 s_2 s_1 s_3) \subseteq a(\Pi_1 s_2 s_1 \Pi_3)$. Hence, by (a) and (b) we have that $l(s', \Pi_4 \Pi_5) \subseteq l(s', \Pi_4 s_2 \Pi_5)$ and, hence, the cost of scheduling it cannot increase.

**Case** $c(A_1) > c(A_2)$:

We show that choosing $\Pi_1 \Pi_2 \Pi_3 s' \Pi_4 s_2 s_1 \Pi_5$ as new order can only decrease the cost for each scene.
Regarding $s_1$, the set of actors in the new order is

$$l(s_1, \Pi_5) = a(s_1) \cup (a(\Pi_5) \cap a(\Pi_1\Pi_2\Pi_3s'\Pi_4s_2))$$

By definition of $l(s, Q)$

$$= (a(s_1) \cup a(\Pi_5)) \cap (a(s_1) \cup a(\Pi_1\Pi_2\Pi_3s'\Pi_4s_2))$$

Distributing $\cup$ over $\cap$

$$= (a(s_1) \cup a(\Pi_5)) \cap (a(s_2) \cup a(\Pi_1\Pi_2s_1\Pi_3s'\Pi_4))$$

By definition of $a(Q)$

$$\subseteq (a(s_2) \cup a(\Pi_5)) \cap (a(s_2) \cup a(\Pi_1\Pi_2s_1\Pi_3s'\Pi_4))$$

By hypothesis of $a(s_1) \subseteq a(s_2)$

$$= l(s_2, \Pi_5)$$

By definition of $l(s, Q)$

which is known to be $A_2$. Hence, the cost of $s_1$ can only decrease in the new schedule.

Now since $a(s_1) \subseteq a(s_2)$ we have that $a(s_2s_1\Pi_5) = a(s_2\Pi_5)$ and, since adding scenes can only increase cost, we have that $a(\Pi_1\Pi_2\Pi_3s'\Pi_4) \subseteq a(\Pi_1s_1\Pi_2\Pi_3s'\Pi_4)$. Thus, $l(s_2, s_1\Pi_5) \subseteq l(s_2, \Pi_5)$, which means the cost of $s_2$ can only decrease.

Let's consider the other scenes. As before, it is clear that the products in $\Pi_1\Pi_2$ and $\Pi_5$ have the same on location actors since the set of scenes before and after remain unchanged. Let us then consider the changes in the on locations actors for $s'$, which can be seen as a general representative of scenes scheduled in between $s_1$ and $s_2$ in the original order. While in the original order the set of on location actors at the time $s'$ is scheduled is $l(s', \Pi_4s_2\Pi_5) = a(s') \cup (a(\Pi_1\Pi_2s_1\Pi_3) \cap a(\Pi_4s_2\Pi_5))$, in the new order the set of on location actors is $l(s', \Pi_4s_2s_1\Pi_5) = a(s') \cup (a(\Pi_1\Pi_2s_1\Pi_3) \cap a(\Pi_4s_2s_1\Pi_5))$. Clearly (a) by definition $a(\Pi_1\Pi_2s_1\Pi_3) \supseteq a(\Pi_1\Pi_2\Pi_3)$ and (b) by hypothesis of $a(s_1) \subseteq a(s_2)$ we have that $a(\Pi_4s_2s_1\Pi_5) = a(\Pi_4s_2\Pi_5)$. Hence, by (a) and (b) we have that $l(s', \Pi_4s_2s_1\Pi_5) \subseteq l(s', \Pi_4s_2\Pi_5)$, which means the set of on location actors when $s'$ is scheduled can only decrease and, hence, the cost of scheduling cannot increase. □

**Example 7.2.7.** Consider the scene scheduling problem of Example 7.1.1. Let us assume that the set of scenes $Q = S \setminus \{s_5\}$ is scheduled after $s_5$. Then, the on location
actors after \{s_5\} are \(o(Q) = \{a_2, a_4\}\). Consider \(s_1\) and \(s_6\). Since \(a(s_6) \subseteq a(s_1)\) and \(o(Q) \cup a(s_6) \supseteq a(s_1)\), \(s_1\) should be scheduled before \(s_6\). This means we should never consider scheduling \(s_6\) next! □

We can modify the pseudo code of Figure 7.2 to take advantage of Lemma 7.2.6 by adding the line

\[
\text{forall } (s_1 \in T) \\
\text{if } (\exists s_2 \in T. a(s_1) \subseteq a(s_2) \land a(S \setminus Q) \cup a(s_1) \supseteq a(s_2)) \ T := T \setminus \{s_1\}
\]

after the line \(T := Q\) and before the \textbf{while} loop. However, this is too expensive in practice. To make this efficient enough we need to precalculate the pairs \(P\) of the form \((s, s')\) where \(a(s) \subseteq a(s')\) and just check that \(s' \in T\), \(s \in T\) and \(a(S \setminus Q) \cup a(s) \supseteq a(s')\) for each pair in \(P\).

Pairwise subsumption, was first used in the solution of [145, 144], although restricted to cases where the difference in the sets is one or two elements. Although no formal proof is given, there is an extensive example in [144] explaining the reasoning for the case where the scenes differ by one element.

### 7.2.5 Optimizing Extra Cost

The \textit{base cost} of a scene scheduling problem is given by \(\sum_{s \in S} d(s) \times c(a(s))\). This is the cost for just paying for the time of the actors of the scenes they actually appear in. Instead of minimizing the total cost, we can minimize the \textit{extra cost} which is the total cost minus the base cost (i.e., the cost of paying for actors that are waiting, rather than playing). We can recover the minimal cost by simply adding the base cost to the minimal extra cost.

To do so we simply need to change the \texttt{cost} and \texttt{lower} functions used in Figure 7.2 as follows:

\[
\text{cost}(s, Q) = d(s) \times c(l(s, Q) \setminus a(s)) \\
\text{lower}(Q) = 0.
\]
The main benefit of this optimization is simply that the cost of computing the lower bounds becomes free.

### 7.3 Bounded Dynamic Programming

We can modify our problem to be a bounded problem. Let \( bnd_{\text{schedule}}(Q, U) \) be the minimal cost required to schedule scenes \( Q \) if this is less than or equal to the bound \( U \), and otherwise some number \( k \) where \( U < k \leq \text{schedule}(Q) \). We can change the dynamic program to take into account upper bounds \( U \) on a solution of interest. The recurrence equation becomes

\[
bnd_{\text{schedule}}(Q, U) = \begin{cases} 
0 & Q = \emptyset \lor U < 0 \\
\min_{s \in Q} d(s) \times c(a(s, Q \setminus \{s\}))+ & \\
\text{bnd}_{\text{schedule}}(Q \setminus \{s\}, U - d(s) \times c(a(s, Q \setminus \{s\}))), & \text{otherwise}
\end{cases}
\]

The only complexity here is that the upper bound is reduced in the recursive relation to take into account the cost of scene \( s \).

Using bounding can have two effects, one positive and one negative. On the positive side, we may be able to determine without much search that a subproblem cannot provide a better solution for the original problem, thus restricting the search. On the negative side, it may increase the search space since we have now multiplied the potential number of subproblems by the upper bound \( U \).

#### 7.3.1 Bounded Best-first Algorithm

Some of the potential subproblem explosion of adding bounds can be ameliorated since if \( \text{schedule}(Q) \leq U \) then \( bnd_{\text{schedule}}(Q, U) = \text{schedule}(Q) \) and, otherwise, \( bnd_{\text{schedule}}(Q, U) \leq \text{schedule}(Q) \) (i.e., \( bnd_{\text{schedule}}(Q, U) \) is a lower bound for \( \text{schedule}(Q) \)). Therefore, we only need to store one answer in the hash table for problem \( Q \) (rather than one per \( U \)): either the value \( \text{OPT}(v) \) indicating we have computed the optimal answer \( v \), or the value \( \text{LB}(v) \) indicating we have determined a lower bound \( v \) on the answer. We assume the hash table is initialized with entries \( \text{NONE} \) indicating no result has been stored. The only time we have to reevaluate a subproblem \( Q \) is if the stored lower bound \( v \) is less than or equal than the current \( U \).
bnd_schedule(Q, U) 
  if (Q = ∅) return 0 
  if (scost[Q] = OPT(v)) return v 
  if (scost[Q] = LB(v) ∧ v > U) return v 
  min := +∞ 
  T := Q 
  while (T ≠ ∅) 
    s := index min_{s ∈ T} cost(s, Q \ {s}) + lower(Q \ {s}) 
    T := T \ {s} 
    if (cost(s, Q \ {s}) + lower(Q \ {s}) ≥ U) break 
    sp := cost(s, Q \ {s}) + bnd_schedule(Q \ {s}, U \ cost(s, Q \ {s})) 
    if (sp < min) min := sp 
    if (min ≤ U) U := min 
    if (min ≤ U) scost[Q] := OPT(min) 
    else scost[Q] := LB(min) 
  return min

Figure 7.6: Pseudo-code for bounded best-first call-based dynamic programming algorithm. bnd_schedule(Q, U) returns the minimum cost required for scheduling the set of scenes Q if it is less than or equal to U. Otherwise it returns a lower bound on the minimal cost.

The code for the bounded dynamic program is shown in Figure 7.6. Note that the hash table handling is slightly more complex, since we can immediately return a lower bound v > U if that is stored in the hash table already.

The key advantage w.r.t. efficiency is that the break in the while loop uses the value U rather than min, since clearly no schedule beginning with s will be able to give a schedule costing less than U in this case. This requires us to update the bound U if we find a new minimum. When the search completes we have either discovered the optimal (if it is less than U), in which case we store it as optimal, or we have discovered a lower bound (> U) which we store in the hash table.

This means we can prune more subproblems. Note that this kind of addition of bounds can be automated [117].
7.3.2 Upper Bounds

Now that we are running a bounded dynamic program, we need an initial upper bound for the original problem. A trivial upper bound is the maximum possible cost, i.e., if all actors are on location at all times:

\[
\left( \sum_{s \in S} d(s) \right) \times \left( \sum_{a \in A} c(a) \right)
\]

To generate a better upper bound, we use a heuristic based on the idea that keeping expensive actors waiting around is bad. Thus, it prioritises expensive actors by attempting to keep their scenes together as much as possible (i.e., as long as this does not imply separating scenes of more expensive actors). To do this, the algorithm maintains a sequence of disjoint sets of scenes (each set corresponding to the scenes kept together for some actors) which provides a partial schedule, i.e, the scenes in a set are known to be scheduled after the scenes in any set to the left and before the scenes in any set to the right. The idea is to (a) only partition sets into smaller sets when this benefits the next actor to be processed, and (b) never to insert new scenes into the middle of the schedule (i.e., scenes are never added to an already formed set, and sets are only added at the beginning or the end of the partial schedule).

Initially the schedule is empty. And the remaining actors are \( R = A \). Then, we select the remaining actor \( a \in R \) with greatest fixed cost \( c(a) \times \left( \sum_{s \in S, a \in a(s)} d(s) \right) \), and we determine the first and last sets in the schedule involving \( a \). If the actor is currently not involved in any set, then we simply add a new set at the end of the schedule with all the scenes in which \( a \) appears. If all the scenes involving \( a \) in are in a single set, we break the set into those involving \( a \) and those not, arbitrary placing the second set afterwards. If all the scenes involving \( a \) are already scheduled, we split the first set that involves the actor into two: first those not involving the actor, and then those involving the actor. We do the same for the last set involving the actor, except that the set involving the actor goes first. This ensures that the scenes involving \( a \) are placed as close as possible without disturbing the scheduling of the previous actors.

If not all the scenes involving \( a \) are already scheduled, we first need to decide
whether to put the set of remaining scenes at the beginning or the end of the current schedule. To do this we calculate the duration for which actor \( a \) will be waiting on location if the scenes are scheduled before or after, and place their remaining scenes wherever it leads to the smallest duration. Then, if we place the scenes afterwards, we split the group where the actor \( a \) first appears into two: first those not involving the actor, and then those involving the actor. Similarly, if the remaining scenes are scheduled at the beginning we split the last group where \( a \) appears into two: first those involving the actor, and those not involving the actor.

This process continues until all actors are considered. We may have some groups which are still not singletons after this process. We order them in any way, since it cannot make a difference to the cost.

Note that this algorithm ensures that the two most expensive actors will never be waiting.

**Example 7.3.1.** Consider the scene scheduling problem from Example 7.1.1. The fixed cost of the actors \( a_1, a_2, a_3, a_4, a_5, a_6 \) are respectively 220, 55, 16, 60, 16, 14. Thus, we first schedule all scenes involving \( a_1 \) in one group \( \{s_1, s_3, s_6, s_8, s_9, s_{10}, s_{11}, s_{12}\} \). We next consider \( a_4 \), which has some scenes scheduled \( (s_1 \text{ and } s_6) \) and some not \( (s_2 \text{ and } s_5) \). Thus, we first need to decide whether to place the set \( \{s_2, s_5\} \) after or before the current schedule. Since the duration for which \( a_4 \) will be waiting on location is 0 in both cases, we follow the default (place it after) and split the already scheduled group into those not involving \( a_4 \) and those involving \( a_4 \), resulting in partial schedule \( \{s_3, s_8, s_9, s_{10}, s_{11}, s_{12}\} \{s_1, s_6\} \{s_2, s_5\} \). The total durations of the groups are 9, 2,
and 4 respectively.

We next consider $a_2$, whose scenes $\{s_4, s_7\}$ are not scheduled. The duration for $a_2$ placing these at the beginning is $9 + 2 = 11$, while placing them at the end is $2 + 4 = 6$. Thus, again we place them at the end, and split the first group, obtaining the partial schedule $\{s_8, s_{10}, s_{12}\} \{s_3, s_9, s_{11}\} \{s_1, s_6\} \{s_2, s_5\} \{s_4, s_7\}$.

We next consider $a_3$, whose scenes are all scheduled and some appear in the first and the last group. We thus split these two groups to obtain $\{s_{10}, s_{12}\} \{s_8\} \{s_3, s_9, s_{11}\} \{s_1, s_6\} \{s_2, s_5\} \{s_7\} \{s_4\}$. Then we consider $a_5$, whose scenes are also all scheduled and appear first in the second group and last in the last group. Splitting these groups has no effect since $a_5$ appears in all scenes in the group so the partial schedule is unchanged. Similarly $a_6$ only appears in one group (the first) so this is split into those containing $a_6$ and those not to obtain $\{s_{10}\} \{s_{12}\} \{s_8\} \{s_3, s_9, s_{11}\} \{s_1, s_6\} \{s_2, s_5\} \{s_7\} \{s_4\}$. The final resulting schedule is shown in Figure 7.7. □

Note that we can easily improve a heuristic solution of a scene scheduling problem by considering swapping the positions of any two pairs of scenes, and making the swap if it lowers the total cost. This heuristic method is explored in [23]. We also tried a heuristic that attempted to build the schedule from the middle by first choosing the most expensive scene and then choosing the next scene that minimizes cost to left or right. However, our experiments indicate that the upper bounds provided by any heuristic have very little effect on the overall computation of the optimal order, probably because the bind_schedule function overwrites the upper bound as soon as it finds a better solution. Hence, we did not explore many options for better heuristic solutions. Instead, we focused on devising better search strategies.

### 7.3.3 Looking Ahead

We can further reduce the search performed in bind_schedule (and schedule) by looking ahead. That is, we examine each of the subproblems we are about to visit, and if we have already calculated an optimal value or correct bound for them, we can use this to get a better estimate of the lower bound cost. Furthermore, we can use this opportunity to change the lower bound function so that it memos any lower bound
7.3. BOUNDED DYNAMIC PROGRAMMING

calculated in the hash table scost. The only modification required is to change the
definition of the lower function to:

\[
\text{lower}(Q)
\]

\[
\begin{align*}
\text{if } (\text{scost}[Q] = \text{OPT}(v)) & \quad \text{return } v \\
\text{if } (\text{scost}[Q] = \text{LB}(v)) & \quad \text{return } v \\
lb := \sum_{s \in Q} d(s) \times c(a(s)) & \quad \%\% \text{ if we are using normal costs} \\
lb := 0 & \quad \%\% \text{ if we are using extra costs} \\
\text{scost}[Q] := \text{LB}(lb) \\
\text{return } lb
\end{align*}
\]

This has the effect of giving a much better lower bound estimate and, hence, reducing
search.

Lookahead is somewhat related to the lower bounding technique used in Russian
Doll Search [150], but in that case all smaller problems are forced to be solved before
the larger problem is tackled, while lookahead is opportunistic, just using results that
are already there.

7.3.4 Better Lower Bounds

If we are storing the lower bound computations, as described in the previous sub-
section, it may be worthwhile spending more time to derive a better lower bound.
Here we describe a rather complex lower bound which is strong enough to reduce the
number of subproblems examined by 1-2 orders of magnitude. We use the following
straightforward result

**Lemma 7.3.2.** Let \( a_i, b_i, i = 1, \ldots, n \) be positive real numbers. Let \( \pi \) be a permuta-
tion of the indices. Define \( f(\pi) = \sum_{i=1}^{n} [a_{\pi(i)} \times \sum_{j=1}^{i} b_{\pi(j)}] \). The permutation \( \pi \) which
minimizes \( f(\pi) \) satisfies \( b_{\pi(1)}/a_{\pi(1)} \leq b_{\pi(2)}/a_{\pi(2)} \leq \ldots \leq b_{\pi(n)}/a_{\pi(n)} \).

This lemma allows us to solve certain special cases of the Talent Scheduling Prob-
lem with a simple sort. Consider the following special case. We have a set of actors
a_1, \ldots, a_n already on location, and a set of scenes s_1, \ldots, s_n where s_i only involves the actor a_i for each i. Then given a schedule s_{\pi(1)}s_{\pi(2)} \ldots s_{\pi(n)} where \pi is some permutation, the cost is given by \( f(\pi) = \sum_{i=1}^{n} c(a_{\pi(i)}) \times \sum_{j=1}^{i} d(s_{\pi(j)}) \). This is of the form required for Lemma 7.3.2, and we can find the optimal scene permutation \( \pi_{opt} \) simply by sorting the numbers \( d(s_i)/c(a_i) \) in ascending order. The minimum cost can then be calculated by a simple summation. Unfortunately, in general, the subproblems for which we wish to calculate lower bounds do not fall under the special case, as scenes generally involve multiple actors. To take advantage of the lemma then, we need to do much more work. The following theorem shows how we can take advantage of Lemma 7.3.2 by relaxing the general case of the Talent Scheduling Problem into the special form required.

**Theorem 7.3.3.** Let \( Q \) be a set of scenes remaining to be scheduled. Let \( A' = o(Q) \), the actors currently on location. Without loss of generality, let \( A' = \{a_1, \ldots, a_n\} \). Let \( Q' \subseteq Q \) be the set of unscheduled scenes that involve at least one actor from \( A' \). Let \( sc(s) = \sum_{a \in A' \cap a(s)} c(a) \). Let \( x(a, s) = 1 \) if \( a \in a(s) \), and 0 otherwise. Let \( w(a, s) = x(a, s) \times c(a)/sc(s) \). Let \( e(a) = \sum_{s \in Q'} w(a, s) \times d(s) \). Let \( f(\pi) = \sum_{k=1}^{n} c(a_{\pi(k)}) \times \sum_{i=1}^{k} e(a_{\pi(i)}) \). A correct lower bound on the extra cost for actors \( A' \) for scenes \( Q' \) is given by \( f(\pi_{opt}) - \sum_{s \in Q'} d(s) \times [sc(s) + \sum_{a \in A' \cap a(s)} c(a)^2/sc(s)]/2 \), where \( \pi_{opt} \) is the permutation of the indices given by sorting \( r(a_i) = e(a_i)/c(a_i) \) in ascending order.

**Proof.** First we describe what each of the defined quantities mean. \( sc(s) \) gives the sum of the cost of the actors for scene \( s \), but only counting the actors which are currently on location. \( w(a, s) \) is a measure of how much actor \( a \) is contributing to the cost of scene \( s \). We have \( 0 \leq w(a, s) \leq 1 \), and \( \sum_{a \in A' \cap a(s)} w(a, s) = 1 \). \( e(a) \) is a weighted sum of the duration of the scenes that \( a \) is involved in, weighted by \( w(a, s) \). \( f(\pi) \) is constructed so that it follows the form required for Lemma 7.3.2 to apply, which we will take advantage of. The actual lower bound is given by the minimum value of \( f(\pi) \), minus a certain constant.

Given any complete schedule that extends the current partial schedule, there is an order in which the on location actors \( a_1, \ldots, a_n \) may finish. Without loss of generality,
7.3. BOUNDED DYNAMIC PROGRAMMING

label the actors so that they finish in the order \(a_1, a_2, \ldots, a_n\) (break ties randomly). We have the following inequalities for the cost of the remaining schedule \(t(a)\) for each of these actors:

\[
t(a_k) \geq c(a_k) \times \sum_{s \in Q' | \exists i, i \leq k, a_i \in a(s)} d(s)
\]

\[
\geq c(a_k) \times \left[ \sum_{i=1}^{k} e(a_i) + \sum_{s \in Q' | a_k \in a(s)} [d(s) \times (1 - \sum_{i=1}^{k} w(a_i, s))] \right]
\]

These inequalities hold for the following reasons. Consider \(a_k\). Any scene which involves any of \(a_1, \ldots, a_k\) must be scheduled before \(a_k\) can leave, since by definition \(a_1, \ldots, a_{k-1}\) leave no later than \(a_k\). So for such scenes \(s\), we must pay \(c(a_k) \times d(s)\) for actor \(a_k\), which gives rise to the first inequality. Now, in the second line, the scene durations from the first line are split up and summed together in a different way, with some terms thrown away. The second line consists of two sums within the outer set of square brackets. A scene which does not involve any of \(a_1, \ldots, a_k\) will not be counted in any \(e(a)\) in the first sum, and is not counted by the second sum which only counts scenes involving \(a_k\). So as required, such durations do not appear in the second line. A scene which involves some of \(a_1, \ldots, a_k\) will have part of its duration counted in the first sum. To be exact, a proportion \(\sum_{i=1}^{k} w(a_i, s) \leq 1\) of it is counted in the first sum. The second sum counts the bits that were not counted in the first sum for scenes that involve \(a_k\). Since the second line never counts more than \(d(s)\) for any scene appearing in the first line, the inequality is valid.

Now, we split the last line of the inequality into its two parts and sum over the actors. Define \(U\) and \(V\) as follows:

\[
U = \sum_{k=1}^{n} c(a_k) \times \sum_{i=1}^{k} e(a_i)
\]

\[
V = \sum_{k=1}^{n} c(a_k) \times \sum_{s \in Q' | a_k \in a(s)} [d(s) \times (1 - \sum_{i=1}^{k} w(a_i, s))]
\]

Then \(U + V\) is a lower bound on the cost for the actor finish order \(a_1, \ldots, a_n\). As can be seen, \(U\) corresponds to \(f(\pi)\) in the theorem. Different permutations of actor finish order \(\pi\) will give rise to different values of \(U\) equal to \(f(\pi)\). By applying Lemma 7.3.2, we can quickly find a lower bound on \(U\) over all possible actor finish orders. That is, for each actor \(a\), we calculate \(r(a) = e(a)/c(a)\). We then sort the actors based on \(r(a)\) from smallest to largest and label them from \(a'_1\) to \(a'_n\). We then calculate \(U\) using finish order \(a'_1, \ldots, a'_n\), which will give us a lower bound on \(U\) over...
all possible actor finish orders.

\[ V, \] on the other hand, although it looks like it depends on the actor finish order, actually evaluates to a constant.

\[ V = \sum_{k=1}^{n} c(a_k) \times \sum_{s \in Q'} d(s) \times (1 - \sum_{i=1}^{k} w(a_i, s)) \]

\[ = \sum_{k=1}^{n} \sum_{s \in Q'} c(a_k) \times [d(s) \times (1 - \sum_{i=1}^{k} w(a_i, s)) + d(s)] \]

\[ = \sum_{s \in Q'} \sum_{k=1}^{n} c(a_k) \times [d(s) \times (1 - \sum_{i=1}^{k} w(a_i, s)) + d(s)] \]

\[ = \sum_{s \in Q'} \sum_{k=1}^{n} c(a_k) \times d(s) - \sum_{s \in Q'} \sum_{k=1}^{n} c(a_k) \times d(s) \times \sum_{i=1}^{k} w(a_i, s) \]

\[ = \sum_{s \in Q'} \sum_{k=1}^{n} c(a_k) \times d(s) - \sum_{s \in Q'} d(s) \times \sum_{k=1}^{n} c(a_k) \times \sum_{i=1}^{k} w(a_i, s) \]

The first double sum is simply the base cost needed to pay each actor for each scene they appear in, and is clearly a constant. Of the second term, only the innermost double sum may be dependent on the actor finish order. Let \( W(s) = \sum_{i=1}^{n} c(a_k) \times c(a_i) \).

\[ 2 \times W(s) = 2 \times \sum_{k=1}^{n} c(a_k) \times \sum_{i=1}^{n} c(a_i) \]

\[ = \sum_{k=1}^{n} c(a_k) \times c(a_i) + \sum_{k=1}^{n} c(a_k) \times c(a_i) \]

\[ = \sum_{k=1}^{n} c(a_k) \times c(a_i) + \sum_{k=1}^{n} c(a_k) \times c(a_i) \]

\[ = \sum_{k=1}^{n} c(a_k) \times c(a_i) + \sum_{k=1}^{n} c(a_k) \times c(a_i) \]

\[ = \sum_{k=1}^{n} c(a_k)^2 \]

\[ W(s) = \frac{[\sum_{k=1}^{n} c(a_k)^2]}{2} \]

which is constant. Now, \( U + V \) gives a lower bound for the total cost. A lower bound for the extra cost is simply \( U + V \) minus the base cost of the actors \( A' \) for the scenes \( Q' \). Luckily, this term already appears as the first term in \( V \). Thus the lower bound for the extra cost is \( f(\pi_{opt}) - \sum_{s \in Q'} d(s) \times W(s) = f(\pi_{opt}) - \sum_{s \in Q'} d(s) \times [\sum_{a \in A' \cap \alpha(a)} c(a)^2 / s(c)] / 2 \) as claimed. \( \square \)

**Example 7.3.4.** Consider the scenes shown in Figure 7.8, where the cost/duration of each actor/scene is 1 for simplicity. To calculate \( f(\pi_{opt}) \), we need to calculate \( r(a) \) and sort them. Since the costs are all 1, we have \( r(a_1) = e(a_1) = 11/6, r(a_2) = e(a_2) = 2, r(a_3) = e(a_3) = 11/6, r(a_4) = e(a_4) = 4/3 \). So we reorder the actors as \( a'_1 = a_4, a'_2 = a_1, a'_3 = a_3, a'_4 = a_2 \) and calculate \( f(\pi) \) using finish order \( a'_1, \ldots, a'_4 \), which gives

\[ f(\pi_{opt}) = 1 \times 4/3 + 1 \times (4/3 + 11/6) + 1 \times (4/3 + 11/6 + 11/6) + 1 \times (4/3 + 11/6 + 11/6 + 2) = \]
7.3.  BOUNDED DYNAMIC PROGRAMMING

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
<th>$s_6$</th>
<th>$s_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>X</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$a_2$</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$a_3$</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>$a_4$</td>
<td>.</td>
<td>.</td>
<td>X</td>
<td>.</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

1. If we are optimizing the extra cost (Section 7.2.5), then to implement this lower bound, we simply need to add the following code into the code for `lower` before the saving of the lower bound in `scost`.

   ```
   A' := o(Q)
   for (a ∈ A')
       r[a] := 0
   for (s ∈ Q')
       a'(s) = a(s) ∩ A'
       total_cost := \sum_{i ∈ a'(s)} c(i)
       total_cost_sq := \sum_{i ∈ a'(s)} c(i)^2
       for (a ∈ a'(s))
           r[a] = r[a] + d(s)/total_cost
       lb = lb - d(s) × (total_cost + total_cost_sq/total_cost)/2
   Sort A' based on r[a] in ascending order
   c := \sum_{i ∈ A'} c(i)
   for (a ∈ A')
       lb = lb + c × r[a] × c(a)
   c = c - c(a)
   ```

Figure 7.8: An original set of remaining scenes, assuming $a_1, a_2, a_3, a_4$ are on location.

16.5, which is a lower bound on $U$ over all actor finish orders. Next, we calculate $\sum_{s ∈ Q} d(s) \times [sc(s) + \sum_{a ∈ A' \cap a(s)} c(a)^2/sc(s)]]/2 = 1 + 1 + 1 + 1 + 3/2 + 3/2 + 2 = 9$. Thus the lower bound for the extra cost at this node is $16.5 - 9 = 7.5$. □
Clearly this is quite an expensive calculation.

### 7.4 Double Ended Search

We will say an actor is *fixed* if we know the first and last scene where the actor appears. Knowing that an actor is fixed is useful, because the cost for that actor is fixed (thus the name) regardless of the schedule of the remaining intervening scenes, if any. For this reason it is beneficial to search for a solution by alternatively placing the next scene in the first remaining unfilled slot and the last remaining unfilled slot, since this will increase the number of fixed actors. Let $B$ denote the set of scenes scheduled at the beginning, and $E$ the set of scenes scheduled at the end. We know the cost of any actor appearing both in scenes of $B$ and scenes of $E$, since we know the duration of the remaining set of scenes $Q = S \setminus B \setminus E$. This strategy was used in the branch-and-bound solution of [23]. A priori this might appear to be a bad strategy since the search space has increased: there are more subproblems of the form “schedule remaining scenes $Q$ given scenes in $B$ are scheduled before and scenes in $E$ are scheduled after (where $B \cup Q \cup E = S$)”, than there are “schedule remaining scenes $Q$ given scenes in $S \setminus Q$ are scheduled before”. However, as we will see in the experiments, this is compensated by the fact that we will get much more accurate estimates on the cost of the remaining schedule.

The change in search strategy causes considerable changes to the algorithm. The sub problems are now defined by $B$ the set of scenes scheduled at the beginning, and $E$ the set of scenes scheduled at the end. The search tries to schedule each remaining scene $s$ at the beginning of the remaining scenes, just after $B$, and the swaps the role of $B$ and $E$ to continue building the schedule. We can thus modify the cost function to ignore the cost of actors already fixed by $B$ and $E$ (i.e., those in $a(B) \cap a(E)$), and take only into account the cost of actors newly fixed by the scene. This can be done as follows:

$$
\text{cost}(s, B, E) = d(s) \times c(l(s, S \setminus B \setminus E \setminus \{s\}) \setminus (a(B) \cap a(E))) \\
+ \sum_{a \in ((a(s) \setminus a(B)) \cap a(E))} d(S \setminus B \setminus E \setminus \{s\}) \times c(a)
$$
7.4. DOUBLE ENDED SEARCH

\[ Q := S \setminus B \setminus E \]
\[ \text{if } (a(Q) \subseteq a(B) \cap a(E)) \text{ return 0} \]
\[ hv := \text{hash lookup}(B, E) \]
\[ \text{if } (hv = OPT(v)) \text{ return } v \]
\[ \text{if } (hv = LB(v) \land v > U) \text{ return } v \]
\[ min := +\infty \]
\[ T := Q \]
\[ \text{while } (T \neq \emptyset) \]
\[ s := \text{index min cost}(s, B, E) + \text{lower}(B \cup \{s\}, E) \]
\[ T := T \setminus \{s\} \]
\[ \text{if } (\text{cost}(s, B, E) + \text{lower}(B \cup \{s\}, E) \geq U) \text{ break} \]
\[ sp := \text{cost}(s, B, E) + \text{bnd_de_schedule}(E, B \cup \{s\}, U - \text{cost}(s, B, E)) \]
\[ \text{if } (sp < min) min := sp \]
\[ \text{if } (min \leq U) U := min \]
\[ \text{if } (min \leq U) \text{ hash_set}(B, E, OPT(min)) \]
\[ \text{else hash_set}(B, E, LB(min)) \]
\[ \text{return } min \]

Figure 7.9: Pseudo-code for bounded best-first call-based dynamic programming algorithm. \texttt{bnd_de_schedule}(Q, B, E) returns the minimum cost required for scheduling the set of scenes \( Q \) if it is less than or equal to \( U \). Otherwise it returns a lower bound on the minimal cost

where the first part adds the cost for scheduling scene \( s \) excluding the fixed actors \( (a(B) \cap a(E)) \), and the second part adds the cost of each actor \( a \) which is newly scheduled by \( s \) (appears in \( (a(s) \setminus a(B)) \)) and already scheduled at the end (appears in \( a(E)) \).

The lower bound cost function also has to change to ignore the actors fixed by \( B \) and \( E \):

\[ \text{lower}(B, E) = \sum_{s \in S \setminus B \setminus E} d(s) \times c(a(s) \setminus (a(B) \cap a(E))) \]

The code for the new algorithm is shown in Figure 7.9. The algorithm first tests whether there are any remaining actors to be scheduled: If \( a(Q) \subseteq a(B) \cap a(E) \)
then all actors playing in scenes of $Q$ are fixed (must be on location for the entire period regardless of $Q$ schedule), and we simply return 0 (since their cost has already been taken into account). Otherwise, the algorithm checks the hash table to find whether the subproblem has been examined before. Note that we replaced the array of subproblems $scost$ by two functions $\text{hash\_lookup}(B, E)$, which returns the value stored for subproblem $B, E$, and $\text{hash\_lookup}(B, E, ov)$, which sets the stored value to $ov$. The remainder of the code is effectively identical to $\text{bnd\_schedule}$ using the new definitions of $\text{cost}$ and $\text{lower}$. The only important thing to note is that the recursive call swaps the positions of beginning and end sets, thus forcing the next scene to be scheduled at the other end.

Note that any solution to the scene scheduling problem has an equivalent solution where the order of the scenes is reversed (a fact that has been noticed by many authors). We are implicitly using this fact in the definition of $\text{bnd\_de\_schedule}$ when we reverse the order of the $B$ and $E$ arguments to make the search double ended, since we treat the problem starting with $B$ and ending in $E$ as equivalent to the problem starting with $E$ and ending in $B$. We can also take advantage of this symmetry when detecting equivalent subproblems (i.e. when looking up whether we have seen the problem before). A simple way of achieving this is to store and lookup problems assuming that $B \leq E$ (that is, in lexicographic order).

\[
\begin{align*}
\text{hash\_lookup}(B, E) &= \begin{cases} 
\text{if } (B \leq E) \text{ scost}[B, E] & \text{else scost}[E, B] 
\end{cases} \\
\text{hash\_set}(B, E, ov) &= \begin{cases} 
\text{if } (B \leq E) \text{ scost}[B, E] := ov & \text{else scost}[E, B] := ov 
\end{cases}
\end{align*}
\]

### 7.4.1 Better Equivalent Subproblem Detection

While taking into account symmetries helps, we can further help the detection of equivalent subproblems by noticing that the cost of scheduling the scenes in $Q = S \setminus B \setminus E$ does not really depend on $B$ and $E$. Rather, it depends on $o(B)$ and $o(E)$, i.e., on the set of actors that will always be on location at the beginning and at the end of $Q$, respectively.

**Example 7.4.1.** Consider the partial schedule of the problem of Example 7.1.1 where
7.4. **DOUBLE ENDED SEARCH**

$B = \{s_1, s_9, s_{12}\}$ and $E = \{s_3, s_5, s_6, s_{11}\}$. The remaining scenes to schedule are $Q = \{s_2, s_4, s_7, s_8, s_9, s_{10}\}$. An optimal schedule of $Q$ (given $B$ and $E$) is shown at the top of in Figure 7.10. The total cost ignoring the fixed actors $a_1$, $a_2$ and $a_4$ is $16 + 8 + 14 = 48$.

Consider the subproblem where $B' = \{s_3, s_{11}, s_5, s_1\}$ and $E' = \{s_9, s_6, s_{12}\}$. The remaining scenes to schedule are still $Q = \{s_2, s_4, s_7, s_8, s_9, s_{10}\}$. Now $o(B') = o(E)$ and $o(E') = o(B)$ and hence any optimal order for the first subproblem can provide an optimal schedule for this subproblem, by reversing the order of the schedule. This is illustrated at the bottom of Figure 7.10. □

We can modify the hash function to take advantage of these subproblem equivalences. We will store the subproblem value on $o(B)$, $o(E)$, and $Q$ under the assumption that $o(B) \leq o(E)$.

\[
\text{hash\_lookup}(B, E) = Q := S \setminus B \setminus E
\]
if $(o(E) < o(B))$ return $scost[o(E), o(B), Q]$
return $scost[o(B), o(E), Q]$

hash_set$(B, E, ov)$

$Q := S \setminus B \setminus E$

if $(o(E) < o(B))$ $scost[o(E), o(B), Q] := ov$
else $scost[o(B), o(E), Q] := ov$

In order to prove the correctness of the equivalence we need the following intermediate result.

**Lemma 7.4.2.** For every $Q, Q' \subseteq S$ such that $Q \cap Q' = \emptyset$ (which is the same as saying $Q' \subseteq S \setminus Q$), we have that $a(Q) \cap a(Q') = o(Q) \cap a(Q') = a(Q') \cap o(Q') = o(Q) \cap o(Q').$

**Proof.** Let us first prove that $o(Q) \cap a(Q') = a(Q) \cap a(Q').$ We have that:

\[
o(Q) \cap a(Q') = (a(Q) \cap a(S \setminus Q)) \cap a(Q')
\]

By definition of $o(Q)$

\[
= a(Q) \cap (a(S \setminus Q) \cap a(Q'))
\]

By associativity of $\cap$

\[
= a(Q) \cap a(Q')
\]

By hypothesis of $Q' \subseteq S \setminus Q$

A symmetric reasoning can be done to prove that $a(Q) \cap o(Q') = a(Q) \cap a(Q').$

To prove that $o(Q) \cap o(Q') = a(Q) \cap a(Q')$ we follow a similar reasoning:

\[
o(Q) \cap o(Q') = (a(Q) \cap a(S \setminus Q)) \cap (a(Q') \cap a(S \setminus Q'))
\]

By definition of $o(Q)$

\[
= (a(Q) \cap a(S \setminus Q')) \cap (a(S \setminus Q) \cap a(Q'))
\]

By associativity of $\cap$

\[
= a(Q) \cap a(Q')
\]

By hypothesis of $Q' \subseteq S \setminus Q$ and $Q \subseteq S \setminus Q'$
Theorem 7.4.3. Let $\Pi_1\Pi_2\Pi_3$ and $\Pi_4\Pi_2\Pi_5$ be two permutations of $S$ such that $o(\Pi_4) = o(\Pi_1)$, $o(\Pi_5) = o(\Pi_3)$. Then, the cost of every scene of $\Pi_2$ is the same in $\Pi_1\Pi_2\Pi_3$ as in $\Pi_4\Pi_2\Pi_5$.

Proof. Without loss of generality, let $\Pi_2$ be of the form $\Pi'_2s\Pi''_2$. We will show that the cost of $s$ is the same in $\Pi_1\Pi_2\Pi_3$ and $\Pi_4\Pi_2\Pi_5$. Now

$$l(s, \Pi'_2\Pi_3) = a(s) \cup (a(\Pi''_2\Pi_3) \cap a(\Pi_1\Pi'_2))$$

By definition of $l(s, Q)$

$$= a(s) \cup ((a(\Pi''_2) \cup a(\Pi_3)) \cap (a(\Pi_1) \cup a(\Pi'_2)))$$

By definition of $a(Q)$

$$= a(s) \cup ((a(\Pi''_2) \cap a(\Pi_1)) \cup (a(\Pi''_2) \cap a(\Pi'_2)) \cup (a(\Pi_3) \cap a(\Pi_1)) \cup (a(\Pi_3) \cup a(\Pi'_2)))$$

Distributing $\cap$ over $\cup$

$$= a(s) \cup ((a(\Pi''_2) \cap o(\Pi_1)) \cup (a(\Pi''_2) \cap a(\Pi'_2)) \cup (o(\Pi_3) \cap a(\Pi_1)) \cup (o(\Pi_3) \cup a(\Pi'_2)))$$

by the Lemma 7.4.2

$$= a(s) \cup ((a(\Pi''_2) \cap a(\Pi_4)) \cup (a(\Pi''_2) \cap a(\Pi'_2)) \cup (o(\Pi_5) \cap a(\Pi_4)) \cup (o(\Pi_5) \cup a(\Pi'_2)))$$

Since $o(\Pi_4) = o(\Pi_1)$ and $o(\Pi_5) = o(\Pi_3)$

$$= a(s) \cup ((a(\Pi''_2) \cap a(\Pi_4)) \cup (a(\Pi''_2) \cap a(\Pi'_2)) \cup (a(\Pi_5) \cap a(\Pi_4)) \cup (a(\Pi_5) \cup a(\Pi'_2)))$$

$$= a(s) \cup ((a(\Pi''_2) \cup a(\Pi_5)) \cap (a(\Pi_4) \cup a(\Pi'_2)))$$

$$= l(s, \Pi'_2\Pi_5)$$
may be able to improve the previous optimizations by ignoring fixed actors whenever performing a double ended search.

**Preprocessing**

The second preprocessing step (concatenating duplicate scenes) can now be applied during search. This is because, given fixed actors $F = a(B) \cap a(E)$, we can apply Lemma 7.2.2 if $a(s_1) \cup F = a(s_2) \cup F$, since the cost of the fixed actors is irrelevant. This means we should concatenate any scenes in $Q = S \setminus B \setminus E$ where $a(s_1) \cup F = a(s_2) \cup F$. We can modify the search strategy in `bnd_de_schedule` to break the scenes in $Q$ into equivalent classes $Q_1, \ldots Q_n$ where $\forall s_1, s_2 \in Q_i, a(s_1) \cup F = a(s_2) \cup F$, and then consider scheduling each equivalence class. In many cases the equivalence class will be of size one!

**Scheduling Actor Equivalent Scenes First**

Lemma 7.2.5 can be extended so that we can always schedule a scene $s$ first where $o(B) = a(s) \cup F$ since the on location actors will include the fixed actors and the extra cost for them will be payed for scene $s$ wherever it is scheduled.

**Pairwise Subsumption**

The extension of Lemma 7.2.6 also holds if $a(s_1) \cup F \subseteq a(s_2) \cup F$ and $a(B) \cup a(s_1) \supseteq a(s_2)$ (since $F \subseteq a(B)$). But this means we need to do a full pairwise comparison of all scenes in $Q = S \setminus B \setminus E$, for each subproblem considered. We did implement this, and although it did cut down search substantially, the overhead of the extra comparison did not pay off. This is the only optimization not used in the experimental evaluation.

**Optimizing Extra Cost**

This is clearly applicable in the doubled ended case, but it complicates the computation of $\text{cost}(s, B, E)$ since we now have to determine exactly which scenes a newly fixed actor occurs in, rather than just adding the cost of the actor for the entire duration of the remaining scenes.
Looking Ahead

This is applicable as before. Note that lower takes the same arguments \((B, E)\) (excluding the upper bound) as bnd_de_schedule. We have to modify the definition of lower\((B, E)\) to make use of hash_lookup and hash_set.

Better Lower Bounds

The same reasoning on better lower bounds can be applied to the set of actors \(o(B) \setminus F\), since the actors in \(F\) will always be on location in the remaining subproblem.

Indeed, we sum the results of the better lower bounds calculated from both ends for \(o(B) \setminus F\) and \(o(E) \setminus F\), since the actors in these sets cannot overlap (by the definition of \(F\)).

Better Equivalent Subproblem Detection

We could improve equivalent subproblem detection by noticing that the fixed actors play no part in determining the schedule of the remaining scenes \(Q = S \setminus B \setminus E\). We could thus build a hash function based on the form of the remaining scenes after eliminating the fixed actors \(F = a(B) \cap a(E)\). But the cost of determining this reduced form seems substantial since, in effect, we have to generate new scenes and hash on sets of them. We have not attempted to implement this approach.

7.5 Experiments

We tested our approach on the two sets of problem instances detailed below. All experiments were run on Xeon Pro 2.4GHz processors with 2GB RAM running Red Hat Linux 5.2. The dynamic programming code is written in C, with no great tuning or clever data structures, and many runtime flags to allow us to compare the different versions easily. The dynamic programming software was compiled with gcc 4.1.2 using -O3. Timings are calculated as the sum of user and system time given by getrusage, since it accords well with wall-clock times for these CPU-intensive programs. For
the problems that take significant time we observed around 10% variation in timings across different runs of the same benchmark.

7.5.1 Structured Benchmarks

The first set of benchmarks are structured problems based on the realistic talent scheduling of Mob Story first used in [23]. We use these problems to illustrate the effectiveness of the different optimizations.

We first extended the benchmarks film103, film105, film114, film116, film118, film119 used in [145], adding three new actors to each problem to bring it to 11, and bringing the number of scenes to 28 (the original problems each involve 8 actors and either 18 or 19 scenes). This gave us 6 base problems of size 11 × 28. These base problems were constructed in such a way that preprocessing did not simplify them (so that the number of “important” actors and scenes was known).

Then, from each base problem we generated smaller problems by removing in turn newly added scenes. In particular, for each base problem we obtained 10 problems ranging from 11 × 27 to 8 × 18, where each problem in the sequence is a subproblem of the larger ones, and the original problem from [145] was included.
Table 7.2: Arithmetic mean subproblems solved for structured problems of size \( n \), and relative increase if the optimization is turned off

<table>
<thead>
<tr>
<th>( n )</th>
<th>Subproblems</th>
<th>7.2.3</th>
<th>7.2.4</th>
<th>7.3.3</th>
<th>7.4.2</th>
<th>7.3.2</th>
<th>7.4.1</th>
<th>7.4</th>
<th>7.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>5091</td>
<td>1.08</td>
<td>1.18</td>
<td>1.02</td>
<td>1.06</td>
<td>10.21</td>
<td>1.00</td>
<td>1.07</td>
<td>0.93</td>
</tr>
<tr>
<td>19</td>
<td>9699</td>
<td>1.12</td>
<td>1.22</td>
<td>1.03</td>
<td>1.09</td>
<td>10.01</td>
<td>11.18</td>
<td>0.99</td>
<td>1.11</td>
</tr>
<tr>
<td>20</td>
<td>10467</td>
<td>1.11</td>
<td>1.26</td>
<td>1.02</td>
<td>1.07</td>
<td>12.36</td>
<td>1.00</td>
<td>1.09</td>
<td>0.95</td>
</tr>
<tr>
<td>21</td>
<td>16154</td>
<td>1.15</td>
<td>1.21</td>
<td>1.02</td>
<td>1.07</td>
<td>15.88</td>
<td>1.00</td>
<td>1.14</td>
<td>1.24</td>
</tr>
<tr>
<td>22</td>
<td>36531</td>
<td>1.17</td>
<td>1.23</td>
<td>1.03</td>
<td>1.12</td>
<td>13.38</td>
<td>0.99</td>
<td>1.21</td>
<td>1.97</td>
</tr>
<tr>
<td>23</td>
<td>42224</td>
<td>1.17</td>
<td>1.32</td>
<td>1.03</td>
<td>1.10</td>
<td>15.73</td>
<td>1.00</td>
<td>1.13</td>
<td>1.52</td>
</tr>
<tr>
<td>24</td>
<td>59349</td>
<td>1.22</td>
<td>1.33</td>
<td>1.02</td>
<td>1.21</td>
<td>18.50</td>
<td>0.98</td>
<td>1.16</td>
<td>1.98</td>
</tr>
<tr>
<td>25</td>
<td>77766</td>
<td>1.25</td>
<td>1.37</td>
<td>1.03</td>
<td>1.17</td>
<td>19.00</td>
<td>0.98</td>
<td>1.19</td>
<td>2.08</td>
</tr>
<tr>
<td>26</td>
<td>136778</td>
<td>1.28</td>
<td>1.40</td>
<td>1.03</td>
<td>1.23</td>
<td>20.49</td>
<td>1.00</td>
<td>1.20</td>
<td>2.51</td>
</tr>
<tr>
<td>27</td>
<td>167232</td>
<td>1.28</td>
<td>1.50</td>
<td>1.03</td>
<td>1.25</td>
<td>23.63</td>
<td>0.99</td>
<td>1.19</td>
<td>2.70</td>
</tr>
<tr>
<td>28</td>
<td>233328</td>
<td>1.31</td>
<td>1.56</td>
<td>1.03</td>
<td>1.27</td>
<td>25.04</td>
<td>0.99</td>
<td>1.19</td>
<td>2.88</td>
</tr>
</tbody>
</table>

From each base problem we also generated smaller problems by randomly removing \( k \) scenes where \( k \) varied from 1 to 10. In particular, for each base problem we obtained 10 problems ranging from \( 11 \times 27 \) to \( 11 \times 18 \), where the sets of removed scenes in differently sized problem are unrelated (as opposed to have a subset/superset relationship).

In total this created 126 core problems.

From each core problem we generated three new variants: equal duration where all durations are set to 1, equal cost where the cost of all actors are set to 1, and equal cost and duration where all durations and costs are set to 1.

We compared the executions of running the dynamic program with all optimizations enabled, and individually turning off each optimization. The optimizations are: scheduling actor equivalent scenes first (Section 7.2.3), pairwise subsumption (Section 7.2.4), looking ahead (Section 7.3.3), concatenating duplicate scenes (Section 7.4.2), upper bounds (Section 7.3.2), better lower bounds (Section 7.3.4), optimizing extra cost (Section 7.2.5), better equivalent subproblem detection (Section 7.4.1), double-ended search (Section 7.4), and bounded dynamic programming (Section 7.3). The average times in milliseconds obtained by running the dynamic program with all optimizations enabled for each size \( n \), are shown in the second column of Table 7.1.
The remaining columns show the relative average time when each of the optimizations is individually turned off. For the last column without bounding, only the problems up to size 22 are shown. Table 7.2 shows the same results in terms of the number of subproblems solved (that is, the number of pairs \((B, E)\) appearing in calls to \text{bnd\_de\_schedule} or \(Q\) in earlier variants).

The tables clearly show that bounded dynamic programming (Section 7.3) is indispensable for solving these problems. Better lower bounding is clearly the next most important optimization, massively reducing the number of subproblems visited. Doubled-ended search (Section 7.4) is also very important except for the fact that better lower bounding (Section 7.3.4) improves the single-ended search much more than it does the double-ended search, so only on the larger examples does it begin to win. Without better lower bounding it completely dominates single-ended search. The next most effective optimization is pairwise subsumption (Section 7.2.4). Looking ahead (Section 7.3.3) and scheduling actor equivalent scenes first (Section 7.2.3) are quite beneficial, as are optimizing extra cost (Section 7.2.5) and better equivalent subproblem detection (Section 7.4.1). The upper bounds optimization (Section 7.3.2) is clearly unimportant, only reducing the number of problems slightly. Note that while some optimizations given more or less constant improvements with increasing size, most are better as size increases.

If we look at the different variants individually (in results not shown) we find that the equal duration variants are slightly (around 7-10%) harder than the core problems, while the equal cost and equal cost and duration variants are 3–4 times harder than the core problems, indicating that cost is very important for pruning.

### 7.5.2 Random Benchmarks

The second set of benchmarks is composed of randomly generated benchmarks. We use these problems to show the effect of number of actors and number of scenes on problem difficulty.

The problems were generated in a manner almost identical to that used in [23]: for a given combination of \(m\) actors and \(n\) scenes we generate for each actor \(i \in \{1, \ldots, m\}\)
7.5. EXPERIMENTS

Table 7.3: Arithmetic mean solving time (ms) for random problems with \( m \) actors and \( n \) scenes.

<table>
<thead>
<tr>
<th>( m )</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>22</th>
<th>24</th>
<th>26</th>
<th>28</th>
<th>30</th>
<th>32</th>
<th>34</th>
<th>36</th>
<th>38</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>20</td>
<td>39</td>
<td>94</td>
<td>141</td>
<td>233</td>
<td>362</td>
<td>685</td>
<td>1365</td>
<td>2291</td>
<td>2977</td>
<td>2408</td>
<td>7101</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>33</td>
<td>85</td>
<td>165</td>
<td>441</td>
<td>650</td>
<td>1981</td>
<td>2531</td>
<td>3179</td>
<td>8901</td>
<td>10690</td>
<td>13426</td>
<td>20907</td>
</tr>
<tr>
<td>12</td>
<td>21</td>
<td>47</td>
<td>149</td>
<td>319</td>
<td>829</td>
<td>2056</td>
<td>3830</td>
<td>6674</td>
<td>10082</td>
<td>13155</td>
<td>20903</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>25</td>
<td>75</td>
<td>255</td>
<td>759</td>
<td>1519</td>
<td>3700</td>
<td>6504</td>
<td>12705</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>41</td>
<td>129</td>
<td>357</td>
<td>1012</td>
<td>3190</td>
<td>8290</td>
<td>20560</td>
<td>55142</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>53</td>
<td>221</td>
<td>535</td>
<td>1463</td>
<td>4412</td>
<td>10904</td>
<td>26240</td>
<td>65042</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>87</td>
<td>248</td>
<td>757</td>
<td>2745</td>
<td>6680</td>
<td>15414</td>
<td>21194</td>
<td>—</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>119</td>
<td>338</td>
<td>997</td>
<td>2855</td>
<td>7499</td>
<td>18672</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.4: Arithmetic mean subproblems solved for random problems with \( m \) actors and \( n \) scenes.

<table>
<thead>
<tr>
<th>( m )</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>22</th>
<th>24</th>
<th>26</th>
<th>28</th>
<th>30</th>
<th>32</th>
<th>34</th>
<th>36</th>
<th>38</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7697</td>
<td>15669</td>
<td>19004</td>
<td>21703</td>
<td>23939</td>
<td>25891</td>
<td>49547</td>
<td>42433</td>
<td>49406</td>
<td>61351</td>
<td>62089</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10</td>
<td>25903</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

(a) a random number\(^1\) \( n_i \in \{2, \ldots, n\} \) indicating the number of scenes actor \( i \) is in, (b) \( n_i \) different random numbers between 1 and \( n \) indicating the set of scenes actor \( i \) is in, and (c) a random number between 1 and 100 indicating the cost of actor \( i \). For each combination of actors \( m \in \{8, 10, 12, 14, 16, 18, 20, 22\} \) and scenes \( n \in \{16, \ldots, 64\} \), we generate 100 problems. Note that, given the above method, a scene might contain no actors while an actor must be involved in at least two scenes (and at most all).

We ensured preprocessing could not simplify any instance.

We ran the instances with a memory bound of 2Gb.

\(^1\)In [23] they generate a number from \{1, \ldots, n\} but actors appearing in only 1 scene are uninteresting (see Section 7.2.2).
Table 7.3 shows the average time in milliseconds obtained for finding an optimal schedule for all random instances of each size which did not run out of memory, while Table 7.4 shows the average number of subproblems solved. The entries — show where less than 80 of the 100 instances solved without running out of memory. The schedules were computed using all optimizations. The results show that while the number of scenes is clearly the most important factor in the difficulty of the problem, if the number of actors is small then the problem difficulty is limited. While increasing the number of actors increases difficulty, as it grows larger than the number of scenes, the incremental difficulty decreases. Note also that the random problems are significantly easier than the structured problems.

While we should be careful when reading these tables, since the difficulty of each 100 random benchmarks considered in each cell can vary remarkably (the standard deviation is usually larger than the average shown), the trend is clear enough.

7.6 Related Work

The talent scheduling problem (which appears as prob039 in CSPLIB [33] where it is called the rehearsal problem) was introduced in [23]. They consider the problem in terms of shooting days instead of scenes so, in effect, all scenes have the same duration. Note, however, that once we make use of Lemma 7.2.2 the requirement for different durations arises in any case. They give one example of a real scene scheduling problem, arising from the film Mob Story, containing 8 actors and 28 scenes. They show that the problem is NP-hard even in the very restricted case of each actor appearing in exactly two scenes and all costs and durations being one, by reduction to the optimal linear arrangement (OLA) problem [55].

In their paper they consider two methods to solve the scene scheduling problem. The first method is a branch and bound search, where they search for a schedule by filling in scenes from both ends in an alternating fashion (double ended search). They optimize on extra cost, and the lower bounds they use are simply the result of fixed costs (so equivalent to the definition of lower in Section 7.4 minus the fixed costs). They do not store equivalent solutions and, hence, are very limited in the size of the
problem they can tackle. Their experiments go up to 14 scenes and 14 actors.

The second method is a simple greedy hill climbing search. Given a starting schedule they consider all possible swaps of pairs of scenes, and move to the schedule after a swap if the resulting cost is less. They continue doing this until they reach a local minimum. On their randomly generated problems the heuristic approach gives answers around 10-12% off optimal regardless of size. They use this algorithm to re-schedule Mob Story with an extra cost of $16,100 as opposed to the hand solution of $36,400. This solution required 1.05 second on their AMDAHL mainframe. In comparison, our best algorithm finds an optimal answer with extra cost $14,600 in 0.1 seconds on a Xeon Pro 2.4GHz processor (which is admittedly very much more powerful). The search only considers 6,605 different subproblems. Note that after preprocessing, it only involves 20 scenes. The optimal solution found is shown in Figure 7.11 (costs are divided by 100).

Adelson et al. [1] define a restricted version of the talent scheduling problem for rehearsal scheduling where the costs of all actors are uniform, and also note how it can be used for an application in archeology. They give a dynamic programming formulation as a recurrence relation, more or less identical to that shown at the beginning of Section 7.2. They report solving an instance (from a real archaeological problem) with 26 “actors” and 16 “scenes” in 84 seconds on a CDC 7600 computer. We were not able to locate this benchmark.

Smith [145, 144] uses the talent scheduling problem as an example of a permutation problem. These papers solved the problem using constraint programming search with
caching of search states, which is very similar to dynamic programming with bounds. This paper considers both scheduling from one end, or from both ends. This paper was the first to use a form of pairwise subsumption, restricted to the case where the scenes differ by at most 2 actors. It also used the preprocessing of merging identical scenes (without proof). This was the first approach (we are aware of) to calculate the optimal solution to the Mob Story problem.

A comparison of the approaches is shown in Table 7.5. The table shows the sizes (after preprocessing). Note that the timing results for [145] are for a 1.7GHz Pentium M PC running ILOG Solver 6.0, whereas our results are for Xeon Pro 2.4GHz processors running gcc on Red Hat Linux. However, note also that there is around 3 orders of magnitude difference between our times and those of [145]. Also the number of cached states in the approach of [145] is around two orders of magnitude bigger than the number of subproblems (which is the equivalent measure). This is probably a combination of our better lower bounds, better detection of equivalent states and better search strategy. A web page with the problems and solutions can be found at www.csse.unimelb.edu.au/~pjs/talent/.

The talent scheduling problem is a generalization of the Optimal Linear Arrangement (OLA) problem (see [23]). This is a very well investigated graph problem, with applications including VLSI design [77], computational biology [82], and linear algebra [131]. The OLA is known to be very hard to solve, it has no polynomial time
approximation scheme unless NP-complete problems can be solved in randomized sub-exponential time [5]. Unfortunately, the problem size in this domain is in the thousands, which means methods that find exact linear arrangements (as dynamic programming does), cannot be applied. Interestingly, there are heuristic methods [88] that use exact methods as part of the entire process, and our algorithm could potentially be applied here.

The talent scheduling problem is highly related to the problem of minimizing the maximum number of open stacks. In this problem there are no durations, or costs and the aim is to minimize the maximum number of actors on location at any time. The problem has applications in cutting, packing and VLSI design problems. Compared to the talent scheduling problem, the open stacks problem has been well studied (see e.g.[160, 161, 162, 157, 42, 9]). The best current solution is our dynamic programming approach [53], but surprisingly almost none of the methods used there to improve the base dynamic programming approach are applicable to the talent scheduling problem. In the end the solutions are quite different, probably because the open stacks problem, while also NP-hard, is fixed parameter tractable [162], as opposed to the talent scheduling problem.

7.7 Conclusion

The talent scheduling problem is a very challenging combinatorial problem, because it is very hard to compute accurate bounds estimates from partial schedules. In this chapter we have shown how to construct an efficient dynamic programming solution by carefully reasoning about the problem to reduce search, as well as adding bounding and searching in the right manner. The resulting algorithm is orders of magnitude faster than other complete algorithms for this problem, and solves significantly larger problems than other methods.

There is still scope to improve the dynamic programming solution, by determining better heuristic orders in which to try scheduling scenes, and possibly determining better dynamic lower bounds by reasoning on the graph of actors that share scenes. One very surprising thing for us, was how much harder the talent scheduling problem
is than the highly related problem of minimizing the maximum number of open stacks.
Chapter 8

Maximum Density Still Life Problem
CHAPTER 8. MAXIMUM DENSITY STILL LIFE PROBLEM

8.1 Introduction

The Game of Life was invented by John Horton Conway and is played on an infinite board made up of square cells. The game takes place through discrete time steps. Each cell $c$ in the board is either alive or dead during each time period. The live/dead state of cell $c$ at time $t+1$, denoted as $\text{state}(c, t+1)$, can be obtained from the number $l$ of live neighbors of $c$ at time $t$ and from $\text{state}(c, t)$ as follows:

$$
\text{state}(c, t+1) = \begin{cases} 
  \text{dead} & \text{if } l < 2 \\
  \text{stable} & \text{if } l = 2 \\
  \text{state}(c, t) & \text{if } l = 3 \\
  \text{dead} & \text{if } l > 3 
\end{cases}
$$

[Death by isolation] [Stable condition] [Birth condition] [Death by overcrowding]

The board is said to be a still life at time $t$ if it is stable under these rules, i.e., it is identical at $t+1$. For example, an empty board is a still life. Given a finite $n \times n$ region where all cells outside must be dead, the Maximum Density Still Life Problem is to compute the maximum density of live cells that can appear in the $n \times n$ region in a still life, or equivalently, the maximum number of live cells that can appear in the $n \times n$ region.

The raw search space of the Still Life Problem has size $O(2^{n^2})$. Thus, it is extremely difficult even for small values of $n$. Previous search methods using IP [18] and CP [19] could only solve up to $n = 9$, while a CP/IP hybrid method with symmetry breaking [19] could solve up to $n = 15$. An attempt using bucket elimination [90] reduced the time complexity to $O(n^22^{3n})$ but increased the space complexity to $O(n^22^{2n})$. This method could solve up to $n = 14$ before it ran out of memory. A subsequent improvement that combined bucket elimination with search [91] used less memory and was able to solve up to $n = 20$.

In this chapter, we combine deep mathematical insights into the Still Life Problem with several powerful search techniques to completely solve the problem for all $n$. This is an incredible improvement. For example, our new approach can solve $n = 100$ in only a matter of hours, whereas the previous best approach would require in the order of $10^{100}$ years and $10^{30}$ bytes of memory to solve $n = 100$. We are in fact able to solve the Still Life problem for all $n$. This is because the Still Life Problem becomes well
behaved mathematically for sufficiently large \( n \) (around \( n > 200 \)), and it is possible to solve all the large cases in constant time.

The overall solution plan has four parts: 1) use complete search with a very strong model to solve the problem for all “small” \( n \) \( (n \leq 50) \), 2) use bounded dynamic programming on a relaxation of the problem to prove upper bounds on live cells for all medium and large \( n \) \( (n > 50) \) in constant time, 3) use a custom search to look for special form solutions to prove lower bounds on live cells for medium \( n \) (around \( 50 < n \leq 200 \)), 4) look for special form periodic solutions that can be tiled to construct arbitrarily large solutions to prove lower bounds on live cells for large \( n \) (around \( n > 200 \)) in constant time. The lower and upper bounds proved in parts 2, 3 and 4 coincide, thus they are the optimums for those \( n \). Each of these parts require deep mathematical insights into the problem as well as clever application of CP techniques. We give a brief overview of them here.

**Part 1** We give a new insightful proof that the maximum density of live cells in the infinite case \( (n = \infty) \) is \( \frac{1}{2} \). The proof is based on counting “wastage”. Wastage is calculated by looking at each \( 3 \times 3 \) pattern and seeing how much space we have “wasted” by not fitting in enough live cells into the local area. This proof allows us to reformulate the Maximum Density Still Life Problem into one of minimizing wastage rather than maximizing the number of live cells. The new model gives very tight lower bounds on wastage that dramatically increases the pruning strength of the model. This model, coupled with a simple lookahead, allows a Lazy Clause Generation solver to solve the problem up to around \( n = 50 \) using complete search.

**Part 2** We conjecture that for sufficiently large \( n \), all wastage which is forced to occur by the still life constraints are forced by only the constraints near the edge of the \( n \times n \) region. That is, only the boundary conditions cause suboptimality compared to the optimal density of \( \frac{1}{2} \) in the infinite case. If this conjecture holds, then it is possible to get a very good or optimal lower bound on the wastage (and thus upper bound on live cells) simply by relaxing the Still Life Problem onto its boundary and solving it, i.e. ignore all constraints other than then those within the first \( k \) rows of
the edge of the \( n \times n \) region for some small \( k \). This relaxed problem has the interesting property that the pathwidth of its constraint graph is \( O(k) \) instead of the \( O(n) \) of the original. There exists various techniques for solving such low pathwidth problems which can reduce the complexity from \( O(2^{-nk}) \) to only \( O(n2^{2k}) \), e.g., caching [145], nogood learning [113, 45], dynamic programming [11], variable elimination [90]. In this case, we use bounded dynamic programming [117]. For fixed and small \( k \), these relaxed problems can be solved in \( O(n) \) time. We will see however that due to the translational symmetry in the problem, the optimal solutions become periodic and we can solve the edge relaxation for all \( n \) in \( O(1) \), giving us very tight upper bounds on the number of live cells.

**Part 3** We conjecture that for sufficiently large \( n \), there always exists optimal solutions of the following form: wastage only exists at the four \( 4 \times 4 \) corners of the board, or in the one row beyond the edge of the board. Based on this conjecture, we search for these special form solutions using a variant of limited discrepancy search with dynamic relaxations as a lookahead. Such a search can find optimal solutions for up to \( n = 200 \) or so. We know that the solution is optimal if the number of live cells in the solution coincides with the upper bound on live cells proved in part 2.

**Part 4** The Still Life Problem becomes mathematically well behaved for sufficiently large \( n \). This raises the possibility that optimal solutions can be constructed in a systematic way. We find optimal solutions for \( n \sim 200 \) which are periodic, and which satisfy certain other constraints. If such solutions are found, then they can be tiled indefinitely to produce arbitrarily large, provably optimal solutions.

### 8.2 Wastage Reformulation

The maximum density of live cells in a still life on an infinite board is known to be \( \frac{1}{2} \) [40]. However, this proof is quite complex and only applies to the infinite case. In this section we provide a much simpler proof that can easily be extended to the finite case and gives much better insight into the possible sub-patterns that can occur in
8.2. WASTAGE REFORMULATION

Theorem 8.2.1. The maximum density of live cells in a still life on an infinite board is \( \frac{1}{2} \).

Proof. We initially assign 2 tokens to each cell in the board. We will redistribute the tokens of each dead cell among its live neighbors according to a particular set of rules. It is clear that if we can prove that after the redistribution, all live cells on the board end up with at least 4 tokens, then there must have been at least as many dead cells as live cells, and thus the density of live cells is \( \leq \frac{1}{2} \).

The rules for the token redistribution is as follows. The tokens of a dead cell are redistributed only to its orthogonal neighboring live cells, i.e. those that share an edge with the dead cell. Table 8.1 shows all possible patterns of orthogonal neighbors (up to symmetries) around a dead cell. Live cells are marked with a black dot, dead cells are unmarked, and cells whose state is irrelevant for our purposes are marked with a “?”. Each pattern indicates the beneficiaries, i.e., the North, East, South or West neighbors that receive 1 token from the center dead cell. We describe as “wastage” the tokens from a dead cell that does not need to be redistributed to any live cell for the proof to work (i.e., it is not needed to reach our target of 4 tokens for each live cell).

As it can be seen from the table, a dead cell gives 1 token to each of its live orthogonal neighbors if it has \( \leq 2 \) live orthogonal neighbors, 1 token to the two opposing live orthogonal neighbors if it has 3, and gives no tokens if it has 0 or 4 orthogonal neighbors. Wastage occurs whenever the dead cell has 0, 1 or 4 live
CHAPTER 8. MAXIMUM DENSITY STILL LIFE PROBLEM

Table 8.2: Contributions to the tokens of a live cell from its South neighbor.

Given this set of redistribution rules, it is sufficient to examine the 3 bordering cells on each side of a live cell to determine how many tokens are obtained from the orthogonal neighbor on that side. For example, the token obtained by the central live cell from its South neighbor is illustrated in Table 8.2.

The tokens obtained by a live cell can therefore be computed by simply adding up the tokens obtained from its four orthogonal neighbors. Since each live cell starts off with 2 tokens, it must receive at least 2 extra tokens to end up with $\geq 4$ tokens. Let us then look at all possible patterns around a live cell and see where the cell will receive tokens from. Table 8.3 shows all possible neighborhoods of a live cell (up to symmetries). For each pattern, it shows the benefactors, i.e., the North, East, South or West neighbors that give 1 token to the live cell, and the resulting amount of wastage, which occurs whenever a live cell receives more than 2 tokens.

Note that the last pattern does not receive sufficient extra tokens, just 1 from the South neighbor. However, the last two patterns always occur together in unique pairs due to the still life constraints (each of the central live cells has 3 neighbors so the row above the last pattern, and the row below the second last pattern must only consist of dead cells). So, as a final redistribution step, within each pair of these two patterns, we transfer the extra token from the second last pattern to the last.

Clearly, after all transfers are done, all live cells end up with $\geq 4$ tokens, and this completes our proof that the maximum density on an infinite board is $\frac{1}{2}$.

The above proof is not only much simpler than that of [40], it also provides us with good insight into how to compute useful bounds for the case in which the board is finite. In particular, it allows us to know exactly how much we have lost from the
8.2. WASTAGE REFORMULATION

Theoretical maximum density by looking at the amount of “wastage” produced by the patterns in the currently labeled cells.

To achieve this, we reformulate the objective function in the Maximum Density Still Life Problem as follows. For each cell \( c \), let \( P(c) \) be the \( 3 \times 3 \) pattern around that cell. Note that if \( c \) is on the edge of the \( n \times n \) region, the dead cells beyond the edge are also included in this pattern. Let \( w(P) \) be the wastage for each \( 3 \times 3 \) pattern as listed in Tables 8.1 and 8.3, with the exception that the last two patterns in 8.3 both have \( w(P) = 0 \). Define \( w(c) \) for each cell \( c \) as follows. If \( c \) is within the \( n \times n \) region, then \( w(c) = w(P(c)) \). If \( c \) is in the row immediately beyond the \( n \times n \) region and shares an edge with it (there are \( 4n \) such cells), then \( w(c) = 1 \) if the cell in the \( n \times n \) region with which it shares an edge is dead, and \( w(c) = 0 \) otherwise. For all other \( c \), let \( w(c) = 0 \). Let \( W = \sum w(c) \) over all cells.

Table 8.3: Possible patterns around a live cell showing token benefactors and any wastage.
Theorem 8.2.2. Wastage and live cells are related by

\[ \text{live cells} = \frac{n^2}{2} + n - \frac{W}{4} \]  \hspace{1cm} (8.1)

Proof. We adapt the proof for the infinite board to the $n \times n$ region. Let us assign 2 tokens to each cell within the $n \times n$ region, and 1 token to each of the 4$n$ cell in the row immediately beyond the edge of the $n \times n$ region. Now, for each dead cell within the $n \times n$ region, partition the token among its live orthogonal neighbors as before. For each dead cell in the row immediately beyond the $n \times n$ region, give its 1 token to the cell in the $n \times n$ region with which it shares an edge. Again, since the last two 3 $\times$ 3 patterns listed above must occur in pairs, we transfer an extra 1 token from one to the other. Note also that the second last pattern of Table 8.3 cannot appear on the South border (which would mean that the last pattern appeared outside the shape) since it is not stable in this position. Clearly, after the transfers, all live cells once again have $\geq 4$ token, and wastage for the 3 $\times$ 3 patterns centered around cells within the $n \times n$ region remain the same. However, since we are in the finite case, we also have wastage for the cells which are in the row immediately beyond the edge of the $n \times n$ region. These dead cells always give 1 token to the neighboring cell which is in the $n \times n$ region. If that cell is live, the token is received. If that cell is dead, that 1 token is wasted. The reformulation above counts all these wastage as follows. The total number of tokens that are used is $2n^2$ from the cells within the $n \times n$ region and $4n$ from the 4$n$ cells in the row immediately beyond the edge, for a total of $2n^2 + 4n$. Now, 4 times live cells will be equal to the total token minus all the token wasted, and hence we end up with Equation (8.1).

We can trivially derive some upper bounds on the number of live cells using this equation. Clearly $W \geq 0$ and, thus, we have: \[ \text{live cells} \leq \left\lfloor \frac{n^2}{2} + n \right\rfloor. \] Also, by the still life constraints, there cannot be three consecutive live cells along the edge of the $n \times n$ region. Hence, there is always at least 1 wastage per 3 cells along the edge and we can improve the bound to: \[ \text{live cells} \leq \left\lfloor \frac{n^2}{2} + n - \left\lfloor \frac{1}{3}n \right\rfloor \right\rfloor. \] While this bound is very close to the optimal value for small $n$, it differs from the true optimum by $O(n)$ and will diverge from the optimum for large $n$. \hfill \Box
8.3 Solving Small \( n \) with Complete Search

The power of a branch and bound algorithm is hugely dependent on how strong a bound we can prove on the objective at each node in the search tree. The stronger the bound we can prove, the earlier we can prune off failed subtrees. The naive Still Life model based on counting the number of live cells is horribly weak, because the upper bound on the number of live cells that propagation can prove is usually very weak and search generally does not fail until the board is at least half filled.

Remodeling the Still Life Problem in terms of minimizing wastage instead of maximizing live cells allows us to propagate much stronger bounds on the objective, as it is easy to tell how much space has already been wasted in the parts of the board that are labeled. Let \( \text{sl}_\text{waste} \) be a width 10 table which specifies the wastage value of each \( 3 \times 3 \) pattern satisfying the still life constraints. For example, the entries corresponding to the first and second patterns in Table 8.3 would be \((0,1,0,0,1,0,0,1,0,0)\) and \((0,1,0,0,1,0,1,0,1,0,1)\). A single table constraint using \( \text{sl}_\text{waste} \) will be sufficient to enforce the still life constraints and set the value of the wastage variable. We propose the following simple model:

\[
\begin{align*}
\text{int: } n; & \quad \% \text{ instance parameter} \\
\text{array } [0..n+1,0..n+1] \text{ of var } 0..1: x; & \quad \% \text{ cell live/dead status} \\
\text{array } [0..n+1,0..n+1] \text{ of var } 0..2: w; & \quad \% \text{ cell wastage} \\
\text{var } 0..2\cdot n\cdot n+4\cdot n: \text{total_wastage}; \\
\text{var } 0..n\cdot n: \text{live_cells}; \\
\end{align*}
\]

\% still life and wastage constraints in \( n \) by \( n \) region
\begin{verbatim}
constraint forall (i,j in 1..n) ( 
    table(sl_waste, [x[i-1,j-1], x[i,j-1], x[i+1,j-1], x[i-1,j], 
    x[i,j], x[i+1,j], x[i-1,j+1], x[i,j+1], x[i+1,j+1], w[i,j]]));
\end{verbatim}

\% boundary conditions
\begin{verbatim}
constraint forall (i in 0..n+1) ( 
\end{verbatim}
x[i,0] = 0 /\ x[0,i] = 0 /\ x[i,n+1] = 0 /\ x[n+1,i] = 0);

class constraint forall (i in 1..n-2) ( 
  sum (j in i..i+2) (x[1,j]) <= 2 /\ 
  sum (j in i..i+2) (x[n,j]) <= 2 /\ 
  sum (j in i..i+2) (x[j,1]) <= 2 /\ 
  sum (j in i..i+2) (x[j,n]) <= 2 
);

% wastage constraints for boundary
constraint forall (i in 1..n) (w[i,0] = 1 - x[i,1]);
constraint forall (i in 1..n) (w[0,i] = 1 - x[1,i]);
constraint forall (i in 1..n) (w[i,n+1] = 1 - x[i,n]);
constraint forall (i in 1..n) (w[n+1,i] = 1 - x[n,i]);

% objective function
constraint total_wastage = sum (i,j in 0..n+1) (w[i,j]);
constraint live_cells = (2*n*n+4*n - total_wastage)/4;

solve maximize live_cells;

This basic model is capable of counting the wastage in the labeled parts of the board and using it to enforce an upper bound on the number of live cells. However, it is also important to get a good lower bound on the wastage that must occur in the parts of the board which are not yet labeled. The simplest bound we can get is that there must be at least 1 wastage per 3 cells along any unlabeled edge. We can implement this by adding a few lines and modifying the constraint on total wastage:

% wastage per 3 edge cells along edge i
array [1..4,1..n/3] of var 1..3: ew;

constraint forall (i in 1..n/3) (
8.3. **SOLVING SMALL N WITH COMPLETE SEARCH**

\[
\begin{align*}
\text{ew}[1,i] &= \sum (j \text{ in } 3i-2..3i) (w[0,j]) \land \\
\text{ew}[2,i] &= \sum (j \text{ in } 3i-2..3i) (w[j,0]) \land \\
\text{ew}[3,i] &= \sum (j \text{ in } 3i-2..3i) (w[n+1,j]) \land \\
\text{ew}[4,i] &= \sum (j \text{ in } 3i-2..3i) (w[j,n+1]) \\
\end{align*}
\]

constraint total_wastage = \sum (i, j \text{ in } 1..n) (w[i,j]) + \\
\sum (i \text{ in } 1..4, j \text{ in } 1..n/3) (ew[i,j]) + \\
\sum (i \text{ in } n/3*3+1..n) (w[0,i] + w[i,0] + w[n+1,i] + w[i,n+1]);

In this modified model, the wastage of groups of three consecutive edge cells are summed together into variables \(\text{ew}[i,j]\) before being added to the objective. Since each \(\text{ew}[i,j]\) variable has a lower bound of 1, this facilitates the “at least 1 wastage per 3 edge cells” lookahead rule. However, this model is still a bit too weak for the bigger instances, and we use a more advanced lookahead based on relaxations. We defer detailed discussion of this to Section 8.4.

The search strategy is also important. We use a labeling strategy where we label from the boundary of the board inwards. We first label the first 3 rows of each edge and each \(8 \times 8\) corner. Thereafter, we label one row in at a time in concentric squares. The reason for this labeling strategy will become much clearer in view of the insights discussed in Section 8.4. Basically, most wastage that is forced to occur by the constraints occurs along the boundary of the \(n \times n\) region, and thus labeling those cells first increases the bound on the objective the quickest, allowing us to detect suboptimal assignments earlier.

Finally, we must note that a certain feature of the solver we used is critical for solving this problem effectively. One major problem with any search strategy for the Still Life problem is that it is possible to make a “mistake” in labeling that makes the subtree unsatisfiable, but propagation may not be able to notice this until many decision levels later. A normal CP solver will take an exponential amount of nodes to backtrack to this mistake and fix it. The solver we use however, is the Lazy Clause Generation solver CHUFFED, which supports conflict analysis and backjumping. Thus
### 8.4 Upper Bounds for Large $n$

To solve the problem for larger $n$, we need more mathematical insight into the problem. We make the following conjecture:

**Conjecture 8.4.1.** For sufficiently large $n$, all forced wastage is caused by the still life constraints within $k$ rows from the edge of the $n \times n$ region, where $k$ is some small, fixed constant.

The conjecture is inspired by the following two facts. Firstly, in the optimal solutions for $n \leq 20$ that were found by previous methods, it was often the case that wastage only appeared in the corners or within the first 3 rows from the edge.

Table 8.4: Optimum number of live cells in the Maximum Density Still Life Problem found by complete search.

<table>
<thead>
<tr>
<th>$n$</th>
<th>opt. time</th>
<th>$n$</th>
<th>opt. time</th>
<th>$n$</th>
<th>opt. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>232</td>
<td>31</td>
<td>497</td>
<td>41</td>
<td>864</td>
</tr>
<tr>
<td>22</td>
<td>253</td>
<td>32</td>
<td>531</td>
<td>42</td>
<td>907</td>
</tr>
<tr>
<td>23</td>
<td>276</td>
<td>33</td>
<td>563</td>
<td>43</td>
<td>949</td>
</tr>
<tr>
<td>24</td>
<td>302</td>
<td>34</td>
<td>598</td>
<td>44</td>
<td>993</td>
</tr>
<tr>
<td>25</td>
<td>326</td>
<td>35</td>
<td>633</td>
<td>45</td>
<td>1039</td>
</tr>
<tr>
<td>26</td>
<td>353</td>
<td>36</td>
<td>668</td>
<td>46</td>
<td>1085</td>
</tr>
<tr>
<td>27</td>
<td>379</td>
<td>37</td>
<td>706</td>
<td>47</td>
<td>1132</td>
</tr>
<tr>
<td>28</td>
<td>407</td>
<td>38</td>
<td>744</td>
<td>48</td>
<td>1181</td>
</tr>
<tr>
<td>29</td>
<td>437</td>
<td>39</td>
<td>782</td>
<td>49</td>
<td>1229</td>
</tr>
<tr>
<td>30</td>
<td>467</td>
<td>40</td>
<td>824</td>
<td>50</td>
<td>1280</td>
</tr>
</tbody>
</table>

CHUFFED can analyze conflicts and immediately backjump to the mistake and fix it without having to waste an exponential amount of time searching in the failed subtree.

In Table 8.4, we show the results for $21 \leq n \leq 50$ ($n \leq 20$ have previously been solved). Clearly, we are already able to solve instances which are much larger than the previous state of the art methods. Optimal solutions for $n = 21$ and $n = 22$ are shown in Figure 8.1. We can solve instances somewhat larger than $n = 50$ using complete search, but the run time grows very quickly. Instead, we use better methods to tackle larger $n$ in the next few sections.
Secondly, we already know that there exist wastage free labellings in the infinite case, thus there is nothing inherent in the still life constraints which force wastage. Instead, it would appear that it is the boundary conditions in the finite case which is forcing the extra wastage to occur, hence the conjecture.

If the conjecture holds, then it should be possible to relax the Still Life problem onto just the boundary variables (variables within $k$ rows from edge) and still derive the same wastage lower bound. This has the advantage that: 1) such a relaxed problem has far fewer variables ($O(nk)$ instead of $O(n^2)$) and thus a much smaller search space, 2) such a relaxed problem has low pathwidth ($O(k)$ instead of $O(n)$) and there exist various techniques that can take advantage of this to reduce the search space even further to $O(n^{2k})$, e.g., caching, nogood learning, dynamic programming, variable elimination.

Theorem III.2.2 tells us that any lower bound we prove for the wastage in the relaxed problem is also a valid lower bound for the original problem. However, this bound may or may not be the optimal bound for the original problem. If we relaxed the problem too much, then the bound we derive from the relaxed problem will be weaker than the optimal bound for the original problem. Now, if Conjecture 8.4.1 holds, then there should exists some small $k$ such that the bound from relaxing the
problem onto a width $k$ boundary should still be optimal for the original problem. We need to find this $k$.

The complexity of solving the relaxation is $O(n2^k)$, so choosing $k$ too large will make the problem intractable. We wish to find the smallest $k$ such that the relaxation is sufficient to prove the optimal bounds for the original problem. We performed a series of experiments to try to guess what this minimal $k$ is. We define the edge still life problem $edge(n,k)$ as follows:

\[
\text{int: } n, k; \quad \% \text{ instance parameters}
\]

\[
\text{array } [0..n+1,0..k+1] \text{ of var } 0..1: x; \quad \% \text{ cell live/dead status}
\]

\[
\text{array } [1..n,0..k] \text{ of var } 0..2: w; \quad \% \text{ cell wastage}
\]

\[
\text{% still life and wastage constraints in } n \text{ by } k \text{ region}
\]

constraint forall (i in 1..n, j in 1..k) (table(wastage, [x[i-1,j-1], x[i,j-1], x[i+1,j-1], x[i-1,j], x[i,j], x[i+1,j], x[i-1,j+1], x[i,j+1], x[i+1,j+1], w[i,j]]));

\[
\text{% boundary conditions}
\]

constraint forall (i in 0..n+1) (x[i,0] = 0);

constraint forall (i in 1..n-2) (sum (j in i..i+2) (x[1,j]) <= 2);

\[
\text{% wastage constraints for boundary}
\]

constraint forall (i in 1..n) (w[i,0] = 1 - x[i,1]);

\[
\text{% objective function}
\]

solve minimize sum (i in 1..n, j in 0..k) (w[i,j]);

Let $m(n,k)$ be the minimum wastage for problem $edge(n,k)$. For each $k$, we calculate the edge wastage ratio $r(k)$ as: $r(k) = \lim_{n \to \infty} m(n,k)/n$. These limits
8.4. UPPER BOUNDS FOR LARGE $N$

Table 8.5: Ratio of wastage to edge cells in an optimal edge for different edge width, and number of patterns that appear in the first three rows.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$r(k)$ (exact)</th>
<th>$r(k)$ (decimal)</th>
<th>castles</th>
<th>blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>1/3</td>
<td>0.333</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3-11</td>
<td>4/11</td>
<td>0.364</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>7/19</td>
<td>0.368</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$\geq$ 13</td>
<td>10/27</td>
<td>0.370</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 8.2: Optimal edge patterns for a width 2 and width 3 boundary relaxation. Wastage is highlighted with a star.

exists and can be calculated in constant time using dynamic programming, because the optimal edge patterns all become periodic for sufficiently large $n$ for any $k$. See Table 8.5 for the values of $r(k)$ and Figure 8.2 for a sample of the optimal edge patterns.

First, we note that $r(1) = r(2) = 1/3 = 0.333$ gives us the trivial edge wastage bound discussed in Section 8.2. The optimal edge pattern for $k = 2$ is unique and periodic and is shown in Figure 8.2(a). However, this optimal width 2 edge cannot be extended to a width 3 edge without introducing additional wastage. For a width 3 edge, $r(3) = 4/11 = 0.364$ and the optimal edge pattern is once again unique and periodic and is shown in Figure 8.2(b). We will refer to the pattern shown in the first 8 columns of Figure 8.2(b) the “castle” pattern and the next 3 columns of Figure 8.2(b) the “block” pattern. The castle pattern has length 8 with 3 wastage and the block pattern has length 3 with 1 wastage. As $k$ increases, $r(k)$ continues to increase slightly. At $k = 13$, we have $r(13) = 10/27 = 0.370$. The optimal edge is no longer unique or periodic, however, all of them are identical within the first 3 rows up to translational symmetry, and are composed of three castle patterns followed by a block pattern, which gives $3 \times 3 + 1 = 10$ wastage per $3 \times 8 + 3 = 27$ cells. All
of the optimal edge patterns only have wastage in the row beyond the edge, which is an interesting fact to note. Beyond, $k = 13$, the value of $r(k)$ plateaus, and we stopped our experimentation at $k = 20$. We show later in Section 8.7 that there exists arbitrarily thick edge patterns which achieve exactly an edge wastage ratio of $r(13) = 10/27$. So in fact, $r(k) = 10/27$ for all $k \geq 13$.

What the edge wastage ratio results show is that the still life constraints up to a depth of 13 are capable of forcing wastage to occur. We should not relax these constraints away, as that will degrade the bound that we can prove using the relaxed problem. Thus we need to look at an edge relaxation of at least width 13. However, the fact that $r(k)$ plateaus after $k = 13$ also suggests that width 13 is sufficient, i.e. the still life constraints beyond a depth of 13 can have no effect on forcing edge wastage, and we can relax them away without changing the bound. This turns out to be correct, as our results later show.

8.5 Solving the Boundary Relaxation

We wish to solve the Still Life problem, relaxed onto a width 13 boundary. The technique we choose to use is bounded dynamic programming [117]. We discuss the choice of this technique and show how it can be applied. First of all, it is necessary to choose one of the methods that can exploit the low pathwidth of the relaxed problem. Of these, dynamic programming is by far the easiest to implement. However, we use a variant of dynamic programming called bounded dynamic programming [117]. The idea behind bounded dynamic programming is that for some kinds of dynamic programs, instead of calculating the exact value for every subproblem, for many subproblems, it is actually sufficient to prove a bound on its value.

**Example 8.5.1.** Consider the recursion: $a(n) = \min(a(n - 1) + 2, a(n - 2) + 5)$.

Suppose we want to calculate the value of $a(n)$. Suppose we have already calculated that $a(n - 1) = 6$. Now, we need to find some information on $a(n - 2)$. However, we note that one of the terms in the $\min$: $a(n - 1) + 2 = 8$, is already known, therefore if it is known that $a(n - 2) \geq 3$, then its exact value is irrelevant as far as the value of $a(n)$ is concerned. We only want to know: is $a(n - 2) \geq 3$, or if it is $< 3$, then
what is its exact value? Thus we only wish to answer a bounded query on the value of $a(n - 2)$, where we do not care what its exact value is if it is above or below a certain range.

On certain types of problems, bounded dynamic programming can be exponentially faster than normal dynamic programming, as there are a lot of subproblems whose exact values are irrelevant and all we need to do is to prove a certain bound on their value. Proving a weak bound is often exponentially faster than finding the exact value. In the case of the Still Life problem, there are many, many ways to label the cells, and most of them lead to large amounts of wastage. Therefore, we often just need to prove that a subproblem is sufficiently bad (has at least a certain amount of wastage), rather than calculate exactly how bad it is. Bounded dynamic programming is a rigorous way to do this.

We illustrate how dynamic programming can be applied with a simplified example. Consider a more constrained version of $\text{edge}(n, k)$ where we have additional boundary conditions which fix the first 2 and last 2 columns. We denote this problem by $\text{edge}(n, k, s_1, s_2, e_1, e_2)$ where the model has the additional lines:

\[
\begin{align*}
\text{% instance parameters} \\
\text{int: } s_1, s_2, e_1, e_2; \\
\text{% additional boundary conditions} \\
\text{constraint } s_1 = \text{sum (i in 0..k-1)} (x[1,i] \times 2^i); \\
\text{constraint } s_2 = \text{sum (i in 0..k-1)} (x[2,i] \times 2^i); \\
\text{constraint } e_1 = \text{sum (i in 0..k-1)} (x[n,i] \times 2^i); \\
\text{constraint } e_2 = \text{sum (i in 0..k-1)} (x[n-1,i] \times 2^i); \\
\end{align*}
\]

As can be seen, we are denoting the value of a column of $k$ cells by a single number in $\{0, \ldots, 2^k - 1\}$, where the value of the cells are being interpreted as the binary digits of this number. We define the Boolean function $s(a, b, c)$ where $a, b, c \in \{0, \ldots, 2^k - 1\}$ as \textit{true} if three consecutive columns labeled as $a, b, \text{and } c$ in that order do not violate the still life constraints, and \textit{false} otherwise. We define the integer function $w(a, b, c)$
where \( a, b, c \in \{0, \ldots, 2^k - 1\} \) as the wastage in the central column if three consecutive columns are labeled as \( a, b, \) and \( c \) in that order.

Let \( m(n, k, s_1, s_2, e_1, e_2) \) be the minimum wastage for problem \( edge(n, k, s_1, s_2, e_1, e_2) \). Then the following recursive formulas hold: \( \forall n \geq 3, s_1, s_2, e_1, e_2 \in \{0, \ldots, 2^k - 1\}, m(n, k, s_1, s_2, e_1, e_2) = \min\{m(n-1, k, s_1, s_2, e_2, e_3) + w(e_1, e_2, e_3) \mid e_3 \in \{0, \ldots, 2^k - 1\}, s(e_1, e_2, e_3)\} \).

Let’s consider why. Consider any solution of \( edge(n, k, s_1, s_2, e_1, e_2) \). The \((n-3)\)th column must have been fixed to something. Let it be denoted by \( e_3 \). To satisfy the still life constraints, we must have \( s(e_1, e_2, e_3) \) be true. For each such solution, \( w(e_1, e_2, e_3) \) gives us the wastage in the \( n-2 \)th column. If we project this solution onto the first \( n-1 \) columns, then we clearly have a solution for \( edge(n-1, k, s_1, s_2, e_2, e_3) \). Hence the minimum wastage among all such solutions is \( m(n-1, k, s_1, s_2, e_2, e_3) + w(e_1, e_2, e_3) \). Taking the minimum of these over all possible \( e_3 \) gives \( m(n, k, s_1, s_2, e_1, e_2) \), and hence the recursive formula.

Such a set of recursive formulas can be solved using dynamic programming in \( O(n) \). We can use precalculated values of \( m(n, k, s_1, s_2, e_1, e_2) \) as a wastage lookahead in the complete search method described in Section 8.3. Given a partially filled edge where the last two columns are given by \( s_1 \) and \( s_2 \), we can look up \( \min_{i,j} m(n, k, s_1, s_2, i, j) \) to get a lower bound on the wastage in the remaining cells of the edge. This will give a bound of approximately \( 10/27 \) wastage per remaining edge cell instead of the trivial \( 1/3 \) wastage per remaining edge cell we had before. Although the difference is small, it is quite crucial for solving the larger \( n \) with complete search.

Now, it turns out that for any \( k \), for sufficiently large \( n \), the values of \( m(n, k, s_1, s_2, e_1, e_2) \) become periodic. This is formalized by the following theorem.

**Theorem 8.5.2.** If for some constants \( M, p, q \), we have: \( \forall s_1, s_2, e_1, e_2, m(M + p, k, s_1, s_2, e_1, e_2) = m(M, k, s_1, s_2, e_1, e_2) + q \), then \( \forall n \geq M, \forall s_1, s_2, e_1, e_2, m(n + p, k, s_1, s_2, e_1, e_2) = m(n, k, s_1, s_2, e_1, e_2) + q \). That is, once \( n \geq M \), the values of \( m \) simply increase by \( q \) every \( p \) cells.

**Proof.** The proof is by induction. The base case of \( n = M \) is true by assumption. Suppose it is true for \( n = t \). Consider \( n = t + 1 \). We have:
Theorem 8.5.2 tells us that if we ever reach an \( n \) where the \( m \) values become periodic, then it remains periodic for all larger \( n \). This always happens due to the translational symmetry in the Still Life Problem. For \( k = 13 \), the values typically become periodic by the time we reach \( n = 200 \). Once this happens, we can calculate closed form equations for \( m(n, k, s_1, s_2, e_1, e_2) \) for all larger \( n \). Thus, we can calculate the values of \( m(n, k, s_1, s_2, e_1, e_2) \) for all \( n \) in constant time.

In the relaxed Still Life Problem, we also need to deal with the corners. However, it is easy to see that the same kind of recursive formulation into a dynamic program is possible. Thus the relaxed problem can be completely solved for all \( n \) using bounded dynamic programming very efficiently. Table 8.6 shows the wastage lower bounds and the corresponding live cell upper bounds we derive for \( 21 \leq n \leq 56 \). We note several things. Firstly, all the bounds calculated by bounded dynamic programming on the relaxed problem are consistent with the optimal live cell values calculated by complete search in Section 8.3. Secondly, the bounds on the relaxed problem are often the optimal bounds for the original problem (the only exceptions being \( n = 24, 26, 28, 38 \)). This is consistent with Conjecture 8.4.1, which stated that for sufficiently large \( n \), the relaxed problem should have the same bound as the original. It would appear that the “sufficiently large \( n \)” in the conjecture is \( n > 38 \).

For \( n \geq 61 \), the wastage lower bounds becomes periodic and is given by the equations in Figure 8.3. The corresponding live cell upper bound is given by the equations in Figure 8.4. We shall see in the next two sections that these live cell upper bounds are in fact achievable, so the equations in Figure 8.4 actually give us the optimal number of live cells for \( n \geq 61 \).
### Table 8.6: Lower bounds on wastage and corresponding upper bounds on live cells

Instances where the bound on the relaxed problem differed from the original problem are indicated with an asterisk.

<table>
<thead>
<tr>
<th>n</th>
<th>waste lb.</th>
<th>live ub.</th>
<th>n</th>
<th>waste lb.</th>
<th>live ub.</th>
<th>n</th>
<th>waste lb.</th>
<th>live ub.</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>40</td>
<td>210</td>
<td>34</td>
<td>56</td>
<td>598</td>
<td>48</td>
<td>76</td>
<td>1181</td>
</tr>
<tr>
<td>21</td>
<td>36</td>
<td>232</td>
<td>35</td>
<td>59</td>
<td>632</td>
<td>49</td>
<td>80</td>
<td>1229</td>
</tr>
<tr>
<td>22</td>
<td>42</td>
<td>253</td>
<td>36</td>
<td>62</td>
<td>668</td>
<td>50</td>
<td>80</td>
<td>1280</td>
</tr>
<tr>
<td>23</td>
<td>43</td>
<td>276</td>
<td>37</td>
<td>60</td>
<td>706</td>
<td>51</td>
<td>80</td>
<td>1331</td>
</tr>
<tr>
<td>24</td>
<td>40</td>
<td>302*</td>
<td>38</td>
<td>64</td>
<td>744*</td>
<td>52</td>
<td>85</td>
<td>1382</td>
</tr>
<tr>
<td>25</td>
<td>46</td>
<td>326</td>
<td>39</td>
<td>67</td>
<td>782</td>
<td>53</td>
<td>84</td>
<td>1436</td>
</tr>
<tr>
<td>26</td>
<td>44</td>
<td>353*</td>
<td>40</td>
<td>64</td>
<td>824</td>
<td>54</td>
<td>87</td>
<td>1490</td>
</tr>
<tr>
<td>27</td>
<td>48</td>
<td>379</td>
<td>41</td>
<td>68</td>
<td>864</td>
<td>55</td>
<td>89</td>
<td>1545</td>
</tr>
<tr>
<td>28</td>
<td>51</td>
<td>407*</td>
<td>42</td>
<td>68</td>
<td>907</td>
<td>56</td>
<td>88</td>
<td>1602</td>
</tr>
<tr>
<td>29</td>
<td>48</td>
<td>437</td>
<td>43</td>
<td>71</td>
<td>949</td>
<td>57</td>
<td>91</td>
<td>1658</td>
</tr>
<tr>
<td>30</td>
<td>53</td>
<td>466</td>
<td>44</td>
<td>73</td>
<td>993</td>
<td>58</td>
<td>92</td>
<td>1717</td>
</tr>
<tr>
<td>31</td>
<td>55</td>
<td>497</td>
<td>45</td>
<td>72</td>
<td>1039</td>
<td>59</td>
<td>92</td>
<td>1776</td>
</tr>
<tr>
<td>32</td>
<td>53</td>
<td>530</td>
<td>46</td>
<td>76</td>
<td>1085</td>
<td>60</td>
<td>97</td>
<td>1835</td>
</tr>
<tr>
<td>33</td>
<td>56</td>
<td>563</td>
<td>47</td>
<td>78</td>
<td>1132</td>
<td>61</td>
<td>96</td>
<td>1897</td>
</tr>
</tbody>
</table>

**Figure 8.3:** Closed form equations for the wastage lower bound for \( n \geq 61 \).

**8.6 Lower Bounds for Large \( n \)**

Lower bounds on live cells can be proved by finding actual solutions to the problem. If the number of live cells in the solution found coincides with the upper bound proved in Section 8.5, then we know that it is an optimal solution and we have solved the

\[
\text{wastage} \geq \begin{cases} 
\lfloor \frac{40}{27}n + 5 \rfloor, & n \equiv 5, 13, 21, 24, 32, 35, 40, 43, 48, 51 \mod 54 \\
\lfloor \frac{40}{27}n + 6 \rfloor, & n \equiv 0, 2, 7, 8, 10, 14, 15, 16, 18, 22, 23, 26, 29, 30, 34, 37, 38, 42, 45, 46, 50, 53 \mod 54 \\
\lfloor \frac{40}{27}n + 7 \rfloor, & n \equiv 1, 3, 4, 9, 11, 12, 17, 19, 20, 25, 27, 28, 31, 36, 39, 41, 44, 47, 49, 52 \mod 54 \\
\lfloor \frac{40}{27}n + 8 \rfloor, & n \equiv 6, 33 \mod 54 
\end{cases}
\]

\[
\text{livecells} \leq \begin{cases} 
\lfloor \frac{n^2}{2} + 17/27n - 2 \rfloor, & n \equiv 0, 1, 3, 8, 9, 11, 16, 17, 19, 25, 27, 31, 33, 39, 41, 47, 49 \mod 54 \\
\lfloor \frac{n^2}{2} + 17/27n - 1 \rfloor, & n \equiv 2, 4, 5, 6, 7, 10, 12, 13, 14, 15, 18, 20, 21, 22, 23, 24, 26, 28, 29, 30, 32, 34, 35, 36, 37, 38, 40, 42, 43, 44, 45, 46, 48, 50, 51, 52, 53 \mod 54 
\end{cases}
\]

**Figure 8.4:** Closed form equations for the live cell upper bound for \( n \geq 61 \).
Figure 8.5: The two best ways to fill a corner (up to symmetry). Wastage is highlighted with a star. Note that there are two units of wastage in the cell in the 4th column and 4th row in the second pattern.

Problem for that $n$. However, finding optimal solutions is very hard, because the default search space of the Still Life Problem is extremely large. The model given in Section 8.3 was good enough for us to solve up to around $n = 50$ using complete search. However, to go beyond that size, we need something that reduces our search space much further. We make the following conjecture:

**Conjecture 8.6.1.** For sufficiently large $n$, there always exists optimal solutions of the following form: wastage only exists at the four $4 \times 4$ corners of the board, or in the one row beyond the edge of the board.

Conjecture 8.6.1 is supported by the experiments described in Section 8.4. We reasoned that for sufficiently large $n$, the edges should follow one of the optimal edge patterns, and we know that none of them have any wastage other than in the one row beyond the edge of the board. Similarly, there are only two ways to label a corner with minimum wastage (see Figure 8.5) and both of them only have wastage within the $4 \times 4$ corners. And finally, the center of the board should be wastage free, because by Conjecture 8.4.1, the constraints in the center of the board do not force any wastage, and any wastage there would simply make the solution suboptimal.

Assuming that Conjecture 8.6.1 holds, we can look only for solutions of this special form. Note that searching only for solutions of this special form gives an incomplete search on the Still Life Problem. However, incomplete search is perfectly sufficient for proving lower bounds on the number of live cells, since the solution itself is the
proof. To solve for all \( n \), we must be able to find an optimal solution for every single \( n \). We use two further techniques to reduce the search space: 1) dynamic relaxations as lookahead, and 2) a customized limited discrepancy search.

In Section 8.4, we described how we can perform a relaxation onto the boundary of the board in order to derive a lower bound on the wastage. We can do the same thing during search. Our search strategy is to label the board 8 rows at a time from top to bottom. Conjecture 8.4.1 tells us that for sufficiently large \( n \), wastage is only forced by the boundary constraints. In a subproblem however, the boundary also includes the values of the cells we have labeled. If we relax the subproblem onto the boundary of the unlabeled region, we should be able to derive a very strong lower bound on the wastage in the unlabeled region of the board. Thus, at each search node, we relax the remaining problem onto: the unlabeled parts of the width 13 boundary, and the 8 unlabeled rows below the last row we labeled (see Figure 8.6).

We note several things. Firstly, the set of variables involved in the relaxation is different at each search depth. Secondly, the boundary conditions of the relaxation are different even for two nodes at the same search depth if their cells in the last two rows are labeled differently. Thus each of these relaxed subproblems are different and have to be solved separately to derive the wastage lower bound. Now, as before, we can solve each of these in \( O(n) \) using dynamic programming. However, we can
do better by noting that the relaxed problem at any node is very similar to the relaxed problem at its parent node, differing only by one column of 8 variables. By appropriately caching the solutions to the relaxed subproblem at the parent node, we can solve the relaxed problem at each node in $O(1)$. The cost is still non-trivial, however, the wastage lower bounds derived by these relaxations are very strong and provide a large amount of pruning, so it is well worth it.

One might wonder why we use a different search order than that described in Section 8.3, where we label the board from outside in rather than from top to bottom. This is because the relaxation lookahead already tells us how much wastage there has to be in the unlabeled parts of the boundary, thus it is unnecessary to actually label it to force the wastage lower bound up. It is better not to label the boundary yet, as there are exponentially many ways to do it, but most of them are probably bad. It is better to fix the variables which are close to those variables already fixed so that we can detect inconsistencies earlier, hence the top to bottom strategy.

We also use the relaxation lookahead as a branching heuristic. When faced with the choice of labeling the next column of 8 variables as any one of the 256 possible values, our relaxation lookahead is able to tell us how much the wastage lower bound will increase by, given each of those choices. We order the choices according to how low the wastage lower bound given by the lookahead is. This is far superior to a naive, greedy branching heuristic which orders the choices based on the wastage in only the labeled part of the board, as it is often the case that greedily minimizing wastage locally will cause much more wastage further on. Our lookahead is capable of seeing the wastage caused further on and will not pick such locally optimal but globally suboptimal choices.

We use a modified version of limited discrepancy search [68]. Firstly, rather than defining a discrepancy as any choice which is not the first choice given by the branching heuristic, we define it as the amount that the wastage lower bound given by the relaxation lookahead would increase by if we made this decision. This means that there can be ties, i.e. multiple choices which are all equally good according to the lookahead. In such cases, we randomly tie-break between them. Secondly, we add randomized restarts to the search. At randomized points in time, the search will
backtrack by a random number of rows. This is important, because despite our lookahead, there still exist vast subtrees where no optimal solutions can be found, and a complete search will take an exponential number of nodes to backtrack out of them. The combination of random restarts, plus random tie-breaking between equally good choices, is very effective. Using these techniques, we are able to find optimal solutions to instances as large as $n = 200$ in several hours on average. An optimal solution for $n = 100$ is shown in Figure 8.7.

Figure 8.7: Optimal solution for $n = 100$. 
8.7 Constructing Optimal Solutions for Arbitrarily Large $n$

Through our experimentation and analysis, it became clear that the Still Life Problem is actually fairly well behaved for sufficiently large $n$. We have the following properties: 1) the wastage lower bound is periodic in $n$, 2) there exists optimal periodic edge patterns which achieves this wastage lower bound, 3) it is “easy” to fill in the center of the board without any wastage. Combining these facts together, it seems possible that with a bit of work, we can construct closed form optimal solutions for arbitrarily large $n$.

In Section 8.5, we already worked out wastage lower bounds for all $n$. If we can construct solutions that achieve these wastage lower bounds for all $n$, then we are done. This can be done by solving instances of the Still Life Problem under additional periodic constraints, so that parts of the solution can be tiled indefinitely to produce arbitrarily large optimal solutions (see Figure 8.8). The initial solution is broken up by two horizontal and two vertical cuts into: 4 corner pieces, 4 edge pieces, and 1 center piece. The edge pieces and center pieces must satisfy periodic constraints so that they can be tiled and still satisfy the still life constraints. Furthermore, there are strong restrictions on the amount of wastage that can occur in these tiled pieces. We know that for sufficiently large $n$, the wastage lower bounds from Section 8.4 has a wastage to edge cell ratio of exactly $\frac{10}{27}$. Thus to hit this lower bound, our 4 periodic edge pieces must have precisely this wastage ratio. Also, the periodic center piece must be completely wastage free.

Now, for the periodic edge pieces to have precisely a wastage to edge cell ratio of $\frac{10}{27}$, their period must be a multiple of 27. Unfortunately, if the center piece is $27 \times 27$, then it is impossible for it to be wastage free, since it has an odd number of cells. Thus the minimum period we can use is $2 \times 27 = 54$. The aim then is to solve an instance with $n = n'$ under additional period 54 constraints and wastage constraints on the edge and center pieces. If an optimal solution can be found, then it can be tiled to create optimal solutions for $n = n' + k \times 54$ for any $k \in \mathbb{N}$. Clearly, to solve for all $n$, this has to be done for each value of $n' \mod 54$, so there are 54 cases. We
found that \( n' \) had to be around 150 or higher before the Still Life problem became sufficiently well behaved that the periodic version was solvable. So we had to solve 54 instances of size 150+, one for each modulus class mod 54, under the additional periodic constraints and wastage constraints.

Rather than directly solving such problems from scratch, we decided to utilize the
8.7. CONSTRUCTING OPTIMAL SOLUTIONS FOR ARBITRARILY LARGE N

solutions that we had already found, and try to extend them into periodic solutions by splicing in a periodic section. We first take an optimal solution for \( n = n' - 54 \). We cut it into two pieces vertically at some point. We then move the two pieces apart by 54 cells and try to fill up the gap with a periodic 54 section which satisfies the additional wastage constraints. This periodic 54 section will end up being two of the periodic edge pieces. Secondly, we take this new solution and cut it horizontally at some point. We then move the two pieces apart by 54 cells and do the same again. This new section will end up being the other two periodic edge pieces plus the center piece. Now we end up with an optimal solution for \( n = n' \) which satisfies the periodic constraints and the wastage constraints. Here is a MiniZinc model for the first step. The second step is analogous.

```
int: n; % instance parameter

% four columns covering the splice point from sol of n = n' - 54
array [1..4,1..n] of 0..1: s;

array [1..54,0..n+1] of var 0..1: x; % cell live/dead status

% still life and wastage constraints in 54 by n region
constraint forall (i in 2..53, j in 1..n) (
    table(sl_waste, [x[i-1,j-1], x[i,j-1], x[i+1,j-1], x[i-1,j],
                   x[i,j], x[i+1,j], x[i-1,j+1], x[i,j+1], x[i+1,j+1], 0]));

% boundary conditions
constraint forall (i in 0..n+1) (
    x[i,0] = 0 \x\ x[i,n+1] = 0);

constraint forall (i in 1..52) (
    sum (j in i..i+2) (x[j,1]) <= 2 \x\ sum (j in i..i+2) (x[j,n]) <= 2
```
constraint forall (i in 1..n) (  
x[1,i] = s[3,i] \&\& x[2,i] = s[4,i] \&\&  
x[n-1,i] = s[1,i] \&\& x[n,i] = s[2,i]);

% wastage constraints for boundary

constraint sum (i in 1..54) (x[i,1]) = 34;
constraint sum (i in 1..54) (x[i,n]) = 34;

solve maximize satisfy;

This splicing does not always succeed as the constraints are very, very strong. Whether it is satisfiable or not depends on the initial solution and the point at which we make the cut. In particular, the cuts must be made at a point where both edges have already transitioned into the optimal edge pattern of 3 castles per 1 block periodic pattern. If we could not solve a particular instance after a reasonable time, we tried a different cut point or tried it using a different solution of $n = n' - 54$.

The success rate was around 80%, so most of them succeeded on the first try. After approximately 3000 hours of computation, we were able to find periodic solutions for all 54 cases, and thus the Still Life Problem was solved for all large $n$. The Still Life shown in Figure 8.8 is one such periodic solution where the center sections can be tiled indefinitely to produce optimal solutions for $n = 100 + k \times 54$.

8.8 Conclusion

We have solved the Maximum Density Still Life Problem for all $n$ by combining deep mathematical insights into the problem with clever application of remodeling, lazy clause generation, bounded dynamic programming, relaxations, and custom search. The complete solution consists of four parts: 1) complete search which can solve $n \leq$
50, 2) bounded dynamic programming with relaxation to prove optimal live cell upper bounds for \( n > 50 \), 3) incomplete search for special form solutions which can prove optimal live cell lower bounds for around \( 50 < n \leq 200 \), 4) incomplete search to find optimal periodic solutions which can be tiled to construct arbitrarily large solutions that prove the optimal live cell lower bounds for around \( n > 200 \). The optimal values for \( n \leq 50 \) are given in Table 8.4, and the optimal values for \( n \geq 51 \) are given by Equation 8.4. Optimal solutions for small and medium \( n \) and periodic optimal solutions for large \( n \) can be found at \texttt{www.csse.unimelb.edu.au/~pjs/still-life/}. The total time taken to completely solve the Maximum Density Still Life Problem for all \( n \) was approximately 3000 hours.
Part IV
Parallelization
Parallelization is an obvious way to try to speed up constraint solving, especially in view of the recent proliferation of multi-core systems. There are a number of ways to parallelize CP solvers. The simplest and most popular is search parallelization, where different parts of the search tree are searched by different processors (e.g., [114, 138, 105]). Another possibility is to parallelize the propagation engine by having different processors execute different propagators (e.g., [132, 167]).

The basic search algorithm described in Figure 2.2 is easy to parallelize. Subproblems arising from different parts of the search tree are solved independently, thus they can be partitioned out to different processors to be solved. Communication between processors is only required when a new solution has been found, or if some form of dynamic load balancing is required. Previous work on parallel search has concentrated mainly on load balancing [130, 138], or on developing an appropriate architecture for parallel search [138, 105].

IV.1 Contributions

In Chapter 9, we improve the state of the art SAT solver MiniSat [39] by modifying its data structures to be more cache-aware. Cache contention can be a serious problem when parallelizing SAT or CP solvers. Although the number of cores in modern processors is growing rapidly, the amount of cache available to these cores do not appear to be growing at the same rate. In most multi-core processor designs, L2 and L3 caches are shared between multiple cores, with the result that each core has a lot less L2 and L3 cache on average than in older single processor designs. This is fine if the cores are using the same set of data. However, in SAT solver and Lazy Clause Generation solvers, the memory access patterns are quite different in different parts of the search tree, hence different cores tend to require different parts of the data at any one time. This leads to serious cache contention issues that severely limit the amount of speedup achievable on multi-core systems. Our data structure improvements are able to speed up the sequential version of MiniSat by around 80%, and speed up a parallel version of MiniSat by around 140%. These improvements carry over to Lazy Clause Generation solvers, which have SAT solvers embedded in them.
In Chapter 10, we address an important theoretical issue which has not been properly addressed in the literature. Many analyses of parallel search and load balancing schemes implicitly or explicitly assume that the total amount of work is fixed (e.g. [89]). This assumption is indeed true for unsatisfiable instances with a fixed search tree. However, it is not true for satisfiable satisfaction problems or for optimization problems. In such problems, finding a (good) solution quickly can dramatically reduce the total amount of work required. Much work in CP is concerned with developing strong branching heuristics which can shape and order the branches of the search tree so that (good) solutions are found early on in the search. However, many parallelization schemes, e.g., splitting the search tree as close to the root as possible [138], focus on work granularity and completely ignore the branching heuristic when partitioning the work. This can result in many of the processors searching in unfruitful parts of the search tree. We analyzed this problem and develop a new work stealing scheme called confidence based work stealing, which partitions the work based on how strong the branching heuristic is at each node. The new parallel algorithm produced near linear or super linear speedup on all the problems we tested.
Chapter 9

Cache Conscious Data Structures for Boolean Satisfiability Solvers
9.1 Introduction

Boolean Satisfiability (SAT) solvers find a wide range of application in areas such as electronic design automation and artificial intelligence. A tremendous amount of research has been undertaken to find ways to speed up SAT solvers. Algorithmic improvements in terms of decision heuristics and learning schemes have made modern SAT solvers orders of magnitude faster than their predecessors. Besides the algorithmic improvements, however, the engineering and design of data structures and the core propagation algorithm has also led to substantial increases in speed [109]. A typical modern SAT solver spends some 70-90% of its time performing Boolean constraint propagation, hence optimization of this inner propagation loop is of great importance.

SAT solving often produces highly non-local memory access patterns. Variable and clause reordering schemes have been investigated in the past [4] to try to increase locality of reference, but the non-locality is inherent in SAT solving and can never be completely eliminated.

Memory issues are even more important in parallel SAT solving and can dramatically affect the speed of a parallel SAT solver [44]. Most current multi-core systems contain L2 or L3 caches that are shared between multiple cores. When more than one thread is used, cache contention occurs and the effective cache capacity for each thread is reduced compared to the sequential case. Cache coherency caused by false sharing can also become an issue. If the private data of different threads are mixed together within the same cache line of memory (as could be the case, for example, if each thread used malloc to allocate memory for small chunks of private data like clauses), then any of those threads writing to their private data would force all the other threads to fetch that cache line from memory again, causing an unnecessary cache miss. Main memory access on a multi-core machine is also typically slower than on single processor machines and does not scale well with the number of cores. This causes a huge problem for memory intensive algorithms such as SAT solving. These issues makes it imperative for us to design data structures and algorithms that are as cache efficient as possible.
In this chapter, we modify MiniSAT 2.0 [39], a very well engineered SAT solver that came first in the industrial category of the SAT 2005 and SAT 2006 competitions, to utilize new data structures and algorithms that significantly improve its caching properties and thus its core propagation speed. We also demonstrate our improvements using our parallel solver PMiniSat, which is a simple parallelization of MiniSat 2.0.

The layout of this chapter is as follows. In Section 9.2, we describe in detail the data structures and algorithms currently employed in MiniSat 2.0. In Section 9.3, we describe the data structures and algorithmic modifications we made. In Section 9.4, we present an experimental evaluation of our changes. In Section 9.5, we discuss related work. Finally in Section 9.6 we conclude.

9.2 Boolean Constraint Propagation

As described in Chapter 2, the Boolean Satisfiability problem is a special case of the Constraint Satisfaction problem where all variables are Boolean, and all constraints are in the form of disjunctive clauses, i.e., constraints of form: \( l_1 \lor \ldots \lor l_k \) where each literal \( l_i \) is either a variable \( x \) or its negation \( \neg x \).

State of the art SAT solvers are based on the DPLL algorithm [35], which interleave decision making with constraint propagation. Due to the specialization to clausal constraints, SAT solvers can use more specialized propagation algorithms than their CP counterparts. Modern SAT solvers utilize a two-watched literal scheme [109] for their core propagation engine. In the two-watched literal scheme, only two of the literals in any clause is being “watched” at any time. Whenever one of those watched literals is assigned a value and becomes “false” in the clause, the clause is examined to see if it is capable of causing a propagation. The two-watched literal scheme represents a vast improvement over earlier counter-based schemes used in older SAT solvers, significantly cutting down on the number of memory accesses and the amount of processing required [99].

In the two watched literal scheme, a “watched list” is created for each literal in the problem (both positive and negative literals). Each clause is attached to the watched
lists of the two literals in it that are being watched. Hence the pointer to each clause exists in exactly two of the watched lists at any time. Whenever a variable becomes assigned, the watched list of the literal that became “false” is examined, and each of the clauses pointed to by the pointers in the watched list is examined to determine if something needs to be done.

When a clause is examined, four different things can happen:

(a) One of the literals in the clause is found to have value “true”. In this case, the clause is already satisfied and nothing needs to be done.

(b) A literal other than the other watched literal is found to be unassigned. In this case, the watched literal that just became “false” is replaced by this new unassigned literal as the watched literal, and the clause is attached to the watched list of the new watched literal.

(c) No other unassigned literal can be found, and the other watched literal is unassigned. In this case, the other watched literal is forced to have value “true” and we have a propagation.

(d) no other unassigned literal can be found, and the other watched literal already has value “false”. In this case we have a conflict.

Pseudo-code for Boolean constraint propagation with watch literals is shown in Figure 9.1. The code takes an assignment $A$ and a decision literal $d$. The queue $Q$ of literals to be processed is a FIFO queue implemented by $\text{push}$, $\text{pop}$, $\text{emptyq}$ and $\text{isempty}$. The set $\text{watch}[l]$ is the set of clauses in $T$ watched on literal $l$. $\text{watch1}[C]$ and $\text{watch2}[C]$ are the two watched literals of clause $C$. In certain places, we treat $A$ and $C$ as sets of literals. We can see the four cases defined above in the if elseif elseif else structure, although the case (a) is split into two since the first condition is common and cheap to check.

9.2.1 Boolean constraint propagation in MiniSat 2.0

MiniSat 2.0 utilizes a two-watched literal scheme. This is implemented in MiniSat 2.0 through a vector of “watched lists”, each of which is a vector of pointers pointing
9.2. BOOLEAN CONSTRAINT PROPAGATION

\[
\text{bcp}(A,d) \\
A := A \cup \{d\} \\
Q := \text{push}(\text{EMPTYQ}, d) \\
\text{while } \neg\text{isempty}(Q) \\
(Q, l) := \text{pop}(Q) \\
\text{foreach } C \in \text{watch}[\neg l] \\
\hspace{1em} \text{if (a) } \text{watch}1[C] \in A \lor \text{watch}2[C] \in A \\
\hspace{1em} \hspace{1em} \text{continue} \\
\hspace{1em} \text{elseif (a) } \exists l' \in C \cap A \\
\hspace{1em} \hspace{1em} \text{watch}1[\text{watch}1[C]] := \text{watch}1[\text{watch}1[C]] - \{C\} \\
\hspace{1em} \hspace{1em} \text{watch}1[C] := l' \\
\hspace{1em} \hspace{1em} \text{watch}[l'] := \text{watch}[l'] \cup \{C\} \\
\hspace{1em} \text{elseif (b) } \exists l' \in C, \neg l' \notin A, l' \neq \text{watch}1[C], l' \neq \text{watch}2[C] \\
\hspace{1em} \hspace{1em} \text{watch}[\neg l] := \text{watch}[\neg l] - \{C\} \\
\hspace{1em} \hspace{1em} \text{watch}[l'] := \text{watch}[l'] \cup \{C\} \\
\hspace{1em} \hspace{1em} \text{if } \neg l = \text{watch}1[C] \\
\hspace{1em} \hspace{1em} \hspace{1em} \text{watch}1[C] := l' \\
\hspace{1em} \hspace{1em} \text{else } \text{watch}2[C] := l' \\
\hspace{1em} \text{elseif (c) } \exists l' \in C, \neg l' \notin A \\
\hspace{1em} \hspace{1em} \text{watch}[\text{watch}1[C]] := \text{watch}[\text{watch}1[C]] - \{C\} \\
\hspace{1em} \hspace{1em} \text{watch}1[C] := l' \\
\hspace{1em} \hspace{1em} \text{watch}[l'] := \text{watch}[l'] \cup \{C\} \\
\hspace{1em} \hspace{1em} A := A \cup \{l'\} \\
\hspace{1em} Q := \text{push}(Q, l') \\
\hspace{1em} \text{else (d) } \\
\hspace{1em} \hspace{1em} \text{return } \text{false} \\
\hspace{1em} \text{return } \text{true}
\]

Figure 9.1: Boolean constraint propagation of the decision literal \(d\) given a current assignment \(A\).

to the bodies of the clauses. Some excerpts of the MiniSat 2.0 data structure code together with their memory layout is shown in Figure 9.2.

The clauses have a special data structure in MiniSat 2.0. A clause consists of an initial 32 bits (\texttt{size\_etc}) that contain the size of the clause as well as a few bit flags, followed by another 32 bits (\texttt{extra}) that contain the “activity” of the clause. This is followed by the literals of the clause itself, each of which take 32 bits. The clause is
class Clause {
    uint32_t size_etc;
    union { float act;
        uint32_t abst; } extra;
    Lit data[0];
    ...
};

template<class T>
class vec {
    T* data;
    int sz;
    int cap;
    ...
};

vec<vec<Clause*> > watch;

Figure 9.2: Data structure used in MiniSat 2.0

produced by mallocing the exact amount of memory needed to store the clause.

9.3 Improving the Cache Behavior

9.3.1 Rationale for changes

The two largest sources of memory consumption in MiniSat 2.0 are the watched lists, and the clauses. They are also the largest sources of cache misses. SAT solving typically exhibit highly non-local memory access patterns, as variable assignments can potentially require any clause to be examined, and clauses can potentially cause any variable to become fixed. Thus at any time, a very wide number of clauses and watched lists may potentially need to be examined. This makes caching highly ineffective as there is very little locality of reference. For typical industrial instances where the number of clauses could number in the hundreds of thousands, cache misses can cause a substantial slowdown in the core propagation engine. It has been found that more cache aware SAT solvers like Chaff and Berkmin are faster than their
9.3. IMPROVING THE CACHE BEHAVIOR

predecessors by up to 3 times due to improved cache performance alone [166].

It may seem that with such awareness of cache issues and with so much work having been done to improve the core propagation engine of SAT solvers, there would be little if any improvements left that can be made. However, that turns out not to be the case. Using Cachegrind, a cache simulator in the Valgrind tool suite, we were able to identify and measure various sources of cache misses. The core of a SAT solver is the inner loop of the propagation engine where a clause is examined to see if it can produce a propagation. This loop is run hundreds of millions of times and constitutes the bulk of the processing in a SAT solver. By running Cachegrind on 10 typical medium sized industrial instances, we were able to find out how cache misses were affecting this inner loop. It was found that the SAT solver spent an average of around 80 instructions, 3.2 L1 misses and 0.64 L2 misses per inner loop on the instances tested (more details are provided in the results section). Current computers have L1 miss penalties of the order of 10-15 cycles, and L2 miss penalties of the order of 100-200 cycles, thus these figures show that a very substantial portion of the run time, perhaps 60-70%, is spent simply waiting for data to be read in. Having confirmed that cache misses are indeed a major problem, we targeted each source of cache misses we found in order to design data structures that could reduce them.

In the following sections, we describe each of our modifications in turn. Our changes are aimed at two things. Firstly, to increase locality of reference by using data structures that can pack the data closely together, and second, to eliminate as much data lookup as possible.

9.3.2 Watched list data structure modification

The data structure currently employed by MiniSat 2.0 to store the watched lists is a “vector” data structure. A vector as implemented in MiniSat 2.0 consists of both a head and a body. The head contains the size of the vector, the amount of memory malloced for the body, and a pointer to the body. The body contains the actual elements.

Cache simulations of MiniSat 2.0 on a set of 119 medium sized industrial instance
(see Appendix 1.1) revealed that reading in the watched lists during Boolean constraint propagation causes a significant number of cache misses. The cache misses can be divided up into two groups: misses from reading the head of the watched list vector, and misses from reading the body of the watched list containing the actual data. Interestingly, the number of cache misses from reading the head of the vectors is close to the number of cache misses from reading the body of the vectors even though the head actually contains no real information at all. This is clearly a source of inefficiency.

Further examination shows the cause of the problem. It turns out that in typical industrial instances, watched lists are often very short. The average length of the watched lists depends on the ratio between the number of clauses and the number of variables. This ratio often lies somewhere between 3-6 for the original problem depending on the problem class, and is largely independent of problem size. As more learnt clauses are created, the ratio will increase. But for a large problem, the number of learnt clauses never exceeds the number of original clauses by much. As a result, watched lists often only contain a small number of elements. For example, among the 119 problems we collected statistics for, 105 of them had over 75% of their watched lists having 16 elements or less, with most problems having around 85-90% of their watched lists being short. Further details can be found in Figure 9.3.

This small number of data elements means that most of the time all of the data in the watched list can be stored in just one cache line of memory. Normally, the cost of reading the vector head would be amortized over the cost of reading in the body of the vector. But since the watched list bodies often only require 1 cache line to be pulled in to read all of its data, the memory read required to read in the head becomes relatively expensive. Reading the head and the body separately basically causes 2 memory reads instead of one and doubles the number of reads required to read in the data and hence doubles the number of cache misses.

The vector data structure was probably chosen because the length of a watched list can change, and so any fixed sized data structure would not work. However, this doubling of the number of memory reads is inefficient. The modification we propose is to use custom vector data structures that can store the first few elements of the
watched list in the head part of the vector, and have the rest stored in the body as usual. Thus for a 64 byte cache line for example, we store the usual vector head information in the first 16 bytes, and use the remaining 48 bytes for storing actual data. This gives enough space to store 12 elements in the watched list head. The C++ definition and a diagram of the data structure is shown in Figure 9.4. Since watched lists are usually short, the data would often fit completely within those 48 bytes and the vector body does not need to be examined. Thus we can read the entire watched list by reading in only 1 cache line of memory. This modification basically halves the number of memory reads and cache misses caused by reading in the watched lists. However, some extra processing is required to both read and write to such a data structure.
class WatchedList {
    T* data;
    int sz;
    int cap;
    T head_data[12];
}

Figure 9.4: Watched list data structure

9.3.3 Clause data structure modification

Examination of typical propagation behavior also revealed some interesting statistics. It was found that clauses are often not fully examined during propagation, rather, only the first few literals are examined. Both case (a) and case (b) as described in Section 9.2 can result in the clause not having to be fully examined, as a true or unassigned literal can be found within the first few literals of the clause. Figure 9.5 shows the statistics on the distribution of how many literals of the clause are examined during each propagation. As can be seen, the probability that a clause has to be examined beyond a certain length drops very rapidly after the first few literals. In a very large proportion of the instances we investigated, some 50-90% of clause examinations terminate after just examining the first literal, and some 85-100% of clause examinations terminate within the first 4 literals.

In the original MiniSat, whenever a clause is examined, an entire cache line of data is likely to be pulled in. Since we often only actually need to read the first few literals of the clause, e.g. 16 bytes out of a 64 byte cache line, pulling in an entire cache line consisting of the rest of the clause is wasteful and will tend to flush out other useful data. Furthermore, MiniSat 2.0 used a simple malloc to allocate memory for clauses, and thus we have no guarantee that clauses do not cross cache line boundaries or that there no wasted space between clauses, etc.

The modification we propose is to split up clauses into a head part and a tail part. The head part consists of the first few literals of the clause, which is the part most likely to be examined, and the tail part consists of the rest of the literals. Since only the head part is likely to be examined in any examination of the clause, the head parts of clauses often have better locality of reference with each other than with their
9.3. IMPROVING THE CACHE BEHAVIOR

Figure 9.5: The histogram over the 119 tested instances of the percentage of clauses where \( \leq k \) literals were examined in the inner loop for Boolean constraint propagation. The histogram shows the results for \( k = 1 \) (only first literal examined), \( k = 4 \) and \( k = 8 \).

own tail parts. Furthermore, since we know their exact size, we can manage their memory allocation manually and ensure that they are well packed along cache line boundaries.

The optimal choice for the length of the head part varies depending on the characteristics of the problem instance. It is necessary for a whole number of the clause heads to fit into each cache line, thus the available choice for the clause head sizes would for example 4, 8, 16, etc literals. If a new watch is often found within the first 4 literals in the problem instance, a choice of 4 would be best. If the instance often requires you to go beyond the first 4 literals to find a new watched literal, then a choice of 8 may be better, etc. However, in general, 4 literals for the head seems to be a good choice, as the statistics shows that in most problems, some 85-100% of clause examinations terminate within the first 4 literals. Furthermore, since most of the original clauses are of length 2 or 3, picking 4 literals for the head will allow us
to pack all of the original clauses very tightly together with little wasted space. Our experiments confirm that this is indeed the optimal length when this modification is implemented alone. We will later show, however, that some of our other very effective modifications alter the statistics so significantly that a choice of 4 literals for the head is no longer the optimum when those other modifications are made. This will be discussed in a later section.

We implement our split clause data structure by having a vector of clause heads, and a vector of clause bodies. If a clause has length 4 or less, we simply store those 4 literals in the clause head. If the clause has length greater than 4, we store the first 3 literals in the head and store an index to the rest of the clause in the 4th spot. The body of the clause is also split up into chunks, forming a linked list where each node fits exactly in one cache line. Having fixed size chunks for both clause heads and clause bodies allows us to easily free and reuse blocks of memory that have already been allocated. There is little if any penalty for splitting up the body into chunks, as the body chunks are rarely examined, and when the end of the cache line is reached, a new cache line would have to be read in in any case even if the clause was stored in a contiguous block of memory. A diagram of our data structure is shown in Figure 9.6.

A side benefit of this scheme is that the clauses can now be referred to by a 32 bit index rather than by a pointer, and on 64 bit systems, this cuts the size of the watched lists by half, further improving cache performance.

In the parallel context, since both the clause packing and the watched list modification pack the bulk of the data belonging to each thread into large contiguous arrays that belong to the thread, cache contention caused by false sharing is almost completely eliminated.

### 9.3.4 First literal

Perhaps the most significant statistic we collected is that, as noted above, in a surprisingly large proportion of time, only one literal in the clause is ever examined. It is often the case that the first literal in the clause is already assigned a value of true and thus the clause examination can immediately terminate. This occurred somewhere
between 50-90% of the time on most of the instances we examined. This is due to the way that the literals in the clause are reshuffled by the propagation algorithm. i.e., literals found to have value true are moved to the first spot, and literals that are made “true” by Boolean constraint propagation on that clause are moved to the first spot. This interesting statistic means that in a significant proportion of the time, a clause is read from memory only to read the value of that first literal. This is tremendously inefficient as each such read would pull in for example a 64 byte cache line just to read 4 bytes of data.

The modification we make is to store the first literal of the clause in the watched list itself along with the index to the clause. Then when the watched list is examined to perform propagation, that literal is first checked to see if it is in fact already true, in which case the clause does not have to be read in at all. And if the literal is not true, then the index is used to read in the clause as per usual. This modification effectively eliminates the memory read required to pull in the clause some 50-90% of the time. The cost of this modification is that the size of the watched lists are now doubled. However, the locality of reference is far better on the watched lists than among randomly located clauses in memory so this modification substantially improves the caching properties of the solver.
9.3.5 Binary clauses

An almost trivial extension to the previous modification is to handle binary clauses specially. Since we are already storing one literal of the clause in the watched list itself, it is quite easy to have each of the literals in the binary clause store the other literal in its watched list. We can simply mark the clause index with a bit flag to indicate whether the clause is binary or not, and if it is it can be handled specially. Since knowing the other literal in the clause means that you know everything about the clause, the clause itself never needs to be read in to perform propagation.

The cache benefit of this modification is to some extent negated by the previous modification as the previous improvement has already made it unnecessary to read in a large proportion of clauses. However, experiments show that a significant enough number of binary clauses lookups get through the previous improvement to make this modification worthwhile.

9.3.6 Ternary clauses

A somewhat more non-trivial change is to handle ternary clauses specially as well. Since the first literal modification in Subsection 9.3.4 requires that we store both a 4 byte clause index and a 4 byte first literal in the watched list for each clause, there is already enough space for us to store two literals, and thus we can store all of the information contained in ternary clauses in the watched lists themselves. The different variations of the watched list element are shown in Figure 9.7. This change complicates certain other aspects of the solver but is certainly possible. The benefit of this modification is less clear, as although we would eliminate all lookups of ternary clauses and thus have fewer cache misses, each of those clauses would now be attached to three watched lists instead of two, and thus the processing required to propagate them may increase by as much as 50%. Experiments seem to indicate that there is some benefit however.

As mentioned in Subsection 9.3.3, the optimal choice of the clause head length depends on the statistics of the problem. For the unmodified solver, a length of 4 is optimal. However, once the first literal modification, the binary clause handling
9.3. IMPROVING THE CACHE BEHAVIOR

and the ternary clause handling in Subsections 9.3.4, 9.3.5 and 9.3.6 are implemented as well, the statistics of the problem changes significantly. All binary and ternary clause lookups are now eliminated, as well as all lookups where the first literal is already true. This represents a significant proportion of the cases where the clause examination terminated within the first 4 literals. Thus once these modifications are made it can no longer be said that most remaining clause lookups terminate within the first few literals. The new optimum was found experimentally to be 16 for our set of instances, which coincides with the size of a cache line on our machine, although the difference in speedup is only around 5-10%.

9.3.7 Packing assignments

Besides the above major modifications, there are also some trivial packing of data that can be done. For example, in the original MiniSat 2.0, a single byte is used to represent the current value of each variable. Because of the very non-local way in which variables are examined, it will often be the case that when the algorithm examines one variables value, an entire cache line of assignment data is pulled in, the rest of which will never be used. Since 2 bits is in fact sufficient to store a variables assignment, it is possible to pack the value of 4 variables into each byte. Such packing means that each cache line contains more assignments and there is a greater chance that some of the other assignments pulled in with the cache line have some use.

9.3.8 Expected benefits

Most of the above changes trade more processing for reduced cache misses. Thus they are not guaranteed to improve performance. A benefit would only be gained if
the underlying assumptions for the modification hold, i.e., the source of cache misses being tackled is a major source of slow down, and the statistical assumptions behind the modification hold for the particular instance you are trying to solve. We expect this to be true for all reasonably large instances, e.g., > 50,000 variables. For small, instances however, cache misses are not an issue and the changes may well cause a slow down.

Another important point to note is that the effects of the modifications do not simply add up. In some cases there are synergies such that the total effect of the modifications is greater than the sum. In others, the effect of one modification is replicated by another to some extent and the total effect is less than the sum.

9.4 Experiments

9.4.1 Sequential solving

In the first set of experiments we test the effectiveness of our modifications in the context of sequential SAT solving using modified versions of MiniSat 2.0. An exhaustive test of each combination of data structure modifications is too time consuming. We test each modification in turn (except for the ternary clause handling as it is difficult to implement alone), all of them together, and certain combinations that produce interesting synergies.

The tests were performed on Xeon Pro 2.4GHz processors with 32kb L1 cache, 4Mb L2 cache, 64 byte cache line and 16-way set associativity. The solver was compiled using gcc 3.4.5 using -O3 optimization. 24 medium sized industrial instances from SAT competition 2007 [134] were used (See Appendix 1.2). These instances come from 9 different problem classes. The original version was run 25 times with different random seeds and the modified versions were run 20 times with different random seeds. This brings the standard error down to around 5%. The instruction count and cache miss counts were collected using the Valgrind cache simulator. For the cache simulation, only 10 instances were tested with 1 run each, so these numbers are merely indicative. The results are shown in Table 9.1.
Table 9.1: Cache conscious optimizations for sequential SAT solving. Comparison of various combinations of optimizations: WL — Watched list modification (9.3.2), PC(n) — Clause modification with length n clause head, n = 16 if labeled as just PC (9.3.3), FL — Store first literal in watched list (9.3.4), BC — Handle binary clauses specially (9.3.5), TC — Handle ternary clauses specially (9.3.6) PA — Pack assigns (9.3.7). The results show the (geometric mean) runtime for the suite, the speedup compared to MiniSat 2.0, I/Loop stands for average number of instructions per inner loop of the propagation engine, L1M/Loop and L2M/Loop stands for the average number of L1 and L2 misses per inner loop of the propagation engine.

<table>
<thead>
<tr>
<th>Modifications</th>
<th>Runtime (s)</th>
<th>Speedup</th>
<th>I/Loop</th>
<th>L1M/Loop</th>
<th>L2M/Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniSat 2.0</td>
<td>59.05</td>
<td>—</td>
<td>80.7</td>
<td>3.21</td>
<td>0.64</td>
</tr>
<tr>
<td>WL</td>
<td>62.17</td>
<td>0.95</td>
<td>97.7</td>
<td>3.02</td>
<td>0.53</td>
</tr>
<tr>
<td>BC</td>
<td>62.07</td>
<td>0.96</td>
<td>93.7</td>
<td>3.30</td>
<td>0.69</td>
</tr>
<tr>
<td>PA</td>
<td>59.17</td>
<td>1.00</td>
<td>99.7</td>
<td>2.89</td>
<td>0.62</td>
</tr>
<tr>
<td>PC</td>
<td>64.39</td>
<td>0.92</td>
<td>97.7</td>
<td>3.14</td>
<td>0.65</td>
</tr>
<tr>
<td>PC(8)</td>
<td>54.85</td>
<td>1.08</td>
<td>97.9</td>
<td>3.05</td>
<td>0.48</td>
</tr>
<tr>
<td>PC(4)</td>
<td>49.78</td>
<td>1.19</td>
<td>98.3</td>
<td>2.95</td>
<td>0.35</td>
</tr>
<tr>
<td>FL</td>
<td>47.91</td>
<td>1.25</td>
<td>83.7</td>
<td>2.58</td>
<td>0.39</td>
</tr>
<tr>
<td>PA+PC(4)</td>
<td>45.51</td>
<td>1.30</td>
<td>99.3</td>
<td>2.63</td>
<td>0.34</td>
</tr>
<tr>
<td>WL+PC+FL</td>
<td>38.96</td>
<td>1.53</td>
<td>104.5</td>
<td>2.44</td>
<td>0.29</td>
</tr>
<tr>
<td>WL+PC+FL+BC</td>
<td>35.14</td>
<td>1.70</td>
<td>99.3</td>
<td>2.23</td>
<td>0.24</td>
</tr>
<tr>
<td>WL+PC+FL+BC+TC</td>
<td>33.16</td>
<td>1.80</td>
<td>89.8</td>
<td>1.78</td>
<td>0.16</td>
</tr>
</tbody>
</table>

24 medium sized industrial instances (Appendix 1.2)

22 hard industrial instances (Appendix 1.3)

<table>
<thead>
<tr>
<th>Modifications</th>
<th>Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniSat 2.0</td>
<td>603.23</td>
<td>—</td>
</tr>
<tr>
<td>WL+PC+FL+BC+TC</td>
<td>318.62</td>
<td>1.89</td>
</tr>
</tbody>
</table>

151 industrial instances from SAT Race 2008 (Appendix 1.4)

<table>
<thead>
<tr>
<th>Modifications</th>
<th>Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniSat 2.0</td>
<td>63.41</td>
<td>—</td>
</tr>
<tr>
<td>WL+PC+FL+BC+TC</td>
<td>41.48</td>
<td>1.53</td>
</tr>
</tbody>
</table>

To test that our improvements generalize, we ran the final version with all modifications on two other sets of instances. One set consists of 22 hard instances taking up to several thousands seconds to solve. These were run without any timeout to test the effect of the cache improvements when the problem is difficult and there is a high
proportion of learnt clauses. The second set consists of the 200 industrial instances used in SAT Race 2008. These contain a wide range of problem classes and problem difficulty, including smaller instances for which our cache improvements are not effective. With a timeout of 900 seconds, only 151 of them can be solved consistently. Figure 9.8 shows a scatter plot comparing the runtime of the modified version with the original version on those 151 instances.

As the figures in Table 9.1 show, the modifications all increased the amount of processing required, but most of them compensated for that with reduced cache misses, leading to substantial speedups. Clause packing, when implemented alone, is most effective for a clause head length of 4, which is what we expect given the large number of binary and ternary clauses in the original problems. Clause packing is able to reduce L1 cache misses by around 10% and L2 cache misses by around 45%, resulting in a speedup of around 19%. The first literal modification also produce a significant speedup, reducing L1 cache misses by around 20% and L2 cache misses by around 40%, giving a speedup of around 25%. The case for the watched list modification, binary clause modification and assignments packing is less clear. The watched list
modification is able to reduce L2 cache misses by around 20%, but the extra processing required has apparently outweighed the benefits. A more efficient implementation of the modification may yield a better speedup. The binary clause modification yielded no speedup whatsoever. In fact, it seems slightly inferior to the original. However, it produces some speedup when coupled with the first literal modification (extra 16%), as the data structure changes used in the first literal modification allow the binary clause modification to be implemented with little extra overhead. Assignments packing also yielded no speedup when implemented alone. However, as a comparison of PA+PC(4) with PC(4) shows, assignments packing can produce a non-negligible speedup (extra 11%) when coupled with other modifications. The best combination of modifications WL+PC+FL+BC+TC was able to reduce L1 misses by around 45% and reduce L2 misses by around 75%, leading to a speedup of 80%.

The average speedup gained from the best combination of modifications is around 80% on the initial 24 instances we examined. There is a large amount of individual variation in the speedups of the instances, with the speedup ranging between 19% to as high as 178%. The average speedup on the 22 hard instances is 89%. Thus the result seems to generalize to hard instances with a higher proportion of learnt clauses. The speedup on the 151 solvable SAT Race instances is only 53%. However, around 40% of those are smaller instances where our cache improvements are not supposed to be effective.

In summary, it can be said that the clause packing, first literal modification, and the special binary clause handling each increase the propagation speed by around 15-25%, and the ternary clause handling increases it by another 10%. The changes caused by the watched list modification and the assignments packing are too inconsistent to be considered a real improvement.

9.4.2 Parallel solving

In the second set of experiments we test the effectiveness of our modifications in the context of parallel SAT solving using the parallel SAT solver PMiniSat. Two versions of PMiniSat were made, one with and one without the data structure modifications.
Table 9.2: Cache conscious optimizations for parallel SAT solving. We compare a parallelized version of MiniSat 2.0, PMiniSat, using the original data structures and the combination of optimizations detailed in Table 9.1. The results are the geometric mean of 20 runs.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Original</th>
<th>WL+PC+FL+BC+TC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runtime (s) Efficiency</td>
<td>Runtime (s) Efficiency Speedup</td>
</tr>
<tr>
<td>1</td>
<td>44.03 —</td>
<td>28.30 —</td>
</tr>
<tr>
<td>2</td>
<td>26.94 0.82</td>
<td>15.19 0.93</td>
</tr>
<tr>
<td>3</td>
<td>21.75 0.67</td>
<td>12.70 0.74</td>
</tr>
<tr>
<td>4</td>
<td>19.06 0.58</td>
<td>10.52 0.67</td>
</tr>
<tr>
<td>5</td>
<td>19.90 0.44</td>
<td>10.15 0.56</td>
</tr>
<tr>
<td>6</td>
<td>21.42 0.34</td>
<td>10.70 0.44</td>
</tr>
<tr>
<td>7</td>
<td>23.41 0.27</td>
<td>10.31 0.39</td>
</tr>
<tr>
<td>8</td>
<td>26.44 0.21</td>
<td>11.03 0.32</td>
</tr>
</tbody>
</table>

The tests were performed on an 8 core Mac with 2x Harpertown Quad core Xeon 2.8GHz processors. Each core has a 32kb L1 cache, and each pair of cores share a 6Mb L2 cache with a 64 byte cache line and 24-way set associativity. The solver was compiled using gcc 4.0.1 using -O3 optimization. We run both versions of the solver with 1 to 8 threads. Again the tests are done on the 24 medium sized industrial instances (Appendix 1.2). Each instance was run 20 times with different random seeds. The results are given in Table 9.2.

The sequential (1 thread) speedup due to the modifications is slightly lower on the Mac than on the Xeon Pro. This is to be expected as the solver has access to a full 6Mb of L2 cache on the Mac compared to only 4Mb on the Xeon Pro, thus cache misses are less of an issue and the cache improvements have a smaller effect. The speedups produced by the modifications clearly increase with increasing number of threads, from 56% speedup for 1 thread up to 140% for 8 threads.

The dramatic drop in efficiency as more threads are used is somewhat surprising at first glance, as it is known from other experiments that PMiniSat is highly efficient algorithmically, achieving an efficiency of 103% at 4 threads and 87% at 8 threads on these 24 instances in terms of the total number of executions of the inner loop
over all threads. The communication cost in PMiniSat is also negligible. Further investigation shows that the relatively low efficiency of PMiniSat on this multi-core machine (with or without cache improvements) can be directly attributed to memory costs.

Our cache simulation numbers show that the average rate of L2 cache misses on the unmodified sequential solver was of the order of 10 million misses per second. With a write-back cache policy, and a 64 byte cache line size, this translates into a fairly massive 1.28 Gb/s of traffic between the processor and the main memory. Thus even though the entire clause database of a medium sized industrial problem may only require 20-50 Mb of memory, the highly non-local memory access pattern forces an enormous amount of memory transfer to occur. The situation is far worse for our parallel solver, as the memory bandwidth requirement will be multiplied by the number of threads. Furthermore, issues such as cache contention, shared data and speculative prefetching by the Harpertown processors will increase the amount of memory transfer that each thread needs even further. In the sequential solver, L2 misses were already consuming some 50-60% of the cpu time. From our numbers, it appears that in the parallel solver, this proportion grows to around 75% for 4 threads and around 90% for 8 threads. Thus when the parallel solver is run with many threads, the limiting factor is not processing power, but the ability of the main memory to handle an extremely large number of memory accesses. On the 8-core Mac that the experiments were run on, the main memory is clearly not fast enough to satisfy all 8 cores.

To illustrate this, memory stress tests using a memory access pattern similar to a SAT solver were run on the 8 core Mac. In the stress tests, 64 Mb of contiguous memory is allocated to each thread. Each thread then reads a random location within its block and increments the value. This loop is performed a billion times. As can be seen in table 9.3, the efficiency drops dramatically with increasing number of threads and the trend is very similar to what is observed on the parallel SAT solver.

In the sequential context, our modifications were able to substantially reduce cache misses. In the parallel context, they provide even more speedup because of the other memory issues that occur. When multiple threads share an L2 cache, capacity
Table 9.3: Parallel memory stress test.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Runtime (s)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29.2</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>22.6</td>
<td>0.65</td>
</tr>
<tr>
<td>4</td>
<td>18.2</td>
<td>0.40</td>
</tr>
<tr>
<td>8</td>
<td>16.8</td>
<td>0.22</td>
</tr>
</tbody>
</table>

contention occurs. Since the vast majority of cache misses in SAT solving are capacity
misses, when two threads share the same cache, the effective L2 cache size for each of
them is roughly halved. On the Harpertown processors, each thread would effectively
have 3Mb or less of L2 cache instead of 6Mb for the sequential case. Thus L2 misses
are far higher and our cache improvements produce more speedup. Main memory
contention is clearly a very serious source of inefficiency for parallel SAT solvers.

Our cache improvements reduce L2 misses, which translates into less main memory
accesses and thus less contention, once again producing more speedup. Although our
numbers show that it would be impractical to run our solver with more than 4 or 5
threads on this particular machine, the speedups illustrate very well how much more
effective our modifications are in the parallel context, as our modifications produce a
far greater speedup of 140% with 8 threads than the 56% with 1 thread.

PMiniSat came 2nd in both the sequential and parallel track of SAT Race 2008 [135],
showing that the cache improvements described in this paper can indeed make a solver
highly competitive with other state of the art SAT solvers.

9.5 Related Work

The inefficiency of having the clauses reside in non-contiguous, randomly allocated
memory locations has been noted before, and array style clause memory manage-
ment have been attempted. For example, in Chaff [109], a large chunk of contiguous
memory is initially allocated for clauses. Memory from this pool is then allocated to
clauses as needed. However, in their implementation, the memory is simply allocated
9.5. RELATED WORK

from the end of the pool, and they do not attempt to reuse freed chunks of memory caused by deleted clauses until an infrequent monolithic garbage collection step compactifies the remaining clauses. This somewhat reduces the effectiveness of using an array for the clauses, as clauses are no longer in contiguous parts of memory due to there being deleted clauses in between. Our implementation allocates fixed sized chunks of memory to clauses, thus it is easy for us to keep a stack of freed chunks that can be reused immediately. This makes better use of memory and also produces better locality of reference.

Our first literal modification bears some resemblance to the Watched Literal Reference List (WLRL) used in [96], however, their WLRL data structure is motivated by different considerations. In their parallel solver MiraXT, clauses are read-only and are shared among multiple threads, thus in order for watched literal propagation to occur, each thread has to remember the two literals in the clause that are attached to their watched lists. This data is stored in an extra data structure called the WLRL. Similar to our first literal modification, examining the WLRL can allow the clause examination to terminate without examining the original clause. However, the authors do not examine the cache behavior of this design or whether it can produce any speedups.

The clause data structures used in CMUSAT [79] is also similar to our first literal modification and is motivated by the same reasons. However, instead of storing the first literal in the watched list itself as we do, they store the two watched literals of each clause in a separate array, essentially creating an abridged version of each clause. This is very similar to the WLRL described above. When a clause is examined, their algorithm first examines these two watched literals to see if they are true. If they are, then the main clause is not examined. It seems to us however, that such a design may not decrease cache misses by much. The examination of the two watched literals, which is stored in a separate array from the watched list, is very likely to cause a cache miss as well, as it has the same highly non-local access pattern that the clauses suffer from. And in the case that the main clause has to be examined, their design may end up causing two misses instead of one. Our modification is superior, as storing the first literal in the watched list itself provides far better locality of reference.
The special handling of binary clauses is not a new idea. In fact, MiniSat 1.13 had a similar implementation that was later removed in MiniSat 2.0 for readability reasons. Special handling for ternary clauses have also been considered before, for example [133] discusses ways in which ternary clause could be handled specially by packing the literals together. Their packing scheme involves packing the 3 literals into only 64 bits. This can substantially reduce the memory footprint when a large majority of clauses are of length 3. However, their implementation limits each literal to 20 bits, with 1 bit reserved for a flag. This means that a maximum of 524,288 variables are allowed. Such a restriction may have been reasonable at the time, but is no longer really reasonable given the sizes of the problems that can be solved today. Our changes do not modify the length of a literal, but do utilize two of its bits for special flags. Thus we retain a limit of 536,870,912 variables, which ought to be sufficient for the foreseeable future.

A data structure that tackles cache misses in a different way is presented in [15]. In their solver PicoSAT, the watched lists are stored as a linked list rather than as a vector. A list head points to the first clause being watched, and each clause contains a link to the next clause on the watched list. This redistributes the memory needed to store the watched list into the clauses themselves. Cache misses from reading watched lists should be substantially reduced. However, this data structure is incompatible with the first literal modification presented in this chapter. Their data structure requires that all clauses be examined in order to find the next clause to be examined, however, the purpose of the first literal modification is to avoid having to examine any clauses in the first place. They also handle binary clauses specially by storing them in a separate stack. Their results show that their linked list based solver with special binary clause handling was ~15% faster than the naive vector based data structure as used in MiniSAT. However, this would seem to be inferior to our modifications, as even just the first literal modification alone gives a ~25% speedup.

A number of cache related optimizations are discussed in [72], which was published after the work in this chapter was initially published. Some of these optimizations are very similar to those contained in this chapter, e.g., compacting the data structures used for storing assignments, using a slab allocator to manage memory. They also
discuss a number of optimizations which are already known, such as maintaining the watch list lazily and moving clause literals into the “clause header” (both already implemented in MiniSat 2.0). The novelty in this paper is the use of prefetching during propagation to avoid cache misses, which reportedly gave $\sim 20\%$ speedup when implemented in MiniSat 2.0.

9.6 Conclusion

In this chapter, we have described a series of data structure and algorithmic modifications that were able to increase the core propagation speed of MiniSat 2.0 by some $80\%$ on a wide range of medium sized industrial instances. An even greater speedup of around $140\%$ was obtained when the modifications were incorporated into the parallel SAT solver PMiniSat running with 8 threads. Cache simulations were used to find the main sources of cache misses in MiniSat 2.0. Propagation statistics were then collected that guided us towards the design of more cache conscious data structures and algorithms. Three of our modifications produced unambiguous speedups of the order of $15\text{--}25\%$ each in the sequential context. Our experiments indicated that memory costs dominate execution in the parallel solver, and hence cache improvements are even more important. Our modifications are relatively easy to implement and can easily be incorporated into other state of the art SAT solvers, most of which also utilize the same two watched propagation as MiniSat 2.0.
CHAPTER 9. CACHE CONSCIOUS DATA STRUCTURES FOR SAT
Chapter 10

Confidence Based Work Stealing
10.1 Introduction

One important criteria for the efficient parallelization of CP solvers is load balancing, i.e., distributing the work so that none of the processors go idle due to a lack of work. Architectures for parallel search in constraint programming typically use work stealing for dynamic load balancing, see for example [105, 114, 138, 130]. In work stealing schemes, whenever a processor runs out of work, it would “steal” some workload from a processor that still has work remaining.

Analysis of work stealing schemes often assume that the amount of work to be done is fixed and is independent of the work stealing scheme (e.g., [89]). While this is true for certain kinds of problems (finding all solutions, proving unsatisfiability), it is not true for others (finding a first solution, finding an optimal solution). Such analyses fail to take into account the fact that the place from which work is stolen in the search tree affects the search strategy, and can have a dramatic effect on the efficiency of the search.

Many approaches choose to steal from as close to the root of the search tree as possible, e.g., [138], as this tends to give the greatest granularity of work and minimizes the overhead of work stealing. However, this is not always the best strategy in terms of search order.

**Effect of work stealing.** Let us consider a relatively simple framework for parallel search. One thread begins with ownership of the entire search tree. When a thread finishes searching the subtree it was responsible for, it will steal an unexplored part of the search tree (a job) from its current owner. This continues until a solution is found or the entire search tree has been searched.

**Example 10.1.1.** The first problem we will consider is a simple model for small instances of the Traveling Salesman Problem, similar to [22]. In our experiments, with 8 threads, stealing left and low (as deep in the tree as possible) requires visiting a number of nodes equal to the sequential algorithm, while stealing high (near the root) requires visiting ~ 30% more nodes on average (see Table 10.1).

The explanation is simple. In one example instance, the sequential algorithm finds
the optimal solution after 47 seconds of CPU time, after which it spends another \( \sim 300 \) seconds proving that no better solution exists. When the parallel algorithm is stealing left and low, all threads work towards finding that leftmost optimal solution, and the optimal solution is found in 47 seconds of CPU time as before (wall clock time \( 47/8 \approx 6 \) seconds). After this, the search takes another 300 seconds of CPU time to conclude. Thus we have perfect linear speedup both in finding the optimal solution, and in proving that no better solution exist.

However, when the parallel algorithm is stealing high, only one thread actually explores the leftmost part of the search tree and works towards that leftmost optimal solution. The other seven threads explore other parts of the search tree, unfruitfully in this case. This time, the optimal solution is found in 47 seconds of wall clock time (376 seconds of CPU time!). The algorithm then spends another 200 seconds of CPU time proving optimality. That is, there is no speedup whatsoever for finding the optimal solution, but linear speedup for proving optimality. Since the optimal solution has been found so much later (376 seconds CPU time instead of 47 seconds), the threads search without the pruning benefits of the optimal solution. Hence, the number of nodes searched drastically increases, leading to a great loss of efficiency. Clearly, this effect gets worse as the number of threads increases. □

It may appear from this example that stealing left and low would be efficient for all problems. However, such a strategy can produce at best linear speedup.

Example 10.1.2. Consider the \( n \)-Queens problem. The search tree is very deep and a top level mistake in the branching will not be recovered from for hours.

Stealing low solves an instance within the time limit iff sequential depth first search solves it within the time limit. This is the case when a solution is in the very leftmost part of the search tree (only 4 instances out of 100, see Table 10.2).

Stealing high, in contrast, allows many areas of the search tree to be explored, so a poor choice at the root of the search tree is not as important. Stealing high results in solving 100 out of 100 instances tested. This is clearly far more robust than stealing low, producing greatly super-linear speedup. □
Veron et al [151] claim that linear and super linear speedups can be expected for branch and bound problems, but they fail to note that finding the optimal solution does not parallelize trivially as shown by Example 10.1.1. Rao and Kumar [125] (and others) show that super linear speedup ought to be consistently attainable for finding the first or the optimal solution for certain types of problems. Their analysis is valid if the search tree is random (i.e., the solutions are randomly distributed), but is not valid in systems where a branching heuristic orders the branches based on their likelihood of yielding a solution. The presence of such a branching heuristic makes linear speedup in finding solutions non-trivial. Gendron and Crainic [57] describe the issue and provide a description of how the issue is handled in several systems. In general, the solutions utilize some kind of best-first criterion to guide how the problem is split up (see e.g., [123, 107]).

Contributions. We give a quantitative analysis of how different work stealing strategies affect the total amount of work performed and explain the relationship between branching heuristic strength and optimal search strategy. We introduce confidence-based work stealing as an adaptive algorithm that, when provided with a user-defined confidence, will steal work in a near optimal manner. The confidence is the estimated ratio of solution densities between the subtrees at each node. This chapter shows that confidence-based work stealing leads to very good algorithmic efficiencies, that is, not many more, and sometimes much less, nodes are explored than for sequential DFS (Depth First Search).

Although the analysis is done in the context of parallel search for constraint programming, the analysis is actually about the relationship between branching heuristic strength and the optimal search order created by that branching heuristic. Thus the analysis actually applies to all complete tree search algorithms whether sequential or parallel. It shows that when the assumptions about branching heuristic strength that lie behind standard sequential algorithms such as DFS, Interleaved Depth First Search (IDFS) [103], Limited Discrepancy Search (LDS) [68] or Depth-bounded Discrepancy Search (DDS) [153] are given to the algorithm as confidence estimates, the algorithm produces the exact same search patterns used in those algorithms. Thus the analysis
and algorithm provides a framework which explains/unifies/produces all those standard search strategies. In contrast to the standard sequential algorithms which are based on rather simplistic assumptions about how branching heuristic strength varies in different parts of the search tree, our algorithm can adapt to branching heuristic strength on a node by node basis, potentially producing search patterns that are vastly superior to the standard ones. The algorithm is also fully parallel and thus we have in effect parallellised DFS, IDFS, LDS and DDS.

Confidence-based search shares the idea to explore more promising parts of the search tree first with other approaches such as impact-based search [126], using solution counting [164] and constraint-level advice [147] for guiding search. However, there are two key differences. First, confidence-based search uses standard branchings heuristics that define the shape of the search tree augmented by confidence. This makes the addition of confidence-based search to an existing constraint programming solver straightforward and allows us to reuse existing constraint models with user-defined branchings. Second, confidence-based search is designed to work in parallel.

10.2 Analysis of Work Allocation

In this section we show quantitatively that the strength of the branching heuristic determines the optimal place to steal work from. We will concentrate on the case of solving a satisfaction problem. The case for optimization is related since it is basically just a series of satisfaction problems.

Preliminary definitions. The basic algorithm for a CP solver is given in Figure 2.2. At each search node, we perform propagation until we reach a fixed point. If the problem is not trivially failed or solved, we call a function \( \text{branch}(C, D') \) which divide the problem up into more constrained subproblems. The function \( \text{branch}(C, D') \) returns an ordered set of decision constraints \( \{b_1, \ldots, b_k\} \). We try to solve each of the subproblems \( (V, D', C \land b_i) \) in turn. The function \( \text{branch}(C, D') \) implements what we call the branching heuristic, and defines a search tree. Each node where we have to branch becomes an internal node, and each node where propagation solved the
subproblem becomes a leaf node. Leaf nodes can either be failure nodes or solution nodes. We define the solution density of a search tree $T$ with $k$ nodes and $l$ solution nodes as $l/k$.

**Optimal split for binary nodes.** For simplicity, assume that visiting each node in the search tree has roughly equal cost. Assuming an oracle that provides accurate information on solution density, work stealing from nodes whose subtrees have the highest solution densities will be optimal.

In practice however, the solution density estimates will not be perfect:

1. Any estimate of the solution density of a subtree will have a very high error, with a substantial chance that the solution density is actually zero.

2. The real solution densities, and hence the errors in the estimate, are highly correlated between nearby subtrees, as they share decision constraints from higher up in the tree.

3. The solution density estimate of a subtree should decrease as nodes in that subtree are examined without finding a solution. The causes are as follows:

   (a) As the most fruitful parts of the subtree are searched, the average solution density of the remaining nodes decrease.

   (b) The correlation between solution densities between nearby subtrees mean that the more nodes have failed in that subtree, the more likely the remaining nodes are to fail as well.

We have to take these issues into account when utilizing solutions densities to determine where to steal work. Given the actual solution density probability distribution for the two branches, we can calculate the expected number of nodes searched to find a solution. We derive the expression for a simple case. Suppose the solution density probability distribution is uniform, that is, it has equal probability of being any value between 0 and $S$ where $S$ is the solution density estimate. Let $A$ and $B$ be the solution density estimates for the left and right branch respectively, and assume
a proportion $p$ and $(1 - p)$ of the processing power is sent down the left and right branch respectively. Then the expected number of nodes to be searched is given by the hybrid function:

$$f(A, B, p) = \begin{cases} \frac{1}{pA}(2 + \ln(\frac{pA}{(1-p)B})) & \text{for } pA > (1-p)B \\ \frac{1}{(1-p)B}(2 + \ln(\frac{(1-p)B}{pA})) & \text{otherwise} \end{cases}$$

(10.1)

The shape of this function does not depend on the absolute values of $A$ and $B$ (which only serve to scale the function), but on their ratio, thus the shape is fixed for any fixed value of $r = A/(A + B)$.

The value of $p$ which minimizes the function for given value of $r$ is well approximated by the straight line $p = r$. In fact, the value of the $f$ function at $p = r$ is no more than 2% higher than the true minimum for any $r$ over the range of $0.1 \leq r < 0.9$. For simplicity, we will make this approximation from now on. This means that it is near optimal to divide the amount of processing power according to the ratio of the solution density estimate for the two branches. For example, if $r = 0.9$, which means that $A$ is 9 times as high as $B$, then it is near optimal to send 0.9 of our processing power down the left branch and 0.1 of our processing power down the right.

**Branching confidence.** Define the confidence of a branching heuristic at each node as the ratio $r = A/(A + B)$. The branching heuristic can be considered strong when $r \to 1$, that is the solution density estimate of the left branch is far greater than for the right branch. In other words, the heuristic is really good at shaping the search tree so that solutions are near the left. In this case, our analysis shows that since $r$ is close to 1, we should allocate almost all our processing power to the left branch every time. This is equivalent to stealing work for search as left and as low as possible. The branching heuristic is weak when $r \approx 0.5$, that is, the solution density estimate of the left branch and right branch are similar because the branching heuristic has no insight into where the solutions are. In this case, our analysis shows that since $r = 0.5$, the processing power should be distributed evenly between left and right branches at each node. This is equivalent to stealing work for search as high as possible.
10.3 Adaptive Work Stealing

Our analysis shows that the optimal work stealing strategy depends on the strength of the branching heuristic. Since we have a quantitative understanding of how optimal work stealing is related to branching heuristic strength, we can design a search algorithm that can automatically adapt and produce “optimal” search patterns when given some indication of the strength of the branching heuristic by the problem model. In this section, we flesh out the theory and discuss the implementation details of the algorithm.

10.4 Dynamically Updating Solution Density Estimates

Now we examine how solution density estimates should be updated during search as more information becomes available.

First we need to relate the solution density estimate of a subtree with root \((C, D)\) with the solution density estimate of its child subtrees (the subtrees rooted at its child states \((C \land c_i, \text{solv}(C \land c_i, D))\)). Consider an \(n\)-ary node. Let the subtree have solution density estimate \(S\). Let the child subtree at the \(i\)-th branch have solution density estimate \(A_i\) and have size (number of nodes) \(k_i\). If \(S\) and \(A_i\) are estimates of average solution density, then clearly: \(S = \sum_{i=1}^{n} A_i k_i / \sum_{i=1}^{n} k_i\), i.e., the average solution density of the subtree is the weighted average of the solution densities of its child subtrees.

**Uncorrelated subtrees.** Assuming no correlation between the solution densities of subtrees, we have that if the first \(j\) child subtrees have been searched unsuccessfully, then the updated solution density estimate is \(S = \sum_{i=j+1}^{n} A_i k_i / \sum_{i=j+1}^{n} k_i\). Assuming that \(k_i\) are all approximately equal, then the expression simplifies to:

\[
S = \sum_{i=j+1}^{n} A_i / (n - j)
\]
For example, suppose $A_1 = 0.3, A_2 = 0.2, A_3 = 0.1$, then initially, $S = (0.3 + 0.2 + 0.1)/3 = 0.2$. After branch 1 is searched, we have $S = (0.2 + 0.1)/2 = 0.15$, and after branch 2 is searched, we have $S = (0.1)/1 = 0.1$. This has the effect of reducing $S$ as the branches with the highest values of $A_i$ are searched, as the average of the remaining branches will decrease.

**Correlated subtrees.** Now we consider the case where there are correlations between the solution density estimates of the child subtrees. The correlation is likely since all of the nodes in a subtree share the constraint $C$ of the parent state. Since the correlation is difficult to model we pick a simple generic model. Suppose the solution density estimates for each child subtree is given by $A_i = \rho A'_i$, where $\rho$ represents the effect on the solution density due to the constraint added at the parent node, and $A'_i$ represents the effect on the solution density due to constraints added within branch $i$. Then $\rho$ is a common factor in the solution density estimates for each branch and represents the correlation between them. We have that:

$$S = \frac{\sum_{i=1}^{n} A_i k_i}{\sum_{i=1}^{n} k_i} = \rho \frac{\sum_{i=1}^{n} A'_i k_i}{\sum_{i=1}^{n} k_i}.$$

Suppose that when $j$ out of $n$ of the branches have been searched without finding a solution, the value of $\rho$ is updated to $\rho \frac{n-j}{n}$. This models the idea that the more branches have failed, the more likely it is that the constraint $C$ added at the parent node has already made solutions unlikely or impossible. Then, after $j$ branches have been searched, we have: $S = \rho \frac{n-j}{n} \sum_{i=j+1}^{n} A'_i k_i / \sum_{i=j+1}^{n} k_i$. Assuming that all $k_i$ are approximately equal again, the expression simplifies to: $S = \rho \frac{n-j}{n} \sum_{i=j+1}^{n} A'_i / (n-j) = \frac{\rho}{n} \sum_{i=j+1}^{n} A'_i = \sum_{i=j+1}^{n} A_i / n$. Equivalently, we can write it as:

$$S = \frac{1}{n} \sum_{i=1}^{n} A_i \text{ and } A_i = 0 \text{ for } 1 \leq i \leq j \quad (10.2)$$

where we update $A_i$ to 0 when branch $i$ fails. The formula can be recursively applied to update the solution density estimates of any node in the tree given a change in solution density estimate in one of its subtrees.
Using confidence. In all of our results, the actual values of the solution densities are not required. We can formulate everything using confidence, the ratio between the solution densities of the different branches at each node. In terms of confidence, when a subtree is searched and fails, the confidence values should be updated as follows:

Let \( r_i \) be the confidence value of the node \( i \) levels above the root of the failed subtree and \( r_i' \) be the updated confidence value. Let \( \bar{r}_i = r_i, \bar{r}_i' = r_i' \) if the failed subtree is in the left branch of the node \( i \) levels above the root of the failed subtree and \( \bar{r}_i = 1 - r_i, \bar{r}_i' = 1 - r_i' \) otherwise. Then:

\[
\bar{r}_i' = (\bar{r}_i - \prod_{k=1}^{i} \bar{r}_k)/(1 - \prod_{k=1}^{i} \bar{r}_k)
\] (10.3)

### 10.5 Confidence Model

Given a confidence at each node, we now know how to steal work “optimally” and how to update confidence values as search proceeds. But how do we get an initial confidence at each node. Ideally, the problem modeler, with expert knowledge about the problem and the branching heuristic can develop a solution density heuristic that gives a confidence value at each node. However, this may not always happen, perhaps due to a lack of time or expertise. We can simplify things by using general confidence models. For example, we could assume that the confidence takes on an equal value \( \text{conf} \) for all nodes. This is sufficient to model general ideas like: the heuristic is strong or the heuristic is weak. Or we could have a confidence model that assigns \( r = 0.5 \) to the top \( d \) levels and \( r = 0.99 \) for the rest. This can model ideas like the heuristic is weak for the first \( d \) levels, but very strong after that, much like the assumptions used in DDS [153].

### 10.6 Algorithm

Given that we have a confidence value at each node, our confidence-based search algorithm will work as follows. The number of threads down each branch of a node is updated as the search progresses. When search for a subtree is finished, the confidence
values of all nodes above the finished subtree are updated as described by Equation (10.3) above.

When work stealing is required, we start at the root of the tree. At each node we use the number of threads down each branch, the confidence value, to determine which branch to take. Given the number of threads down each branch is currently $a$ and $b$ respectively then if $|(a + 1)/(a + b + 1) - r| \leq |a/(a + b + 1) - r|$ we go left, otherwise right (i.e., which move would split the work closer to the confidence value). We continue this process until we find an unexplored node, at which point we steal the subtree with that unexplored node as root.

There is an exception to this. Although we may sometimes want to steal as low as possible, we cannot steal too low, as then the granularity would become too small and communication costs will dominate the runtime. Thus we dynamically determine a granularity bound under which threads are not allowed to steal, e.g., 15 levels above the average fail depth. If the work stealing algorithm guides the work stealing to the level of the granularity bound, then the last unexplored node above the granularity bound is stolen instead. The granularity bound is dynamically adjusted to maintain a minimum average subtree size so that work stealing does not occur more often than a certain threshold.

Since the confidence values are constantly updated, the optimal places to search next changes as search progresses. In order for our algorithm to adapt quickly, we do not require a thread to finish the entire subtree it has stolen before stealing again. Instead, after a given restart time has passed, the thread returns the unexplored parts of its subtree to a master coordinating work stealing and steals work again from the top. The work frontier is stored at the master and the work is resumed when work stealing guides a thread to the area again (similar to the idea used in interleaving DFS [103]).

**Example with reasonably high confidence.** Suppose we know that the branching heuristic is reasonably strong, but not perfect. We may use $\text{conf} = 0.8$.

Suppose we have 8 threads. Initially, all confidence values are 0.8. When the 8 threads attempt to steal work at the root, thread 1 will go down the left hand side.
Thread 2 will go down the left hand side as well. Thread 3 will go down the right hand side. Thread 4 will go down the left hand side, etc, until 6 threads are down left and 2 threads are down right. At node $b$, we will have 5 threads down the left and 1 thread down the right, and so on. The work stealing has strongly favored sending threads towards the left side of each node because of the reasonably high confidence values of 0.8. The result is shown in Figure 10.1(a).

Suppose as search progresses the subtree starting at node $d$ finishes without producing a solution. Then we need to update the confidence values. Using Equation (10.3), the confidence value at node $b$ becomes 0, and at node $a$ 0.44. Now when the threads steal work from the root, the situation has changed. Since one of the most fruitful parts of the left branch has been completely searched without producing a solution, it has become much less likely that there is a solution down the left branch. The updated confidence value reflects this. Now the threads will be distributed such
that 4 threads are down the left branch and 4 threads are down the right branch, as shown in Figure 10.1(b).

Next, perhaps the subtree starting at node $j$ finishes. The confidence value at node $e$ then becomes 0, the confidence value at node $b$ remains 0 and the confidence value at node $a$ becomes 0.14. The vast majority of the fruitful places in the left branch has been exhausted without finding a solution, and the confidence value at the root has been updated to strongly favor the right branch. The threads will now be distributed such that 7 threads go down the right and 1 thread goes down the left, as shown in Figure 10.1(c).

Next, suppose the subtree starting at node $f$ finishes. The confidence value at node $c$ becomes 0 and the confidence value at node $a$ becomes 0.44. Since the most fruitful part of the right branch has also failed, the confidence value now swings back to favor the left branch more, as shown in Figure 10.1(d). This kind of confidence updating and redistribution of threads will continue, distributing the threads according to the current best solution density estimates. In our explanation here, for simplicity we only updated the confidence values very infrequently. In the actual implementation, confidence values are updated after every job is finished and thus occur much more frequently and in much smaller sized chunks.

**Example with low confidence.** For the second example, suppose we know that the heuristic is very bad and is basically random. We may use $conf = 0.5$, i.e., the initial solution density estimates down the left and right branch are equal.

Suppose we have 4 threads. Initially, all the confidence values are 0.5. The threads will distribute as shown in Figure 10.2(a). This distributes the threads as far away from each other as possible which is exactly what we want. However, if the search tree is deep, and the first few decisions that the each threads makes are wrong, all threads may still get stuck and never find a solution.

This is where the restart limit kicks in. After a certain time threshold is reached, the threads abandon their current search and begin work stealing from the root again. Since the confidence values are updated when they abandon their current job, they take a different path when they next steal work. For example, if the thread down
node \( e \) abandons after having finished a subtree with root node at depth 10, then the confidence at node \( e \) becomes 0.498, the confidence at node \( b \) becomes 0.499, and the confidence at node \( a \) becomes 0.4995. Then when a thread steals work from the root, it will again go left, then right. When it gets to node \( e \) however, the confidence value is 0.498 and there are no threads down either branch, thus it will go right at this node instead of left like last time. The result is shown in Figure 10.2(b). The updated confidence value has guided the thread to an unexplored part of the search tree that is as different from those already searched as possible. This always happens because solution density estimates are decremented whenever a part of a subtree is found to have failed, so the confidence will always be updated to favor the unexplored parts of the search tree.

**Emulating standard search patterns.** As some other examples, we briefly mention what confidence models lead to some standard search patterns. DFS: \( \text{conf} = 1 \), \( \text{restart} = \infty \). IDFS: \( \text{conf} = 1 \), \( \text{restart} = 1000 \). LDS: \( \text{conf} = 1-\epsilon \), \( \text{restart} = 1 \) node. DDS: \( \text{conf} = 0.5 \) if depth < \( d \), \( 1-\epsilon \) if depth \( \geq d \), \( \text{restart} = 1 \) node.
10.7 Experiments

Confidence-based work stealing is implemented using Gecode 3.0.2 [56] with an additional patch to avoid memory management contention during parallel execution. The benchmarks are run on a Mac with $2 \times 2.8 \text{ GHz}$ Xeon Quad Core E5462 processors with 4Gb of memory. However, due to memory limitations, we could not run the large instances of n-Queens or Knights on this machine. We run those two benchmarks on a Dell PowerEdge 6850 with $4 \times 3.0 \text{ GHz}$ Xeon Dual Core Pro 7120 CPUs with 32Gb of memory. 8 threads are used for the parallel search algorithm. We use a time limit of 20 min CPU time (2.5 min wall clock time for 8 threads), a restart time of 5 seconds, and a dynamic granularity bound that adjusts itself to try to steal no more than once every 0.5 seconds. We collected the following data: wall clock runtime, CPU utilization, communication overhead, number of steals, total number of nodes searched and number of nodes explored to find the optimal solution.

Optimization problems. In our first set of experiments we examine the efficiency of our algorithm for three optimization problems from Gecode’s example problems. The problems are: Traveling Salesman Problem (TSP), Golomb-Ruler and Queens-Armies. A description of these problems can be found at [56]. We use the given search heuristic (in the Gecode example file) for each, except for TSP where we try both a strong heuristic based on maximizing cost reduction and a weak heuristic that just picks variables and values in order. For TSP, we randomly generated many instances of an appropriate size for benchmarking. Only the size 12 and size 13 instances of Golomb Ruler, and only the size 9 and size 10 instances of Queen-Armies, are of an appropriate size for benchmarking. We use the simple confidence model with $conf = 1, 0.66$ and $0.5$. The results are given in Table 10.1.

It is clear that in all of our problems, runtime is essentially proportional to the number of nodes searched, and it is highly correlated to the amount of time taken to find the optimal solution. The quicker the optimal solution is found, the fewer the nodes searched and the lower the total runtime. The communication cost, which
Table 10.1: Experimental results for optimization problems with simple confidence model. The results show: wall clock runtime in seconds (Runtime), speedup relative to the sequential version (Speedup), and runtime efficiency (RunE) which is Speedup/8, CPU utilization (CPU%), communication overhead (Comm%), number of steals (Steals), total number of nodes explored (Nodes), the algorithmic efficiency (AlgE) the total number of nodes explored in the parallel version versus the sequential version, the number of nodes explored to find the optimal solution (Onodes), and the solution finding efficiency (SFE) the total number of nodes explored in the parallel version to find the optimal versus the sequential version. Values for Runtime, CPU%, Comm%, Steals, Nodes, and Onodes are the geometric mean of the instances solved by all 4 versions.

<table>
<thead>
<tr>
<th>conf</th>
<th>Runtime</th>
<th>Speedup</th>
<th>RunE</th>
<th>CPU%</th>
<th>Comm%</th>
<th>Steals</th>
<th>Nodes</th>
<th>AlgE</th>
<th>Onodes</th>
<th>SFE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>313.3</td>
<td>—</td>
<td>—</td>
<td>100%</td>
<td>0%</td>
<td>—</td>
<td>5422k</td>
<td>—</td>
<td>1572k</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>38.2</td>
<td>7.25</td>
<td>0.91</td>
<td>96.5%</td>
<td>0.4%</td>
<td>708</td>
<td>5357k</td>
<td>1.01</td>
<td>1589k</td>
<td>0.99</td>
</tr>
<tr>
<td>0.66</td>
<td>47.2</td>
<td>5.88</td>
<td>0.74</td>
<td>93.7%</td>
<td>0.1%</td>
<td>319</td>
<td>6657k</td>
<td>0.81</td>
<td>5130k</td>
<td>0.31</td>
</tr>
<tr>
<td>0.5</td>
<td>48.0</td>
<td>5.77</td>
<td>0.72</td>
<td>92.9%</td>
<td>0.1%</td>
<td>467</td>
<td>6747k</td>
<td>0.80</td>
<td>5275k</td>
<td>0.30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conf</th>
<th>Runtime</th>
<th>Speedup</th>
<th>RunE</th>
<th>CPU%</th>
<th>Comm%</th>
<th>Steals</th>
<th>Nodes</th>
<th>AlgE</th>
<th>Onodes</th>
<th>SFE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>347.8</td>
<td>—</td>
<td>—</td>
<td>99.0%</td>
<td>0%</td>
<td>—</td>
<td>7.22M</td>
<td>—</td>
<td>1.15M</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>46.7</td>
<td>7.45</td>
<td>0.93</td>
<td>99.4%</td>
<td>0.6%</td>
<td>1044</td>
<td>6.96M</td>
<td>1.04</td>
<td>1.09M</td>
<td>1.06</td>
</tr>
<tr>
<td>0.66</td>
<td>45.8</td>
<td>7.60</td>
<td>0.95</td>
<td>96.9%</td>
<td>0.1%</td>
<td>379</td>
<td>7.02M</td>
<td>1.03</td>
<td>1.10M</td>
<td>1.05</td>
</tr>
<tr>
<td>0.5</td>
<td>41.6</td>
<td>8.36</td>
<td>1.05</td>
<td>97.5%</td>
<td>0.1%</td>
<td>304</td>
<td>6.36M</td>
<td>1.14</td>
<td>0.96M</td>
<td>1.20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conf</th>
<th>Runtime</th>
<th>Speedup</th>
<th>RunE</th>
<th>CPU%</th>
<th>Comm%</th>
<th>Steals</th>
<th>Nodes</th>
<th>AlgE</th>
<th>Onodes</th>
<th>SFE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>562</td>
<td>—</td>
<td>—</td>
<td>100%</td>
<td>0%</td>
<td>—</td>
<td>9.71M</td>
<td>—</td>
<td>1.07M</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>69.0</td>
<td>8.15</td>
<td>1.02</td>
<td>99.3%</td>
<td>0.2%</td>
<td>572</td>
<td>8.96M</td>
<td>1.08</td>
<td>0.81M</td>
<td>1.33</td>
</tr>
<tr>
<td>0.66</td>
<td>59.0</td>
<td>9.54</td>
<td>1.19</td>
<td>99.3%</td>
<td>0.1%</td>
<td>346</td>
<td>7.58M</td>
<td>1.28</td>
<td>0.49M</td>
<td>2.21</td>
</tr>
<tr>
<td>0.5</td>
<td>65.2</td>
<td>8.63</td>
<td>1.08</td>
<td>99.3%</td>
<td>0.1%</td>
<td>259</td>
<td>8.42M</td>
<td>1.15</td>
<td>0.66M</td>
<td>1.63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conf</th>
<th>Runtime</th>
<th>Speedup</th>
<th>RunE</th>
<th>CPU%</th>
<th>Comm%</th>
<th>Steals</th>
<th>Nodes</th>
<th>AlgE</th>
<th>Onodes</th>
<th>SFE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>602</td>
<td>—</td>
<td>—</td>
<td>100%</td>
<td>0%</td>
<td>—</td>
<td>13.6M</td>
<td>—</td>
<td>845k</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>87.1</td>
<td>6.91</td>
<td>0.86</td>
<td>99.3%</td>
<td>0.7%</td>
<td>1521</td>
<td>14.5M</td>
<td>0.94</td>
<td>1878k</td>
<td>0.45</td>
</tr>
<tr>
<td>0.66</td>
<td>86.3</td>
<td>6.98</td>
<td>0.87</td>
<td>98.8%</td>
<td>0.2%</td>
<td>1143</td>
<td>14.5M</td>
<td>0.96</td>
<td>2687k</td>
<td>0.31</td>
</tr>
<tr>
<td>0.5</td>
<td>86.0</td>
<td>7.00</td>
<td>0.87</td>
<td>99.5%</td>
<td>0.2%</td>
<td>983</td>
<td>14.5M</td>
<td>0.95</td>
<td>2816k</td>
<td>0.30</td>
</tr>
</tbody>
</table>

includes all work stealing and synchronization overheads, is less than 1% for all problems.
The strong heuristic in TSP is quite strong. Using \( \text{conf} = 1 \) achieves near perfect algorithmic efficiency, where algorithm efficiency is defined as the total number of nodes searched in the parallel algorithm vs the sequential algorithm. Other values of \( \text{conf} \) clearly cause an algorithmic slowdown. The optimal solution is found on average 3.2 and 3.3 times slower for \( \text{conf} = 0.66 \) and 0.5 respectively, resulting in an algorithmic efficiency of 0.81 and 0.80 respectively. The opposite is true when the weak heuristic is used. Using \( \text{conf} = 1 \) or 0.66 allows us to find the leftmost optimal solution in approximately the same number of nodes as the sequential algorithm, but using \( \text{conf} = 0.5 \) to reflect that the heuristic is weak allows the algorithm to find the optimal solution even faster, producing an algorithmic efficiency of 1.14.

The branching heuristic in Golomb Ruler is a greedy heuristic that selects the minimum possible value for the variable at each stage. This is a reasonable heuristic but by no means perfect. It turns out that for Golomb Ruler 12 and 13, the optimal solution does not lie directly in the left-most branch, and a certain degree of non-greediness leads to super-linear solution finding efficiencies.

The results for Queens-Armies show little difference depending on confidence. Clearly the heuristic is better than random at finding an optimal solution, and solution finding efficiency degrades slightly as we ignore the heuristic. But the overall nodes searched are almost identical for all confidence values, as the work required for the proof of optimality make up the bulk of the run time, and the proof of optimality parallelizes trivially regardless of confidence.

**Satisfaction problems.** In our second set of experiments we examine the efficiency of our algorithm for three satisfaction problems from Gecode’s examples [56]. The problems are: \( n \)-Queens, Knights, and Perfect-Square.

The sequential version solved very few instances of \( n \)-Queens and Knights. Furthermore, all those solves are extremely fast (< 3 sec) and are caused by the search engine finding a solution at the very leftmost part of the search tree. Most of the time spent in those runs is from moving down to the leaf of the search tree rather than actual search and is not parallelizable, thus comparison of the statistics for the parallel vs sequential algorithms on those instances is not meaningful as there is very
Table 10.2: Experimental results for satisfaction problems with simple confidence model

<table>
<thead>
<tr>
<th>conf</th>
<th>Solved</th>
<th>Runtime</th>
<th>Speedup</th>
<th>RunE</th>
<th>CPU%</th>
<th>Comm%</th>
<th>Steals</th>
<th>Nodes</th>
<th>AlgE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>4</td>
<td>2.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1859</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>10.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1835</td>
<td>86.6%</td>
</tr>
<tr>
<td>0.66</td>
<td>29</td>
<td>18.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15108</td>
<td>81.6%</td>
</tr>
<tr>
<td>0.5</td>
<td>100</td>
<td>2.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14484</td>
<td>65.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conf</th>
<th>Solved</th>
<th>Runtime</th>
<th>Speedup</th>
<th>RunE</th>
<th>CPU%</th>
<th>Comm%</th>
<th>Steals</th>
<th>Nodes</th>
<th>AlgE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>7</td>
<td>0.22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1212</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0.26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1150</td>
<td>68.1%</td>
</tr>
<tr>
<td>0.66</td>
<td>13</td>
<td>0.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8734</td>
<td>48.0%</td>
</tr>
<tr>
<td>0.5</td>
<td>21</td>
<td>0.66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8549</td>
<td>35.2%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>conf</th>
<th>Solved</th>
<th>Runtime</th>
<th>Speedup</th>
<th>RunE</th>
<th>CPU%</th>
<th>Comm%</th>
<th>Steals</th>
<th>Nodes</th>
<th>AlgE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>15</td>
<td>483.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>213k</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>72.3</td>
<td>6.68</td>
<td>0.83</td>
<td></td>
<td></td>
<td></td>
<td>216k</td>
<td>0.99</td>
</tr>
<tr>
<td>0.66</td>
<td>14</td>
<td>71.2</td>
<td>6.78</td>
<td>0.85</td>
<td></td>
<td></td>
<td></td>
<td>218k</td>
<td>0.98</td>
</tr>
<tr>
<td>0.5</td>
<td>82</td>
<td>8.9</td>
<td>54.02</td>
<td>6.75</td>
<td></td>
<td></td>
<td></td>
<td>32k</td>
<td>6.64</td>
</tr>
</tbody>
</table>

little work to parallelize. The number of instances solved is a more interesting statistic and is a better means of comparison. The parallel algorithm beats the sequential algorithm by an extremely large margin in terms of the number of instances solved.

The $n$-Queens and Knights problems both have very deep subtrees. Once the sequential algorithm fails to find a solution in the leftmost subtree, it will often end up stuck forever. Modeling the fact that the branching heuristic is very weak at the top by using $conf = 0.5$ clearly produce a super linear speedup. The parallel algorithm solves 100 out of 100 instances of $n$-Queens, compared to 4 out of 100 instances for the sequential algorithm or the parallel algorithm with $conf = 1$. The speedup cannot be measured as the sequential algorithm does not terminate for days when it fails to find a solution quickly. Similarly the parallel algorithm with $conf = 0.5$ solved 21 instances of Knights compared to 7 for the sequential and the parallel version with $conf = 1$.

The branching heuristic for the Perfect Square problem is better than random, but is still terribly weak. Using $conf = 0.5$ to model this once again produces super
Table 10.3: Experimental results showing Nodes and algorithmic efficiency (AlgE) using accurate confidence values, where we follow the confidence value to degree $\alpha$.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Seq</th>
<th>$\alpha = 1$</th>
<th>$\alpha = 0.5$</th>
<th>$\alpha = 0$</th>
<th>$\alpha = -0.5$</th>
<th>$\alpha = -1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Golomb-Ruler 12</td>
<td>5.31M</td>
<td>2.24M 2.37</td>
<td>3.48M 1.53</td>
<td>4.27M 1.24</td>
<td>10.8M 0.49</td>
<td>10.6M 0.50</td>
</tr>
<tr>
<td>Golomb-Ruler 13</td>
<td>71.0M</td>
<td>53.2M 1.34</td>
<td>57.6M 1.23</td>
<td>61.9M 1.15</td>
<td>74.8M 0.95</td>
<td>111M 0.64</td>
</tr>
</tbody>
</table>

linear speedup, solving 82 instances out of 100 compared to 15 out of 100 for the sequential algorithm. We can compare run times for this problem as the sequential version solved a fair number of instances and those solves actually require some work (483 sec on average). The speedup in this case is 54 using 8 threads.

Using accurate confidence values. So far, we have tested the efficiency of our algorithm using simple confidence models where the confidence value is the same for all nodes. This does not really illustrate the algorithm’s full potential. We expect that the algorithm should perform even better when node-specific confidence values are provided. In our third set of experiments, we examine the efficiency of our algorithm when node specific confidence values are provided.

Due to our lack of domain knowledge, we will not attempt to write a highly accurate confidence heuristic. Rather, we will simulate one by first performing an initial full search of the search tree to find all solutions, then produce confidence estimates for the top few levels of the search tree using several strategies like, follow the measured solution density exactly, follow it approximately, ignore it, go against it, etc, to see what effect this has on runtime. Let $\alpha$ quantify how closely we follow the measured confidence value and let $conf$ be the measured confidence value. Then we use the following formula for our confidence estimate: $conf' = \alpha \times conf + (1 - \alpha) \times 0.5$. If $\alpha = 1$, then we follow it exactly. If $\alpha = -1$, we go against it completely, etc. We use the Golomb-Ruler problem for our experiment as the full search tree is small enough to enumerate completely. The results are shown in Table 10.3.

The results show that using confidence values that are even a little biased towards the real value is sufficient to produce super linear speedup. And not surprisingly, going against the real value will result in substantial slowdowns.
10.8 Conclusion

By analyzing work stealing schemes using a model based on solution density, we were able to quantitatively relate the strength of the branching heuristic with the optimal place to steal work from. This leads to an adaptive work stealing algorithm that can utilize confidence estimates to automatically produce “optimal” work stealing patterns. The algorithm produced near perfect or better than perfect algorithmic efficiency on all the problems we tested. In particular, by adapting to a steal high, interleaving search pattern, it is capable of producing super linear speedup on several problem classes. The real efficiency is lower than the algorithmic efficiency due to hardware effects, but is still quite good at a speedup of at least 7 at 8 threads. Communication costs are negligible on all problems even at 8 threads.
Chapter 11

Conclusion
In this thesis, we have presented important contributions to a number of areas in Constraint Programming, including nogood learning, symmetry breaking, relaxations, dominance and parallelization.

Nogood learning is undoubtedly one of the most important techniques for improving combinatorial optimization. Its application in the specialized field of Boolean Satisfiability led to solvers which are orders of magnitudes faster than their predecessors and to massive industrial adoption. Similar gains are now being seen when nogood learning techniques are applied to CP solvers. In Chapter 3, we presented a new nogood learning technique based on constraint projection which can be used to exploit subproblem dominances that arise when different search paths lead to equivalent subproblems. Our new method can provide orders of magnitude speedup on appropriate problems. It is competitive with Lazy Clause Generation and can be exponentially stronger on some problems. An important avenue of future work is to understand more fully the strengths and weaknesses of these different nogood learning techniques so that we can combine their strengths together.

Symmetry breaking techniques are critical for solving problems with symmetries efficiently. In Chapter 4, we presented a new symmetry breaking technique called SBDS-1UIP, which uses symmetric versions of the 1UIP nogoods derived by Lazy Clause Generation solvers to prune symmetric parts of the search space. We showed that SBDS-1UIP can exploit at least as many symmetries as the original SBDS method, and that it can exploit types of symmetries that no previous general symmetry breaking technique is capable of exploiting. In particular, SBDS-1UIP can exploit any symmetry that holds in the sub-component of the problem which caused the failure. This is a fundamental step forward in symmetry breaking, as SBDS-1UIP can be strictly stronger than complete methods like SBDS or SBDD. An interesting avenue of future work would be to explore what other kinds of symmetries can be exploited using different types of nogoods, e.g., the nogoods derived using constraint projection as described in Chapter 3.

Partially broken symmetries frequently occur in industrial problems. Such almost symmetries can often still be exploited for significant speedups. Although there has been some work on almost symmetries, most of it is theoretical and there exists
only one previous implementation of a general method. In Chapter 5, we formalized almost symmetries and presented two new general methods for exploiting almost symmetries. The first is to treat almost symmetries as conditional symmetries and exploit them by posting conditional symmetry breaking constraints. The second is to modify SDBS-1UIP to handle almost symmetries. Both techniques are capable of producing orders of magnitude speedups on appropriate problems. This work is an important contribution to the still immature, but promising field of almost symmetry breaking.

Dominance techniques are another important way to improve combinatorial optimization. Identifying dominance relations allow us to prune off parts of the search tree which are suboptimal, and can often lead to large speedups. In Chapter 6, we identified 4 different dominance relations for the Minimization of Open Stacks Problem (MOSP), which when exploited, provide some 5-6 orders of magnitude speedup. In Chapter 7, we identified 2 dominance relations for the Talent Scheduling Problem which provides \( \sim 2 \) times speedup. Dominance is a generalization of symmetry. However, unlike symmetry, it is not well studied in the literature, despite its huge potential for improving combinatorial optimization. An important avenue of future work would be to develop a fuller understanding of dominance and to develop generic dominance methods that can be applied across a wide range of problems.

Relaxation techniques can provide very significant speedups on appropriate problems. In Chapter 6, we identified an effective relaxation to the Minimization of Open Stacks Problem that allows us to speed up the proof of optimality on the hardest instances by some 3-4 orders of magnitude. In Chapter 7, we presented a lower bound calculation for the Talent Scheduling Problem based on relaxations which provides around 1 order of magnitude speedup. In Chapter 8, we tackled the extremely hard Maximum Density Still Life Problem. The previous state of the art could only solve up to \( n = 20 \). By using an appropriate relaxation technique, we were able to change the asymptotic complexity of the problem from \( O(2^n) \) to \( O(n) \). This relaxation technique, combined with remodeling, bounded dynamic programming and customized search, allowed us to completely solved the Maximum Density Still Life Problem for all \( n \). An interesting avenue of future work would be to see if these relaxation
techniques can be generalized and applied to other problems as well.

Parallelization is another way to improve combinatorial optimization. In Chapter 9, we identified cache contention as a serious bottleneck that severely limits the amount of speedup achievable by parallel SAT and LCG solvers on multi-core systems. We alter the data structures used in the state of the art SAT solver MiniSat [39] to be more cache aware, leading to a sequential SAT solver that is some 80% faster on average and a parallel SAT solver that is some 140% faster on average.

In Chapter 10, we examined the interaction between branching heuristics and dynamic load balancing schemes used in parallel solvers. Many commonly used load balancing schemes focus on work granularity and completely ignore the branching heuristic, causing many of the processors to search in unfruitful parts of the search tree. We analyze this issue and develop a new work stealing scheme which partitions work based on how strong the branching heuristic is at each node. This analysis addresses an important theoretical issue that has largely been ignored in the literature, and is critical for the efficient parallelization of CP solvers. Our new parallel algorithm produced near linear or super linear speedup on all the problems we tested. An interesting avenue of future work would be to find a way to automate the process of estimating the strength of the branching heuristic at each node, as our current method requires the problem modeler to provide this information.

In this thesis, we have presented many new and powerful techniques for improving combinatorial optimization. The application of these techniques should provide significant speedups on a wide range of combinatorial optimization problems, translating into time, resource and monetary savings for industry.
Bibliography


Appendix A

Appendix

1.1 119 industrial SAT instances

These examples were taken from the SAT Competition and SAT Race instances available at
http://www.satcompetition.org/ and

1.2 24 medium sized industrial instances

AProVE07-04.cnf
AProVE07-06.cnf
AProVE07-15.cnf
AProVE07-22.cnf
blocks-4-ipc5-h21-unknown.cnf
clauses-4.cnf
cube-9-h10-unsat.cnf
cube-9-h11-sat.cnf
dated-10-15-u.cnf
dated-5-11-u.cnf
manol-pipe-c10nidw_s.cnf
manol-pipe-c6nidw_i.cnf
mizh-md5-47-3.cnf
mizh-sha0-35-2.cnf
mizh-sha0-35-4.cnf
mizh-sha0-35-5.cnf
total-10-11-u.cnf
total-5-11-u.cnf
total-5-13-u.cnf
uts-105-ipc5-h26-unsat.cnf
uts-105-ipc5-h27-unknown.cnf
uts-106-ipc5-h28-unknown.cnf
uts-106-ipc5-h29-unknown.cnf
uts-106-ipc5-h30-unknown.cnf

1.3 22 hard industrial instances

AProve07-02.cnf
AProve07-03.cnf
AProve07-06.cnf
AProve07-08.cnf
AProve07-09.cnf
AProve07-16.cnf
AProve07-21.cnf
AProve07-27.cnf
blocks-4-ipc5-h22-unknown.cnf
dated-10-11-u.cnf
dated-10-13-u.cnf
dated-5-15-u.cnf
dated-5-17-u.cnf
manol-pipe-c10nidw_s.cnf
manol-pipe-c7bidw_i.cnf
manol-pipe-c7nidw.cnf
manol-pipe-f9b.cnf
1.4 200 INDUSTRIAL INSTANCES FROM SAT RACE 2008

manol-pipe-g10bidw.cnf
total-10-13-u.cnf
uts-106-ipc5-h30-unknown.cnf
uts-106-ipc5-h31-unknown.cnf
uts-106-ipc5-h32-unknown.cnf

1.4 200 industrial instances from SAT Race 2008

aloul-chnl11-13
aloul-sr06-chnl10-13 *
anbul-dated-5-15-u *
anbul-part-10-13-s
anbul-part-10-15-s
babic-dspam-vc1080
babic-dspam-vc949
babic-dspam-vc973
cmu-bmc-barrel6
cmu-bmc-longmult13 *
cmu-bmc-longmult15 *
een-pico-prop00-75
een-pico-prop05-75 *
een-pico-sr06-pr02-02 *
een-tipb-sr06-par1
een-tipb-sr06-tc6b
een-tip-sat-nusmv-t5.B
een-tip-sat-texas-tp-5e
een-tip-sat-vis-eisen
fuhs-aprove-15 *
fuhs-aprove-16 *
goldb-heqc-alu4mul *
goldb-heqc-dalumul *
goldb-heqc-desmul *
goldb-heqc-frg1mul
goldb-heqc-i8mul *
goldb-heqc-k2mul
goldb-heqc-rotmul *
goldb-heqc-x1mul
grieu-vmpc-27 *
grieu-vmpc-31
grieu-vmpc-s05-05s *
grieu-vmpc-s05-24s *
grieu-vmpc-s05-27r *
hoons-vbmc-lucky7 *
hoons-vbmc-s04-05 *
hoons-vbmc-s04-06 *
hoons-vbmc-s04-07 *
hoons-vbmc-s04-07 *
ibm-2002-04r-k80 *
ibm-2002-11r1-k45 *
ibm-2002-18r-k90 *
ibm-2002-20r-k75 *
ibm-2002-22r-k60
ibm-2002-22r-k75 *
ibm-2002-22r-k80 *
ibm-2002-23r-k90
ibm-2002-24r3-k100 *
ibm-2002-25r-k10 *
ibm-2002-29r-k75 *
ibm-2002-30r-k85
ibm-2002-31_1r3-k30 *
ibm-2004-01-k90 *
ibm-2004-1_11-k80 *
ibm-2004-23-k100 *
ibm-2004-23-k80 *
ibm-2004-29-k25 *
ibm-2004-29-k55 *
ibm-2004-3_02_3-k95 *
jarvi-eq-atree-9 *
manol-pipe-c10b *
manol-pipe-c10bi_s *
manol-pipe-c10id_s *
manol-pipe-c10ni_s *
manol-pipe-c10nid_i
manol-pipe-c10nid_s *
manol-pipe-c10nidw
manol-pipe-c6b_i *
manol-pipe-c6bid_i *
manol-pipe-c6bidw_i *
manol-pipe-c6id *
manol-pipe-c6idw *
manol-pipe-c6n *
manol-pipe-c6ni_s *
manol-pipe-c6nid_s *
manol-pipe-c6nidw *
manol-pipe-c7_i *
manol-pipe-c7_i *
manol-pipe-c7id *
manol-pipe-c7idw *
manol-pipe-c7idw_s *
manol-pipe-c8_i *
manol-pipe-c8b_i *
manol-pipe-c8b_i *
manol-pipe-c8n *
manol-pipe-c8nid *
manol-pipe-c8nidw *
manol-pipe-c9n_i *
manol-pipe-cha05-113 *
manol-pipe-cha05-143 *
manol-pipe-f6b *
manol-pipe-f6i *
manol-pipe-f6n *
manol-pipe-f6nid *
manol-pipe-f7idw *
manol-pipe-f7nidw *
manol-pipe-f9b *
manol-pipe-g10bid_i
manol-pipe-g10idw *
manol-pipe-g10ni *
manol-pipe-g10nid *
manol-pipe-g6bid *
manol-pipe-g7n *
manol-pipe-g8b *
manol-pipe-g8bidw *
manol-pipe-g8n *
manol-pipe-g8nidw *
marijn-philips *
maris-s03-gripper11
miza-sr06-md5-47-03 *
miza-sr06-md5-48-01 *
miza-sr06-sha0-35-03 *
mizh-md5-47-3 *
mizh-md5-47-4 *
mizh-md5-47-5 *
mizh-md5-48-2 *
mizh-md5-48-5 *
mizh-sha0-35-3 *
mizh-sha0-35-4 *
mizh-sha0-36-1 *
mizh-sha0-36-3 *
mizh-sha0-36-4 *
narain-vpn-clauses-10
narain-vpn-clauses-8
narai-vpn-10s *
narai-vpn-sat05-02s *
1.4. 200 INDUSTRIAL INSTANCES FROM SAT RACE 2008

narai-vpn-sat05-07 *
palac-sn7-ipc5-h16 *
palac-uts-106-ipc5-h34 *
post-c32s-col400-16 *
post-c32s-gcdm16-22 *
post-c32s-gcdm16-23 *
post-c32s-ss-8
post-cbmc-aes-d-r1
post-cbmc-aes-d-r2
post-cbmc-aes-ee-r2
post-cbmc-aes-ee-r3
post-cbmc-aes-ele *
post-cbmc-zfcp-2.8-u2
schup-12s-abp4-1-k31 *
schup-12s-bc56s-1-k391
schup-12s-motst-2-k315 *
schup-12s-s04-abp4 *
schup-12s-s04s2-09 *
schup-12s-s04-valves
simon-mixed-s02bis-01 *
simon-mixed-s02bis-03 *
simon-mixed-s02bis-05 *
simon-mixed-s02bis-05 *
simon-s02b-r4b1k1.1 *
simon-s02b-r4b1k1.2 *
simon-s02-f2clk-50 *
simon-s03-fifo8-400 *
simon-s03-w08-15 *
stric-bmc-ibm-10 *
stric-bmc-ibm-12 *
vange-col-abb313GPIA-9-c
vange-color-inc-54 *
velev-dlx-uns-1.0-05
velev-engi-uns-1.0-4nd *
velev-eng-uns-1.0-04 *
velev-eng-uns-1.0-04a *
velev-fvp-sat-3.0-07 *
velev-fvp-sat-3.0-12 *
velev-fvp-sat-3.0-b18 *
velev-live-2.0-02 *
velev-live-sat-1.0-01 *
velev-live-sat-1.0-03 *
velev-live-sat-1.0-05 *
velev-npe-1.0-02 *
velev-npe-1.0-03 *
velev-npe-1.0-9dlx-b71 *
velev-pipe-1.0-08 *
velev-pipe-1.0-09 *
velev-pipe-1.1-03 *
velev-pipe-1.1-05 *
velev-pipe-oun-1.0-04
velev-pipe-oun-1.1-05
velev-pipe-sat-1.0-01 *
velev-pipe-sat-1.0-03 *
velev-pipe-sat-1.1-01 *
velev-pipe-sat-1.1-03 *
velev-pipe-sat-1.1-09 *
velev-pipe-uns-1.0-08
velev-pipe-uns-1.0-14 *
velev-pipe-uns-1.1-03 *
velev-pipe-uns-1.1-05
velev-pipe-uns-1.1-07
velev-pipe-uns-1.1-08
velev-sss-1.0-05 *
velev-sss-1.0-cl
velev-vliw-sat-2.0-02 *
1.4. 200 INDUSTRIAL INSTANCES FROM SAT RACE 2008

velev-vliw-sat-2.0-04 *
velev-vliw-sat-4.0-03 *
velev-vliw-sat-4.0-b4 *
velev-vliw-sat-4.0-b8 *
velev-vliw-uns-2.0-02
velev-vliw-uns-2.0-iq1
velev-vliw-uns-2.0-iq2
velev-vliw-uns-2.0-uq5
velev-vliw-uns-4.0-9
velev-vliw-uns-4.0-9-i1

* instances that were consistently solvable within the 900 second timeout and used to produce Figure 9.8.

Note: there were files with the exact same name but different content in the different sets of instances used for SAT Race 2008. The duplicate filenames are not an error.
Author/s: 
Chu, Geoffrey G.

Title: 
Improving combinatorial optimization

Date: 
2011

Citation: 

Persistent Link: 
http://hdl.handle.net/11343/36679

Terms and Conditions: 
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.