DECOMPOSITION AND SYMMETRY IN CONSTRAINT OPTIMIZATION PROBLEMS

by

Matthew Kitching

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2010 by Matthew Kitching
Abstract

Decomposition and Symmetry in Constraint Optimization Problems

Matthew Kitching
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2010

This thesis presents several techniques that advance search-based algorithms for solving Constraint Optimization Problems (COPs). These techniques exploit structural features common in such problems. In particular, the thesis presents a number of innovative algorithms, and associated data structures, designed to exploit decomposition and symmetry in COPs.

First, a new technique called component templating is introduced. Component templates are data structures for efficiently representing the disjoint sub-problems that are encountered during search. Information about each disjoint sub-problem can then be reused during search, increasing efficiency.

A new algorithm called OR-decomposition is introduced. This algorithm obtains many of the computational benefits of decomposition without the need to resort to separate recursions. That is, the algorithm explores a standard OR tree rather than an AND-OR tree. In this way, the search algorithm gains greater freedom in its variable-ordering compared to previous decomposition algorithms.

Although decomposition algorithms such as OR-decomposition are effective techniques for solving COPs with low tree-width, existing decomposition algorithms offer little advantage over branch and bound search on problems with high tree-width. A new method for exploiting decomposition on problems with high tree-width is presented. This technique involves detecting and exploiting decompositions on a selected subset of the
problem’s objectives. Such decompositions can then be used to more efficiently compute additional bounds that can be used by branch and bound search.

The second half of the thesis explores the use of symmetries in COPs. Using component templates, it is possible to exploit dynamic symmetries that appear during search when some of the variables of a problem have been assigned a value. Symmetries have not previously been combined with decomposition in COPs.

An algorithm called Set Branching is presented, which exploits almost-symmetries in the values of a variable by clustering similar values together, then branching on sets of values rather than on each single value.

The decomposition and symmetry algorithms presented in this thesis increase the efficiency of constraint optimization solvers. The thesis also presents experimental results that test these algorithms on a variety of real world problems, and demonstrate performance improvements over current state-of-the-art techniques.
3.8 Conclusion ................................................................. 58

4 OR-Decomposition .................................................... 59
  4.1 Introduction ............................................................. 59
  4.2 Solving WCSPs .......................................................... 60
  4.3 Decomposition without Separate Recursions ..................... 61
  4.4 Formal Results .......................................................... 69
  4.5 Local Propagation ..................................................... 73
  4.6 Multiple Partial Variable-Ordering ............................... 75
  4.7 Experimental Results ............................................... 76
  4.8 Related Work .......................................................... 88
  4.9 Future Work ............................................................. 88
  4.10 Conclusions ............................................................ 89

5 High Tree-Width Decomposition .................................... 91
  5.1 Introduction ............................................................. 91
  5.2 Background .............................................................. 92
  5.3 Decomposition Bounding ............................................ 93
  5.4 Formal Results .......................................................... 100
  5.5 Finding Decomposition Subsets .................................. 101
  5.6 Related Work ............................................................ 104
  5.7 Experiments ............................................................. 105
  5.8 Applicability ............................................................. 119
  5.9 Future Work ............................................................. 119
  5.10 Conclusions ............................................................ 120

6 Symmetric Templating ................................................. 121
  6.1 Introduction ............................................................. 121
  6.2 Template Symmetries ............................................... 124
Chapter 1

Introduction

Many practical combinatorial problems involve finding an optimal solution to the problem from a set of possible solutions. These problems include stand-alone optimization problems, as well as sub-problems that occur in the operation of larger AI systems. Stand-alone problems include allocation problems [18], scheduling [108], bioinformatics [99, 107, 46], combinatorial auctions [100], CP networks [98] and probabilistic reasoning [85]. With so many practical applications, a great deal of effort has been devoted to the development of efficient optimization solvers.

Constraint Optimization Problems are a formalism for representing a wide range of such optimization problems. Finding the optimal solution is, in general, computationally intractable. However, many problems can be solved in practice by using algorithms that exploit structure that commonly appears in practical problems.

Decomposition and symmetries are two such structural features that have great potential for improving the efficiency of solving Constraint Optimization Problems. However, to exploit this structure new algorithms and algorithmic techniques must be developed. This thesis aims to increase the efficiency of Constraint Optimization Solvers by providing a number of new algorithms that exploit decomposition and symmetries in problem instances.
This chapter gives an overview of this thesis. The chapter begins by exploring the relevance of Constraint Optimization Problems to the academic and industrial community. The chapter then provides an outline of the thesis.

1.0.1 Constraint Optimization Problems

A key research area of Artificial Intelligence (AI) is Automated Reasoning. Automated reasoning defines modeling languages that encode difficult real-world problems (e.g., planning, scheduling). Inference procedures solve the original problem expressed in the modeling language and return a solution that can be translated back into a solution for the original problem. Thus, an efficient and robust solver for the modeling language can solve a wide variety of difficult problems.

Propositional logic is one such modeling language. Solving problems expressed in propositional logic is known as the Satisfiability Problem (SAT). SAT was the first problem shown to be NP-complete [22] in 1971. Since then, a large body of research has been devoted to developing algorithms for solving SAT, resulting in extremely efficient algorithms [45, 49] that are able to solve a wide and useful range of practical problems.

Unfortunately, it is often difficult to express complex problems efficiently using propositional logic. Constraint Satisfaction Problems (CSPs) offers a more expressive modeling language for representing problems [52, 33]. In CSPs, problems are specified by a set of variables and a set of constraints that must be satisfied. CSPs provide a more robust modeling framework, although there are some problems on which SAT solvers are more efficient.

The formalism of Mixed Integer Programming (MIP) solves optimization problems in which some variables must take integer values, while others may take real values [111]. Although MIP solvers are efficient at solving many optimization problems, the formalism is restrictive, and as a result, it is sometimes difficult to model complex problems efficiently. Even when the problems can be modeled, MIP is difficult for the mathemati-
Recently the formalism of Constraint Optimization Problems (COPs) has been the focus of considerable research. In COPs, the goal is to minimize (or maximize) a set of cost functions while simultaneously satisfying a set of constraints. Since COPs specify problems using utility functions, the programmer can easily specify preferences or costs that should be maximized or minimized. Many types of problems can be encoded as COPs, including discrete optimization problems, and most probable explanation problems [85].

Once a problem is encoded as a COP, a black box COP solver can be used to find the solution which minimizes (or maximizes) the utility functions. This task is typically much more difficult than simply finding a single solution to a problem, due to the fact that all solutions that satisfy the set of constraints must be considered (implicitly or explicitly) when finding a solution that minimizes (or maximizes) the utility functions.

As a result of their increased complexity, advanced algorithms must be used in order to solve even moderately sized COPs. Inference-based algorithms such as Variable Elimination (VE) are useful for solving COPs with low tree-width (tree-width is a structural property of COPs that is described in Chapter 2), guaranteeing a solution in time and space exponential in the tree-width of the problem. Unfortunately, not all practical problems have low tree-width, and on such problems, the space requirements of VE become prohibitive. Recently, search-based algorithms exploiting decomposition have been developed that also successfully solve problems with lower tree-width. Although search based decomposition techniques are not as space-sensitive as VE, they still have a number of drawbacks. Current techniques impose restrictions on the variable-ordering used in search that can impede the bounding methods the algorithm must use. Furthermore these techniques are ineffective on problems with high tree-width.

Exploiting problem symmetry is another well-studied technique for reducing the size of the search space in CSPs, which potentially can also be applied to COPs. Unfortu-
nately, the cost of detecting symmetries can be large, and many problems contain few or no symmetries. Thus, although symmetries can dramatically improve the efficiency of constraint solvers, for many problems there is little advantage to exploiting them. In some cases, the overhead of computing symmetries can increase the time it takes the solver to find a solution.

This thesis presents new decomposition and symmetry algorithms to solve COPs. These algorithms improve on state-of-the-art decomposition-based search algorithms by allowing complete variable-ordering freedom, and the exploitation of decomposition on problems with high tree-width. These innovations allow decomposition to be effective on a much broader set of problems. In addition, this thesis presents an efficient algorithm for exploiting symmetries in COPs. Finally, the thesis presents an algorithm that exploits almost-symmetries; thus, allowing for improvements on problems possessing no true symmetries.

1.0.2 Thesis Structure

Chapter 2 is devoted to a review of current research related to decomposition and symmetry in COPs. The chapter defines key concepts and algorithms referenced throughout this thesis. The benefits and failings of current algorithms will be described, helping to frame the relevance of contributions made by this thesis.

Chapters 3-7 contain the innovations made by this thesis and are divided into two main parts. The first part, contained in Chapters 3-5, explores how decompositions can be exploited in COPs.

- Chapter 3 develops the technique of component templates, which are used to efficiently detect and cache the repeated sub-problems that arise from problem decomposition during search.

- Chapter 4 shows how complete variable-ordering freedom can be obtained by ex-
exploiting decomposition in a single depth-first search tree.

- Chapter 5 shows how decomposition can be exploited on COPs with high treewidth.

A method called component templating is introduced in Chapter 3 [59]. Component templates are data structures that exploit problem structure to efficiently cache the computed solutions (or partial solutions) of the disjoint sub-problems that are found while solving a COP. Component templates form the backbone for much of the work presented in this thesis. The algorithms in Chapter 4 and Chapter 5 use component templates to find decompositions quickly in partially solved COPs, while Chapter 6 uses component templates to exploit symmetries between component templates.

An algorithm called OR-Decomposition is presented in Chapter 4 [60]. This algorithm allows decomposition to be exploited in a standard Branch and Bound tree. OR-Decomposition provides increased variable-ordering flexibility when compared to current decomposition algorithms, potentially reducing the size of the search space explored, and reducing the time required to solve the problem. Experimental results are presented that demonstrate the benefits of using OR-Decomposition.

Decomposition techniques, including OR-Decomposition, are only effective on problems that decompose frequently during search. In Chapter 5, a new algorithm called decomposition bounding is introduced [61]. The technique involves modifying Branch and Bound to detect and exploit decompositions on a selected subset of the objectives of the problem. Decompositions over this subset are exploited to compute tighter bounds, which allows Branch and Bound to prune more of its search space.

The second part of the thesis explores the use of symmetries in COP. The section includes the following two chapters:

- Chapter 6 describes how component templates can be used to exploit dynamic symmetries more effectively (i.e., symmetries that only appear during search after
some of the variables have been instantiated).

- Chapter 7 describes how almost-symmetries can be exploited by branching on sets of values.

Chapter 6 extends the idea of component templates introduced in Chapter 3 [59]. Symmetries are calculated between component templates, thereby allowing efficient symmetry detection during search. Caching allows the solutions found for symmetric sub-problems to be re-used, thereby reducing size of the search space.

Although symmetries are a useful tool for efficiently solving COPs in which they are present, often problems contain no true symmetries. To overcome this problem, Chapter 7 presents an algorithm which attempts to find and exploit values which are similar, but not necessarily symmetric [62].

The conclusions of this thesis are presented in Chapter 8, which summarizes and assesses the work presented and identifies possibilities for future work.

1.0.3 Thesis Statement

This thesis presents a number of algorithms that reduce the required time to solve Constraint Optimization Problems. The algorithms exploit decomposition and symmetry in order to reduce the size of the search space, while minimizing the computational cost of exploiting decompositions and symmetries.
Chapter 2

Solving COP with Search

2.1 Introduction

In this chapter, current literature related to decomposition and symmetry in Constraint Optimization Problems is reviewed, providing the background required to understand the contributions made by this thesis. This chapter also outlines how the new techniques presented in this thesis relate to existing algorithms designed to exploit decomposition and symmetries. An outline of Chapter 2 is as follows.

The chapter formally defines Constraint Optimization Problems (COPs), as well as other key concepts and notation that will be used throughout the thesis. Based on these fundamentals, two standard algorithms for solving Constraint Optimization Problems are introduced. First, a dynamic-programming based algorithm called Variable Elimination (VE) is discussed [112, 32], followed by a time-exponential search-based algorithm called Branch and Bound [64].

The next section of the chapter introduces decomposition-based COP solvers. The AND/OR search algorithm, complexity results associated with AND/OR search, and common extensions to this algorithm are reviewed [73].

The chapter then discusses the current use of symmetries in constraint programming.
Conditional variable symmetries and value symmetries are defined. Current symmetry based techniques are also described, including SBDD and SBDS [10, 43]. The chapter also presents a description of commonly used variable-ordering techniques for search based algorithms.

Finally, the manner in which the contributions of this thesis advance the techniques described in the current literature are outlined; thus, framing the relevance of this thesis.

2.2 Variables, Constraints, and Objectives

A Constraint Optimization Problem is defined by a set of variables, a set of domains, and a set of functions over the variables. This section defines these concepts.

Definition 2.2.1 A **Variable** is an object that must be assigned a value in any complete solution to a Constraint Optimization Problem. Each variable has an associated discrete **Domain**, which represents the set of possible values that the variable can take. Upper case letters denote variables (i.e., \(V_1, V_2, V_3\)), and lower case letters denote values (i.e., \(x, y, z\)). We use the notation \(V_1 \leftarrow y\) or \(V_1 = y\) to denote variable \(V_1\) being assigned the value \(y\). We use the notation \(\text{Dom}[V]\) to denote the domain of variable \(V\).

Definition 2.2.2 A **Constraint** is a poly-time checkable function over a set of variables, which maps assignments of the variables to true/false. The set of variables are called the **scope** of the constraint. The scope of a constraint \(c\) is denoted by \(\text{scope}(c)\). A **tuple** over a set of variables, \(A\), is a set of assignments for all variables in \(A\). A constraint can be viewed as a set of allowed tuples over the scope of the constraint, where each tuple represents a set of assignments of the variables in the scope of the constraint that is mapped to true by the constraint.

Definition 2.2.3 An **Objective Function** is identical to a constraint except that it maps assignments to the variables in its scope to a real valued cost (rather than true/false).
We denote the cost of objective $o$, under the assignment, $A$, by $o(A)$. For convenience, we allow $A$ to be a superset of the variables in scope($o$), i.e., $o(A)$ is equivalent to $o(A \cap \text{scope}(o))$.

For example, consider the objective described in Figure 2.1. Here, the assignment $A \leftarrow 0, B \leftarrow 1, C \leftarrow 0$ achieves the lowest cost for the objective.

Variables, domains, constraints, and objectives form the basis for Constraint Optimization Problems, a powerful method for representing complex combinatorial optimization problems. We use the symbol $\mathcal{P}$ to represent individual constraint optimization problems.

### 2.3 Constraint Optimization Problems

In this section, several types of Constraint Optimization Problems are reviewed, including Constraint Satisfaction Problems (CSP), Weighted Constraint Satisfaction Problems (WCSP), Maximum Constraint Satisfaction Problems (MAX-CSP), and Most Probable Explanation Problems (MPE).
Definition 2.3.1 A Constraint Optimization Problem (COP), $\mathcal{P}$, is specified by a tuple, $\langle \text{Vars}, \text{Dom}, \text{Obj}, \otimes, \text{min/max} \rangle$, where Vars is a set of variables; for each $V \in \text{Vars}$, $\text{Dom}[V]$ is the domain of $V$; Obj is a set of objective functions, $\otimes$ is a combinatorial operator $\otimes \in \{\Pi, \Sigma\}$, and min/max specifies whether the problem is a minimization or maximization problem.

The solution to a COP is a set of assignments to all the variables that minimize (or maximize) the cost of $\otimes$ where $\otimes$ is a combinatorial operator applied to all of the objective functions in Obj.

Definition 2.3.2 A Weighted Constraint Satisfaction Problem (WCSP), is a COP where the combination operator is $\Sigma$. A solution to a WCSP is the assignment of values to variables that minimizes (or maximizes) the sum of costs of the objectives in Obj. Unless otherwise specified, the techniques described in this thesis are applicable to WCSPs where the following three properties hold:

1. Each objective, $o_i \in \text{Obj}$, is dependent on a set of variables where $\text{scope}(o_i) \subseteq \text{Vars}$.

2. Each objective, $o_i \in \text{Obj}$, maps assignments to the variables in $\text{scope}(o_i)$ to a real value.

3. For any complete assignment $A$ to Vars, $\text{Obj}(A) = \sum_i o_i(A)$.

In this framework, hard constraints can be expressed as objectives that map satisfying assignments to 0 and violating assignments to $\infty$.

This thesis will be concerned primarily with WCSPs, although there is a strong relationship between the different types of COPs. Thus, algorithms developed for WCSPs are often applicable to other COPs. The following notations relating to WCSPs are used throughout this thesis:
Notation 2.3.3 Given a WCSP \((\text{Vars}, \text{Dom}, \text{Obj})\), let \(\mathcal{A}\) denote any set of assignments to some of the variables of \(\text{Vars}\); let \(\text{varsOf}(\mathcal{A})\) denote the set of variables assigned by \(\mathcal{A}\); let \(\text{cost}(\mathcal{A}, \mathcal{P})\) denote the sum of the costs of all objectives in the WCSP, \(\mathcal{P}\), that are fully instantiated by \(\mathcal{A}\); and let \(\text{mincost}(\mathcal{P})\) denote the minimum cost of any solution \(\mathcal{A}\) to \(\mathcal{P}\) (i.e., the optimal objective value achievable in \(\mathcal{P}\)).

A set of assignments, \(\mathcal{A}\), reduces the original WCSP, \(\mathcal{P}\), to a smaller WCSP, \(\mathcal{P}|_{\mathcal{A}}\). The variables of \(\mathcal{P}|_{\mathcal{A}}\) are the variables of \(\mathcal{P}\) not assigned in \(\mathcal{A}\) (\(\mathcal{P}.\text{Vars}\backslash\text{varsOf}(\mathcal{A})\)). The domains of these variables are the same as in \(\mathcal{P}\). The objectives of \(\mathcal{P}|_{\mathcal{A}}\) are those that contain at least one unassigned variable and are obtained by restricting the original objectives of \(\mathcal{P}\) by \(\mathcal{A}\). That is, for any objective \(o_i \in \text{Objs}\), if \(\text{scope}(o_i) \not\subseteq \text{varsOf}(\mathcal{A})\) then the reduction of \(o_i\) by \(\mathcal{A}\), \(o_i|_{\mathcal{A}}\), is a new objective function with \(\text{scope}(o_i|_{\mathcal{A}}) = \text{scope}(o_i) \backslash \text{varsOf}(\mathcal{A})\). On any set of assignments \(\alpha\) to the variables in \(\text{scope}(o_i|_{\mathcal{A}})\) it holds that \(o_i|_{\mathcal{A}}(\alpha) = o_i(\mathcal{A} \cup \alpha)\).

Definition 2.3.4 A Constraint Satisfaction Problem (CSP) is similar to a COP except that a CSP contains a set of constraints, \(\text{Cons}\), rather than a set of objectives, \(\text{Obj}\). A solution to a CSP is a complete assignment, \(\mathcal{A}\), such that for all \(c \in \text{Cons}\), \(c(\mathcal{A}|\text{scope}(c)) = \text{true}\), i.e., the assignment satisfies all constraints.

Every CSP can be represented as a WCSP by replacing each constraint \(c \in \text{Cons}\) by an objective, \(o\), where \(\text{scope}(c) = \text{scope}(o)\) and \(c(\mathcal{A}) = \text{true}\) if and only if \(o(\mathcal{A}) = 0\) and \(c(\mathcal{A}) = \text{false}\) if and only if \(o(\mathcal{A}) = \infty\). An assignment \(\mathcal{A}\) is a solution to the CSP if and only if \(\mathcal{A}\) is a solution to the WCSP solution and \(\text{Obj}(\mathcal{A}) = 0\).

Definition 2.3.5 A Maximum Constraint Satisfaction Problems (MAX-CSP), is a CSP where the solution is a complete set of assignments, \(\mathcal{A}\), such that the number of constraints, \(c \in \text{Cons}\), where \(c(\mathcal{A}|\text{scope}(c)) = \text{false}\) (i.e., the number of constraints violated by \(\mathcal{A}\)) is less than or equal to the number of constraints violated by any other complete set of assignments.
Every MAX-CSP can be represented as a WCSP by replacing every constraint, \( c \in \text{Cons} \), by an objective, \( o \), where \( \text{scope}(c) = \text{scope}(o) \) and \( c(A) = \text{true} \) if and only if \( o(A) = 0 \) and \( c(A) = \text{false} \) if and only if \( o(A) = 1 \). \( A \) is a solution to the MAX-CSP if and only if \( A \) is a solution to the WCSP.

**Definition 2.3.6** A Bayes Net, \( \mathcal{P} \), is specified by a tuple, \( \langle \text{Vars}, \text{Dom}, G, \text{CPT} \rangle \), where \( \text{Vars} \) is a set of variables. For each \( V \in \text{Vars} \), \( \text{Dom}[V] \) is the domain of \( V \). \( G \) is an acyclic directed graph, the nodes of which are equal to the set of variables. \( \text{CPT} \) is a set of conditional probability functions, one for every variable in \( \text{Vars} \). For every variable, \( V_i \in \text{Vars} \), with \( P_i \) being the set of parents of \( V_i \) in \( G \), there exists a single function, \( \text{Pr}_i \in \text{CPT} \), with \( \text{scope}(\text{Pr}_i) = \{V_i\} \cup P_i \). For any assignment, \( A \), to the variables in \( P_i \), \( \text{Pr}_i(V_i = d, A) = \text{Pr}_i(V_i = d | A) \); i.e., the probability that \( V_i \) takes on the value, \( d \), given that the parents of \( V_i \) have taken on the values specified in \( A \).

A Bayes Net specifies a joint distribution over the variables in \( \text{Vars} \) as follows: for any complete assignment \( A \) to the variables in \( \text{Vars} \), \( \text{Pr}(A) = \Pi_i \text{Pr}_i(A | \text{scope}(\text{Pr}_i)) \).

**Definition 2.3.7** Given a Bayes Net \( \langle \text{Vars}, \text{Dom}, G, \text{CPT} \rangle \), the Most Probable Explanation (MPE) problem involves finding a complete assignment, \( A \), such that \( \text{Pr}(A) \geq \text{Pr}(A') \) for all other complete assignments \( A' \).

MPE is a COP with \( \Pi \) as the combination operator. All of the algorithms presented in this thesis can be adapted to use \( \Pi \) as the combination operator rather than \( \Sigma \). However, for simplicity, an MPE can be converted to a WCSP by converting every \( \text{Pr}_i \in \text{CPT} \) to \( \log_{\text{Pr}_i} \), where \( \text{scope}(\log_{\text{Pr}_i}) = \text{scope}(\text{Pr}_i) \); and where, for any assignment, \( A \), to the variables in \( \text{scope}(\text{Pr}_i) \), \( \log_{\text{Pr}_i}(A) = \log(\text{Pr}_i(A)) \).

This thesis concentrates on solving the WCSP problem. However, the definitions above demonstrate that a range of problems can be formulated as COPs. In fact, the experimental sections in this thesis include many MAX-CSP and MPE problems.
2.4 Tree-Decompositions

The following section describes some graph theoretic notation that will frequently be used in this thesis.¹

Given a hypergraph $H = (V, E)$, let $\pi = V_1, ..., V_n$ be an ordering of the variables in $V$. This ordering induces a series of hypergraphs $H_1, ..., H_n$ where where $H_n = H$ and $H_{i-1}$ is induced by eliminating $V_i$ from $H_i$ and then merging all of the edges that contain $V_i$.

**Definition 2.4.1** The *induced-width* of $H$ under $\pi$ is the size of the largest hyperedge in all the hypergraphs $H_1, ..., H_n$, where the size of the hyperedge is the number of variables in the hyperedge.

**Definition 2.4.2** The *elimination-width* of $H$ is the minimum induced-width over all orderings $\pi$. Typically, we will use $w$ to denote the induced-width of a particular ordering and $w^*$ to denote the elimination-width of $H$.

**Definition 2.4.3** Let $H = (V,E)$ be a hypergraph. A *tree-decomposition* of $H$ is a binary tree, $T$, such that each node of $T$ is labeled with a subset of $V$ in the following way:

1. The labels of the leaf nodes of $T$ are in a 1-1 correspondence with the hyper-edges $E \in H$.

2. Given labels for the leaf nodes, every internal node $n$ contains $v \in V$ in its label, if and only if $n$ is on a path between two leaf nodes, $l_1$ and $l_2$, whose labels contain $v$.

The tree-width of a tree-decomposition, $T$ for $H$, is the maximum size of any labellings in $T$ minus one. The tree-width of $H$ is the minimum tree-width over all tree-decompositions of $H$.

¹This thesis uses the graph theoretic notation defined by Bacchus, Dalmau, and Pitassi [9], although alternate definitions exist.
Example 2.4.4 Figure 2.2 represents a valid tree-decomposition on the graph in Figure 2.1.

The following list gives the elimination of the variables of Figure 2.1 subject to the ordering $\pi = A, B, C, D, E, F, G, H$.


$H_7 = (B,C),(B,E),(C,F),(C,D),(D,F),(D,G),(E,F),(F,G),(G,H)$

$H_6 = (C,E),(C,F),(C,D),(D,F),(D,G),(E,F),(F,G),(G,H)$

$H_5 = (E,F,D),(D,F),(D,G),(E,F),(F,G),(G,H)$

$H_4 = (E,F),(D,F),(E,F),(F,G),(G,H)$

$H_3 = (F,G),(G,H)$

$H_2 = (G),(G,H)$

$H_1 = (H)$

In the example above, the induced-width of $\pi$ is 3, since $(E,F,D) \in H_5$ is the largest
hyperedge produced. It can be shown that if a graph has tree-width \( w^* \), then there is an ordering, \( \pi \), of the vertices such that the induced-width of \( H \) under \( \pi \) is at most \( w^* \). Similarly, if a hypergraph, \( H \), has elimination-width \( w^* \), then \( H \) has a tree-decomposition with tree-width of at most \( w^* \). Thus, for any hypergraph \( H \), its elimination-width and tree-width are equal.

Although an ordering with small induced-width provides many theoretical and practical advantages, Arnborg, Corneil, and Proskurowski have shown that finding the tree-width of a graph is an NP-hard problem [5, 6]. These authors also demonstrated that determining whether a problem has a tree-width below a constant \( k \) is fixed parameter tractable in \( k \).

Several methods for approximating tree-width and induced-width exist based on the structure of the problem. Popular heuristics include: randomization [12], maximum cardinality search [105], minimum degree [97], minimum fill-in (minfill) [63], and QuickBB [47]. This thesis will use the minfill heuristic to compute orderings with better (i.e., smaller) induced-width, although any of the methods mentioned above can be used.

To apply the notions of tree-width and elimination-width to COPs, a natural graph called the Constraint Graph is associated with every COP.

**Definition 2.4.5** A **Constraint Graph** is a graph, \( G = (V, E) \), representing a COP. The vertices of \( G \) are in one to one correspondence with the variables of the COP, while edges connect nodes if and only if both of the end-point variables of the edge are in the scope of the same constraint or objective function.

This representation is often referred to as the primal representation [33]. Although there are many other ways to represent COPs, including using the hypergraph defined by the scopes of the objectives and the dual labeling [33], the primal representation is sufficient for this thesis.

**Example 2.4.6** The following example illustrates an application of the basic WCSP
framework. Given the map in Figure 2.3, comprised of four countries, A, B, C, and D, each country must be coloured with one of two colours, red or green. The map must be coloured so that the number of adjacent countries in the same colour is minimized. The constraint graph of the problem is shown in Figure 2.4.

The problem can be modeled as a WCSP using four variables, one for each country A, B, C, and D. The domain of each variable is \{r, g\}, which represents the two colours, red and green. In addition, five objectives must be included, \(o(A, B), o(A, C), o(B, C), o(B, D), \) and \(o(C, D)\), where each objective, \(o(X, Y)\), maps the tuple \(⟨X = r, Y = r⟩\) and \(⟨X = g, Y = g⟩\) to cost one, and the tuples \(⟨X = r, Y = g⟩\) and \(⟨X = g, Y = r⟩\) to cost zero. Thus, \(\mathcal{P}\) is specified by the following tuple:

\[
⟨\text{Vars} = \{A, B, C, D\}, \text{Dom} = \{r, g\}, \text{Obj} = \{o(A, B), o(A, C), o(B, C), o(B, D), o(C, D)\}⟩.
\]

A visual representation of \(\mathcal{P}\) can be seen in Figure 2.5. The edges in the figure rep-
resent tuples with cost one, representing the penalty when two adjacent countries are coloured the same colour. Figure 2.6 shows the best solution for the WCSP, given by the assignments \( A \leftarrow g, B \leftarrow r, C \leftarrow r, D \leftarrow g \). The cost of the solution is one (objective \( o(B, C) \)).

## 2.5 Solving WCSPs

WCSPs provide a powerful tool for modeling relevant problems, including allocation problems [18], scheduling [108], bioinformatics [99, 107, 46], combinatorial auctions [100], CP networks [98] and probabilistic reasoning [85]. The following section, we describe some of the standard techniques for solving WCSPs.

Branch and Bound search will be the basis for all the work in this thesis; however, it is useful to look at competitive algorithms in order to situate the contributions made in this thesis. Therefore, this section also introduces several other algorithms for solving WCSPs.

### 2.5.1 Backtrack-Free Search

A relationship between tree-width and complete search was first proposed by Freduer in 1982 [41]. He proved that backtrack-free search can be obtained when the constraint graph has induced-width, \( w \), by applying strong \( w \)-consistency to the problem. This algorithm applies inference of sufficient strength so that no backtracks need to be performed.

In 1987, Dechter and Pearl [35] introduced directional arc consistency. Their work demonstrated that by applying directional arc consistency in the direction of the variable branching ordering, backtrack-free search can be achieved. This leads to a \( O(n(d)^w^*) \) time and space algorithm for achieving backtrack-free search where \( w^* \) is the tree-width of the constraint graph, \( n \) is the number of variables in the problem, and \( d \) is the maximum domain size of every variable in the problem. Given the space requirements, however, if
a problem has even moderately large tree-width, this algorithm is impractical.

### 2.5.2 Variable Elimination

The basic Variable Elimination (VE) algorithm, also known as adaptive-consistency, was introduced by Dechter and Pearl in 1989 [36].

The following two operations are used in VE:

1. The merger or sum of a set of objectives: Given two objectives, g and f, their sum is defined to be a new objective \( (g+f) \) such that \( \text{scope}(g+f) = \text{scope}(g) \cup \text{scope}(f) \), and for any assignment, \( \mathcal{A} \), to the variables in \( \text{scope}(g+f) \), we have 
\[
(g+f)(\mathcal{A}) = g(\mathcal{A}|\text{scope}(g)) + f(\mathcal{A}|\text{scope}(f)).
\]
The sum of a set of objectives can be computed iteratively by utilizing associativity: 
\[
g+f+h = (g+f) + h.
\]

2. Elimination of a variable: Given an objective, f, and a variable, \( X \) where \( X \in \text{scope}(f) \), the elimination of \( X \) from \( f \) is defined to be a new objective, \( f \setminus X \), such that \( \text{scope}(f \setminus X) = \text{scope}(f) - \{X\} \), and for any assignment, \( \mathcal{A} \), to the variables in \( \text{scope}(f \setminus X) \), it holds that 
\[
f(\mathcal{A}) = \min_d \in \text{Dom}(X) f(\mathcal{A} \cup \{X = d\}).
\]

Given a constraint program \( C = \langle V, \text{Dom}, O \rangle \) with related constraint network \( H = \langle V, E \rangle \), and ordering \( \pi = V_1, ..., V_n \), VE proceeds as follows:

1. create \( n \) buckets, each labeled with variable \( V_i \)

2. for \( i = n \) to \( 1 \),
   - put all unplaced objectives mentioning \( V_i \) into \( \text{bucket}_i \). Thus every objective is associated with the bucket of its highest indexed variable.

3. for \( i = 1 \) to \( n \),
   - sum together all objectives in bucket \( i \), yielding a new objective 
\[
f_i = \sum_{g \in \text{bucket}_i} g
\]
• eliminate the variable \( V_i \) from \( f_i \) to obtain the new objective \( f_i \setminus V_i \). Add \( f_i \setminus V_i \) into the bucket of the highest indexed variables in its scope.

If at any point a relation with no allowable tuples is created, then there is no solution. If no such empty relation is generated, the WCSP has a solution and that solution can be extracted by working backwards through the elimination steps.

This algorithm is time and space exponential in \( w \), the width of the particular elimination ordering used by \( \text{VE} \). Since \( w \geq w^* \), the tree-width of the WCSP, \( \text{VE} \)'s space requirement becomes prohibitive when the algorithm is applied to many real-world problems. As a result, many variations have been proposed to make \( \text{VE} \) more practical. These variations perform search until either a tree remains [30, 85], or until the remaining constraint graph has a reasonable tree-width [92, 32]. At this point, \( \text{VE} \) is applied to the reduced problems. These techniques can reduce the space complexity of \( \text{VE} \) by performing search until the space requirements of \( \text{VE} \) becomes manageable.

### 2.5.3 Branch and Bound

Branch and Bound (\( \text{B&B} \)), described in Algorithm 1, is a common technique for solving WCSPs using backtracking search, and is the basis for many of the algorithms presented in this thesis.

The Global Upper Bound (\( \text{gub} \)) represents a known upper-bound on \( \text{mincost}(\mathcal{P}) \). Starting with a known upper-bound, \( \text{gub} \) is updated during search when complete assignments with lower costs are found. As a preprocessing step, a \( \text{gub} \geq \text{mincost}(\mathcal{P}) \) is found for the problem \( \mathcal{P} \). \( \text{B&B} \) uses a bounding function called \( \text{getBounds} \), which returns a value such that \( \text{getBounds}(\mathcal{P}) \leq \text{mincost}(\mathcal{P}) \). This bounding function must work on the reduction of \( \mathcal{P} \) by any set of assignments, \( \mathcal{A} \).

\( \text{B&B} \) proceeds by building up partial variable assignments in a depth-first manner, while using bounding to prune the search space. For any set of assignments, \( \mathcal{A} \),
cost(\(A, P\)) + \text{mincost}(P|_A) is the minimal cost that can be achieved by any extension of \(A\) on the WCSP \(P\).

Each recursion takes as input a set of assignments, \(A\), and the reduced problem \(P|_A\). Initially, the algorithm is invoked on the empty set of assignments and the original problem \(P\).

1 \text{BB (}A, C)\)

\[
/* \text{If } \text{mincost}(C) + \text{cost}(A, P) < \text{GUB, then GUB will be updated to be } \\
\text{mincost}(C) + \text{cost}(A, P) */
\]

2 \begin{align*}
3 & \quad \text{if } (\text{getBounds}(C) + \text{cost}(A, P)) \geq \text{GUB} \text{ then} \\
4 & \quad \quad \text{return} \\
5 & \quad \text{if } (|C.Vars| = 0) \text{ then} \\
6 & \quad \quad \text{GUB} = \text{cost}(A, P); \text{ return} \\
7 & \quad \text{choose } (\text{a variable } V \in C.Vars) \\
8 & \quad \text{foreach } d \in \text{Dom}[V] \text{ do} \\
9 & \quad \quad \text{BB (}A \cup \{V = d\}, C|_{V=d})
\end{align*}

10 \text{end}

\text{Algorithm 1: Branch and Bound (B&B)}

If the current set of assignments, \(A\), makes it impossible to find a better solution (Line 3) the algorithm returns. If there are no variables remaining in \(P.Vars\), then search is at a leaf node and \text{B&B} can update \text{GUB} and return. Otherwise, \(A\) can be extended by choosing any unassigned variable and testing each of its values using a recursion on the augmented set of assignments \(A \cup \{V = d\}\) and the reduced problem \(C|_{V=d}\). The recursion will either return with \text{GUB} unchanged or updated, depending on whether a solution with lower cost exists in the sub-tree. Every time \text{GUB} is updated, the algorithm can also record the complete assignment responsible for the new best value.
Chapter 2. Solving COP with Search

(Line 6). The last recorded complete assignment will be a solution to the problem.

The time complexity of B&B is $O(d^n)$, where $d$ is the maximum domain size of the WCSP and $n$ is the number of variables in the problem. The space complexity of B&B is $O(d \cdot n)$. Although this worst case time complexity is large, in practice B&B is often able to prune large sections of the search space if a strong bound is provided by the getBounds function.

2.6 Bounding Functions

B&B search uses a bounding function to calculate a valid lower-bound during search. In this section, a review of several bounding functions will be given.

In most cases in this thesis, a generic function call getBounds will be used, which can refer to any of the bounding functions described below. In certain cases, algorithms also contain specific references to the Local Consistency technique, although in most of those cases, the ideas can also be applied to other bounding techniques such as virtual arc consistency, mini-buckets, or linear relaxations [24, 31, 93, 94, 55].

2.6.1 Local Consistency

Enforcing Local Consistency [24, 67, 28], is a powerful bounding function that is often used in WCSP solvers. In this section, an outline of enforcing local consistency is presented.

Let $o_{ij}$ denote a binary objective between the variables $V_i$ and $V_j$, where $o_{ij}(a, b)$ is the cost of assigning $a$ to $V_i$ and $b$ to $V_j$. Let $o_i$ denote a unary objective for variable $V_i$, where $o_i(a)$ is the cost of assigning $a$ to $V_i$. If the WCSP does not initially specify a unary objective, $o_i$, for variable $V_i$, then an initially unary objective, $o_i$, that maps all values of $V_i$ to zero is added to the WCSP. Finally, a zero arity lower-bound objective, $O_\emptyset$, is also added to the WCSP with initial value zero. When local consistency is used
as the lower bounding function, getBounds returns the current value of $O_\emptyset$.

Enforcing Local Consistency in WCSPs involves modifying the WCSP into an equivalent problem by flowing value towards $O_\emptyset$. Two problems are equivalent if and only if the cost of every complete assignment is preserved. The value of $O_\emptyset$ is a valid lower-bound for the WCSP given the current assignment. In addition to generating a lower-bound, local consistency can also remove unfeasible values from the variable domains, thus improving the efficiency of B&B’s search.

The method uses the following two atomic transformations, both of which are guaranteed to preserve equivalence:

1. The first transformation is a **unary transformation**: Let $\alpha$ be the minimum cost $o_i(a)$ over $a \in \text{Dom}[V_i]$. If $\alpha > 0$, then $\alpha$ can be added to $O_\emptyset$ and subtracted from every $o_i(a), a \in \text{Dom}[V_i]$. Note that any complete assignment must assign some value to $V_i$, therefore the complete assignment must incur a cost of at least $\alpha$. Hence for any complete assignment, $A$, the cost prior to applying a unary transformation would involve a contribution, $s$, from $o_i$ with $s \geq \alpha$. After the transformation, the cost of $A$ will involve an additional $\alpha$ from $O_\emptyset$ as well as $s - \alpha$ from $o_i$. Hence, any unary transformation creates an equivalent problem.

   A value $b \in V_i$ is a unary support for a $V_i$ if $V_i(b) = 0$.

2. The second transformation is a **binary transformation**. Consider two objectives $o_{ij}$ and $o_i$, and a value $a \in \text{Dom}[V_i]$. Let $\alpha$ be the minimum cost of $o_{ij}(a, b)$ over all $b \in \text{Dom}[V_j]$. If $\alpha > 0$ then $\alpha$ can be added to $o_i(a)$ and subtracted from $o_{ij}(a, b)$ for all $b \in \text{Dom}[V_j]$. Given a binary transformation from $a$ into $V_j$, for any complete assignment, $A$, one of two situations exists. In the first case, $V_i \neq a \in A$. In this case there is no contribution to the cost of the complete assignment from $o_i(a)$ or $o_{ij}(a, b)$ for any $b \in \text{Dom}[V_j]$ before or after the binary transformation.

   In the second case, $V_i = a \in A$. If the assignment $V_j = b$ is in $A$ then before
the transformation there is a contribution of \( s_1 \) from \( o_i(a) \) and \( s_2 \) from \( o_{ij}(a, b) \), where \( s_2 \geq \alpha \). After the transformation, there is a contribution of \( s_1 + \alpha \) from \( o_i(a) \) and \( s_2 - \alpha \) from \( o_{ij}(a, b) \). Hence, any binary transformation creates an equivalent problem.

A value \( b \in V_j \) is a support for a value \( a \in V_i \) along \( o_{ij} \), if \( o_{ij}(a, b) = 0 \). A value \( b \in V_j \) is a full support for a value \( a \in V_i \) along \( o_{ij} \), if \( o_{ij}(a, b) = 0 \) and \( o_j(b) = 0 \).

By the repeated application of these two atomic transformations, costs are added to \( O_\emptyset \), which is always a valid lower-bound for the reduced WCSP. The \textit{getBounds} function returns the value of \( O_\emptyset \).

Several variations of local consistency exists. The following is a brief overview of the different types of local consistency:

1. A variable, \( V_i \), is said to be \textbf{node consistent} if: (1) for all values \( a \in \text{Dom}[V_i] \), \( o_i(a) \neq \infty \); and (2) \( V_i \) has a unary support. A WCSP is node consistent (NC) if every variable is node consistent.

2. Variable \( V_i \) is \textbf{arc consistent} (AC) if every value \( a \in \text{Dom}[V_i] \) has a support in every objective \( o_{ij} \in \text{Obj} \). A WCSP is arc consistent (AC) if every variable is arc and node consistent.

3. Given a variable-ordering, \(<\), variable \( V_i \) is \textbf{directional arc consistent} (DAC) if for every value \( a \in V_i \), \( V_i \) has a full support in every objective \( o_{ij} \in \text{Obj} \), where \( j > i \). A WCSP is DAC if every variable is DAC and node consistent. A WCSP is \textbf{full directional arc consistent} (FDAC) if it is AC and DAC.

4. Variable \( V_i \) is \textbf{existential arc consistent} (EAC) if there exists \( a \in \text{Dom}[V_i] \) such that \( o_i(a) = 0 \) and value \( a \) has a full support on every objective \( o_{ij} \in \text{Obj} \). A WCSP is existential arc consistent if every variable is EAC and node consistent. A WCSP is \textbf{existential directional arc consistent} (EDAC) if it is EAC, and FDAC.
2.6.2 Mini-Buckets Elimination

Mini-Bucket Elimination (MBE) is a variation of the VE algorithm described in Section 2.5.2 that returns a lower-bound for the problem. MBE avoids the space and time complexity problem of VE by partitioning large buckets into smaller subsets, called mini-buckets, which are processed independently [31, 37].

Given a constraint program $C = \langle V, Dom, O \rangle$, with related constraint graph $H = \langle V, E \rangle$, ordering $\pi = v_1, \ldots, v_n$, and a bounding parameter $i$, MBE proceed as follows:

1. create $n$ buckets, each labeled with variable $V_s$
2. for $s = n$ to $1$,
   - put all unplaced objectives mentioning $V_s$ into $bucket_s$.
3. for $s = 1$ to $n$,
   - sum together objectives in bucket $i$ until adding an additional objective to
     the newly created objective would result in an objective with more than $i$
     variables. At this point, the step is repeated for all unprocessed objectives.
     Thus, many multiple objectives can be created in each bucket.
   - eliminate the variable $V_i$ from each objectives $f_i \in bucket_i$. Add each new
     objective, $f_i \backslash V_i$, into the bucket of the highest indexed variables in its scope.

The quality of the lower-bound depends on the parameter $i$ (called the i-bound). MBE creates objectives with no more than i-bound variables. The complexity of the algorithm is time and space $O(d^i)$ where $i \leq n$. When $i \geq w$, MBE coincides with VE.

2.7 Decomposition-Based Search

Even with a state-of-the-art bounding function, solving a WCSP to optimality is often beyond the abilities of B&B search. Decomposition is one technique that can solve
problems that standard B&B is unable to solve. In particular, on problems with low tree-width, exploiting decomposition can have a dramatic theoretical and practical effect. Here, a standard decomposition-based search algorithm called AND/OR search is introduced.

2.7.1 Components

As the variables of a WCSP are assigned during backtracking search, the problem can break into disjoint parts. Consider \( P \), consisting of the variables \( \{A,B,M,X,Y\} \) and two objectives: \( o_1(A,B,M) \) and \( o_2(M,X,Y) \). If variable \( M \) is assigned value \( m \), then \( P|_{M=m} \) will consist of two disjoint sub-problems that share no variables. One sub-problem contains unassigned variables \( A \) and \( B \), with the objective \( o_1^{M=m}(A,B) = o_1(M=m,A,B) \). The other sub-problem contains unassigned variables \( X \) and \( Y \) with the objective \( o_2^{M=m}(X,Y) = o_2(M=m,X,Y) \). These disjoint sub-problems are called components.

The two components described above, created by \( M = m \), can be solved independently: any solution \( \langle A = a, B = b \rangle \) to the first component, and solution \( \langle X = x, Y = y \rangle \) to the second component, can be combined with \( M = m \) to obtain a solution to the original WCSP. Since the worst case complexity of solving a WCSP is exponential in the number of variables, decomposition into components can yield significant computational gains. Components are formally defined in the following definition:

Definition 2.7.1 In backtracking search, each node of the search tree, \( n \), corresponds to a set of variable assignments. A component is a subset of the original problem that has been isolated by a set of assignments.

A component, \( \kappa \), of a WCSP, \( P = \langle Vars, Dom, Obj \rangle \), is a tuple \( \langle \kappa.Vars, \kappa.Obj, \kappa.A \rangle \), where \( \kappa.Vars \subseteq Vars \), \( \kappa.Obj \subseteq Obj \) and \( \kappa.A \) is a set of assignments \( \{V_i = a_i, \ldots\} \), such that:
1. $\mathbf{\kappa}.\text{Obj} = \bigcup_{V \in \mathbf{\kappa}.\text{Vars}} \{o | o \in \text{Obj} \land V \in \text{scope}(o)\},$

2. $\text{vars}(\mathbf{\kappa}.\mathbf{A}) \cup \mathbf{\kappa}.\text{Vars} = \bigcup_{o \in \mathbf{\kappa}.\text{Obj}} \text{scope}(o),$

3. $\text{vars}(\mathbf{\kappa}.\mathbf{A}) \cap \mathbf{\kappa}.\text{Vars} = \emptyset,$

4. $\mathbf{\kappa}$ is minimal. That is, there is no other tuple $p' = (\mathbf{\kappa}'.\text{Vars}, \mathbf{\kappa}'.\text{Obj}, \mathbf{\kappa}'.\mathbf{A})$ that satisfies conditions 1-3 and for which $\mathbf{\kappa}'.\text{Vars} \subseteq \mathbf{\kappa}.\text{Vars}, \mathbf{\kappa}'.\text{Obj} \subseteq \mathbf{\kappa}.\text{Obj}, \mathbf{\kappa}'.\mathbf{A} \subseteq \mathbf{\kappa}.\mathbf{A}$ and with least one of these sets being a strict subset.

In other words, a component contains a set of variables and all of the objectives over these variables. Furthermore, these objectives are isolated from the rest of the problem by a set of assignments, $\mathbf{\kappa}.\mathbf{A}$. All of the variables in the scope of the objectives are either variables in the component or are instantiated in $\mathbf{\kappa}.\mathbf{A}$. Furthermore, none of the variables of the component (the variables in $\mathbf{\kappa}.\text{Vars}$), are assigned in $\mathbf{\kappa}.\mathbf{A}$.

Search with decomposition relies on detecting components. When dynamic variable-ordering is used, the cost of detecting components can be high.

### 2.7.2 Thesis Contributions: Component Templating

Chapter 3 describes a technique that can reduce the cost of dynamically detecting components. The technique is a data structure called a component template that is used to cache the constraint graph of components encountered during search [59]. When a component is encountered in search, a template is created which represents the constraint graph of the component. Whenever components that share the same constraint graph are encountered during search, the template can be used to quickly detect its existence. In addition, the template can be used to store information about the component, which is used to exploit symmetries in Chapter 6.
2.7.3 Search with decomposition

Decomposition has been successfully used in Constraint Satisfaction in solvers such as BTD [83, 57, 106, 56]. A technique used in solving WCSPs is AND/OR search with Branch and Bound (AND/OR), represented in Algorithm 2 [73, 77]. As variable assignments are made during backtracking search, the WCSP can be separated into components. Assigning the variables of one component has no effect on the bound of any other component. AND/OR works by invoking a separate recursion for each component generated during search.

In AND/OR, components are created and destroyed as the algorithm performs its search. The search starts with the call $\kappa = P$, with $\kappa.lb \leq \text{mincost}(\kappa) \leq \kappa.ub$ and an initial upper-bound, UB. Each recursion solves a single component, $\kappa$, (created by the current set of assignments) subject to a target cost of UB. To solve $\kappa$, an unassigned variable, $V \in \kappa$, is selected, and every domain value, $d \in \text{Dom}[V]$, is tested by reducing $\kappa$ by the assignment $V = d$. The reduced component, $\kappa|_{V=d}$, is first separated into a set of sub-components $\mathcal{K}^d$ (Line 8). Each sub-component, $\kappa^d \in \mathcal{K}^d$, is solved independently (Line 13). In order to achieve a total cost of less than UB for $\kappa$ under the assignment $V = d$, the sum of the lower-bounds over all components in $\mathcal{K}^d$ must be less than $\text{UB} - \Delta^d$ (where $\Delta^d$ is the immediate cost of making the assignment $V = d$, computed on Line 7).

Thus, each sub-component, $\kappa^d \in \mathcal{K}^d$, must achieve a value of no greater than $\text{UB} - \Delta^d$ minus the sum of the lower-bounds of all of the other sub-components in $\mathcal{K}^d$ (Line 12). Since each recursive call updates the lower-bound of a sub-component in $\mathcal{K}^d$, search can abort if the sum of the (now updated) lower-bounds of the sub-components exceeds $\text{UB} - \Delta^d$ (Line 11).

When the algorithm has completed testing the assignment $V = d$, bounds under this

\footnote{Note that the paradigm of AND/OR search, i.e., exploiting decomposition through separate recursions, can be used in various contexts [73, 76]. In this thesis, however, we will always be using AND/OR in conjunction with branch and bound.}
AND-OR \((\kappa, \text{UB})\)

/* On entry \((\kappa.lb, \kappa.ub)\) must be valid bounds on \(\text{mincost}(\kappa)\). If \(\text{mincost}(\kappa) < \text{UB}\), then compute exact bounds \(\kappa.lb = \text{mincost}(\kappa) = \kappa.ub\). Else compute valid bounds such that \(\text{UB} \leq \kappa.lb \leq \text{mincost}(\kappa) \leq \kappa.ub\). */

begin
  if \((\kappa.lb < \text{UB} \land \kappa.lb \neq \kappa.ub)\) then
    choose (a variable \(V \in \kappa.Vars\))
    foreach \(d \in \text{Dom}[V]\) do
      UB = min(UB, \kappa.ub)
      \(\Delta^d = \text{cost}(V = d, \kappa)\)
      \(\mathcal{K}^d = \text{toComponents}(\kappa|_{V=d})\)
      foreach \(\kappa^d \in \mathcal{K}^d\) do
        \((\kappa^d.lb, \kappa^d.ub) = \text{getBounds}(\kappa^d)\)
      endforeach \(\kappa^d \in \mathcal{K}^d\) while \((\sum_{\kappa^d \in \mathcal{K}^d} \kappa^d.lb < \text{UB} - \Delta^d)\) do
        UB_{\kappa,d} = UB - \Delta^d - \sum_{\kappa' \in \mathcal{K} \land \kappa' \neq \kappa^d} \kappa'.lb
        AND-OR \((\kappa^d, \text{UB}_{\kappa,d})\)
      endforeach \(\kappa^d \in \mathcal{K}^d\)
      \((lb^d, ub^d) = \sum_{\kappa^d \in \mathcal{K}} (\kappa^d.lb, \kappa^d.ub)\)
      \(\kappa.ub = \min(\kappa.ub, ub^d + \Delta^d)\)
    endforeach \(d \in \text{Dom}[V]\)
  end
end

Algorithm 2: AND-OR search with Branch And Bound
setting of \( V \) are computed in Line 14. The upper-bound for the component may be lower than the \( \text{ub} \). In that case, the algorithm will continue to compute the input component’s minimum cost, by trying the remaining assignments to \( V \). However, since \( V = d \) achieves at most cost \( \kappa \cdot \text{ub} \), any new assignment to \( V \) needs to achieve an even lower cost; therefore \( \text{ub} \) can be lowered (Line 6) before attempting the next assignment to \( V \).

After all values for \( V \) have been tried, the bounds for \( \kappa \) can be updated in Line 16. Once search has concluded testing the assignment \( V = d \), all of the data structures in \( K^d \) can be deleted. Thus, the space requirements of the algorithm remain polynomial.

During search with decomposition, the same component may be encountered many times. Thus, it is natural to cache the computed bounds for these components so that updated bounds can be used when the same component is encountered again. Cached bounds can be used instead of the function \( \text{getBounds} \) (Line 10).

2.7.4 Relating Variable Elimination to Decomposition Search

The algorithms \( \text{VE} \) and \( \text{AND/OR} \) using caching are closely related [8, 27]. Mateescu has analyzed this relationship in detail [77]. Mateescu showed the only differences between \( \text{VE} \) and \( \text{AND/OR} \) with caching under a fixed variable-ordering are cosmetic:

“(1) different direction of exploring a common search space (top down for search vs. bottom-up for inference); (2) different assumption of control strategy (depth-first for search and breadth-first for inference).” [77].

\( \text{VE} \), however, does not allow useful techniques such as no-good learning and backjumping [30, 58]. Moreover, \( \text{VE} \) must use a static variable-ordering.

Decomposition during search can be done efficiently when a static variable-ordering is used. In this case, preprocessing can reduce most of the overhead associated with decomposition [27]. When a dynamic variable-ordering is used, it is necessary to compute the decompositions that are created during search. Caching also becomes more expen-
sive to implement (e.g., see the methods presented in [101]). However, static variable-orderings can lead to larger search trees. It was proved in [7] that for some problem instances, dynamic variables orderings can yield a super-polynomial speedup over any static variable-ordering. There is also empirical evidence to demonstrate that despite the higher cost of decomposition, search with decomposition performs better with dynamic variable-orderings on many problem instances [75].

Although decomposition during search is an effective technique for solving problems that frequently decompose, there are a number of limitations associated with AND/OR. When AND/OR solves independent sub-problems, it is often forced to solve components to completion before assigning variables from any other component. It is possible that search could backtrack more quickly by assigning a small number of variables from each of the components.

2.7.5 Decomposition and Tree-Width

Exploiting decomposition (i.e., separate recursions) improves the worst-case complexity from $2^{O(n)}$ to $n^{O(w^*)}$, where $n$ is the number of variables of the problem and $w^*$ is the treewidth of the constraint graph [27]. If caching is also employed, the worst-case complexity is improved further, from $n^{O(w)}$ to $n \cdot 2^{O(w^*)}$ [11, 101, 4].

AND/OR improves the theoretical worst-case running time over B&B whenever $w < n$, where $w$ is the elimination width of an ordering, and $n$ is the number of variables in the problem. However, predicting whether AND/OR will improve the time to solve a problem requires a more nuanced analysis than simply testing whether $w < n$. The time to solve a problem using any search based algorithm is roughly equal to the product of the number of nodes expanded and the time to process each node. Due to the overheads associated with exploiting decomposition, the time to process each node in AND/OR is typically larger than in B&B. Therefore, AND/OR needs to expand significantly fewer nodes than B&B to improve on the time to solve a problem. Thus, in order to predict
the effectiveness of decomposition, it is important to estimate whether the number of nodes expanded using AND/OR will be significantly less than B&B.

When AND/OR or B&B assigns a value to a variable, the bounding algorithm is often able to prune entire sub-trees of the search space. In addition, when a variable is assigned a value, bounding algorithms are able to prune values from the domain of unassigned variables. Therefore, it is more likely that a node will be pruned by bounding or inference when more variables are assigned values. Since decomposition will not avoid any search nodes until the problem has decomposed, if \( w^* \) approaches \( n \), the nodes pruned by decomposition will be located near the bottom of the search space. Thus, when \( w^* \) approaches \( n \), the nodes that are avoided by decomposition are also likely to be avoided by bounding and propagation. Therefore, rather than simply examining whether \( w^* < n \), the ratio of \( w^* \) to \( n \) more accurately predicts the benefits of decomposition.

In this thesis, the terms ‘low tree-width’ and ‘high tree-width’ are used to denote the ratio of \( w^* \) to \( n \). Intuitively, a problem that has low tree-width has a large ratio between the number of variables in the problem and the tree-width of the problem. If a problem has a low tree-width, it is possible to find decompositions early in the search space, and the time to solve problems using AND/OR is typically less than when B&B is used. A problem with high tree-width has a small ratio between the number of variables in the problem and the tree-width of the problem. If a problem has a high tree-width, the problem is unlikely to benefit from decomposition. This thesis considers a problem to be low tree-width if \( w < n/5 \), and a problem to be high tree-width if \( w \geq n/5 \).

2.7.6 Thesis Contribution: OR-Decomposition

In Chapter 4, an algorithm called OR-Decomposition is presented that is able to exploit decomposition in a standard backtracking search tree (an OR tree). OR-Decomposition can assign variables with complete freedom. Further, OR-Decomposition is able to work on different components concurrently while retaining the space efficiency of depth-first
search. In this way, incremental information gathered from any component can be used to improve the bounding information for all of the other components.

Decomposition-based search algorithms, including OR-Decomposition, are effective on problems with low tree-width. Unfortunately, they have limited value on problems with high tree-width, since decompositions rarely emerge during search.

2.7.7 Thesis Contribution: Decomposition Bounding

In Chapter 5, a method called Decomposition Bounding is proposed which exploits decomposition on problems with high tree-width. The algorithm detects and exploits decompositions on a selected subset of the problem’s objectives. Decompositions over this subset, generated during search, are exploited to compute tighter bounds, in turn allowing search to prune more of its search space that would have otherwise been explored. This thesis then presents a greedy heuristic that selects an appropriate subset of objectives; i.e., one that readily decomposes during search and yet still can provide strong bounds.

2.8 Symmetry

The first half of this thesis provides algorithms that exploit decompositions in COPs. The second half exploits symmetries and almost-symmetries in COPs.

Exploiting symmetry has been another useful tool for solving constraint programs. Many different definitions of symmetries in CSP have been proposed. Cohen, Jeavons, Jefferson, Petrie, and Smith provide a comprehensive overview of the literature [20, 21]. The literature often describes symmetries in terms of variable symmetries and value symmetries.

A variable symmetry is a permutation of the set $V' \in Vars$ (i.e., a bijection from variables onto variables) that preserves the set of solutions. A value symmetry is a
permutation of the set $\text{Dom}[V]$ (i.e., a bijection from the values of a variable to values of the same variable) that preserves the set of solutions.

In most methods, symmetries are computed at the start of search; thus, only global symmetries are exploited. However, local symmetries may exist in a constraint problem. These symmetries exist only when the original constraint problem has been reduced by search and inference into smaller sub-problems. Thus, even if no global symmetries exist, symmetries may emerge after a partial assignment of variables. The detection and exploitation of local symmetries is often referred to as conditional symmetry detection.

2.8.1 Computing Conditional Variable Symmetry

Puget has proposed a method for automatic detection of variable and value symmetries in CSPs [90]. The method creates a coloured graphical representation of the constraint graph. This representation consists of the union of many individual constraints sub-graphs which represents admissible values of a constraint.

Since Puget’s method forms the basis for the algorithm presented in Chapter 6, his algorithm will be discussed in greater detail here. The graph is constructed as follows:

- A vertex is added for each variable.
- A vertex is added for every value in the domain of each variable.
- A vertex is added for each consistent assignment $\{X = x_1, Y = y_2, ..., Z = z_n\}$ for each constraint.
- A vertex is added for each constraint.

The variable vertices are of the same colour. Value vertices are all of a second colour. Consistent assignment vertices are all of a third. Finally, each constraint vertex is represented by a fourth colour. Each constraint vertex is linked to every consistent assignment vertex that satisfies the constraint. The vertex for consistent assignment
Figure 2.7: Symmetry Graph

\{X = x_1, Y = y_2, ... Z = z_n\} is linked to every value vertex \{x_1, y_2, ..., z_n\}, and each value vertex is linked to its corresponding variable vertex.

An automorphism-detection algorithm called NAUTY [80] detects automorphisms in the coloured graph. The automorphisms must map a vertex of a given colour to a vertex of the same colour.

For example, suppose we are given the following CSP, \(\mathcal{P} = (\{X, Y\}, \{1, 2, 3\}, C_{XY})\), where \(C_{XY}\) is the equality constraint between variables \(X\) and \(Y\). The following vertices are added:

- A vertex is added for variables \(X\) and \(Y\).

- A vertex is added for values \(X = 1, X = 2, X = 3, Y = 1, Y = 2, Y = 3\).

- A vertex is added for each consistent assignment \(A_1 = \{X = 1, Y = 1\}\), \(A_2 = \{X = 2, Y = 2\}\), and \(A_3 = \{X = 3, Y = 3\}\).

- A vertex is added for the equality constraint.

The completed graph is seen in Figure 2.7. This graph is tested for automorphisms, and one symmetry is detected, which maps \(X \rightarrow Y\) and each variable value combination \(X = x\) to \(Y = x\).
2.8.2 Symmetry-Breaking during Search (SBDS)

Gent and Smith propose a generic method, SBDS, to break global symmetries during search so that only unique solutions are considered (i.e. that only one exemplar of each symmetric equivalence class of solutions is considered). The method assumes symmetries are pre-calculated and are used as an array of symmetries [10, 44]. At each node, search backtracks if a symmetric problem has already been tested. The overhead of testing for symmetries can be large.

2.8.3 Symmetry by Dominance-Detection (SBDD)

A conditional symmetry detection algorithm called SBDD was proposed by [43]. The algorithm detects local symmetric no-goods by traversing the ancestors of a node. When a right node is traversed, then the left child of that node is a no-good. Symmetries are detected within these ancestral no-goods, and if a symmetric no-good is found, then the current node is pruned.

The algorithm is successful at detecting symmetric no-goods, but the overhead in detecting symmetries is large, often resulting in longer running time since more time is spent detecting the symmetries than is saved.

2.8.4 Symmetry-Breaking Constraints

Symmetries can also be exploited by adding symmetry-breaking constraints to a problem [89, 26, 3, 2, 40, 68, 91, 109]. The introduction of constraints transforms the original problem into one containing fewer symmetries. The introduced constraints guarantee that if a solution exists in the original problem, a solution also exists in the transformed problem.

The use of these techniques does have a downside: local symmetries are difficult to detect since their constraints are local to sub-trees. Furthermore, symmetry-breaking
constraints may prune the first solution that would have been found, resulting in longer search, although techniques for avoiding this problem have been developed [91, 109]. Finally, a super-polynomial number of constraints may often be required to break all symmetries. Consequently, many algorithms add only a subset of the symmetry-breaking constraints [96], thus implementing only partial symmetry-breaking.

### 2.8.5 Symmetry in Constraint Optimization

Symmetries have been extensively studied in the context of CSPs, however, there has been far less work done for COPs. Prestwich and Beck used dominance relationships to prune the search space, although the dominance relationships are hand-encoded for each problem [88]. Walsh has proposed a method that breaks symmetries in optimization problems by adding symmetry-breaking constraints to the problem [110], however these ideas have not yet been implemented.

Symmetry-breaking can be a useful tool on certain problems. However, there are a number of drawbacks to current symmetry techniques. The high cost of symmetry detection means that on many problems exploiting symmetries results in a longer running time due to the overhead associated with symmetry detection.

### 2.8.6 Thesis Contributions: Symmetry Templating

In Chapter 6, the templating idea proposed in Chapter 3 is used to perform automatic symmetry detection between templates created during search. A symmetry between two templates is valid for every instance of the template. Thus, a single symmetry detection computation can be used to detect symmetries between all the instances of the two templates. Amortizing the cost of symmetry detection across all instances of a template in this way is key in making automatic conditional symmetry detection cost effective.
2.8.7 Thesis Contribution: Set Branching

Although techniques for exploiting symmetries can yield significant performance improvements, such techniques are not useful when the problem has few global or conditional symmetries. In Chapter 7 a technique called Set Branching clusters the values of a variable’s domain into sets. The sets are constructed such that values in the same set are similar, although not necessarily symmetric. Search can then branch on these sets of values rather than branching on each individual value, thereby reducing the branching factor at search nodes. Constructing a collection of sets such that branching on these sets will still allow effective bounding is the aim of the clustering algorithm. Thus, computed bounds, in conjunction with the reduced branching factor, significantly reduces the size of the explored search space.

2.9 Variable-Ordering Heuristics

Search based algorithms can use a variety of variable-ordering heuristics. Variable-ordering has a large impact on search [33, 75]. Standard techniques for selecting variables are min-domain [50], min-dom/deg [14], and two-sided Jersolow [53]. The min-domain heuristic selects the variable with the smallest domain, min-dom/deg selects the variable with the lowest domain size divided by its degree, where a variable’s degree is equal to the number of constraints it participates in. The two-sided Jersolow heuristic considers both the variable’s domain size and the average cost of the objective functions in which the variable appears. Mateescu [77], describes these variable-orderings as semantic-based heuristics, as they all aim to reduce the size of the search tree based on the context and current assignments to the variables.
2.9.1 Graph-Based Heuristics

Mateescu [77] also defines graph-based heuristics that aim to maximize the number of decompositions created during search. These heuristics first calculate a tree-decomposition.

A tree-decomposition is defined by a set of labels, each containing a set of variables. A graph-based heuristic follows the tree-decomposition by maintaining an active label at each node in search. When search begins, the label at the root of the tree-decomposition is active. For example, in Figure 2.8, when the algorithm begins solving the problem, the top-most label containing the variables $D$ and $F$ is made active. Variables in the active label will be assigned until all variables in the label are assigned a value. A tree-decomposition has the following property: Once all of the variables in a label, $n$, are instantiated (and thus all variables in the labels of all parents of $n$ have already been instantiated) then the union of the variables in the labels of the nodes of each sub-tree below $n$ are isolated and form a component. Thus, after variables $D$ and $F$ are assigned
in Figure 2.8, the problem will decompose into two components, the first with variables $A$, $B$, $C$, and $E$, and the second with variables $G$ and $H$. A decomposition-based search algorithm will then select one of the two components to solve, and the label of the root of the corresponding subtree will be made active. In Figure 2.9 for example, if the search algorithm chooses to solve the component with variables $A$, $B$, $C$, and $E$, then the label containing variables $C$ and $E$ is made active, otherwise the label containing variable $G$ is activated.

By assigning variables based on the active label, search attempts to decompose the problem early in the search space, thus maximizing the effect that decomposition algorithms have on the search space. A secondary benefit of statically following a tree-decomposition is that components are immediately detected when all the variables in a label are assigned a value.

### 2.9.2 Dynamic Variable-Ordering

Dynamic Variable-Ordering (DVO) uses a semantic variable-ordering heuristic to select variables. Hence, the variable-ordering can vary from branch to branch of the search tree. Since a fixed static ordering is not being used during search, the technique of using a statically computed tree-decomposition to detect when components appear during search can no longer be used. Instead, decomposition-based search algorithms must dynamically test for decompositions during search. The search algorithm may select any uninstantiated variable to branch on next. However, this can make the problem less likely to decompose, since there is nothing guiding search towards decompositions. For example, consider the problem seen in Figure 2.8. The solver might begin by assigning the variable, $H$, some value. Although choosing $H$ might be advantageous because a semantic variable-ordering algorithm determines that $H$ will likely lead to better bounds early in the tree, the problem is no closer to decomposing.

With AND/OR search, there is never full freedom in variable-ordering, even when
a full dynamic variable-ordering is used. Restrictions on variable-ordering are imposed by the recursive nature of the algorithm. When a problem decomposes into components, \textbf{AND/OR} must select a single component to work on first. When solving this component, no variable from any other component may be branched on. For example, suppose \textbf{AND/OR} assigns the variables \(D\) and \(F\) in Figure 2.8. The problem then decomposes into two disjoint sub-problems; the first component contains the variables \(A, B, C,\) and \(E\); and, the second component contains the variables \(G\) and \(H\). If \textbf{AND/OR} first attempts to solve the first component, it will be unable to assign variables \(G\) or \(H\) until it has completed its work on the first component. A similar situation arises if it first attempts to solve the second component.

\subsection{2.9.3 Static Variable-Ordering}

Static Variable-Orderings (SVO) are orderings where the variables are branched on in a fixed order. Thus, the variable-ordering is the same along every branch of the search tree. Typically SVOs use graph-based heuristics to compute their fixed variable-ordering. A static variable-ordering can be computed by following a initially computed tree-decomposition. For example, consider the problem seen in Figure 2.8. As a preprocessing step, the solver orders the variables in the topmost label \((D\) and \(F)\) in some way, then proceed to the next label, adding variables not yet in the ordering. During search, the solver chooses the first unassigned variable in the ordering.

Although this approach finds decompositions early in the search space, the solver has no freedom to deviate from the ordering.

\subsection{2.9.4 Partial Variable-Ordering}

Partial Variable-Ordering (PVO) uses a combination of both semantic and graph-based heuristics [75]. Given a tree-decomposition and an active label, search is free to dynamically select from any unassigned variable in the active label. For example, consider the
problem seen in Figure 2.8. Initially, the label containing variables $D$ and $F$ is active. PVO uses a semantic variable-ordering heuristic to select a variable from the active label, until all variables from the label are assigned. At this point, a child label is made active. Thus, PVO attempts to create decompositions early in the search space, but has more variable-ordering freedom than SVO. However, PVO has less freedom than DVO as it must choose a variable from the currently active label rather than any unassigned variable.
Chapter 3

Component Templates

3.1 Introduction

In this chapter, a method is presented that increases the efficiency of search with decomposition when dynamic variable-orderings are used. When decomposition is exploited during search, the same components can appear multiple times. Significant computational efficiencies can be gained by remembering (caching) information about these components, particularly bounds on their optimal value, when they are encountered during search and re-using that information when components are subsequently encountered. However, caching the bounds of components (component caching) can entail significant overheads.

In Section 2.9.1 (see page 38), graph-based variable-ordering heuristics were introduced. Graph-based heuristics are designed to decompose a problem early in the search space, thus maximizing the effect of decomposition algorithms. Using graph-based variable-ordering heuristics also offers a second advantage. Any component can be indexed by the instantiation of the variables it shares with the labels of its ancestor nodes. This instantiation of variables can be used to index into a cache of the component’s values [27]. As a result, component detection is easily accomplished by a simple analysis of the
tree-decomposition, and caching can be implemented by indexing from the nodes of the tree-decomposition. This is the scheme implemented in many AND/OR solvers [72]. However, this scheme only works if the tree-decomposition is followed exactly.

Later in this thesis, it is demonstrated that B&B often benefits greatly by more freedom in its variable-ordering; therefore it is sometimes more beneficial to violate the tree-decomposition, since the tree-decomposition construction considers only the scopes of the objectives and not their values. Thus, a tree-decomposition cannot take into account the fact that some objectives may have very high cost. If a variable has a very high cost, then instantiating the variable might quickly drive up the bounds generated during search, allowing a bounding function to more efficiently prune the search space.

In this chapter, a method is presented that increases the efficiency of component caching, especially when dynamic variable-orderings are used. As a result, this method improves the overall efficiency of search with decomposition. The chapter presents a novel way to represent the common structure of an entire set of components in a single data structure called a component template. Thus, the individual components that are instances of a template can share a single representation. Component templates allow fast lookup of cached results through array indexing, making caching considerably more time and space efficient. In addition, component templates allow efficient component detection during search.

This chapter applies component templating to Weighted Constraint Satisfaction Problems (WCSPs) introduced in Definition 2.3.2 (see page 10). However, component templating also can be used in other applications of search with decomposition and caching, for example MAX-CSP, #CSP, and MPE.
3.2 Background

3.2.1 Components

In backtracking search, at each node of the search tree, \( n \), there is a set of variable assignments, \( \mathcal{A} \). Recall from the definition of components (Section 2.7.1 on page 25), that a component is a subset of the original problem that has been isolated by \( \mathcal{A} \).

At each node, \( n \), some set of assignments, \( \mathcal{A} \), have been made, and some set of variables, \( \mathcal{U} \), remain unassigned. If the components at node \( n \) are \( \kappa_1, \ldots, \kappa_k \), then they must satisfy the following four conditions:

1. \( \kappa_i.\text{Vars} \subseteq \mathcal{U} \) (the variables of the components are unassigned); and,
2. \( \kappa_i.\mathcal{A} \subseteq \mathcal{A} \) (the current assignments isolate the variables of the component); and,
3. \( \bigcup \kappa_i.\text{Vars} = \mathcal{U} \) (every unassigned variable is in some \( \kappa_i.\text{Vars} \)); and,
4. \( \kappa_i.\text{Vars} \cap \kappa_j.\text{Vars} = \emptyset \) for \( i \neq j \) (every unassigned variable appears in exactly one component).

Example 3.2.1 Consider a WCSP with variables \( \{A, B, M, N, X, Y\} \) and objectives \( o_1(A, B, M), o_2(X, Y, N), \) and \( o_3(M, N) \). At a node, \( n \), where only the assignment \( M = a \) has been made, there will be exactly two components: \( \kappa_{ab} \) defined by the tuple \( \langle \kappa_{ab}.\text{Vars} = \{A,B\}, \kappa_{ab}.\text{Obj} = \{o_1\}, \kappa_{ab}.\mathcal{A} = \{M = a\} \rangle \), and \( \kappa_{xyn} \) defined by the tuple \( \langle \kappa_{xyn}.\text{Vars} = \{X,Y,N\}, \kappa_{xyn}.\text{Obj} = \{o_2, o_3\}, \kappa_{xyn}.\mathcal{A} = \{M = a\} \rangle \). All of the four conditions described above hold for components \( \kappa_{ab} \) and \( \kappa_{xyn} \).

Computing the components that exist at a node of the search tree is easily accomplished by standard algorithms for detecting the connected components of a graph, e.g., union-find or depth-first search [25]. An undirected graph, \( G_n \), can be defined that includes a node for every objective, and a node for every uninstantiated variable. \( G_n \)
contains an edge between two nodes if and only if one of the nodes is a variable node \( V \), the other node is an objective node \( o \), and \( V \in \text{scope}(o) \).

**Observation 3.2.2** The connected components of \( G_n \) correspond to the minimal components of \( n \). In particular, \( \kappa \) is a component at \( n \) if and only if there exists a connected component, \( c \) of \( G_n \), such that \( \kappa.\text{Vars} \) is the set of variable nodes in \( c \); \( \kappa.\text{Obj} \) are the set of objective nodes of \( c \); and, \( \kappa.\text{A} \) are the assignments made to the instantiated variables of these objectives.

The set of minimal components can be calculated by incrementally adding connected objectives to \( \kappa.\text{Obj} \) and connected variables to \( \kappa.\text{Vars} \) until there are no more edges in \( G_n \) to follow. This is precisely what CONNECTED COMPONENT algorithms do (these algorithms also compute minimal components). \( \kappa.\text{A} \) is computed after \( \kappa.\text{Vars} \) and \( \kappa.\text{Obj} \) have been finalized.

Note that any objective that has been fully instantiated at the node, \( n \), will become an isolated node in \( G_n \); i.e., a single node component. These fully instantiated objectives form components with no variables, one objective, and the assignments required to fully instantiate that objective.

**Example 3.2.3** Suppose in Example 3.2.1, the assignment \( N = b \) is made, so that \( M = a \) and \( N = b \) are the only two assignments. This assignment will generate three components. Component \( \kappa_{ab} \) is unaffected by the new assignment, but component \( \kappa_{xy} \) is now split into two components \( \kappa_{xy} = (\{X, Y\}, o_2, N = b) \) and \( \kappa_{mn} = (\{\}, o_3, \{N = b, M = a\}) \). The component \( \kappa_{mn} \) contains no variables, just a single fully instantiated objective. The component \( \kappa_{xy} \) does not contain \( M = a \) in its assignment set; although its parent component \( \kappa_{xy} \) contained \( M = a \) in its assignment set. The assignment, \( M = a \), is not needed to isolate \( \kappa_{xy} \) from the rest of the problem.

After backtracking from the assignment \( M = a \), suppose that the assignments \( M = b \) and \( N = b \) are made. These assignments will generate three components, including the
component $\kappa_{xy} = \langle \{X, Y\}, o_2, N = b \rangle$. $\kappa_{xy}$ is identical to the component generated from the assignments $M = a$ and $N = b$. If search found an optimal solution to $\kappa_{xy}$ under the assignments $M = a$ and $N = b$, that solution can be re-used under the assignments $M = b$ and $N = b$.

### 3.2.2 Computation Benefits of Components

If $\kappa$ is a component, then the cost of any assignment, $A$, to its variables, $\kappa.Vars$, is equal to the sum of its objectives, $\kappa.Obj$, evaluated at the set of assignments, $A \cup \kappa.A$. Note that the objectives of $\kappa$ are functions that include only $\kappa.Vars$ and the assignments in $\kappa.A$. Thus, any complete assignment to $\kappa.Vars$, combined with $\kappa.A$, fully instantiates all of the objectives in $\kappa.Obj$ (yielding a single numeric cost for each objective which can then be summed). For any set of assignments, $\kappa.A$, that isolate the component, the **minimal cost** of $\kappa$, $\text{mincost}(\kappa)$, is the minimum cost that can be achieved by any assignment to the variables $\kappa.Vars$:

$$\text{mincost}(\kappa) = \min_{A: A \text{ is an assignment to } \kappa.Vars} \sum_{o \in \kappa.Obj} o(A \cup \kappa.A).$$

In the above equation, recall from Section 2.2.3 (page 8) that for convenience, we allow $o(A)$ to be a superset of the scope of $o$. An assignment that achieves the minimal cost solves the component. Note that a component that corresponds to a fully instantiated objective has a cost equal to the cost of the instantiated objective. Note also that the cost of a component can be computed by examining assignments to only the component’s variables; the rest of the problem can be ignored.

**Proposition 3.2.4** Let $\kappa_1, \ldots, \kappa_k$ be the set of components at $n$. For any complete set of assignments that extends the assignments already made at $n$, the minimal cost that can be obtained is $\sum_{i=1}^{k} \text{mincost}(\kappa_i)$.

This proposition follows from the fact that the minimum cost of components are independent of each other. Computationally, this proposition means that each component
can be solved independently. While solving a component, additional variable assignments can generate further decompositions, which in turn can be solved independently. In other words, decomposition can be applied recursively.

3.3 Templates

Component templates are used to represent the shared information of a group of components, each of which then becomes an instance of the template. The basic idea is quite simple, with most of the innovation arising from the ways in which templates can be exploited algorithmically.

From the definitions above, it can be observed that for any two components, \( \kappa_1 \) and \( \kappa_2 \), if \( \kappa_1.\text{Vars} = \kappa_2.\text{Vars} \neq \emptyset \), then \( \kappa_1.\text{Obj} = \kappa_2.\text{Obj} \). Furthermore, the variables assigned in \( \kappa_1.\mathcal{A} \) are identical to the variables assigned in \( \kappa_2.\mathcal{A} \). In fact, the only difference between two non-equal components containing the same (non-empty) set of variables is that the particular values assigned in \( \kappa_1.\mathcal{A} \) and \( \kappa_2.\mathcal{A} \) differ. As a consequence of the values being different, \( \text{mincost}(\kappa_1) \) and \( \text{mincost}(\kappa_2) \) may also differ (since the objectives are being minimized subject to the differing values in \( \kappa_1.\mathcal{A} \) and \( \kappa_2.\mathcal{A} \)).

Component templates represent all of the components that have an identical set of variables. Formally, a component template, \( \mathcal{T} = \langle \mathcal{T}.\text{Vars}, \mathcal{T}.\text{Obj}, \mathcal{T}.\mathcal{D} \rangle \), is: a set of variables which are represented by \( \mathcal{T}.\text{Vars} \); a set of objectives which are represented by \( \mathcal{T}.\text{Obj} \); and a second set of variables which are represented by \( \mathcal{T}.\mathcal{D} \), disjoint from \( \mathcal{T}.\text{Vars} \), called the dependency variables. Every set of assignments, \( \mathcal{A} \), to the variables in \( \mathcal{T}.\mathcal{D} \) generates an instance of the template, which is denoted by \( \mathcal{T}(\mathcal{A}) \). Every instance \( \mathcal{T}(\mathcal{A}) \) is a component, defined by the tuple \( \langle \mathcal{T}.\text{Vars}, \mathcal{T}.\text{Obj}, \mathcal{A} \rangle \). In other words, an instance of a template is a component that has variables and objectives identical to those of the template and in which \( \mathcal{A} \) is a set of assignments to the dependency variables of the template.
Example 3.3.1 For instance, consider the component $\kappa_{ab} = \langle \{A, B\}, o_{1}, M = a \rangle$ seen in the previous example. The component template $\mathcal{T}_{ab} = \langle \{A, B\}, \{o_{1}\}, \{M\} \rangle$ includes $\kappa_{ab}$ as one of its instances. In this case $\mathcal{T}_{ab}(M = a)$ is the component $\kappa_{ab}$.

Using Templates During Search. As described above, the components at each node, $n$, can be determined by a CONNECTED COMPONENTS algorithm run on the graph $G_{n}$. Note, however, that $G_{n}$ contains only variables and objectives. $G_{n}$ does not include the actual values assigned to the instantiated variables. Hence, the algorithm actually identifies a set of templates. The components at the node, $n$, are the particular instances of these templates determined by the assignments at $n$.

Once a template is detected for the first time, a data structure is created to represent the template and stored in a template cache. The template data structure can then be used to efficiently detect any instance of the template that might be created at future nodes of the search.

3.3.1 Templates in Backtracking Search

In order to understand the way in which templates are used in backtracking search, it is necessary to understand the following observation:

Observation 3.3.2 Let $\mathcal{A}$ be the set of assignments made at node $n$, and let $\mathcal{T}$ be a component template. If $\mathcal{A}$ instantiates all of the variables in $\mathcal{T}.D$ and none of the variables in $\mathcal{T}.\text{Vars}$, then $\mathcal{T}(\mathcal{A}|^{\mathcal{T}.D})$ is one of the components at $n$, where $\mathcal{A}|^{\mathcal{T}.D}$ is the subset of $\mathcal{A}$ that assigns the variables in $\mathcal{T}.D$.

The observation holds for the following two reasons:

1. The instance $\mathcal{T}(\mathcal{A}|^{\mathcal{T}.D})$ is isolated from the rest of the problem by $\mathcal{A}$.

2. If $\mathcal{A}$ assigned any of the variables in $\mathcal{T}.\text{Vars}$, $\mathcal{T}$ would not be a component at $n$, rather some reduction of $\mathcal{T}$ would exist at $n$. 
Once a template has been created, i.e., once it has been detected by the \textit{connected components} computation, it can be used in future search to make component detection more efficient. In particular, a lazy watch technique [81] can be employed to detect when all of the variables of \( \mathcal{T}.\mathcal{D} \) have been instantiated at a node of the search tree. An unassigned variable is designated as a watch variable. When a watch variable is assigned, the other variables in \( \mathcal{T}.\mathcal{D} \) can be checked to see if any of them remain unassigned. If an unassigned variable remains, the unassigned variable becomes the new watch variable for \( \mathcal{T} \). If no unassigned variables remain, an instance of \( \mathcal{T} \) may exist as a component at the current search node. Note that if the watch variable of \( \mathcal{T} \) is not assigned, no work needs to be done to check for instances of \( \mathcal{T} \).

If all the variables in \( \mathcal{T}.\mathcal{D} \) have been assigned, and all the variables of \( \mathcal{T}.\mathcal{Vars} \) remain unassigned, then \( \mathcal{T} \) will be triggered and \( \mathcal{T}.\mathcal{Vars} \) forms a component at the current node. All of the template’s variables and objectives can then be removed from graph \( G_n \), reducing the size of \( G_n \). \textit{connected components} can then be run on this smaller remaining graph to identify the other components at the current node. Triggering components and reducing the size of \( G_n \) in this way can yield a non-trivial improvement in the total time needed to perform component detection.

Once a template has been triggered, information must be accessed about the particular instance, \( \mathcal{T}(\mathcal{A}|\mathcal{T}.\mathcal{D}) \), that exists at the current search node. Associated with each template is a value cache that is used to store the upper and lower bounds on the cost of its instances (solutions can also be stored for solved instances). If \( \mathcal{T}(\mathcal{A}) \) is a template instance, then \( \mathcal{T}(\mathcal{A}).lb \) and \( \mathcal{T}(\mathcal{A}).ub \) will denote the stored lower and upper bounds on \( \text{mincost}(\mathcal{T}(\mathcal{A})) \). If the instance has never been seen before, these bounds are given some default initial values.\(^1\)

\(^1\)The default initial bounds can be generated in a variety of ways. See Section 2.6 on page 21 for examples.
The search in the subtree below the current node will either compute the value of the instance (making $T(A).lb = T(A).ub$), compute better bounds on its value, or backtrack without updating these bounds. When a template is triggered, accessing an instance’s bounds can be very efficient. In particular, each variable in $T.D$ has a finite domain of values, and each instance $T(A)$ is defined by the values assigned to these variables. Thus the instance’s defining sequence of values can be used as an index into a multi-dimensional array. However, instances of the template may never be encountered during the search (because of branch and bound pruning). If the multi-dimensional array is too large, the instance’s values can be used as a hash code to index into a small hash table more suitable for storing sparse data.

**Example 3.3.3** In the previous example, when the assignment $M = a$ is made for the first time, a new template, $T_{ab} = \langle \{A, B\}, \{o_1\}, \{M\} \rangle$, is created by detecting the connected component $\langle \{A, B\}, o_1 \rangle$. The instance of this template $T(M = a)$, so detected, corresponds to the component $\kappa_{ab}$. Search proceeds over the variables $A$ and $B$, returning the upper and lower bounds of component $\kappa_{ab}$ under the instantiation $M = a$. The lower and upper bounds are the bounds that can be achieved by the objectives in $T_{ab}.Obj$ over all possible values for $A$ and $B$, subject to $M = a$. These bounds are stored in the template cache as $T_{ab}(M = a).lb$ and $T_{ab}(M = a).ub$, i.e., as bounds indexed by the assignment $M = a$.

If the assignment $M = k$ is made later in search, while $A$ and $B$ are still unassigned, then $T_{ab}$ is triggered and the bounds on the new instance $T_M(M = k)$ are retrieved from the template’s value cache. If $k = a$, then the cached upper and lower bounds can be re-used at this new search node.
3.4 AND/OR search with Component Caching

AND/OR search with Component Caching (\texttt{AND/OR+CC}), described in Algorithm 3, is an extension of \texttt{AND/OR} (Algorithm 2, page 28) that uses the techniques in this chapter to perform search efficiently with recursive decompositions. \texttt{AND/OR+CC} attempts to find the minimum cost for a single component, defined as a template instance, $\mathcal{T}(A)$, and is given an upper-bound, $\text{ub}$. \texttt{AND/OR+CC} can abort its computation as soon as it discovers that $\mathcal{T}(A)$ cannot achieve a cost lower than $\text{ub}$. Even if the computation is aborted, however, the routine still stores the best bounds it was able to compute before termination (Line 20). Storing the bounds produced by a partial computation of a component’s optimal cost allows a better integration with Branch and Bound. \texttt{AND/OR+CC} uses a call to \texttt{getBounds} (just like \texttt{B&B}) in order to calculate initial lower and upper bounds for a component.

\texttt{AND/OR+CC} begins in Line 3 by checking whether (a) $\mathcal{T}(A).lb \geq \text{ub}$ (in which case the required bound cannot be achieved) or (b) $\mathcal{T}(A).lb = \mathcal{T}(A).ub$ (in which case the component has already been fully solved). In either of these cases, \texttt{AND/OR+CC} can immediately return.

If neither of these cases hold, Line 4 checks whether the upper-bound of the component is lower than $\text{ub}$. If the upper-bound of the component is lower than $\text{ub}$, the algorithm will eventually compute the component’s minimum cost (the computation cannot be aborted by bounding), and it is more efficient to set $\text{ub}$ to the upper-bound of the component, as this is a tighter bound. A variable, $V$, is then selected for assignment. Line 7 determines the new templates formed when $V$ is assigned a value. The templates are calculated as follows: first, triggered templates are removed from the constraint graph; second, a connected component algorithm finds the remaining connected components, each of which forms a new template.

The value $\Delta d$ (Line 9) is the sum of the costs of every objective $o \in \mathcal{T}(A).Obj$ that has become fully instantiated as a result of the assignment of the value $d$ to the variable $V$. 
Algorithm 3: AND/OR Branch and Bound search with Component Caching.

```plaintext
AND/OR+CC (T(A),UB)
/* On entry T(A) must have valid bounds (T(A).lb, T(A).ub). If
  mincost(T(A)) < UB then compute T(A).lb = mincost(κ) = T(A).ub. Else
  compute valid bounds on T(A) such that UB < T(A).lb ≤ T(A).ub */
begin
  if (T(A).lb < UB ∧ T(A).lb ≠ T(A).ub) then
    if UB > T(A).ub then
      UB := T(A).ub + 1
    V := select variable from T.Vars to branch on
    K := Find the templates contained in the constraint graph consisting of
    T.Obj and T.Vars − {V}
    foreach d ∈ Dom[V] do
      Δd = \sum_{o_i ∈ \text{scope}(o_i) \cap T.Vars = \{V\}} o_i(V = d, A)
      foreach T_i ∈ K do
        \[ A_i := (A \cup \{V = d\})|_{T_i.D} \]
        \[ T_i(A_i).lb = \text{MAX}(\text{getBounds}(T_i(A_i)), \text{getCache}(T_i(A_i)).lb) \]
        \[ T_i(A_i).ub = \text{MIN}(\text{getBounds}(T_i(A_i)), \text{getCache}(T_i(A_i)).ub) \]
      foreach T_i ∈ K (while \( \sum_i T_i(A_i).lb + Δd < UB \)) do
        UB_i := UB - (Δd + \sum_{j ≠ i} T_j(A_j).lb)
        AND/OR+CC (T_i(A_i),UB_i)
      \( (1b^d, ub^d) = (\sum_i T_i(A_i).lb + Δd, \sum_i T_i(A_i).ub + Δd) \)
      UB := min(UB,ub^d)
      (T(A).lb, T(A).ub) = (\text{min}_d(1b^d), \text{min}_d(ub^d))
    setCache(T(A).lb, T(A).ub)
  end
```
As \( V \) is assigned different values, different instances of the templates will be created and solved. For each template, Line 11 identifies the new template instance (components) created by the current value of \( V \). The lower-bound of a component is the maximum of a bounding function and a cache lookup (Line 12). The upper-bound of a component is the minimum of a bounding function and a cache lookup (Line 13). The components are solved recursively in Lines 14–16.

Under the assignment \( V = d \), the minimum cost that can be achieved for \( T(A) \) is \( \Delta d + \sum_{T_i(A_i) \in K} \min \text{cost}(T_i(A_i)) \). This cost must be less than \( \text{ub} \) otherwise we can reject \( V = d \) and try another value for \( V \). Hence, we must have \( \text{ub} > \Delta d + \sum_{T_i(A_i) \in K} \min \text{cost}(T_i(A_i)) > \Delta d + \sum_{T_i(A_i) \in K} T_i(A_i).lb \). \( V = d \) can be abandoned at any time in Line 14 if \( \Delta d + \sum_{T_i(A_i) \in K} T_i(A_i).lb \) becomes greater than or equal to \( \text{ub} \) (this sum changes as the component bounds are updated by the various recursive calls). Each component will be solved in a separate recursion, and this condition implies that for each component, \( T_i(A_i) \), we must have \( \min \text{cost}(T_i(A_i)) < \text{ub} - \Delta d + \sum_{j \neq i} \min \text{cost}(T_j(A_j)) < \text{ub} - \Delta d + \sum_{j \neq i} T_j(A_j).lb \). Thus, this last expression, computed and stored in the variable \( UB_i \) (Line 15), is a valid upper bound for the component \( T_i(A_i) \). This is the bound passed to the recursive call solving \( T_i(A_i) \) in Line 17. When \( \text{AND/OR+CC} \) has completed testing the assignment \( V = d \), bounds for \( T(A) \) under this setting of \( V \) are computed in Line 17.

These bounds may be lower than the initial upper-bound, \( i.e., \text{ub}^d < \text{ub} \). In that case, \( \text{AND/OR+CC} \) will continue to compute the input component’s minimum cost, by trying the remaining assignments to \( V \). However, since \( V = d \) achieves at most cost \( \text{ub}^d \), any new assignment to \( V \) needs to achieve an even lower cost; therefore, \( \text{ub} \) can be lowered (Line 18) before attempting the next assignment to \( V \).

After all values for \( V \) have been tried, the bounds for the input component, \( T(A) \), can be updated. Finally, the updated bounds of component \( T(A) \) are stored in the cache (Line 20). The updated bounds can re-used if the same component is encountered later in search.
It is not difficult to see that $\text{AND/OR+CC}$ is an optimization of the $\text{AND/OR}$. It simply uses caching to avoid duplicated work. Therefore, its correctness follows directly from the correctness of the $\text{AND/OR}$.

**Constraint Propagation.** $\text{AND/OR+CC}$ can also employ constraint propagation over the hard objectives (the original constraints) to prune domain values that would necessarily lead to costs of $\infty$. Domain pruning will reduce the iterations performed at Line 8 and the algorithm requires no other changes.

### 3.5 Empirical Results

Component templating is implemented and tested on the Maximum Density Still Life problem.

The following specific algorithms are tested:

1. Branch and Bound ($\text{BB}$), Algorithm 1 (see page 20).

2. AND/OR ($\text{AND/OR}$), Algorithm 2. This version searches for components and solves them separately, i.e., it performs search with decomposition. However, $\text{AND/OR}$ does not use templates, nor does it cache already solved components.

3. AND/OR with templates ($\text{AND/OR+T}$), Algorithm 3, is a template version of $\text{AND/OR}$. Its only improvement over $\text{AND/OR}$ is the use of templates to improve component detection. There is no template cache to store bounds for the template instances.

4. AND/OR with Component Caching ($\text{AND/OR+CC}$), extends $\text{AND/OR+T}$ by activating the template cache to store computed bounds for the template instances.

$\text{BB}$ uses Dynamic Variable-Ordering, while the remaining algorithms use Partial Variable-Ordering (see Section 2.9 on page 37 for details). These four different algorithms
all use the semantic heuristic Domain/Degree, which minimizes the size of the domain
divided by the degree of the variable. All algorithms perform GAC constraint propagation
on the hard objectives to prune domain values.

All experiments were run on a 2.2 GHz Pentium IV with 6GB of memory. The time
limit for each run was set to 10,000 seconds. In the experiments conducted, the space
used in caching never exceeded available memory, so it was not necessary to prune the
cache during search.

3.5.1 Problem Set

The Maximum Density Still Life (MDSL) problem involves finding maximum density
stable configurations in Conway’s game of Life [66]. MDSL has an $N \times N$ grid of cells
that can be either dead or alive, implicitly surrounded by a boundary of dead cells. A
live cell is stable if and only if it is surrounded by 2 or 3 live cells, and a dead and stable
if and only if it is surrounded by 0, 1, 4, 5, 6, 7, or 8 live cells. The goal is to find a
stable configuration of the cells that has a maximum number of live cells.

The encoding of MDSL is basic. The problem has $N^2$ boolean variables, each one
representing the alive/dead status of a single cell; $N^2$ unary objective functions, one
for each cell, assigning cost zero to live cells and cost one to dead cells; and, $N^2$ hard
objective functions, one for each cell, which assign cost zero to stable settings of the cell
given its neighbour’s values and cost $\infty$ to unstable settings.

The most effective solvers for this problem employ an extensive amount of domain
specific information [66]. However, symmetric templating is entirely generic. The aim of
testing Component Templating on MDSL is to evaluate the effectiveness of the algorithm.

3.5.2 MDSL results

Table 3.1 shows the performance of the six tested algorithms in terms of time in sec-
onds, while Table 3.2 shows the performance of the six algorithms in terms of nodes
expanded. Decomposition (AND/OR) yields significant improvements over standard Branch and Bound (BB) in decreasing the size of the search tree. For the smaller problems, AND/OR takes more time, due to its larger overhead. However, this overhead is quickly recouped as the problems get larger and the reduction in the size of the search tree achieved by decomposition becomes more significant. Since AND/OR+T does not employ caching, it does not reduce the size of AND/OR’s search tree (except for heuristic reasons). It does, however, provide a non-trivial improvement in efficiency. In particular, AND/OR+T shows that template triggering improves the efficiency of detecting components. The times in the table demonstrate that the overhead of detecting components is non-trivial.

AND/OR+CC makes a further improvement over the previous three algorithms by activating storage of bounds in the template, which results in a significant decrease in the size of the search space. There is a corresponding decrease in time because fewer nodes are expanded.

<table>
<thead>
<tr>
<th>Size</th>
<th>AND/OR+CC</th>
<th>AND/OR+T</th>
<th>AND/OR</th>
<th>BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0.1</td>
<td>0.1</td>
<td>0.24</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1.1</td>
<td>2.0</td>
<td>2.4</td>
<td>2.0</td>
</tr>
<tr>
<td>7</td>
<td>8.5</td>
<td>19.8</td>
<td>41.8</td>
<td>55.1</td>
</tr>
<tr>
<td>8</td>
<td>82.3</td>
<td>316.6</td>
<td>408.8</td>
<td>2207</td>
</tr>
<tr>
<td>9</td>
<td>4662.4</td>
<td>&gt;10000</td>
<td>&gt;10000</td>
<td>&gt;10000</td>
</tr>
<tr>
<td>10</td>
<td>7845.4</td>
<td>&gt;10000</td>
<td>&gt;10000</td>
<td>&gt;10000</td>
</tr>
</tbody>
</table>

Table 3.1: Time taken in CPU seconds, fastest time in bold
### Table 3.2: Nodes Expanded, fewest nodes in bold

<table>
<thead>
<tr>
<th>Size</th>
<th>AND/OR+CC</th>
<th>AND/OR+T</th>
<th>AND/OR</th>
<th>BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1646</td>
<td>1646</td>
<td>1646</td>
<td>767</td>
</tr>
<tr>
<td>5</td>
<td>7744</td>
<td>8146</td>
<td>8146</td>
<td>5364</td>
</tr>
<tr>
<td>6</td>
<td>93335</td>
<td>206141</td>
<td>177540</td>
<td>177522</td>
</tr>
<tr>
<td>7</td>
<td>644175</td>
<td>1.5*10^6</td>
<td>2.1*10^6</td>
<td>4.7*10^6</td>
</tr>
<tr>
<td>8</td>
<td>6.1*10^6</td>
<td>2.4*10^7</td>
<td>2.3*10^7</td>
<td>3.1*10^8</td>
</tr>
<tr>
<td>9</td>
<td>9.7*10^7</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>10</td>
<td>5.6*10^8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

### 3.6 Related Work

Referencing components by a lookup table has been proposed for Bayesian networks [27, 1]. The approach uses a static variable-ordering, so each variable is associated with a single set of components. Similarly, lookup tables have also been used in COPs when static variable-ordering is used [73, 77, 29].

Component templating is also used by algorithms presented in future chapters. Chapter 4 and Chapter 5 use templates to quickly determine if a problem has decomposed, and to cache bounds. Chapter 6 uses templates to efficiently detect conditional symmetries.

### 3.7 Discussion

Without caching, **AND/OR** (Algorithm 2 on page 28) requires only space linear in the number of variables in addition to the space required to store the original problem. However, by adding template caching, the space requirements of **AND/OR+CC** can be considerably increased. In fact, **AND/OR+CC** can easily run out of memory when solving hard large problems such as those seen in [79].
It should be noted however that template caching simply increases the efficiency of
the algorithm, it is not required for the algorithm’s correctness. Thus, if templates are
taking up too much space they can be pruned from memory and the algorithm can stop
caching them during search. This will mean that Line 12 and 13 will only be able to
retrieve default bounds computed by \textit{getBounds}, which will increase the size of the search
space the algorithm must explore.

Nevertheless, Mateescu and Dechter [79] propose a number of techniques for reducing
the memory requirements of caching that could be exploited for templates on larger
instances.

\section{Conclusion}

The chapter introduces a new technique called component templating that reduces the
computational cost associated with decomposition-based search. Component templates
reduce certain computational overheads associated with decomposition-based search when
dynamic variable-orderings are used. Component Templates can also store additional in-
formation, such as bounds on the component. They can also be used to more efficiently
exploit conditional symmetries (discussed in Chapter 6).
Chapter 4

OR-Decomposition

4.1 Introduction

This chapter explores a new method for exploiting decomposition on WCSPs with low tree-width problems (low tree-width was introduced in Section 2.7.5 on page 30).

In Chapter 3, it was noted that decomposition can reduce the worst-case time complexity from $2^{O(n)}$ to $n^{O(w)}$. When caching is added to decomposition-based search algorithms, the worst-case time complexity is further reduced from $n^{O(w)}$ to $n \cdot 2^{O(w)}$ [9].

The downside to using decomposition-based search algorithms is that they can reduce the effectiveness of the bounding techniques that are essential for solving WCSPs. The standard method for exploiting decomposition during search is to invoke a separate recursion for each component generated during the search, yielding an AND/OR search tree [27, 73, 59, 29]. By solving each component in a separate recursion, AND/OR has no ability to interleave the solving of its current components. As will be explained below, this inability can lead to inefficiencies in the search.

In this chapter, an algorithm is presented that is able to exploit decomposition in a standard backtracking search tree (an OR tree), which gives the algorithm complete freedom in its variable-ordering. The algorithm is able to switch between working on
different components by simply selecting a variable from a different component. In this way, information can be incrementally gathered for any component, and can be used to improve the bounding information for all of the other components. As a result of this improvement, it is possible that an entire collection of components can be rejected without ever having to solve any of the components to optimality. In addition, the useful bounding technique of local propagation (soft-arc consistency) [24] is integrated with the algorithm.

This chapter will begin by presenting the new algorithm and illustrating some of its properties. Empirical results demonstrating the potential of the approach are then discussed.

### 4.2 Solving WCSPs

The techniques presented in this chapter are most beneficial on WCSPs of low tree-width. However, because of the effectiveness of bounding, problems with high tree-width can often be solved efficiently with the algorithm presented in this chapter.

**Branch and Bound** (B&B, Algorithm 1 on page 20) is a standard technique for solving WCSPs using backtracking search. As noted in Chapter 2, B&B works by building up partial variable assignments in a depth-first manner, and uses a bounding function to prune the search space. Each recursion is passed a WCSP, $\mathcal{P}$, (a reduction of the original WCSP by the current set of assignments) and an upper-bound $\text{ub}$. B&B tries to compute $\text{mincost}(\mathcal{P})$, subject to the condition that it can abort its computation as soon as it can conclude that $\text{mincost}(\mathcal{P}) \geq \text{ub}$.

**Branch and Bound with Decomposition:** A more recent technique used in solving WCSPs is search with decomposition (AND/OR, Algorithm 2 on page 28) [27, 73, 59, 29]. In Chapter 3, AND/OR was presented using component templates (AND/OR search with Component Caching, Algorithm 3 on page 52). As variable assignments are
made during backtracking search, the WCSP can become separated into components (see Section 2.7.1 on page 25). **AND/OR** proceeds by solving each of these components in a separate search tree by calling itself recursively on each component.

### 4.3 Decomposition without Separate Recursions

Although **AND/OR** gains computational advantage from breaking the problem into independent sub-problems, it can make bounding less effective by restricting itself to solving one component at a time.

As discussed in Chapter 3, when solving a collection of components, \( \kappa_1, \ldots, \kappa_n \), subject to an upper-bound, \( \text{ub} \), the best upper-bound that can be used when solving \( \kappa_i \) is \( \text{ub} - \Delta^d + \sum_{\kappa_j \neq \kappa_i} \text{lb}(\kappa_j) \) (see **AND/OR+CC** on page 52, Line 15). That is, the other components might all achieve an optimal cost equal to their lower-bounds, so the best bound on the cost of \( \kappa_i \) is the upper-bound, \( \text{ub} \), minus the sum of the lower-bounds of all other components. \( \Delta^d \) represents the cost of objectives that were fully instantiated by the variable assignment, and can be viewed as components over no variables. If little is known about components \( \kappa_1, \ldots, \kappa_{i-1}, \kappa_{i+1}, \ldots, \kappa_n \), their lower-bounds might be small compared to their optimal costs, and thus **AND/OR** might be forced to solve \( \kappa_i \) under a very weak bound (i.e., a high upper-bound).

**Example 4.3.1** Consider solving a component, \( \kappa \), with \( \text{ub} = 100 \). Say that **AND/OR** branches on variable \( V \), making the assignment \( V = d \), and that this adds zero to the cost \( (\Delta^d = 0) \) while breaking \( \kappa \) into five components, \( \kappa_1, \ldots, \kappa_5 \). If \( \text{mincost}(\kappa_i) = 25 \), the assignment \( V = d \) eventually must be rejected since any assignment that includes \( V = d \) can only achieve a minimal cost of 125. Suppose that for each component, \( \kappa_i \), the initial lower-bound, \( \kappa_i.\text{lb} \), is 10. The upper-bound, \( \text{ub}_1 \) (the upper-bound used for solving \( \kappa_i \), see **AND/OR+CC** on page 52, Line 15) will be 100 - 4 * 10 = 60. Hence, search will be forced to solve \( \kappa_1 \) to optimality. This search will update \( \kappa_1.\text{lb} \) to 25 (the optimal cost of
\( \kappa_1 \). When AND/OR solves \( \kappa_2 \), \( \text{ub}_2 \) will be set to \( 100 - (3 \times 10 + 25) = 45 \), and the search will also be forced to solve \( \kappa_2 \) to optimality. When AND/OR solves \( \kappa_3 \), \( \text{ub}_3 \) will be set to \( 100 - (2 \times 10 + 50) = 30 \), and \( \kappa_3 \) must be solved to optimality. When AND/OR solves \( \kappa_4 \), \( \text{ub}_4 \) will then be set to \( 100 - (1 \times 10 + 75) = 15 \) and the search of \( \kappa_4 \) can terminate as soon as it determines that \( \kappa_4.\text{lb} \geq 15 \). After this step, \( \sum \kappa_i.\text{lb} \geq 100 \) and \( V = d \) is rejected. Computing the optimal value for \( \kappa_1-\kappa_3 \) can be very expensive. A much shallow search of all of the components could have served to move the lower-bounds of every component to 20 or higher, so that \( V = d \) could be rejected without having to solve any of the components to optimality.

The key contribution of this chapter is to demonstrate how the computational benefits of decomposition can be obtained without having to perform separate recursions for each component. The proposed method exploits the ideas originally presented in [7] for counting problems where the benefits of decomposition are obtained in a regular backtracking search tree (OR-tree). The ideas of [7] are extended in a non-trivial way so that bounding can be exploited.

The new algorithm, OR-Decomposition, is shown in Algorithm 4. Like B&B, and unlike AND/OR, the algorithm takes as input the entire remaining problem (AND/OR takes as input a single component). However, unlike B&B, instead of the input being regarded as a single reduced WCSP, \( \mathcal{P} \), the input is broken up into a set of components, \( \mathcal{K} \). The aim is to solve all of components in \( \mathcal{K} \) (i.e., compute bounds such that \( \forall \kappa \in \mathcal{K} : \kappa.\text{lb} = \kappa.\text{ub} \) or equivalently \( \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb} = \sum_{\kappa \in \mathcal{K}} \kappa.\text{ub} \)). The recursion can abort if the combined cost of these components is greater than or equal to the passed upper-bound \( \text{ub} \) (i.e., when \( \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb} \geq \text{ub} \)).

If neither \( \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb} = \sum_{\kappa \in \mathcal{K}} \kappa.\text{ub} \) nor \( \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb} \geq \text{ub} \) have been met, some unassigned variable is chosen and all of its values are tried. On each instantiation, the component, \( \tau \), containing \( V \) might be split up into a new collection of components. The new collection of components replaces \( \tau \) in the recursive call (Line 12). As in B&B,
1 OR-Decomp (K, ub)

    /* On entry each κ ∈ K must have valid bounds (κ.lb, κ.ub). If
     * \[ \sum_{\kappa \in K} \text{mincost}(\kappa) < \text{ub} \] then compute exact bounds for every
     * κ ∈ K: κ.lb = mincost(κ) = κ.ub. Else compute valid bounds on the
     * components in K such that ub ≤ \[ \sum_{\kappa \in K} \kappa.lb \]. */

2 begin

3 if (\[ \sum_{\kappa \in K} \kappa.lb < \text{ub} \land \sum_{\kappa \in K} \kappa.lb \neq \sum_{\kappa \in K} \kappa.ub \]) then

4     choose (any variable V such that \( \exists \tau \in K \) with \( V \in \tau.Vars \) and

5     \( \tau.lb \neq \tau.ub \))

6     \text{AddConstraint}(\tau, \tau.ub)

7 foreach \( d \in \text{Dom}[V] \) while \[ \sum_{\kappa \in K} \kappa.lb \leq \text{ub} \] do

8         UB = min(UB, \[ \sum_{\kappa \in K} \kappa.ub \])

9         \( \Delta^d = \text{cost}(V = d, \tau) \)

10        \( K^d = \text{toComponents}(\tau|V=d) \)

11 foreach \( \kappa^d \in K^d \) do

12    \( (\kappa^d.lb, \kappa^d.ub) = \text{getBounds}(\kappa^d) \)

13 \( K' = (K - \tau \cup K^d) \)

14 \text{OR-Decomp} (K', \text{ub} - \Delta^d)

15 \[ (lb^d, ub^d) = \sum_{\kappa^d \in K^d} (\kappa^d.lb, \kappa^d.ub) \]

16 \( \tau.ub = \min(\tau.ub, ub^d + \Delta^d) \)

17 \text{RemoveConstraint}(\tau, \tau.ub)

18 \( \tau.lb = \max(\tau.lb, \min_{d \in \text{Dom}[V]} lb^d + \Delta^d) \)

18 end

\textbf{Algorithm 4}: Decomposition and Bounding in a Standard Backtracking Tree
the upper-bound is updated to account for the cost of making the assignment $V = d$. Note that unlike AND/OR, all of the remaining components are passed to the recursive call (Line 13). Thus, the sub-tree search below this invocation can choose to branch on variables from any component in any order—search is not constrained to branch only on the remaining variables of a single passed component as in AND/OR.

On return from the search below, the bounds of the newly generated components in $\mathcal{K}^d$ (and the other components not in $\mathcal{K}^d$) may have been updated, and the upper-bound of $\tau$ can be updated (Line 15). The algorithm will continue to compute the minimum cost of $\tau$, by trying the remaining assignments to $V$. However, since $V = d$ achieves at most cost $ub^d + \Delta^d$, any new assignment to $V$ needs to achieve an even lower cost. Therefore, $ub$ can be lowered (Line 7) before attempting the next assignment to $V$. After trying all of the values for $V$, the lower-bound of $\tau$ can be updated (Line 17) to be the minimum lower-bound achieved over all the values in the domain of $V$.

Note that in the search below, it is possible to branch on the other components in $\mathcal{K}$ and to update their bounds. Hence, sufficient information can be gathered to abort the for loop before trying all of the values of $V$. This motivates the while test during the for loop (Line 6). Note also that if any of these components are solved, i.e., if exact bounds on their value are computed, search will never branch on the variables in the components again: the branch variable must be from an unsolved component (Line 4). That is, if under $V$’s first value, some component, $\kappa' \in \mathcal{K} : \kappa' \neq \tau$, is solved, then in the sub-trees generated by other assignments to $V$, search will never branch on any of the variables of $\kappa'$ again. This is one manner in which decomposition is exploited. If all of the components of $\mathcal{K}$ other than $\tau$ are solved in the search under $V$’s first value (which may occur if $ub - \Delta^d > \sum_{\kappa \in \mathcal{K} \land \kappa \neq \tau} \text{mincost}(\kappa)$) then OR-Decomposition will obtain all of the computational benefits of decomposition.

**Example 4.3.2** Consider a WCSP, $\mathcal{P}$ with two objectives, $o_1(A, B, C) = A + B + C$ and $o_2(C, D, E) = C + D + E$, where all of the variables have domain $\{0, 1\}$. Thus,
mincost(\mathcal{P}) = 0 is obtained when all variables have been set to zero. Suppose also that 
getBounds always returns the weakest bounds possible for the remaining reduced WCSP. 
Suppose further that OR-Decomposition first branches on \( C = 0 \), which splits the 
problem into two components \( \kappa_1 = \{ o_1(\text{A, B, C} = 0) \} \) and \( \kappa_2 = \{ o_2(\text{C} = 0, \text{D, E}) \} \).

Consider the search trees generated for OR-Decomposition, AND/OR, and B&B 
after the assignment \( C = 0 \) is made if all three algorithms always assign the value 1 
before 0. The search by B&B, shown in Fig 4.1, will attempt 20 variable assignments. 
In contrast, the search by AND/OR, shown in Fig 4.2, is smaller attempting only 12 
assignments. AND/OR is able to detect that the problem consists of two independent 
components and solve them independently. OR-Decomposition, shown in Fig 4.3, also 
searches only 12 nodes. It also exploits decomposition, but in a different search tree. In 
particular, under the left-most instantiation of the variables A and B, \( \kappa_2 \) is solved to 
optimality. Hence, the search need not branch on D and E again until a different value 
is assigned to C.

However, sometimes bounding in OR-Decomposition can interfere with solving the 
components in \( \mathcal{K} \) independently.

Example 4.3.3 For example, suppose that \( \mathcal{K} \) contains two components, \( \kappa_1 \) and \( \kappa_2 \), 
where \( \kappa_1 \) contains only a single unassigned variable, V, with values \{a, b, c\}. Say OR-
Decomposition first branches on V, then after each value is assigned to V, OR-
Decomposition will attempt to solve \( \mathcal{K}' = \{ \kappa_2 \} \) (\( \mathcal{K}' \) will be an empty set of components 
since V is \( \kappa_1 \)'s final value). Depending on \( \Delta^d \), OR-Decomposition will try to solve \( \kappa_2 \) 
under different, perhaps too stringent bounds. For example, suppose mincost(\( \kappa_2 \)) = 10, 
ub = 15, and the cost of assigning a, b, and c to V is \( \Delta^a = 7, \Delta^b = 6 \), and \( \Delta^c = 3 \) 
respectively. OR-Decomposition attempts to solve \( \kappa_2 \) under both \( V = a \) and \( V = b \) 
will fail (although \( \kappa_2 \).lb will increase). Only when \( V = c \) will OR-Decomposition solve 
\( \kappa_2 \) under a bound that is greater than mincost(\( \kappa_2 \)). Although \( \kappa_2 \) is solved to optimality
Figure 4.1: Search Space of B&B

Figure 4.2: Search Space of AND/OR

Figure 4.3: Search Space of OR-Decomposition
under $V = c$, it would have been more efficient to solve $\kappa_2$ to optimality under $V = a$. As the number of variables in $\kappa_1$ increases, these repeated attempts to solve $\kappa_2$ can multiply. Some savings can, however, occur since each solution attempt can tighten the bounds on $\kappa_2$. Nevertheless, a multiplicative effect can occur destroying independence.

Thus, on the positive side OR-Decomposition can interleave the solving of current components by branching on variables from different components at each recursion. Interleaving the solving of components can refine incrementally the bounds of many different components so that a collection of components can be rejected without having to solve any component to optimality, while still obtaining many of the benefits of decomposition. On the negative side, however, bounding can sometimes interfere with the benefits of decomposition, as illustrated in Example 4.3.3.

A couple of simple ideas can mitigate the negative effects of bounding on decomposition. The first idea is to force a component to be solved to optimality if the search continues to return to the same component. The second idea is to force a component to be solved to optimality if that component contains a small number of variables. In the example above, the first method would solve $\kappa_2$ (i.e., find $\text{mincost}(\kappa_2)$) after having returned to it some number of times. The second method would solve $\kappa_1$ (finding that $V = c$ is the correct assignment to make), before advancing to $\kappa_2$ since $\kappa_1$ is small. In our implementation of OR-Decomposition, we did not find an effective way of utilizing the first idea: a fixed count of the number of times search could return to a component before forcing the component to be solved to optimality often degraded overall performance. The second idea, forcing the solution of a component when it is small, was effective. In our implementation we forced the solution of any component whose variables had a product domain size of 20 or less.

As noted earlier in this chapter, AND/OR with caching has the worst case time complexity is $n^{O(1)}2^w$. OR-Decomposition does not maintain the same worst case complexity results. However, in practice both AND/OR and OR-Decomposition
solve instances in far less time than \( n \cdot 2^{O(w)} \) (see the experimental section of this chapter).

### 4.3.1 Local Bounding

There is one further aspect of **OR-Decomposition** that has yet to be explained: the two lines **AddConstraint** and **RemoveConstraint** that bracket the for loop over \( V \)'s values. The intuition for these lines is that the current component, \( \tau \), has an upper-bound, \( \tau.ub \), that is initialized when \( \tau \) is first added to \( K \). \( \tau.ub \) is updated after each value for \( V \) has been attempted (Line 15). Thus, in the search below the current node, it is never effective to instantiate the variables of \( \tau.Vars \) to values that will cause \( \tau \) to achieve a value greater than \( \tau.ub \), which may happen even though the global bound of \( \sum_{\kappa \in \mathcal{K}} \kappa.lb \leq ub \) still holds. The easiest way to enforce this local bound on the settings of \( \tau.Vars \) appears to be to post a constraint on the search below. The constraint prohibits the sum of the costs of the objectives in \( \tau \) from every exceeding \( \tau.ub \). Note that the strength of this constraint increases as tighter bounds on \( \tau.ub \) are obtained.

**Example 4.3.4** For example, suppose that \( \tau \) contains objectives \( o_1(A, B, C) \), \( o_2(A, E, F) \), and \( o_3(F, G) \) and that \( \tau.ub = 10 \). If **OR-Decomposition** branches on \( A = a \) with \( \Delta^a = 3 \) and then at some later point on \( F = f \) with \( \Delta^f = 3 \). At this point, **OR-Decomposition** will have broken \( \tau \) into three sub-components: \( \kappa_1 = \{o_1(A = a, B, C)\} \), \( \kappa_2 = \{o_2(A = a, E, F = f)\} \), and \( \kappa_3 = \{o_3(F = f, G)\} \), and accumulated an immediate cost of \( \Delta^a + \Delta^f = 6 \). Therefore, if \( \kappa_1.lb + \kappa_2.lb + \kappa_3.lb > (10 - 6) \), then **OR-Decomposition** can immediately backtrack to the deepest point to which a variable of \( \tau \) has been instantiated (in this case to undo the assignment \( F = f \)). In other words, under the assignments \( A = a \) and \( F = f \), \( \tau \) cannot achieve its optimal value—it is already exceeding a known upper-bound on its optimal value. More formally, in the sub-tree below, if \( S \) is the set of components that has been generated from \( \tau \), and \( A \) is the set of assignments that have been made to variables of \( \tau \), then necessarily, \( \sum_{\kappa \in S} \kappa.lb + \text{cost}(A, \tau) \leq \tau.ub \).
Other ways of implementing this local bound condition exist, but utilizing a hard constraint is a simple method. There are other potential ways in which the local bounds could be used, including, for example, utilizing a sophisticated propagation of the added constraint. In the current implementation, OR-Decomposition only checks this constraint and backtracks when it is violated, backtracking far enough so as to change an assignment to a variable of the component associated with the constraint.

4.4 Formal Results

Theorem 4.4.1 Assume that on entry to OR-Decomposition \( \forall \kappa \in \mathcal{K} \), it holds that \((\kappa.\text{lb} \leq \text{mincost}(\kappa) \leq \kappa.\text{ub})\). If \( UB > \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa) \), then OR-Decomposition will return with all components of \( \mathcal{K} \) solved, i.e., \( \forall \kappa \in \mathcal{K} : (\kappa.\text{lb} = \text{mincost}(\kappa) = \kappa.\text{ub}) \). If, on the other hand \( UB \leq \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa) \), then OR-Decomposition computes valid lower-bounds and upper-bounds for every \( \kappa \in \mathcal{K} \), and will return with \( UB \leq \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb} \).

The proof uses the following two Lemmas.

Lemma 4.4.2 Assume that on entry to OR-Decomposition, \( \kappa.\text{lb} \leq \text{mincost}(\kappa) \leq \kappa.\text{ub} \) for all \( \kappa \in \mathcal{K} \). If OR-Decomposition immediately returns after Line 3, Theorem 4.4.1 is satisfied.

Proof: If OR-Decomposition immediately returns after Line 3, then either \( \sum_{\kappa \in \mathcal{X}} \kappa.\text{lb} = \sum_{\kappa \in \mathcal{X}} \kappa.\text{ub} \) or \( \sum_{\kappa \in \mathcal{X}} \kappa.\text{lb} \geq UB \). If \( \sum_{\kappa \in \mathcal{X}} \kappa.\text{lb} \geq UB \), then by the precondition, \((\forall \kappa \in \mathcal{K} : (\kappa.\text{lb} \leq \text{mincost}(\kappa) \leq \kappa.\text{ub}))\), it holds that \( \sum_{\kappa \in \mathcal{X}} \text{mincost}(\kappa) \geq \sum_{\kappa \in \mathcal{X}} \kappa.\text{lb} \geq UB \). If \( \sum_{\kappa \in \mathcal{X}} \kappa.\text{lb} = \sum_{\kappa \in \mathcal{X}} \kappa.\text{ub} \), then by the precondition \((\forall \kappa \in \mathcal{K} : (\kappa.\text{lb} \leq \text{mincost}(\kappa) \leq \kappa.\text{ub}))\), it holds that \( \forall \kappa \in \mathcal{K} : (\kappa.\text{lb} = \text{mincost}(\kappa) = \kappa.\text{ub}) \). So we see that if all components are not solved, i.e., \( \sum_{\kappa \in \mathcal{X}} \kappa.\text{lb} \neq \sum_{\kappa \in \mathcal{X}} \kappa.\text{ub} \), and OR-Decomposition returns immediately after Line 3, then it must be the case that \( UB \leq \sum_{\kappa \in \mathcal{X}} \text{mincost}(\kappa) \) and it returns with \( UB \leq \sum_{\kappa \in \mathcal{X}} \kappa.\text{lb} \). By assumption
valid bounds were passed to OR-Decomposition, and hence these (unchanged) bounds remain valid on return.

For a set of components, \( \mathcal{K} \), let \( |\mathcal{K}| \) denote the number of variables in the components of \( \mathcal{K} \), i.e., \( |\mathcal{K}| = |\bigcup_{\kappa \in \mathcal{K}} \kappa.\text{Vars}| \).

**Lemma 4.4.3** Assume that Theorem 4.4.1 holds when \( |\mathcal{K}| < n \). If on entry to OR-Decomposition with \( |\mathcal{K}| = n \) it holds that \( \kappa.\text{lb} \leq \text{mincost}(\kappa) \leq \kappa.\text{ub} \) for all \( \kappa \in \mathcal{K} \), then this condition still holds when OR-Decomposition returns.

**Proof:** First, note that the assignment that led to \( \text{mincost}(\tau) \) must include one of \( V = d, d \in \text{Dom}[V] \). Thus, \( \text{mincost}(\tau) = \min_{d \in \text{Dom}[V]}(\sum_{\kappa^d \in \mathcal{K}^d} \text{mincost}(\kappa^d) + \Delta^d) \).

By the assumption that Theorem 4.4.1 holds when \( |\mathcal{K}| < n \), the recursive call made on Line 13 (which solves a set of components with \( n-1 \) variables) returns with \( \forall \kappa' \in \mathcal{K}':(\kappa'.\text{lb} \leq \text{mincost}(\kappa') \leq \kappa'.\text{ub}) \). Since \( \mathcal{K}^d \subseteq \mathcal{K}' \), it holds that \( \forall \kappa^d \in \mathcal{K}^d:(\kappa^d.\text{lb} \leq \text{mincost}(\kappa^d) \leq \kappa^d.\text{ub}) \). Since \( lb^d = \sum_{\kappa^d \in \mathcal{K}^d} (\kappa^d.\text{lb}) \) and \( ub^d = \sum_{\kappa^d \in \mathcal{K}^d} (\kappa^d.\text{ub}) \), it holds that \( lb^d \leq \sum_{\kappa^d \in \mathcal{K}^d} \text{mincost}(\kappa^d) \leq ub^d \). Therefore, \( \min_{d \in \text{Dom}[V]}(lb^d + \Delta_d) \leq \min_{d \in \text{Dom}[V]}(\sum_{\kappa^d \in \mathcal{K}^d} \text{mincost}(\kappa^d) + \Delta_d) \) \( \leq \min_{d \in \text{Dom}[V]}(ub^d + \Delta_d) \). Thus \( \tau.\text{lb} \leq \text{mincost}(\tau) \leq \tau.\text{ub} \).

Now we turn to the proof of Theorem 4.4.1. By Lemma 4.4.2, if OR-Decomposition immediately returns after Line 3, then Theorem 4.4.1 is satisfied. Thus, our proof assumes that OR-Decomposition does not immediately return after Line 3.

The proof of Theorem 4.4.1 is by induction on \( |\mathcal{K}| \).

**Base Case:** If \( |\mathcal{K}| = 1 \) then OR-Decomposition selects the remaining unassigned variable, \( V \), and assigns \( V \) each value, \( d \in \text{Dom}[V] \). For each \( d \in \text{Dom}[V] \), \( \Delta^d \) is the exact cost of every objective in \( \tau \). Since \( |\mathcal{K}'| = 0 \), \( \tau.\text{lb} \) and \( \tau.\text{ub} \) will be set to minimum \( \Delta^d \).
calculated. Thus, it holds that $\forall \kappa \in \mathcal{K}: (\kappa.\text{lb} = \text{mincost}(\kappa) = \kappa.\text{ub})$.

Inductive Hypothesis: Assume Theorem 4.4.1 holds for $\mathcal{K}$ where $|\mathcal{K}| < n$.

Inductive Step: Prove that Theorem 4.4.1 holds for $\mathcal{K}$ where $|\mathcal{K}| = n$.

We prove the inductive step by contradiction. By Lemma 4.4.3, $\forall \kappa \in \mathcal{K}: (\kappa.\text{lb} \leq \text{mincost}(\kappa) \leq \kappa.\text{ub})$. Thus, the only way for Theorem 4.4.1 to not hold is if one of the following two cases holds:

Case 1: $\text{ub} \leq \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa)$ and $\text{ub} > \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb}$.

Let $lb^x + \Delta^x = \min_{d \in \text{Dom}[V]}(lb^d + \Delta^d) = \tau.\text{lb}$. When the recursive call is made for $V = x$ (Line 13), the following inequality holds:

$$\text{UB} \leq \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa)$$
$$= \sum_{\kappa \in \mathcal{K}, \kappa \neq \tau} \text{mincost}(\kappa) + \text{mincost}(\tau)$$
$$\leq \sum_{\kappa \in \mathcal{K}, \kappa \neq \tau} \text{mincost}(\kappa) + \sum_{\kappa^x \in \mathcal{K}^x} \text{mincost}(\kappa^x) + \Delta^x$$
$$= \sum_{\kappa^x \in \mathcal{K}^x} \text{mincost}(\kappa^x) + \Delta^x$$

Therefore, $\text{UB} - \Delta^x \leq \sum_{\kappa^x \in \mathcal{K}^x} \text{mincost}(\kappa^x)$. By the inductive hypothesis, OR-Decomposition will return with $\text{UB} - \Delta^x \leq \sum_{\kappa^x \in \mathcal{K}^x} \kappa^x.\text{lb}$; thus, $\text{UB} \leq \sum_{\kappa^x \in \mathcal{K}^x} \kappa^x.\text{lb} + \Delta^x = \sum_{\kappa^x \in \mathcal{K}^x} \kappa^x.\text{lb} + \sum_{\kappa^x \in \mathcal{K}^x} \Delta^x = \sum_{\kappa^x \in \mathcal{K}^x} \kappa^x.\text{lb} + \tau.\text{lb}$. Since $\mathcal{K}' - \mathcal{K}^x = \mathcal{K} - \tau$, it holds that $\text{UB} \leq \sum_{\kappa \in \mathcal{K}, \kappa \neq \tau} \kappa.\text{lb} + \tau.\text{lb} = \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb}$; a contradiction.

Case 2: $\text{ub} > \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa)$, and not all components are solved in $\mathcal{K}$ are solved to optimality. In other words, $\exists \kappa \in \mathcal{K}$ for which $\kappa.\text{lb} < \text{mincost}(\kappa)$ or $\text{mincost}(\kappa) < \kappa.\text{ub}$.

Let $lb^a + \Delta^a = \min_{d \in \text{Dom}[V]}(lb^d + \Delta^d) = \tau.\text{lb}$. When the recursive call is made for $V = a$, the recursive call will return with $lb^a < \text{mincost}(\tau)$ only if $\text{ub} - \Delta^a \leq$
\[ \sum_{\kappa' \in \mathcal{K}} \text{mincost}(\kappa'), \] which implies that \( \text{UB} \leq \sum_{\kappa' \in \mathcal{K}^x} \text{mincost}(\kappa') + \Delta^x \). However, since \( \mathcal{K}' = \mathcal{K} + \mathcal{K}^a - \tau = \mathcal{K} + \mathcal{K}^a - (\mathcal{K}^a + \Delta^a) \), this contradicts the assumption that \( \text{UB} > \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa) \). Thus, \( \kappa.\text{lb} = \text{mincost}(\kappa) \).

Let \( \sum_{\kappa' \in \mathcal{K}^x} \text{mincost}(\kappa^x) + \Delta^x = \text{mincost}(\tau) \) (there is at least one value in the domain of \( V \) for which this is true). When the recursive call is made for \( V = x \) (Line 13), the following inequality holds:

\[
\text{UB} > \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa) \\
= \sum_{\kappa \in \mathcal{K} \setminus \tau} \text{mincost}(\kappa) + \text{mincost}(\tau) \\
= \sum_{\kappa \in \mathcal{K} \setminus \tau} \text{mincost}(\kappa) + \sum_{\kappa^x \in \mathcal{K}^x} \text{mincost}(\kappa^x) + \Delta^x \\
= \sum_{\kappa' \in \mathcal{K}'} \text{mincost}(\kappa') + \Delta^x
\]

Therefore, \( \text{UB} - \Delta^x > \sum_{\kappa' \in \mathcal{K}'} \text{mincost}(\kappa') \). By the inductive hypothesis, \textbf{OR-Decomposition} will return with \( \forall \kappa' \in \mathcal{K}' : (\kappa'.\text{lb} = \text{mincost}(\kappa') = \kappa'.\text{ub}) \).

Since \( \mathcal{K}' \supseteq \mathcal{K}^x \), \( \forall \kappa^x \in \mathcal{K}^x : (\kappa^x.\text{lb} = \text{mincost}(\kappa^x) = \kappa^x.\text{ub}) \). Thus \( \text{lb}^x = \sum_{\kappa^x \in \mathcal{K}} \text{mincost}(\kappa^x) = \text{ub}^x \). Since \( \tau.\text{ub} = \min_{d \in \text{Dom} \{ \mathcal{V} \}} (\text{ub}^d + \Delta^d) \), it holds that \( \tau.\text{ub} \leq \text{ub}^x + \Delta^x = \text{lb}^x + \Delta^x = \text{mincost}(\tau) \). However, by Lemma 4.4.3, \( \tau.\text{lb} \leq \tau.\text{ub} \); thus, \( \text{mincost}(\tau) = \tau.\text{ub} \).

As stated above, when the assignment \( V = x \) is solved, \( \forall \kappa' \in \mathcal{K}' : (\kappa'.\text{lb} = \text{mincost}(\kappa') = \kappa'.\text{ub}) \). Since \( \mathcal{K} \supseteq \mathcal{K}' \cup \tau \), it holds that \( \forall \kappa \in \mathcal{K} : (\kappa.\text{lb} = \text{mincost}(\kappa) = \kappa.\text{ub}) \); a contradiction.

\[ \square \]

The space requirements of \textbf{OR-Decomposition} are also worth investigating. After the value, \( V = d \), has been tried, all newly generated components, \( \mathcal{K}^d \), can be discarded. If the input WCSP, \( \mathcal{P} \), has \( n \) variables, then there can be at most \( n \) components in \( \mathcal{K} \) (each component must contain at least one variable), and \textbf{OR-Decomposition} can descend a path of at most length \( n \). Thus, at most \( O(n^2) \) space is needed to store the
active components during the algorithm’s operation, above and beyond the space initially needed to represent the input problem, $\mathcal{P}$.

The algorithm’s performance can be considerably enhanced by remembering previously encountered components in a cache. Thus, after new bounds on $\tau$ have been computed, at Line 17, these bounds (perhaps exact) can be stored in the cache and reused whenever $\tau$ is encountered again in the search. Caching is an important part of the implementation, and the template techniques described in Chapter 3 are used to make caching more efficient.

Of course with caching we no longer have an $O(n^2)$ bound on OR-Decomposition’s space requirements. Nevertheless, the results of [8] demonstrate that there exists a variable-ordering under which OR-Decomposition’s space requirements will be bounded by $n^{O(1)}2^{O(w)}$ even if all components encountered during search are cached. In practice, however, even this bound can be impractical. Nevertheless, an important feature of caching (both with OR-Decomposition and AND/OR) is that it serves only to improve the algorithms’ performance. Items can be pruned from the cache to save space without affecting the algorithm’s correctness.

\section{4.5 Local Propagation}

An important technique when solving WCSPs is local propagation, or soft-arc consistency, described in detail in Section 2.6.1 (see page 21). The key intuition behind soft-arc consistency is repeated here. Soft-arc consistency works by “sweeping” values from the sub-objectives to a zero-arity sub-objective, $O_\emptyset$. Two sweeping transformations are employed. First, values can be swept between a unary objective $o_1(V)$ and any binary objective involving $V$, e.g., $o_2(V,X)$. Second, values can be swept from a unary objective $o_1(V)$ into the zero-ary objective, $O_\emptyset$. These two types of transformations are equivalence-preserving in the sense that the cost of every complete assignment is pre-
To add local propagation to **OR-Decomposition**, $\Delta^d$ (Line 8) is set to be the total value swept to $O_\emptyset$ as a result of applying local propagation, after the assignment $V = d$ is made. In addition, to accommodate local bounding, the constraints added at Line 5 are enforced by ensuring that in the search below, the sum of the lower-bounds of all of the components generated from $\tau$, plus the total value swept to $O_\emptyset$ from variables of $\tau$, always remains $\leq \tau.ub$. This requires instrumenting the propagation mechanism to keep track of the values swept to $O_\emptyset$ from different variables of the problem.

**Caching:** Local propagation can also interfere with caching. Caching stores the bounds computed for components, and then re-uses these bounds if the components reappear during search. When the same component appears multiple times during search, local propagation might have moved a different amount of cost into or out of the component. This can invalidate the cached bounds.

To exploit caching in the presence of local propagation, the bounds must be adjusted to be independent of the current propagation before they are stored in the cache. These bounds also must be adjusted to account for the current propagation when they are retrieved from the cache. Recently, such a technique was developed when a fixed tree-decomposition is used to guide the search [29]. With a fixed tree-decomposition, the components that will arise during search can be predicted in advance.

In **OR-Decomposition**, however, the order in which the variables are instantiated is unconstrained, that is, a fixed tree-decomposition is not used. Rather, components are detected dynamically whenever they are created by the instantiated variables. Nevertheless, this chapter generalizes the techniques of [29], in order to compute the cost that has flowed into and out of the components as they are generated during search.

For every value $d \in V_i \in \mathcal{P}.V$, a backtrackable data structure, $\Delta^d_i$, is used to store the cost modifications for the value. Each structure, $\Delta^d_i$, is an array of size $|\mathcal{P}.V|$. Element $\Delta^d_i[j]$ represents the flow between the value $V_i[d]$ and the variable $V_j$. When a binary
transformation of size $\alpha$ is made between variable $o_{ij}$ and $o_i(d)$, the cost of $\alpha$ is added to $\Delta^d[j]$ ($\alpha$ will be negative if value is flowed out of $o_i(d)$ into $o_{ij}$).

Suppose a component, $\langle \kappa.\text{Vars}, \kappa.\text{Obj}, \kappa.A \rangle$, is encountered during search. Let $A$ be the set of assignments to the variables of $\kappa.A$. The total flow affecting the component’s value is $\sum_{V_j \in \kappa.A} \sum_{V_i \in \kappa.\text{Vars}} (\Delta^d_j[V_j = d] \in A)$. Intuitively, the sum is the flow that has been made from every variable, $V_i \in \kappa.\text{Vars}$, into or out of every variable $V_j = d$ assigned in $\kappa.A$. Note that a flow between the unassigned variables of a component will have no effect on the optimal value of the component, since the transformations create equivalent problems. Similarly, flows between the component’s variables and other variables not assigned in $\kappa.A$ have no effect on the value of the component. This is because any such flow must have traversed through some variable of $\kappa.A$ and thus will be accounted for in the above expression. In particular, the variables of $\kappa.A$ separate the variables of $\kappa$ from all other variables of the problem.

Using these flows, the cached bounds can be adjusted so that the bounds are made independent of the context when they are to be stored in the cache and made compatible with the current context when retrieved from the cache.

### 4.6 Multiple Partial Variable-Ordering

We develop a new variable-ordering heuristic called Multiple Partial Variable-Ordering (MPVO) to exploit OR-Decomposition’s ability to interleave the processing of the active components.

Active labels, semantic variable-ordering, graph-based variable-ordering, and the PVO heuristic were introduced in Section 2.9 (see page 37). Given a tree-decomposition and an active label, PVO uses a semantic variable-ordering heuristic to select a variable from the active label. When all variables from the label are assigned, a single child label is made active based on the component being solved [77].
MPVO is a new variable-ordering heuristic based on PVO, in which multiple labels can be active. MPVO uses a semantic variable-ordering heuristic to select from any variable in any active label. At the beginning of search, only the top label of the tree-decomposition is active. When every variable of a label, $l$, is assigned a value, every label that is a child of $l$ is made active. For example, suppose the WCSP with the constraint graph represented in Figure 4.4 is solved using OR-Decomposition, with the tree-decomposition seen in Figure 4.5. Initially, the label containing the variables $D$ and $F$ is active, and MPVO uses a semantic variable-ordering heuristic to select an unassigned variable. Once $D$ and $F$ are both assigned a value, the label containing $C, E$ and the label containing $G$ are made active. A semantic variable-ordering heuristic is then used to choose between the variables $C, E$, and $G$.

AND/OR is not compatible with MPVO since AND/OR must commit to solving a single component. For example, suppose a WCSP with the constraint graph represented in Figure 4.4 is solved using AND/OR with the tree-decomposition seen in Figure 4.5. After assigning variables $D$ and $F$, AND/OR must commit to solving either the component $\kappa_1$ containing $C, E$, or the component $\kappa_2$, containing $G$. If, for example, AND/OR chooses to solve $\kappa_1$ first, it will explore a sub-tree in which it can branch only on the variables of $\kappa_1$: $A, B, C,$ and $E$. After solving $\kappa_1$, it will turn to $\kappa_2$, exploring a sub-tree that branches only the variables of $\kappa_2$: $G$ and $H$. Thus, AND/OR must branch on all of the variables $\{A, B, C, E\}$ before branching on any variable $\{G, H\}$ or vice versa. OR-Decomposition, on the other hand, is free to branch on $B$, then $G$, then $A$, etc. That is, by exploiting its greater freedom in variable-ordering, OR-Decomposition can interleave the solving of $\kappa_1$ and $\kappa_2$.

### 4.7 Experimental Results
In this experimental section, B&B, AND/OR, and OR-Decomposition are implemented using local propagation. The algorithms are tested on both Weighted Constraint Satisfaction Problems (WCSP) and Most Probable Explanation (MPE) problems from Bayesian networks.

Before discussing the algorithms tested, it is necessary to note certain features of OR-Decomposition. OR-Decomposition can interleave the branching of variables from different active components at each recursion. OR-Decomposition using the MPVO variable-ordering is free to select a variable from any active label. In order to isolate the effect of interleaving, the experimental section in this chapter modifies the MPVO heuristic in two ways. The first modification is MPVO+Alternate. Suppose at node $n$, a variable, $V$, is selected from an active label, $l$. At any child node of $n$, MPVO+Alternate does not allow a variable from label $l$ to be selected (if there is an active label $m \neq l$). In other words, MPVO+Alternate forces components to be solved
by interleaving whenever possible. The second modification is \textbf{MPVO+Same}. At any child node of \( n \), \textbf{MPVO+Same} forces the selection of a variable from the label \( l \) to be selected (if there is an unassigned variable in label \( l \)). In other words, \textbf{MPVO+Same} forbids interleaving, thus making \textbf{OR-Decomposition} more like standard \textbf{AND/OR}.

The following specific algorithms are tested:

1. \textbf{BB} (Algorithm 1 on page 20), with FDAC local propagation implemented in the solver Toolbar [16];

2. \textbf{AND/OR} (Algorithm 2 on page 28), with caching and FDAC local propagation using PVO (a variable-ordering that follows a tree-decomposition);

3. \textbf{OR-Decomp}, (Algorithm 4), with caching and FDAC local propagation using MPVO;

4. \textbf{OR-Decomp+Alternate} (Algorithm 4), with caching and FDAC local propagation using \textbf{MPVO+Alternate};

5. \textbf{OR-Decomp+Same} (Algorithm 4), with caching and FDAC local propagation using \textbf{MPVO+Same}.

The semantic heuristic used for all algorithms is an adaptation of the Jersolow heuristic that has previously been used for solving WCSPs [16]. Intuitively, this heuristic considers both the variable’s domain size and the average cost of the objective functions in which the variable appears.

All algorithms utilize a value ordering determined by the unary objectives, \( o_i(V_i) \), used during local propagation; that is, the values for variable \( V_i \) are ordered by lowest unary cost, \( o_i(V_i) \). Tree-decompositions are computed using a min-fill algorithm (for those algorithms that utilized a tree-decomposition). \textbf{AND/OR} orders its components so as to solve the largest component first. \textbf{OR-Decomposition} does not need to order components since it is able to solve its components concurrently.
All experiments were run with 1200 second timeouts on 2.66GHz machines with 8GB of memory. If a problem was not solved to optimality in 1200 seconds, the term DNF (Did Not Finished) is used. In the experiments conducted, the space used in caching never exceeded available memory, so it was not necessary to prune the cache during search.

### 4.7.1 Problem Sets

The following benchmarks are tested in this experimental section:

**Star-Graphs** are randomly generated, binary MAX-CSP problems built from a set of cliques of variables.\(^1\) The domain size of every variable is 10. For every objective \(o_{ij}\), seven randomly selected tuples have cost zero, while the remaining tuples have cost one. The MAX-CSP problems are defined by the tuple \(\langle k, c, p \rangle\), where \(k\) is the number of cliques \((K_1, \ldots K_k)\), \(c\) is the number of variables in the center clique \((K_1)\), and \(p\) is the number of variables in cliques \(K_2, \ldots K_k\). Every variable in \(K_1\) is connected to every other variable in the problem. For example, the Star Graph \(\langle 5, 2, 3 \rangle\) is shown in Figure 4.6. By increasing the number of cliques, \(k\), the number of decompositions in the problem instance is increased. By increasing the number of variables, \(p\), the complexity of each decomposable component in the problem instance is increased. Thus, it is possible to create problems with many non-trivial decompositions. In our experiments we keep the number of variables in the center clique, \(c\), constant.

The **Radio Link Frequency Assignment Problem** (RLFAP) assigns frequencies to a set of radio links in such a way that all the links may operate together without noticeable interference. The RLFAP instances are cast as binary WCSPs [18]. The benchmark family includes 46 problems.

The **Earth Observing Satellite** (SPOT5) problems select from a set of candidate

\(^1\)Star-Graphs are a family of problems that have never previously been used. Star-Graphs are specifically designed to test complex decomposition problems.
photographs, a subset of photographs such that some imperative constraints are satisfied while the total importance of the selected photographs is maximized. The problems have been formulated as WCSPs with binary and ternary constraints in the SPOT5 benchmark [13]. The benchmark family consists of 22 problems.

The GridNetworks (Grid) problems involve computing the setting of the variables in a Bayesian network that has maximum probability (an MPE problem). The Bayesian network is a $N \times N$ grid containing conditional probability tables (CPTs) filled with values that are either chosen uniformly at randomly from the interval (0,1), or are randomly assigned 0 or 1 with equal probability. The problem instances tested involve values of $N$ ranging between 10 and 38, where 90% of the CPT’s entries are 0 or 1 [102]. The benchmark family consists of 13 problems.

The UAI instances are a benchmark of circuits used in formal verification and diagnosis. The problem set has been converted into n-ary WCSPs [17]. The benchmark family consists of 13 problems.
Figure 4.7: Time in seconds: $< 3, 5, 13 >$ Star-Graph. A point $(X,Y)$ represents the time taken to solve by AND/OR (X-axis) versus the time taken to solve by OR-Decomp (Y-axis)

### 4.7.2 Star-Graph Results

Figure 4.7 plots the comparative time needed to solve 100 randomly generated $< 3, 5, 13 >$ Star-Graph instances by the AND/OR and OR-Decomp algorithms. A point, $(X,Y)$, represents the time taken to solve a problem by AND/OR (X-axis) versus the time taken to solve the same problem by OR-Decomp (Y-axis). Figure 4.7 shows that every instance is solved faster by OR-Decomp.

Table 4.1 summarizes the number of instances solved, total time in seconds to solve, and total nodes expanded for different sets of Star-Graphs using AND/OR, OR-Decomp, and BB. Each set of Star-Graphs is comprised of 100 instances. When an instance is not solved, a time of 1200 seconds is added to the total time. OR-Decomp is proven to be faster than both AND/OR and BB across all sets of instances.

Table 4.2 summarizes the number of instances solved, total time in seconds to solve, and total nodes expanded for different sets of Star-Graphs using OR-Decomp+Same
## Table 4.1: Star-Graphs: Number of Problems Solved, Time in Seconds to solve all instances, Nodes Expanded to solve all instances. Each set contains 100 instances

<table>
<thead>
<tr>
<th>Star-Graph</th>
<th>BB</th>
<th>AND/OR</th>
<th>OR-Decomp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#s</td>
<td>time</td>
<td>nodes</td>
</tr>
<tr>
<td>&lt;3,5,13&gt;</td>
<td>100</td>
<td>14948</td>
<td>2.4*10^8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;4,5,13&gt;</td>
<td>100</td>
<td>92963</td>
<td>1.4*10^9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;5,5,13&gt;</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;6,5,13&gt;</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;3,5,14&gt;</td>
<td>99</td>
<td>98898</td>
<td>1.5*10^10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;4,5,14&gt;</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;5,5,14&gt;</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Table 4.2: Star-Graphs: Number of Problems Solved, Time in Seconds to solve all instances, Nodes Expanded to solve all instances. Each set contains 100 instances

<table>
<thead>
<tr>
<th>Star-Graph</th>
<th>OR-Decomp+Same</th>
<th>OR-Decomp+Alternate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#s</td>
<td>time</td>
</tr>
<tr>
<td>&lt;3,5,13&gt;</td>
<td>100</td>
<td>6530</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;4,5,13&gt;</td>
<td>100</td>
<td>23624</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;5,5,13&gt;</td>
<td>100</td>
<td>45822</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;6,5,13&gt;</td>
<td>100</td>
<td>70664</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;3,5,14&gt;</td>
<td>100</td>
<td>25500</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;4,5,14&gt;</td>
<td>97</td>
<td>88510</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;5,5,14&gt;</td>
<td>17</td>
<td>111879</td>
</tr>
</tbody>
</table>
Table 4.3: Number of Problems Solved (1200 second timeout)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>RLFAP (46)</th>
<th>Spot5 (22)</th>
<th>Grids (12)</th>
<th>UAI (13)</th>
<th>Total (94)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB</td>
<td>19</td>
<td>9</td>
<td>4</td>
<td>11</td>
<td>43</td>
</tr>
<tr>
<td>AND/OR</td>
<td>27</td>
<td>14</td>
<td>3</td>
<td>8</td>
<td>52</td>
</tr>
<tr>
<td>OR-Decomp</td>
<td>29</td>
<td>14</td>
<td>3</td>
<td>11</td>
<td>57</td>
</tr>
</tbody>
</table>

and OR-Decomp+Alternate. Both algorithms are able to solve all instances; however, OR-Decomp+Alternate solves the instances faster than OR-Decomp+Same, indicating that the ability of OR-Decomposition to interleave components increases its efficiency.

The time to solve $\langle k, c, p \rangle$ Star-Graph instances using OR-Decomp+Alternate in Table 4.2 is similar to the time needed to solve identical instances using OR-Decomp, as shown in Table 4.1.

The results for OR-Decomp+Alternate and OR-Decomp+Same are not reported for the remaining benchmarks, since similar effects can be seen on other instances.

### 4.7.3 Benchmark Summary

Table 4.3 summarizes the number of problems solved by the various algorithms on the remaining benchmarks. Of the 94 total problem instances, BB solves 43 problems across the four benchmarks, AND/OR solves 52 problems, and, OR-Decomp solves the most problems at 57.

### 4.7.4 RLFAP Results

Table 4.4 contains the results of the RLFAP benchmarks for BB, AND/OR and OR-Decomp. Let $n$ be the number of variables in the problem and let $w$ be the induced-width, $w$, as found by a min-fill heuristic used to compute a tree decomposition for the problem. Problems where $w > n/5$ are considered high tree-width problems, and
<table>
<thead>
<tr>
<th>Instance</th>
<th>n</th>
<th>w</th>
<th>time</th>
<th>nodes</th>
<th>time</th>
<th>nodes</th>
<th>time</th>
<th>nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELAR6-SUB0</td>
<td>16</td>
<td>7</td>
<td>0.16</td>
<td>2900</td>
<td>0.04</td>
<td>506</td>
<td>0.29</td>
<td>1873</td>
</tr>
<tr>
<td>CELAR6-SUB1-24</td>
<td>14</td>
<td>9</td>
<td>2.73</td>
<td>37700</td>
<td>13.64</td>
<td>13895</td>
<td>10.51</td>
<td>140342</td>
</tr>
<tr>
<td>CELAR6-SUB1</td>
<td>14</td>
<td>9</td>
<td>44.19</td>
<td>289323</td>
<td>DNF</td>
<td>DNF</td>
<td>85.81</td>
<td>629095</td>
</tr>
<tr>
<td>CELAR6-SUB2</td>
<td>16</td>
<td>10</td>
<td>16.77</td>
<td>55235</td>
<td>101.9</td>
<td>305792</td>
<td>48.97</td>
<td>77309</td>
</tr>
<tr>
<td>CELAR6-SUB3</td>
<td>18</td>
<td>10</td>
<td>234.5</td>
<td>911561</td>
<td>DNF</td>
<td>DNF</td>
<td>85.81</td>
<td>629095</td>
</tr>
<tr>
<td>CELAR6-SUB4-20</td>
<td>22</td>
<td>11</td>
<td>10.68</td>
<td>157295</td>
<td>DNF</td>
<td>DNF</td>
<td>65.52</td>
<td>431188</td>
</tr>
<tr>
<td>CELAR6-SUB4</td>
<td>22</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>631.49</td>
<td>1815209</td>
<td>48.97</td>
<td>77309</td>
</tr>
<tr>
<td>CELAR7-SUB0</td>
<td>16</td>
<td>9</td>
<td>0.07</td>
<td>1150</td>
<td>0.09</td>
<td>1384</td>
<td>1.32</td>
<td>29243</td>
</tr>
<tr>
<td>CELAR7-SUB1-20</td>
<td>14</td>
<td>9</td>
<td>0.14</td>
<td>3748</td>
<td>0.19</td>
<td>4479</td>
<td>1</td>
<td>11825</td>
</tr>
<tr>
<td>CELAR7-SUB1</td>
<td>14</td>
<td>9</td>
<td>5.21</td>
<td>21429</td>
<td>7.09</td>
<td>228278</td>
<td>17.04</td>
<td>115634</td>
</tr>
<tr>
<td>CELAR7-SUB2</td>
<td>16</td>
<td>10</td>
<td>43.61</td>
<td>139535</td>
<td>DNF</td>
<td>DNF</td>
<td>57.12</td>
<td>228049</td>
</tr>
<tr>
<td>CELAR7-SUB3</td>
<td>18</td>
<td>10</td>
<td>484.43</td>
<td>183247</td>
<td>DNF</td>
<td>DNF</td>
<td>630.22</td>
<td>1133279</td>
</tr>
<tr>
<td>CELAR7-SUB4-22</td>
<td>22</td>
<td>11</td>
<td>7.11</td>
<td>82089</td>
<td>27.45</td>
<td>231038</td>
<td>16.54</td>
<td>94552</td>
</tr>
<tr>
<td>graph07</td>
<td>141</td>
<td>32</td>
<td>29.39</td>
<td>206707</td>
<td>DNF</td>
<td>DNF</td>
<td>640.15</td>
<td>2539727</td>
</tr>
<tr>
<td>graph12</td>
<td>252</td>
<td>47</td>
<td>10.89</td>
<td>159601</td>
<td>3.35</td>
<td>17523</td>
<td>8.29</td>
<td>41216</td>
</tr>
<tr>
<td>graph13reduc</td>
<td>454</td>
<td>142</td>
<td>222.63</td>
<td>51694</td>
<td>DNF</td>
<td>DNF</td>
<td>237.79</td>
<td>552865</td>
</tr>
<tr>
<td>scen06-16reduc</td>
<td>82</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>238.64</td>
<td>777249</td>
<td>198.96</td>
<td>355286</td>
</tr>
<tr>
<td>scen06-16</td>
<td>100</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>794.36</td>
<td>3421633</td>
<td>279.56</td>
<td>462795</td>
</tr>
<tr>
<td>scen06-18reduc</td>
<td>82</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>162.41</td>
<td>473064</td>
<td>146.22</td>
<td>336754</td>
</tr>
<tr>
<td>scen06-18</td>
<td>100</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>479.53</td>
<td>1851673</td>
<td>177.99</td>
<td>363176</td>
</tr>
<tr>
<td>scen06-20reduc</td>
<td>82</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>83.05</td>
<td>273535</td>
<td>102.1</td>
<td>279549</td>
</tr>
<tr>
<td>scen06-20</td>
<td>100</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>222.79</td>
<td>968318</td>
<td>114.61</td>
<td>265798</td>
</tr>
<tr>
<td>scen06-22reduc</td>
<td>81</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>58.47</td>
<td>233905</td>
<td>50.47</td>
<td>129872</td>
</tr>
<tr>
<td>scen06-22</td>
<td>99</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>155.88</td>
<td>717882</td>
<td>62.81</td>
<td>145928</td>
</tr>
<tr>
<td>scen06-24reduc</td>
<td>81</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>32.36</td>
<td>131029</td>
<td>14.26</td>
<td>47026</td>
</tr>
<tr>
<td>scen06-24</td>
<td>99</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>98.45</td>
<td>479126</td>
<td>34.17</td>
<td>121034</td>
</tr>
<tr>
<td>scen06-30reduc</td>
<td>81</td>
<td>10</td>
<td>210.63</td>
<td>1901130</td>
<td>0.34</td>
<td>2860</td>
<td>0.65</td>
<td>4392</td>
</tr>
<tr>
<td>scen06-30</td>
<td>99</td>
<td>10</td>
<td>DNF</td>
<td>DNF</td>
<td>3.75</td>
<td>27892</td>
<td>1.8</td>
<td>12647</td>
</tr>
<tr>
<td>scen09</td>
<td>200</td>
<td>7</td>
<td>7.23</td>
<td>173382</td>
<td>1.09</td>
<td>1685</td>
<td>0.21</td>
<td>1454</td>
</tr>
<tr>
<td>scen10</td>
<td>200</td>
<td>7</td>
<td>0.7</td>
<td>9589</td>
<td>0.19</td>
<td>720</td>
<td>0.11</td>
<td>761</td>
</tr>
</tbody>
</table>

Table 4.4: RLFAP Instances - Time in Seconds (1200 second timeout) and Nodes Expanded
problems where $w \leq n/5$ are considered low tree-width problems. The RLFAP problem set has both high tree-width and low tree-width instances. On problems with low tree-width decomposition offers a much greater potential speedup over B&B. On the scen instances, OR-Decomp and AND/OR outperform BB. These instances have low tree-width, where the decompositions found are non-trivial to solve. OR-Decomp solves 10 of the 12 scen instances in Table 4.4 faster than AND/OR due to the added flexibility gained by interleaving variable selection.

Table 4.5 summarizes the results of the RLFAP benchmarks for BB, AND/OR and OR-Decomp. On both high tree-width and low tree-width problems, OR-Decomp solves more instances, more quickly than AND/OR. However, on problems with high tree-width, decomposition techniques offer few theoretical or practical advantages over branch and bound search. Longer running times on some instances may be caused by the overhead required to maintain decomposition information, or the variable-ordering heuristic used by the decomposition search. Although BB performs poorly on low tree-width instances, BB solves the high tree-width problems faster than both AND/OR and OR-Decomp.

### 4.7.5 Spot5 Results

Table 4.6 shows the results of the three algorithms on 14 Spot5 instances. AND/OR and OR-Decomp both solve the same number of instances, whereas BB solves only 9 of the instances. AND/OR outperformed OR-Decomp on the majority of these problems. The results appear to show that variable-ordering freedom is not as important
Chapter 4. OR-Decomposition

<table>
<thead>
<tr>
<th>Instance</th>
<th>BB</th>
<th>AND/OR</th>
<th>OR-Decomp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
<td>w</td>
<td>time</td>
</tr>
<tr>
<td>1502</td>
<td>209</td>
<td>4</td>
<td>0.06</td>
</tr>
<tr>
<td>29</td>
<td>82</td>
<td>14</td>
<td>2.1</td>
</tr>
<tr>
<td>404</td>
<td>100</td>
<td>19</td>
<td>69.25</td>
</tr>
<tr>
<td>503</td>
<td>143</td>
<td>9</td>
<td>DNF</td>
</tr>
<tr>
<td>54</td>
<td>67</td>
<td>11</td>
<td>0.06</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>b29</td>
<td>82</td>
<td>14</td>
<td>2.1</td>
</tr>
<tr>
<td>b404</td>
<td>100</td>
<td>19</td>
<td>69.25</td>
</tr>
<tr>
<td>b408</td>
<td>200</td>
<td>33</td>
<td>DNF</td>
</tr>
<tr>
<td>b42</td>
<td>190</td>
<td>18</td>
<td>DNF</td>
</tr>
<tr>
<td>b503</td>
<td>143</td>
<td>8</td>
<td>DNF</td>
</tr>
<tr>
<td>b505</td>
<td>240</td>
<td>16</td>
<td>DNF</td>
</tr>
<tr>
<td>b54</td>
<td>67</td>
<td>9</td>
<td>0.06</td>
</tr>
<tr>
<td>b8</td>
<td>8</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.6: Spot5 Instances - Time in Seconds (1200 second timeout) and Nodes Expanded

on these instances. **AND/OR** is also more likely to solve components to optimality, which may enable **AND/OR** to re-use solutions found during search more frequently. In the Future Work section of this chapter, the prospect of combining **AND/OR** with **OR-Decomposition** is explored.

### 4.7.6 Grid Results

The Grid problems can be solved more quickly when the variable-ordering is flexible (see Table 4.7). The added flexibility of **OR-Decomp**, therefore, allows it to solve the instances faster than **AND/OR**. Although **OR-Decomp** does not impose any variable-ordering restrictions, the MPVO heuristic forces search to follow a tree-decomposition ordering. **BB** is not similarly restricted, thus, **BB** is able to solve one additional problem.
Chapter 4. OR-Decomposition

4.7.7 UAI Results

The time needed to solve UAI instances is reduced when decomposition is used (see Table 4.8). **OR-Decomp** solves the entire set of instances faster than any other algorithm. Both **OR-Decomp** and **AND/OR** could solve more instances than **BB**.

4.7.8 Experimental Conclusions

These experiments illustrate the way in which the increased freedom in variable-ordering afforded by **OR-Decomp** yields improved results over **AND/OR**. The Star-Graph re-
results demonstrate that the gains by **OR-Decomp** are the result of interleaving the solving of components. The remainder of the experimental results deal with problems coming from real-world problems. These tests show that **OR-Decomp** is able to solve more instances than its counterparts. Only on Spot5 instances does **AND/OR** yield faster results than **OR-Decomp**. The Spot5 results demonstrate that a combination of **OR-Decomp** and **AND/OR** might yield better results than using either algorithm in isolation.

### 4.8 Related Work

This chapter is related to previous work on decomposition in constraint programming [83, 71, 56, 73, 70, 72, 79], belief networks [27], CNF [7], counting [34, 48], and integer linear programs [74]. A comprehensive analysis of many of these advances can be found in [77]. Earlier algorithms, however, force search to commit to a single component when a problem is decomposed. **OR-Decomposition** improves on these algorithms by allowing search to work on multiple components simultaneously. Thus, **OR-Decomposition** gains many of the benefits of decomposition while allowing complete variable-ordering freedom.

### 4.9 Future Work

Although **OR-Decomposition** allows multiple components to be solved concurrently, sometimes there is an advantage to solving a single component to optimality. For example, when a problem decomposes into multiple components, **OR-Decomposition** may backtrack before solving any component to optimality, whereas **AND/OR** may solve some of the components to optimality before backtracking. When these same components are encountered later in search, **AND/OR** can dismiss the component if optimality has been proven.
Chapter 4. OR-Decomposition

It would be natural to try to combine OR-Decomposition with AND/OR. OR-Decomposition and AND/OR could be integrated by using a heuristic to estimate whether it is more effective to interleave the solving of components (as seen in OR-Decomposition), or to solve each component sequentially (as seen in AND/OR).

A second avenue for future research would be to integrate OR-Decomposition with Variable Elimination. Variable Elimination can be used to solve quickly small sub-problems near leaf nodes. Using Variable Elimination with AND/OR+BB has been successful [78], and this idea could be applied to enhance OR-Decomposition.

4.10 Conclusions

In this chapter we have experimented with a depth-first AND/OR algorithm. Dechter showed that an AND/OR search space can be explored in a breadth-first manner [76]. Breadth-first AND/OR does allow for interleaving of components, as the search can alternate between different nodes. However, this approach raises the inherent space complexity of the algorithm. As pointed out above, OR-Decomposition can be run without a cache (or with a fixed size cache). It can preform interleaving while only using $O(n^2)$ space. Breadth-first search requires much more space for its operation and there is no option for reducing its space requirements by turning off caching. We preformed a few experiments with breadth-first search and found, for example, that it was only able to solve 3 of the 46 RLFAP benchmarks before running out of memory, compared to 29 solved by OR-Decomposition.

Exploiting decomposition can significantly reduce the time required to solve WCSPs. Unfortunately, AND/OR solves only one active component at a time due to its recursive nature. In this chapter OR-Decomposition, a novel search method, designed to address this weakness was presented and tested. OR-Decomposition is able to exploit decomposition, while allowing complete freedom to branch on any unassigned variable of
any active component. Tested on a number of benchmarks, OR-Decomposition shows improvements over current competitive algorithms.
Chapter 5

High Tree-Width Decomposition

5.1 Introduction

Exploiting decomposition can reduce the worst-case time complexity of solving WCSPs from $2^{O(n)}$ to $n^{O(1)}2^{O(w)}$, where $n$ is the number of variables and $w$ is the tree-width of the constraint graph [27, 7] generated by the objectives (see Section 2.7.3 on page 27). Algorithms such as AND/OR [73], OR-Decomposition [60], and BTD [29] have all exploited successfully the theoretical benefits of decomposition in WCSPs to obtain significant performance improvements.

All of these algorithms exploit the fact that as variables are instantiated during B&B, the problem can decompose into independent components that can be solved separately. The tree-width of the problem’s constraint graph provides a bound on the number of variables that need to be instantiated before the problem decomposes. Hence, when the problem has low tree-width, independent components will frequently appear during search and can be exploited by decomposition algorithms. When a problem has high tree-width, however, a large number of variables must be instantiated before the problem decomposes. The theoretical advantages of decomposition erode on problems with high tree-width, and the practical advantages over ordinary B&B fade (for example, see the
RLFAP experiments presented in Section 4.5).

This chapter demonstrates how bounds can be computed from a selected subset of the problem’s objectives, and used by B&B to prune the search space. Unlike other techniques such as mini-buckets, these bounds are not computed prior to search; rather, they can be computed during B&B search with little extra overhead. In particular, decomposition over the selected subset of objectives can be utilized to efficiently compute these bounds, without interfering with the normal operation of the B&B search. Since only a subset of the objectives are selected, effective decompositions can be obtained over this subset during search even though the complete problem does not decompose due to its high tree-width.

This chapter presents a new algorithm that increases the quality of bound information during B&B by exploiting decompositions found in a subset of objectives of the original WCSP. A greedy algorithm for selecting an appropriate subset of the objectives is then offered, and the chapter concludes with empirical results demonstrating the potential of the approach.

5.2 Background

The techniques offered in this chapter are applicable to problems with high tree-width, which in this chapter will be defined to be problems with induced-width $\geq |\text{Vars}|/5$. The results offered in the experimental section illustrate that when the tree-width of a WCSP is near $|\text{Vars}|$, the techniques described in this chapter offer considerable advantages over B&B and other current decomposition techniques. However, as the tree-width decreases, the advantages erode compared to standard decomposition algorithms.

Branch and Bound (B&B) is a standard technique for solving WCSPs using backtracking search (See Algorithm 1 on page 20 for details). B&B builds up partial variable assignments in a depth-first manner while using bounding to prune the search space.
**Branch and Bound with Decomposition:** In Chapter 3, the AND/OR search algorithm that uses component templates for caching was presented (See Algorithm 3 on page 52 for details). When the reduced problem splits into components, AND/OR search solves each component in an independent recursive call. Component templates can cache the updated lower and upper bounds search has computed for each component so that these new bounds can be re-used later in the search. As mentioned in Chapter 3, AND/OR search does not explore a standard backtracking search tree like that explored by B&B. AND/OR search is a mixture of backtracking and divide and conquer: it searches in the space of partial variable assignments like backtracking, but exploits decomposition by solving each component in a separate computation like divide and conquer.

In Chapter 4, OR-Decomposition was introduced as an alternate algorithm for exploiting decomposition in WCSPs (See Algorithm 4 on page 63). OR-Decomposition exploits decomposition while searching a standard backtracking search tree, using the caching techniques of [7], rather than separate recursions, to obtain the computational advantages of decompositions. In other words, it explores a standard OR tree rather than an AND-OR tree.

### 5.3 Decomposition Bounding

If a problem has high tree-width, it will not split into disjoint components until search has descended far down the search tree. For example, if the problem has an objective, the scope of which includes all of the variables (e.g., a global constraint) then it will never split into disjoint components during search. Hence, AND/OR, BTD, and OR-Decomposition all have limited value on problems with high tree-width.

The key contribution of this chapter is to demonstrate ways in which decomposition can be exploited on problems with high tree-width in order to efficiently compute
useful bounds. The new algorithm, called **Decomposition Bounding** (DB), employs a standard Branch and Bound search while simultaneously using the techniques of **OR-Decomposition** to exploit decompositions. Unlike **OR-Decomposition**, however, the algorithm does not look for decompositions of the entire problem because on problems of high tree-width, these occur infrequently. Rather, DB selects a subset of the problem’s objectives and looks for decompositions over this subset. Decompositions of this subset are exploited to compute bounds from this subset of objectives. These bounds can then be used by the overall Branch and Bound search to more effectively prune its search space. DB employs the techniques used in **OR-Decomposition** to exploit decomposition in a standard backtracking search tree, thus, the bounding techniques in DB can be used without interfering with B&B.

The idea of computing bounds from a relaxation of the problem has been used before in techniques such as mini-buckets [31], and linear relaxation [111]. In Section 5.6, DB will be compared to these algorithms.

**DB** is illustrated in Algorithm 5. Some preprocessing is performed on the original problem, \( \mathcal{P} \), before invoking DB. First, \( \mathcal{P} \) is split into two sub-problems: let \( \mathcal{P}^D \) be the decomposable sub-problem, where \( \mathcal{P}^D.\text{Vars} \) and \( \mathcal{P}^D.\text{Dom} \) remain unchanged from the original problem, but \( \mathcal{P}^D.\text{Obj} \subseteq \mathcal{P}.\text{Obj} \) is a selected subset of the original objectives (the choice of this subset is discussed below). The second sub-problem, \( \mathcal{P}^C \), is the complement of \( \mathcal{P}^D \), where \( \mathcal{P}^C.\text{Vars} \) and \( \mathcal{P}^C.\text{Dom} \) remain unchanged from the original problem, and \( \mathcal{P}^C.\text{Obj} = \mathcal{P}.\text{Obj} - \mathcal{P}^D.\text{Obj} \), that is, the objectives not included in the decomposable sub-problem.

Second, \( \mathcal{P}^D \) is split into a set of components, \( \mathcal{K} \). This can be accomplished efficiently by Component Templates during search (See Section 3.3 on page 47 for details). In DB, the function \textit{toComponents} performs such a computation. For each component \( \tau \in \mathcal{K} \), a valid lower-bound is computed using the bounding function \textit{getBounds}(\( \tau \)). The lower-bound is stored in the field \( \tau.\text{lb} \).
$\textbf{DB} (A, \mathcal{P}_{\text{cur}}, \mathcal{K}, \mathcal{C})$

/* On entry, each $\kappa \in \mathcal{K}$ must have a valid lower-bound, $\kappa.\text{lb}$. The call returns with valid lower-bounds for each $\kappa \in \mathcal{K}$. In addition, if $\text{mincost}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P}) < \text{gub}$, then gub will be updated to be $\text{mincost}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P})$. */

begin

$B_O = \text{getBounds}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P})$

$B_D = \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb} + \text{getBounds}(\mathcal{C}) + \text{cost}(A, \mathcal{P})$

if ($B_O \geq \text{gub} \lor B_D \geq \text{gub}$) then

return

if (|$\mathcal{P}_{\text{cur}}.\text{Vars}$| = 0) then

$\text{gub} = \text{cost}(A, \mathcal{P})$; return

choose (a variable $V \in \mathcal{P}_{\text{cur}}.\text{Vars}$)

$\tau =$ the component in $\mathcal{K}$ such that $V \in \tau.\text{Vars}$

foreach $d \in \tau.\text{Dom}[V]$ do

$\Delta^d = \text{cost}(V = d, \tau)$

$\mathcal{K}^d =$ toComponents($\tau|_{V=d}$)

foreach $\kappa^d \in \mathcal{K}^d$ do

$\kappa^d.\text{lb} = \text{MAX}(\text{getBounds}(\kappa^d), \text{getCache}(\kappa^d))$

$\text{DB}(A \cup \{V=d\}, \mathcal{P}_{\text{cur}}|_{V=d}, \mathcal{K} - \tau \cup \mathcal{K}^d, \mathcal{C}|_{V=d})$

$\text{lb}^d = \sum_{\kappa^d \in \mathcal{K}^d} \kappa^d.\text{lb} + \Delta^d$

$\tau.\text{lb} = \text{MAX}(\tau.\text{lb}, \text{MIN}_{d \in \text{Dom}[V]} \text{lb}^d)$

setCache($\tau.\text{lb}$)

end

Algorithm 5: Decomposition Bounding
DB takes the following parameters as input: the current assignment $A$; the current problem $P_{\text{cur}}(P|A)$; the current decomposable sub-problem maintained as a set of components, $K$ (equivalent to $\text{toComponents}(P^D|A)$); and, the current complementary problem $C$ ($P^C|A$). Initially, DB is invoked with an empty set of assignments; the original problem, $P$; the decomposable sub-problem broken up into components, $K$; and the complement sub-problem $P^C$.

DB operates much like B&B: it returns if no solution exists with cost less than GUB (Line 5), and it updates GUB and returns if there are no remaining uninstantiated variables (Line 8). Otherwise, DB selects some unassigned variable and calls itself recursively on each possible assignment to that variable (Line 16). (Since the variables of $P^D$ are the same as $P$, there is always a component $\tau \in K$ containing the chosen variable.)

The difference between DB and B&B lies in the additional bounds test $B_D \geq \text{gub}$ (line 5), where $B_D = \text{cost}(A, P) + \sum_{\kappa \in K} \kappa.\text{lb} + \text{getBounds}(C)$. To understand this test, note that $A$ can be rejected if any lower-bound on $\text{cost}(A, P) + \text{mincost}(P^D|A) + \text{mincost}(P^C|A)$ is greater than or equal to the cost of the current best known complete assignment. A lower-bound on this sum is also a lower-bound on $\text{cost}(A, P) + \text{mincost}(P|A)$, the minimal cost that can be achieved for $P$ by any extension of $A$. In particular, $\text{mincost}(P^D|A) + \text{mincost}(P^C|A) \leq \text{mincost}(P|A)$ since a lower cost can be achieved by optimizing the objectives of $P^D|A$ and $P^C|A$ independently of one another. Clearly, $\text{getBounds}(C) = \text{getBounds}(P^C|A) \leq \text{mincost}(P^C|A)$, and since $K$ is $P^D|A$ broken into independent components, $\sum_{\kappa \in K} \kappa.\text{lb} \leq \text{mincost}(P^D|A)$.

Good bounds are computed and cached on the various components of $K$ that are encountered during search, which makes the bound, $B_D$ effective. This computation of good bounds “goes along for the ride” during the Branch and Bound search. Whenever the assignment $V = d$ is made by the search, the component $\tau \in K$ containing $V$ is reduced. The assignment generates an immediate cost, $\Delta^d$, for $\tau$ determined by the objectives of $\tau$ that it fully instantiates, and reduces $\tau$ into a set of components $K^d$. 
Thus, \( lb^d = \Delta^d + \sum_{k^d \in \mathcal{K}^d} k^d \cdot lb \) (line 17) represents a lower-bound on the value that \( \tau \) can achieve, given that \( V = d \). Since \( V \) must be assigned some value, the optimal value, \( \mincost(\tau) \), is lower-bounded by the minimum \( lb^d \) taken over all \( d \in \text{Dom}[V] \). Note that the lower-bound, \( \tau \cdot lb \), is also a valid lower-bound; thus, the tightest lower-bound on \( \tau \)'s value is the maximum of these two bounds (line 18).

Typically, \( lb^d \) will be tighter than \( \tau \cdot lb \), as \( lb^d \) is derived from more information: i.e., from trying the various values of the variable \( V \) and examining the components that \( \tau \) decomposes into under these assignments (however, in the case where \( \tau \cdot lb \) was retrieved from a cache lookup, it is possible for \( \tau \cdot lb \) to be the tighter bound). Furthermore, since the components into which \( \tau \) decomposes are passed to the recursive call (line 16), their lower-bounds may be refined further during the search of the sub-tree below, making the \( lb^d \) values even more informative. Other components of \( \mathcal{K} \) are also passed to the recursive call, and the lower-bounds of the components of \( \mathcal{K} \) can also be improved by the search below. Since the optimal solution to these components is independent of the value assigned to \( V \), these improved bounds can serve to prune the search required when testing the subsequent values of \( V \).

Finally DB also caches the computed component lower-bounds (line 19). When new components are generated, DB checks the cache to determine if it contains a tighter lower-bound for those components (line 15). Since the decomposable sub-problem, \( \mathcal{P}^D \), is constructed so that it has low tree-width (see the next section), it is often the case that the same component can be generated many times during search. Thus, caching improves DB's performance.

Consider DB running on the problem seen in Figure 5.1, where a) represents the primal graph of one of the RLFAP benchmarks. This WCSP, \( \mathcal{P} \), has only binary objectives, represented by the edges between nodes (variables). The second graph, b), is the primal graph of \( \mathcal{P}^D \).

Suppose the following assignments are made: \( V_8 \leftarrow a, V_7 \leftarrow x \). \( \mathcal{K}^d \) will include
components whose variables are \( \{V_0, V_1, V_{12}, V_{13}\} \), \( \{V_2, V_3, V_{10}, V_{11}\} \), \( \{V_4, V_5\} \) and \( \{V_5, V_6\} \).

Note that \( \mathcal{P} \) has not decomposed at this point in search. Suppose that \( V_0 \) is selected as the next variable to assign (Line 9 of Algorithm 5). After assigning \( V_0 \leftarrow a \), search descends into the sub-tree. From the exploration of this sub-tree, a tighter lower-bound on the component \( \{V_0, V_1, V_{12}, V_{13}\} \) can be found. This lower-bound is still valid even after search retracts from \( V_7 \leftarrow x \), and can be used to bound the problem for different assignments to \( V_8 \) and \( V_7 \).

The components found in \( \mathcal{D} \) are generally not components of the reduced original problem, \( \mathcal{P}_{\text{cur}} \), since \( \mathcal{D} \) contains only a subset of the objectives of \( \mathcal{P}_{\text{cur}} \). Consider using AND-OR search to exploit the decompositions of \( \mathcal{D} \) while solving \( \mathcal{P}_{\text{cur}} \). If AND-OR invokes a separate recursion for each component of \( \mathcal{D} \), it could not simultaneously be solving \( \mathcal{P}_{\text{cur}} \) as \( \mathcal{P}_{\text{cur}} \) cannot be separated in this way. Still it would be possible to invoke AND-OR on each component of \( \mathcal{D} \), ignoring \( \mathcal{P}_{\text{cur}} \). However, in this case AND-OR might require considerable extra work over and above the standard B&B search that is being used to solve \( \mathcal{P}_{\text{cur}} \). Decomposition bounding, on the other hand, by exploiting the caching ideas of OR-Decomposition, is piggy-backing on the standard B&B search being used to solve \( \mathcal{P}_{\text{cur}} \) to perform a decomposed computation of bounds on the components of \( \mathcal{D} \). It does not interfere with the basic operation of the B&B search, adding only a slight overhead at each node of that search.

Value-pruning is an important technique used by many bounding techniques. During
search, values can be pruned if it can be inferred that no extension of the current partial assignment containing this value can lead to a solution better than the current GUB. In DB, value prunings can be shared between the various sub-problems; for example, suppose DB branches on variable V, and from D it can be inferred that \( d \in \text{Dom}[V'], V' \neq V \) cannot be extended to a better solution. In this case, \( d \) is removed from \( D . \text{Dom}[V'] \), but it can also be removed from \( \mathcal{P}_{\text{cur}} . \text{Dom}[V'] \), which may allow \( \mathcal{P}_{\text{cur}} \) to make additional inferences and value-prunings. The one exception is that values in \( D \) cannot be pruned due to value-prunings inferred by \( \mathcal{P}_{\text{cur}} \) or \( \mathcal{C} \). This is because \( D \) must build valid lower-bounds based only on the objectives of \( D . \text{Obj} \) in order to cache and re-use these bounds. DB, therefore, tests all values of \( \tau . \text{Dom}[V] \) (that are not pruned by the objectives in \( D \)) rather than only the values in \( \mathcal{P}_{\text{cur}} . \text{Dom}[V] \) at Line 11.

Although DB can only decrease the number of nodes Branch and Bound must search, it requires extra work be performed at each node. First, it must do extra work to detect components (calls to toComponents) and to store and look-up the cache. However, the templating techniques described in Chapter 3 can be utilized to decrease this overhead, e.g., by allowing low-cost component-detection via watched variables and efficient caching.

This overhead can be further reduced by using weaker, cheaper, getBounds functions on these \( \mathcal{C} \) and \( D \). In particular, since the quality of bounds found for \( D \) is derived largely from information gathered during search, a cheaper getBounds function can be used on \( D \). For example, when soft local consistency is used to calculate bounds, DB can use the effective but costly FDAC [67] algorithm when calling getBounds(\( \mathcal{P}_{\text{cur}} \)), but use the much faster AC algorithm for getBounds on \( D \) and \( \mathcal{C} \) (See Section 2.6.1 on page 21 for details of these algorithms).
5.4 Formal Results

**Theorem 5.4.1** If on entry to DB it holds that \( \forall \kappa \in \mathcal{K} : (\kappa.\text{lb} \leq \text{mincost}(\kappa)) \), then DB returns with valid lower-bounds for each \( \kappa \in \mathcal{K} \). In addition, if \( \text{mincost}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P}) < \text{gub} \), then gub will be updated to be \( \text{mincost}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P}) \).

The proof uses the following Lemma:

**Lemma 5.4.2** Assume that \( \forall \kappa \in \mathcal{K} : (\kappa.\text{lb} \leq \text{mincost}(\kappa)) \) and \( \sum_{\kappa \in \mathcal{K}} \kappa.\text{lb} + \text{getBounds}(\mathcal{C}) + \text{cost}(A, \mathcal{P}) \geq \text{gub} \), then \( \text{mincost}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P}) \geq \text{gub} \).

**Proof:** The proof of Theorem 5.4.1 is by induction on the number of variables in \( \mathcal{P}_{\text{cur}}.\text{Vars} \).

Base Case, \( |\mathcal{P}_{\text{cur}}.\text{Vars}| = 0 \). First note that since there are no unassigned variables there can be no components in \( \mathcal{K} \), so all bounds are vacuously valid. Second, when \( |\mathcal{P}_{\text{cur}}.\text{Vars}| = 0 \), it holds that \( B_{\text{O}} = \text{getBounds}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P}) = \text{mincost}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P}) = \text{cost}(A, \mathcal{P}) \), as getBounds returns the exact answer (zero) when \( |\mathcal{P}_{\text{cur}}.\text{Vars}| = 0 \). Also by Lemma 5.4.2, if \( B_{D} \geq \text{gub} \) then \( B_{O} \geq \text{gub} \). Thus if DB returns via the test at Line 5, \( \text{mincost}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P}) \geq \text{gub} \) and GUB need not be updated. Otherwise we must have \( \text{mincost}(\mathcal{P}_{\text{cur}}) + \text{cost}(A, \mathcal{P}) < \text{gub} \) and GUB will be updated to this value at Line 8.

Inductive Hypothesis: Assume Theorem 5.4.1 holds for \( \mathcal{P}_{\text{cur}} \) where \( |\mathcal{P}_{\text{cur}}.\text{Vars}| < n \).

Inductive Step: Prove that Theorem 5.4.1 holds for \( \mathcal{P}_{\text{cur}} \) where \( |\mathcal{P}_{\text{cur}}.\text{Vars}| = n \).

If DB returns with \( B_{O} \geq \text{gub} \lor B_{D} \geq \text{gub} \), the valid bounds are unchanged and by
Lemma 5.4.2 the theorem does not require that \( gub \) be updated. Otherwise, \( DB \) tries each value \( d \in \text{Dom}[V] \). If \( \text{mincost}(P_{\text{cur}}) + \text{cost}(A, P) < gub \), let \( \alpha \) be an assignment for which \( \text{cost}(\alpha, P_{\text{cur}}) = \text{mincost}(P_{\text{cur}}) \), and let \( V = x \in \alpha \) (i.e., \( V = x \) is part of a minimum cost assignment for \( P_{\text{cur}} \)). When \( V = x \) is solved recursively, \( gub \) will be updated to be \( \text{mincost}(P_{\text{cur}}) + \text{cost}(A, P) \) by the Inductive Hypothesis.

The proof that the call returns with valid lower-bounds for each \( \kappa \in \mathcal{K} \) is as follows. The assignment that led to \( \text{mincost}(\tau) \) must include one of \( V = d, d \in \text{Dom}[V] \). Thus \( \text{mincost}(\tau) = \min_{d \in \text{Dom}[V]}(\sum_{\kappa^d \in \mathcal{X}^d} \text{mincost}(\kappa^d) + \Delta^d) \). By the inductive hypothesis, the recursive call made on Line 16 returns with \( \forall \kappa^d \in \mathcal{K}^d : (\kappa^d.\text{lb} \leq \text{mincost}(\kappa^d)) \). Since \( lb^d = \sum_{\kappa^d \in \mathcal{X}^d} (\kappa^d.\text{lb}) \), it holds that \( lb^d \leq \sum_{\kappa^d \in \mathcal{X}^d} \text{mincost}(\kappa^d) \). Therefore \( \tau.\text{lb} = \min_{d \in \text{Dom}[V]}(lb^d + \Delta^d) \leq \min_{d \in \text{Dom}[V]}(\sum_{\kappa^d \in \mathcal{X}^d} \text{mincost}(\kappa^d) + \Delta^d) = \text{mincost}(\tau) \). By the inductive hypothesis, all components \( \kappa \in \mathcal{K}, \kappa \neq \tau \) will return from any recursive calls with valid lower-bounds. Since \( \tau.\text{lb} \leq \text{mincost}(\tau), \forall \kappa \in \mathcal{K} \), it holds that \( (\kappa.\text{lb} \leq \text{mincost}(\kappa)) \).

\[ \square \]

5.5 Finding Decomposition Subsets

\( DB \) exploits decompositions found in a decomposable sub-problem, \( P^D \). Intuitively, \( P^D \) should have a low tree-width, allowing ready decomposition, but still should support the generation of effective bounds. To generate effective bounds, \( P^D \) should include objectives of high cost. Here, an algorithm is proposed which greedily selects objectives of high cost to be included in \( P^D.\text{Objs} \).

In order to accommodate hard constraints (tuples with cost \( \infty \)) in the proposed algorithm, an initial step maps each cost, \( \infty \), to a large but no infinite cost. For each objective, \( o \), in problem \( P \), let \( \text{max}(o) \) be the maximum cost that can be contributed by objective \( o \) (it is unlikely that any assignment will contribute \( \text{max}(o) \) from all the objectives). Formally, let \( T \) be the set of all possible assignments to the variables in
scope(o). Then \( \max(o) = (\max_{\alpha \in T} o(\alpha)) | o(\alpha) \neq \infty \). For any complete assignment to the problem that does not violate a hard constraint, the maximum cost contributed by objective \( o \) is \( \max(o) \). Thus, if a feasible solution to \( P \) exists, then \( \min\text{cost}(P) \leq (\sum_{o \in O_{obj}} \max(o)) \). For each tuple with cost \( \infty \), the \( \infty \) cost is replaced by the finite cost, \( (\sum_{o \in O_{obj}} \max(o)) + 1 \). Algorithm 6 requires that all costs are finite. Replacing \( \infty \) cost with \( (\sum_{o \in O_{obj}} \max(o)) + 1 \) assures that hard constraints maintain their importance relative to soft constraints, yet still have finite costs.

After modifying the problem to eliminate \( \infty \) costs, the cost of each objective is estimated by computing its expected cost. For an objective, \( o \), let \( T \) be the set of all possible assignments to the variables in \( \text{scope}(o) \). Then, \( E(o) = (\sum_{\alpha \in T} o(\alpha)) / |T| \) is the expected cost of \( o \); that is, the average cost added by \( o \) over all possible instantiations of its variables. The expected cost is calculated by summing the cost of all possible instantiations of its variables. If the size of \( \text{scope}(o) \) makes this computation intractable, \( E(o) \) can be estimated in other ways.\(^1\) For a set of objectives, \( O \), define \( E(O) = \sum_{o \in O} E(o) \).

\[\begin{align*}
1 & \text{Create Decomposable Problem } (P) \\
2 & \text{begin} \\
3 & \hspace{1em} \text{Objs} = P.\text{Objs} - \text{HighArityObjectives} \\
4 & \hspace{1em} \text{SortedObjs} = \text{sort}(\text{Objs}) \\
5 & \hspace{1em} \text{NumberOfObjs} = |P.\text{Vars}| * \log_2(|P.\text{Vars}|) * \lambda \\
6 & \hspace{1em} \text{for } (\text{objectives} = 0; \text{objectives} < \text{NumberOfObjs}) \text{ do} \\
7 & \hspace{2em} \text{PD.Objs} = \text{PD.Objs} \cup \text{SortedObjs}[\text{objectives}] \\
8 & \hspace{2em} \text{TreeDec} = \text{FindTreeDecomposition(} \text{PD.Objs} \text{)} \\
9 & \hspace{2em} \text{PD.Objs} = \text{PD.Objs} \cup \text{AddObjs(} \text{TreeDec, Objs} \text{)} \\
10 & \text{end} \\
\end{align*}\]

\(1\)Typically, objectives with large scope have some other structure that allows them to be represented compactly. That structure can be exploited to estimate the expected cost.
Given a problem with \( n = |\mathcal{P}.Vars| \) variables, after all objectives with high arity have been removed, Algorithm 6 selects a number of objectives (NumberObjs) to include in the decomposable sub-problem. The objectives are selected with the goal that the primal graph defined by these objectives is decomposable, yet is dense enough to give meaningful bounds. Experimentation shows that by setting NumberObj to be \( n \log_2(n) \times \lambda \), with \( \lambda \) equal to 0.35, the resulting constraint graphs for the problem sets have low tree-width, yet have sufficiently high expected cost to produce meaningful bounds.

Algorithm 6 first eliminates high arity objectives from \( \mathcal{P} \) (those objectives with arity greater than \( n \times \lambda \)), and then orders the remaining objectives from highest expected cost to lowest expected cost. The highest cost objectives are then added to \( \mathcal{P}^D.\text{Obj} \). Next, a tree-decomposition based on \( \mathcal{P}^D.\text{Obj} \) is computed (different options exist for computing reasonably good tree-decompositions). Finally, additional objectives are added to \( \mathcal{P}^D.\text{Obj} \), provided their inclusion does not violate the calculated tree-decomposition (e.g., all unary objectives are added to \( \mathcal{P}^D.\text{Obj} \)).

For example, consider Figure 5.2 a), which represents the primal graph of one of the RLFAP benchmarks. This WCSP, \( \mathcal{P} \), has only binary objectives, represented by the edges between nodes (variables). The second graph, b), is the primal graph of \( \mathcal{P}^D \) computed by Algorithm 6 when run on \( \mathcal{P} \) WCSP with \( \lambda = 0.35 \). A tree-decomposition of width 3 could be computed for \( \mathcal{P}^D \), while for \( \mathcal{P} \), the computed tree-decomposition has width 9. Furthermore, \( E(\mathcal{P}^D.\text{Obj})/E(\mathcal{P}.\text{Obj}) = 0.76 \); in other words, 76% of the expected cost of the entire problem is found in the objectives of \( \mathcal{P}^D \).

Thus, \( \mathcal{P}^D \) will be able to exploit many more decompositions than \( \mathcal{P} \), but still contains a large majority of the expected weights contained in \( \mathcal{P} \).
5.6 Related Work

Mini-bucket elimination and variable splitting both use the technique of relaxing the WCSP in order to exploit decomposition (mini-buckets are described in detail in Section 2.6.2 on page 24) [37, 19]. However, DB can offer advantages over both algorithms. Mini-buckets and variable splitting operate over all objectives, and they relax the connectivity between the objectives by assuming that some of the shared variables are actually independent. The relaxations made by mini-buckets and variable splitting are guided entirely by the desire to reduce the tree-width—the weight of the objectives is not considered. Greedily selecting a subset of highly weighted objectives (Algorithm 6) represents a new approach and potentially could be applied to enhance these techniques as well.

Since variable splitting and mini-buckets operate over all objectives, they are unable to deal with high arity constraints efficiently. For example, when a WCSP includes a single objective with scope equal to $\text{Vars}$, DB simply excludes the objective from $\mathcal{D}$, whereas variable splitting duplicates every variable of original problem.

A second advantage is seen when dynamic variable-ordering is used. In Chapter 4, it was shown that solvers benefit from increased flexibility in variable-ordering. Mini-buckets can use either static, or dynamic mini-buckets. When static mini-buckets are used, the mini-bucket algorithm is run as a preprocessing step, and the results stored. In order to exploit static mini-buckets, the variable-ordering used during search must follow
the ordering used in the mini-bucket calculation. When dynamic mini-buckets are used, mini-buckets are calculated during search; thus, search is able to use dynamic variable-ordering. However, the overhead of computing mini-bucket bounds during search is costly [71]. On many benchmarks, dynamic mini-buckets explores several orders of magnitude fewer nodes than static mini-buckets, yet still takes longer than static mini-buckets to prove optimality. Marinescu and Dechter determined that when enough memory is available, the static mini-bucket heuristics with relatively large i-bounds solve problems faster than dynamic mini-buckets [71]. In contrast to dynamic mini-buckets, DB computes bounds over the decomposable set as B&B search proceeds, thus significantly reducing overhead when dynamic variable-ordering is used.

DB is also related to a compilation technique used in Clone [86] to solve MAX-SAT problems. Clone uses variable splitting technique to relax the problem, and then compiles the relaxed problem into deterministic decomposable negation normal form (d-DDNF). The d-DDNF representation can be queried to find valid bounds during search. Although the overhead of d-DDNF is not as high as dynamic mini-buckets, empirically, the overhead does seem to be significant, as it is generally outperformed by other bounding algorithms for MAX-SAT, such as MiniMaxSat [54]. Most telling, Clone is particularly outperformed on problems of high tree-width [87].

DB also has similarities with Multi-Objective Russian Doll Search [95]. In Multi-Objective Russian Doll Search, sets of variables are used to generate bounds using a series of Russian Doll searches. The sets are intuitively similar to the decomposable problem and complement problem described in this chapter.

5.7 Experiments

In this section, the B&B, OR-Decomposition, and DB algorithms are tested on WCSPs with high tree-width. The algorithms are tested on both Weighted Constraint
Satisfaction Problems (WCSP) and Most Probable Explanation (MPE) problems from Bayesian networks.

Before discussing the specific algorithms tested, it is necessary to note certain features of DB. DB is not designed for problems of low tree-width. On such problems, traditional decomposition algorithms like AND/OR and OR-Decomposition will typically be better choices. DB is able to prune the Branch and Bound search space, either when $B_D \geq \text{gub}$ or when $B_O \geq \text{gub}$ (Line 5 of DB). In other words, DB can use either bounds computed via decomposition over the sub-problem $\mathcal{P}^D (B_D)$, or the ordinary bounds computed by Branch and Bound ($B_O$). Bounds computed via decomposition benefit from assigning variables that decompose the problems, while ordinary bounds from assigning the most costly variables. Hence, there is some tension in the choice of which variable to instantiate next. Variables that encourage $\mathcal{P}^D$ to decompose can be chosen to enable better bounds $B_D$ from $\mathcal{P}^D$. Alternatively, variables that Branch and Bound would normally select (using the semantic heuristic with which it is operating) may be chosen to enable better bounds from the full problem $\mathcal{P}$.

In addition to testing the performance of DB, we also designed some experiments to test the effectiveness Algorithm 6 which selects the decomposable subproblem, $\mathcal{P}^D$. We implemented and tested the opposite of Algorithm 6 and used this ANTI technique to select $\mathcal{P}^D$. Instead of selecting objectives with high expected cost, the ANTI technique selects objectives with low expected costs.

All algorithms will use one of two variable-ordering heuristics. DVO (Dynamic Variable-Ordering) orders variables solely by a semantic variable-ordering. DVO is described in detail in Section 2.9 on page 37. Multiple Partial Variable-Ordering (MPVO) is guided by a graph-based heuristic, but uses a semantic variable-ordering to choose a variable from the unassigned variables in the set of active labels. MPVO is described in detail in Section 4.6 on page 75.

The following specific algorithms were tested:
1. **BB**, Algorithm 1, with FDAC local propagation [67] using DVO implemented in the solver Toolbar [16];

2. **OR-Decomp**, Algorithm 4, with FDAC local propagation using MPVO;

3. **DB+D**, Algorithm 5, with FDAC local propagation using DVO;

4. **DB+M**, Algorithm 5, with FDAC local propagation using MPVO;

5. **DB+ANTI+M**, Algorithm 5, with FDAC local propagation using MPVO, and the anti-heuristic described above for selecting the objectives of $\mathcal{P}^D$.

**DB+M** and **DB+ANTI+M** gives priority to generating good bounds from $B_D$ by encouraging decomposition in $\mathcal{P}^D$, while **DB+D** gives priority to generating bounds from the full problem. As mentioned in Section 5.3, **DB+D**, **DB+M**, and **DB+ANTI+M** utilize FDAC for bounding the full problem, $\mathcal{P}^{cur}$, and utilize and weaker forward checking to bound the sub-problems $\mathcal{D}$ and $\mathcal{E}$.

Two semantic heuristic are tested on all algorithms. The first semantic heuristic is Domain/Degree (**DOM/DEG**), which minimizes the size of the domain divided by the degree of the variable. Intuitively, **DOM/DEG** considers both the branching factor and the potential for simplifying the problem after assigning a value to the variable. The second semantic heuristic is the Jersolow heuristic (**JER**), which has previously been used for solving WCSPs [16]. Intuitively, the **JER** heuristic considers both the variable's domain size and the average cost of the objective functions in which the variable appears. A detailed description of the variable-ordering heuristics can be found in Section 2.9 on page 37.

All algorithms utilize a value ordering determined by the unary objectives used during local propagation. That is, the values for variable $V_i$ are ordered by lowest unary cost $o_i(V_i)$. For those algorithms that utilized a tree-decomposition, these decompositions are computed using a min-fill algorithm.
All experiments reported here were run with 1200 second timeouts and were conducted on 2.66GHz machines with 8GB of memory. If a problem was not solved to optimality in 1200 seconds, the term DNF (Did Not Finished) is used. If a problem was not solved in 1200 seconds by any of the algorithms, no results are reported. In the experiments conducted, the space used in caching never exceeded available memory, so it was not necessary to prune the cache during search.

5.7.1 Problem Sets

The Earth Observing Satellites with Global Constraints (SPOT5+Global) problems were first discussed in Chapter 4 of this thesis. They involve selecting a subset of candidate photographs such that some imperative constraints are satisfied while the total importance of the selected photographs is maximized. The problems have been formulated as WCSPs with binary and ternary constraints in the SPOT5 benchmark [13]. The SPOT5 benchmark contains 42 problems. The original SPOT5 problems have relatively low tree-width, so to make them applicable for DB two global constraints are added to every instance.\(^2\)

First, a parity constraint is imposed on the assigned values, requiring that the sum of the assigned values be equal to 0 mod 2. Second, a constraint is imposed requiring that the sum of the assigned values be less that 0.9 times the sum of the maximum values in the variable domains. Basic forward checking is used for both of these constraints. Since the scopes of introduced constraints contain all of the problem’s variables, the tree-width of the problem is equal to the number of variables minus one. Algorithm 6 generates a decomposable sub-problem that simply excludes these two global constraints.

The Radio Link Frequency Assignment Problems (RLFAP) were first discussed in Chapter 4 of this thesis. They involve assigning frequencies to a set of radio links in

\(^2\)Note that it can often be the case that in practice when solving a WCSP various additional situation specific constraints have to be considered.
such a way that all the links may operate together without noticeable interference. The RLFAP instances are cast as binary WCSPs [18]. The benchmark family includes 46 problems. Unlike the Spot5+Global benchmarks, these problems are unchanged, since many of the problems have relatively high tree-width in their original form.

### 5.7.2 Spot5+Global Results

Table 5.1 shows the results of the Spot5+Global benchmark with the semantic heuristic DOM/DEG. The results show that DB+D yields a significant improvement over BB, providing a 6 to 16 fold performance improvement on five of the problems, roughly the same performance on four of the problems, and about a 50% decrease in performance on one (very easy) problem. However, when the variable-ordering gives preference to generating decompositions over \( P^D \), there is an even more profound improvement, with DB+M solving two problems the other algorithms cannot solve.

Table 5.2 shows the results on the Spot5+Global problem suite using the Jersolow variable-ordering heuristic. The Jersolow heuristic yielded a considerable improvement in the performance of BB, so that it could now solve all of the instances solved by DB+M. However, there was still a 52 and 90 fold improvement over BB on the two difficult instances that both algorithms solved (b404 and 404). Jersolow provides a significant speed up for BB but DB+M is still better on most problems. Unlike BB, both DB+M and DB+D took longer to solve the instances with Jersolow compared to Dom/Deg, although the reasons for this differing behavior are unknown.

### 5.7.3 RLFAP Results

Table 5.3 shows the results of running Algorithm 6 on RLFAP problems. The induced-width (IW) of the tree-decomposition of the original problem, \( P \), found by our min-fill algorithm, and the induced width found for the generated decomposable sub-problem \( P^D \) are shown as is the proportion of expected objective cost allocated to \( P^D \). Only prob-
<table>
<thead>
<tr>
<th>Instance</th>
<th>BB</th>
<th></th>
<th>DB+D</th>
<th></th>
<th>DB+M</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>time</td>
<td>nodes</td>
<td>time</td>
<td>nodes</td>
<td>time</td>
</tr>
<tr>
<td>b1502</td>
<td>212938</td>
<td>8.76</td>
<td>4872</td>
<td>1.19</td>
<td>664</td>
<td>0.54</td>
</tr>
<tr>
<td>b29</td>
<td>348437</td>
<td>3.08</td>
<td><strong>4451</strong></td>
<td><strong>0.25</strong></td>
<td>8841</td>
<td>0.35</td>
</tr>
<tr>
<td>b404</td>
<td>7265661</td>
<td>94.35</td>
<td>383716</td>
<td>10.4</td>
<td><strong>6082</strong></td>
<td><strong>0.41</strong></td>
</tr>
<tr>
<td>b503</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td><strong>102429</strong></td>
<td><strong>7.68</strong></td>
</tr>
<tr>
<td>b54</td>
<td>15210</td>
<td>0.17</td>
<td>4279</td>
<td>0.18</td>
<td><strong>2989</strong></td>
<td><strong>0.15</strong></td>
</tr>
<tr>
<td>b8</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>1502</td>
<td>220514</td>
<td>9.4</td>
<td>10901</td>
<td>1.64</td>
<td><strong>1508</strong></td>
<td><strong>0.62</strong></td>
</tr>
<tr>
<td>29</td>
<td>348437</td>
<td>3.21</td>
<td><strong>4451</strong></td>
<td><strong>0.24</strong></td>
<td>9187</td>
<td>0.37</td>
</tr>
<tr>
<td>404</td>
<td>10805167</td>
<td>148.39</td>
<td>3209139</td>
<td>158.74</td>
<td>86106</td>
<td><strong>3.1</strong></td>
</tr>
<tr>
<td>503</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td><strong>5849048</strong></td>
<td><strong>370.99</strong></td>
</tr>
<tr>
<td>54</td>
<td>35169</td>
<td>0.4</td>
<td>19378</td>
<td>0.71</td>
<td><strong>9565</strong></td>
<td><strong>0.35</strong></td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Spot5+Global with DOM/DEG, best times and fewest nodes in **bold**.
### Table 5.2: Spot5+Global with Jersolow, best times and fewest nodes in **bold**.

<table>
<thead>
<tr>
<th>Instance</th>
<th>BB</th>
<th></th>
<th></th>
<th>DB+D</th>
<th></th>
<th></th>
<th>DB+M</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>time</td>
<td>nodes</td>
<td>time</td>
<td>nodes</td>
<td>time</td>
<td>nodes</td>
<td>time</td>
</tr>
<tr>
<td>b1502</td>
<td>13219</td>
<td>1.27</td>
<td>6995</td>
<td>1.55</td>
<td>1011</td>
<td>0.59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b29</td>
<td>196606</td>
<td>2.38</td>
<td>15804</td>
<td>0.63</td>
<td>7956</td>
<td>0.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b404</td>
<td>4237997</td>
<td>92.82</td>
<td>952956</td>
<td>41.66</td>
<td>28996</td>
<td>1.55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b54</td>
<td>9926</td>
<td>0.18</td>
<td>4674</td>
<td>0.2</td>
<td>3706</td>
<td>0.19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b8</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1502</td>
<td>12508</td>
<td>1.18</td>
<td>5436</td>
<td>1.31</td>
<td>1563</td>
<td>0.65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>196606</td>
<td>2.37</td>
<td>15804</td>
<td>0.64</td>
<td>4973</td>
<td>0.29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>404</td>
<td>6875312</td>
<td>140.24</td>
<td>3794055</td>
<td>230.2</td>
<td>46827</td>
<td>2.14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>24418</td>
<td>0.38</td>
<td>16067</td>
<td>0.55</td>
<td>70250</td>
<td>2.11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Spot5+Global with Jersolow, best times and fewest nodes in **bold**.

problems with $IW(\mathcal{P}) \geq (|\mathcal{P}.\text{Vars}|/5)$ (high tree-width instances introduced in Section 2.7.5 on page 30) are reported. The induced-width is computed from a min-fill ordering, which only approximates the true tree-width. Nevertheless, the results indicate that Algorithm 6 often produces a $\mathcal{P}^D$ that has significantly lower tree-width than $\mathcal{P}$, while still retaining many of the most costly objectives.

Table 5.4 shows time results on the RLFAP problems with $IW(\mathcal{P}) \geq (|\mathcal{P}.\text{Vars}|/5)$ (high tree-width instances). Table 5.5 shows the number of nodes expanded on the same problems. The tables show the results obtained from BB, OR-Decomp, and the Decomposition Bounding algorithms DB+D, DB+M, and DB+ANTI+M.

The benchmark family includes 46 problems, although only 24 of the problems have $IW(\mathcal{P}) \geq (|\mathcal{P}.\text{Vars}|/5)$ (ten of these problems are not solved by any solver). The results show that DB+D and DB+M generally perform better than the other algorithms, displaying the fastest performance on 11 out of the 14 problems. DB+M dominates
<table>
<thead>
<tr>
<th>Instance</th>
<th>NVar</th>
<th>IW($\mathcal{P}$)</th>
<th>IW($\mathcal{P}^D$)</th>
<th>$\frac{E[\mathcal{P}^D]}{E[\mathcal{P}]}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELAR6-0</td>
<td>16</td>
<td>7</td>
<td>3</td>
<td>0.90</td>
</tr>
<tr>
<td>CELAR6-1-24</td>
<td>14</td>
<td>9</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>CELAR6-1</td>
<td>14</td>
<td>9</td>
<td>3</td>
<td>0.75</td>
</tr>
<tr>
<td>CELAR6-2</td>
<td>16</td>
<td>10</td>
<td>3</td>
<td>0.76</td>
</tr>
<tr>
<td>CELAR6-3</td>
<td>18</td>
<td>10</td>
<td>4</td>
<td>0.80</td>
</tr>
<tr>
<td>CELAR6-4-20</td>
<td>22</td>
<td>11</td>
<td>4</td>
<td>0.84</td>
</tr>
<tr>
<td>CELAR7-0</td>
<td>16</td>
<td>9</td>
<td>4</td>
<td>0.97</td>
</tr>
<tr>
<td>CELAR7-1-20</td>
<td>14</td>
<td>9</td>
<td>4</td>
<td>0.8</td>
</tr>
<tr>
<td>CELAR7-1</td>
<td>14</td>
<td>9</td>
<td>4</td>
<td>0.86</td>
</tr>
<tr>
<td>CELAR7-2</td>
<td>16</td>
<td>10</td>
<td>4</td>
<td>0.94</td>
</tr>
<tr>
<td>CELAR7-4-22</td>
<td>22</td>
<td>11</td>
<td>4</td>
<td>0.96</td>
</tr>
<tr>
<td>graph07</td>
<td>141</td>
<td>32</td>
<td>23</td>
<td>0.99</td>
</tr>
<tr>
<td>graph13reduc</td>
<td>454</td>
<td>142</td>
<td>92</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 5.3: Algorithm 6’s results on the RLFAP problems
<table>
<thead>
<tr>
<th>Instance</th>
<th>n</th>
<th>$IW(\mathcal{P})$</th>
<th>$BB$</th>
<th>OR-Decomp</th>
<th>$DB+ANTI+M$</th>
<th>$DB+D$</th>
<th>$DB+M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELAR6-SUB0</td>
<td>16</td>
<td>7</td>
<td>0.18</td>
<td>0.14</td>
<td>0.6</td>
<td>0.55</td>
<td>0.46</td>
</tr>
<tr>
<td>CELAR6-SUB1-24</td>
<td>14</td>
<td>9</td>
<td>9.29</td>
<td>10.66</td>
<td>9.89</td>
<td>3.41</td>
<td>2.81</td>
</tr>
<tr>
<td>CELAR6-SUB1</td>
<td>14</td>
<td>9</td>
<td>124.61</td>
<td>104.15</td>
<td>122.94</td>
<td>34.99</td>
<td>35.51</td>
</tr>
<tr>
<td>CELAR6-SUB2</td>
<td>16</td>
<td>10</td>
<td>729.51</td>
<td>268.37</td>
<td>545.63</td>
<td>38.86</td>
<td>38.05</td>
</tr>
<tr>
<td>CELAR6-SUB3</td>
<td>18</td>
<td>10</td>
<td>DNF</td>
<td>670.78</td>
<td>DNF</td>
<td>521.19</td>
<td>287.86</td>
</tr>
<tr>
<td>CELAR6-SUB4-20</td>
<td>22</td>
<td>11</td>
<td>59.66</td>
<td>18.4</td>
<td>52.24</td>
<td>3.84</td>
<td>5.3</td>
</tr>
<tr>
<td>CELAR6-SUB4reduc</td>
<td>20</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>556.24</td>
<td>338.38</td>
</tr>
<tr>
<td>CELAR7-SUB0</td>
<td>16</td>
<td>9</td>
<td>0.31</td>
<td>0.32</td>
<td>0.48</td>
<td>0.27</td>
<td>0.14</td>
</tr>
<tr>
<td>CELAR7-SUB1-20</td>
<td>14</td>
<td>9</td>
<td>0.21</td>
<td>0.26</td>
<td>0.22</td>
<td>0.48</td>
<td>0.49</td>
</tr>
<tr>
<td>CELAR7-SUB1</td>
<td>14</td>
<td>9</td>
<td>7.13</td>
<td>8.66</td>
<td>27.6</td>
<td>8.36</td>
<td>9.81</td>
</tr>
<tr>
<td>CELAR7-SUB2</td>
<td>16</td>
<td>10</td>
<td>166.3</td>
<td>284.6</td>
<td>408.3</td>
<td>109.88</td>
<td>111.73</td>
</tr>
<tr>
<td>CELAR7-SUB4-22</td>
<td>22</td>
<td>11</td>
<td>1020.59</td>
<td>93.8</td>
<td>DNF</td>
<td>37.55</td>
<td>186.38</td>
</tr>
<tr>
<td>graph07</td>
<td>141</td>
<td>32</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>511.39</td>
</tr>
<tr>
<td>graph13reducmore</td>
<td>454</td>
<td>142</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>973.47</td>
</tr>
</tbody>
</table>

Table 5.4: Time in Seconds: High tree-width RLFAP problems with DOM/DEG, best times in **bold**.

**DB+ANTI+M** on every instance, illustrating that the selection of objectives for the decomposable problem is fundamental to the performance of **DB**.

Table 5.6 shows the results from the same algorithms applied to RLFAP instances with $IW(\mathcal{P}) < (|\mathcal{P}.Vars|/5)$ (low tree-width instances). On these instances, the decomposition technique **OR-Decomp** dominates both **DB+D** and **DB+M**. Clearly, if the tree-width is low, exploiting decompositions over only a subset of objectives is not as effective as exploiting decompositions over all the objectives. However, the tree-width of instances can be quickly estimated by heuristics as a preprocessing step. If the tree-width is low, traditional decomposition algorithms can be used instead of **DB**.
### Table 5.5: Nodes expanded: High tree-width RLFAP problems with DOM/DEG, fewest nodes in **bold**.

<table>
<thead>
<tr>
<th>Instance</th>
<th>( n )</th>
<th>( IW(\mathfrak{D}) )</th>
<th>BB</th>
<th>OR-Decomp</th>
<th>DB+ANTI+M</th>
<th>DB+D</th>
<th>DB+M</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELAR6-SUB0</td>
<td>16</td>
<td>7</td>
<td>3625</td>
<td>1462</td>
<td>1067</td>
<td>34924</td>
<td>28322</td>
</tr>
<tr>
<td>CELAR6-SUB1-24</td>
<td>14</td>
<td>9</td>
<td>373182</td>
<td>178460</td>
<td>166807</td>
<td>45882</td>
<td><strong>30017</strong></td>
</tr>
<tr>
<td>CELAR6-SUB1</td>
<td>14</td>
<td>9</td>
<td>5.6\times10^6</td>
<td>893052</td>
<td>1.9\times10^6</td>
<td><strong>91998</strong></td>
<td>106814</td>
</tr>
<tr>
<td>CELAR6-SUB2</td>
<td>16</td>
<td>10</td>
<td>4.7\times10^7</td>
<td>859585</td>
<td>3.5\times10^6</td>
<td><strong>91465</strong></td>
<td>92994</td>
</tr>
<tr>
<td>CELAR6-SUB3</td>
<td>18</td>
<td>10</td>
<td>DNF</td>
<td>2.7\times10^6</td>
<td>2.9\times10^7</td>
<td><strong>1.2\times10^6</strong></td>
<td>1.5\times10^6</td>
</tr>
<tr>
<td>CELAR6-SUB4-20</td>
<td>22</td>
<td>11</td>
<td>2.5\times10^6</td>
<td>255186</td>
<td>2.1\times10^6</td>
<td><strong>33529</strong></td>
<td>57387</td>
</tr>
<tr>
<td>CELAR6-SUB4reduc</td>
<td>20</td>
<td>11</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>1.8\times10^6</td>
<td><strong>941522</strong></td>
</tr>
<tr>
<td>CELAR7-SUB0</td>
<td>16</td>
<td>9</td>
<td>4079</td>
<td>2205</td>
<td>2587</td>
<td>1829</td>
<td><strong>1424</strong></td>
</tr>
<tr>
<td>CELAR7-SUB1-20</td>
<td>14</td>
<td>9</td>
<td>7907</td>
<td>5097</td>
<td>5313</td>
<td>35459</td>
<td>39307</td>
</tr>
<tr>
<td>CELAR7-SUB1</td>
<td>14</td>
<td>9</td>
<td>153636</td>
<td>85982</td>
<td>591620</td>
<td><strong>61430</strong></td>
<td>65586</td>
</tr>
<tr>
<td>CELAR7-SUB2</td>
<td>16</td>
<td>10</td>
<td>1.7\times10^6</td>
<td>570860</td>
<td>5.3\times10^6</td>
<td>609819</td>
<td>592776</td>
</tr>
<tr>
<td>CELAR7-SUB4-22</td>
<td>22</td>
<td>11</td>
<td>6.2\times10^6</td>
<td><strong>476342</strong></td>
<td>1.7\times10^8</td>
<td>1.4\times10^6</td>
<td>1.2\times10^7</td>
</tr>
<tr>
<td>graph07</td>
<td>141</td>
<td>32</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td><strong>1.3\times10^7</strong></td>
</tr>
<tr>
<td>graph13reducmore</td>
<td>454</td>
<td>142</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td><strong>3.8\times10^7</strong></td>
</tr>
</tbody>
</table>
### Table 5.6: Time in Seconds: Low tree-width RLF AP problems using DOM/DEG, best times in bold.

<table>
<thead>
<tr>
<th>Instance</th>
<th>$n$</th>
<th>$IW(P)$</th>
<th>$BB$</th>
<th>$OR$-Decomp</th>
<th>$DB+D$</th>
<th>$DB+M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph12</td>
<td>252</td>
<td>47</td>
<td>5.48</td>
<td><strong>5.38</strong></td>
<td>29.41</td>
<td>DNF</td>
</tr>
<tr>
<td>scen06-20reduc</td>
<td>82</td>
<td>11</td>
<td>DNF</td>
<td><strong>563.17</strong></td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>scen06-22reduc</td>
<td>81</td>
<td>11</td>
<td>DNF</td>
<td><strong>321.54</strong></td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>scen06-22</td>
<td>99</td>
<td>11</td>
<td>DNF</td>
<td><strong>956.67</strong></td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>scen06-24reduc</td>
<td>81</td>
<td>11</td>
<td>DNF</td>
<td><strong>76.38</strong></td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>scen06-24</td>
<td>99</td>
<td>11</td>
<td>DNF</td>
<td><strong>196.4</strong></td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>scen06-30reduc</td>
<td>81</td>
<td>11</td>
<td>DNF</td>
<td><strong>1.62</strong></td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>scen06-30</td>
<td>99</td>
<td>10</td>
<td>DNF</td>
<td><strong>2.16</strong></td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>scen09</td>
<td>200</td>
<td>7</td>
<td>5.5</td>
<td><strong>0.09</strong></td>
<td>32.15</td>
<td>16.37</td>
</tr>
<tr>
<td>scen10</td>
<td>200</td>
<td>7</td>
<td>0.39</td>
<td><strong>0.04</strong></td>
<td>1.88</td>
<td>2.4</td>
</tr>
</tbody>
</table>

### Table 5.7: Number Solved and Time in Seconds for RLFAP instances with DOM/DEG

<table>
<thead>
<tr>
<th>Instances</th>
<th>BB</th>
<th>OR-Decomp</th>
<th>DB+P</th>
<th>DB+M</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#s</td>
<td>Time</td>
<td>#s</td>
<td>Time</td>
<td>#s</td>
</tr>
<tr>
<td>RLFAP</td>
<td>13</td>
<td>15329</td>
<td>21</td>
<td>7183</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 5.7: Number Solved and Time in Seconds for RLFAP instances with DOM/DEG
Table 5.7 summarizes the results for all RLFAP instances using DOM/DEG variable-ordering. The **Combined** algorithm uses a MIN-FILL algorithm to calculate the induced-width of instances as a pre-processing step. If \( IW(P) < (|P.Vars|)/5 \), then **OR-Decomp** is launched, otherwise **DB+M** is launched. The total time for the pre-processing step is a moderate 5.66 seconds for all 24 instances. **Combined** is able to solve three more instances than any other algorithm.

Table 5.8 shows the results from the same algorithms performed on RLFAP instances where \( IW(P) \geq (|P.Vars|)/5 \) (high tree-width instances), using the Jersolow variable-ordering heuristic. On this set of experiments, no algorithm clearly outperforms the others. When the Jersolow variable-ordering heuristic is used, **DB** does not exhibit the same performance improvement that is seen when the DOM/DEG variable-ordering heuristic is used. The lack of improvement is probably caused by the fact that Algorithm 6 selects objectives with high expected cost, while the Jersolow variable-ordering heuristic chooses variables to assign that are likely to have adjacent edges with high cost. Thus, when Jersolow variable-ordering is used, variables will be selected that have adjacent objectives which are also likely to be in \( P^D.\text{Obj} \). When the variables are assigned, these binary objectives are immediately transformed into unary objectives. Since **DB** operates by building up strong bounds during search, eliminating objectives early in the search space will reduce the effectiveness of the bounding provided by \( P^D \).

Finally, Figure 5.3 summarizes the results of varying parameter \( \lambda \) on the high tree-width RLFAP instances using **DB+M** and DOM/DEG variable-ordering. The parameter \( \lambda \) is used by Algorithm 6 to determine the number of objectives to include in \( P^D \). The X-axis represents the value assigned to \( \lambda \), while the Y-axis represents the total time to solve the problem set. If a problem is not solved to optimality within 1200 seconds, a time of 1200 is added to the total. When \( \lambda \) is small, only a small number of objectives are included in \( P^D \), and no meaningful bounds are found. When \( \lambda \) is large, the tree-width of \( P^D \) is large, and fewer decompositions exist in \( P^D \). There is, however, a large range of
<table>
<thead>
<tr>
<th>Instance</th>
<th>n</th>
<th>IW(?)</th>
<th>BB</th>
<th>OR-Decomp</th>
<th>DB+D</th>
<th>DB+M</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELAR6-SUB0</td>
<td>16</td>
<td>7</td>
<td>0.16</td>
<td>0.29</td>
<td>0.29</td>
<td>0.24</td>
</tr>
<tr>
<td>CELAR6-SUB1-24</td>
<td>14</td>
<td>9</td>
<td>2.73</td>
<td>10.51</td>
<td>3.06</td>
<td><strong>1.72</strong></td>
</tr>
<tr>
<td>CELAR6-SUB1</td>
<td>14</td>
<td>9</td>
<td>44.19</td>
<td>85.81</td>
<td>46.2</td>
<td><strong>20.73</strong></td>
</tr>
<tr>
<td>CELAR6-SUB2</td>
<td>16</td>
<td>10</td>
<td><strong>16.77</strong></td>
<td>48.97</td>
<td>22.34</td>
<td>26.9</td>
</tr>
<tr>
<td>CELAR6-SUB3</td>
<td>18</td>
<td>10</td>
<td>234.5</td>
<td>266.49</td>
<td>260.83</td>
<td><strong>174.82</strong></td>
</tr>
<tr>
<td>CELAR6-SUB4-20</td>
<td>22</td>
<td>11</td>
<td>10.68</td>
<td>65.52</td>
<td>15.42</td>
<td><strong>5.57</strong></td>
</tr>
<tr>
<td>CELAR6-SUB4reduc</td>
<td>20</td>
<td>11</td>
<td>DNF</td>
<td>1099.14</td>
<td>DNF</td>
<td><strong>478.93</strong></td>
</tr>
<tr>
<td>CELAR6-SUB4</td>
<td>22</td>
<td>11</td>
<td>DNF</td>
<td><strong>630.22</strong></td>
<td>DNF</td>
<td>DNF</td>
</tr>
<tr>
<td>CELAR7-SUB0</td>
<td>16</td>
<td>9</td>
<td><strong>0.07</strong></td>
<td>1.32</td>
<td>0.15</td>
<td>0.22</td>
</tr>
<tr>
<td>CELAR7-SUB1-20</td>
<td>14</td>
<td>9</td>
<td><strong>0.14</strong></td>
<td>1</td>
<td>0.2</td>
<td>0.34</td>
</tr>
<tr>
<td>CELAR7-SUB1</td>
<td>14</td>
<td>9</td>
<td>5.21</td>
<td>17.04</td>
<td><strong>4.84</strong></td>
<td>4.96</td>
</tr>
<tr>
<td>CELAR7-SUB2</td>
<td>16</td>
<td>10</td>
<td><strong>43.61</strong></td>
<td>57.12</td>
<td>51.25</td>
<td>74.31</td>
</tr>
<tr>
<td>CELAR7-SUB3</td>
<td>18</td>
<td>10</td>
<td><strong>484.43</strong></td>
<td>DNF</td>
<td>529.83</td>
<td>DNF</td>
</tr>
<tr>
<td>CELAR7-SUB4-22</td>
<td>22</td>
<td>11</td>
<td><strong>7.11</strong></td>
<td>16.54</td>
<td>9.06</td>
<td>18.57</td>
</tr>
<tr>
<td>graph07</td>
<td>141</td>
<td>32</td>
<td><strong>29.4</strong></td>
<td>640</td>
<td>51.8</td>
<td>DNF</td>
</tr>
<tr>
<td>graph13reducmore</td>
<td>454</td>
<td>142</td>
<td><strong>222.6</strong></td>
<td>DNF</td>
<td>360.4</td>
<td>DNF</td>
</tr>
</tbody>
</table>

Table 5.8: Time in Seconds: High tree-width RLFAP problems using Jersolow, best times in **bold**.
values for which \( \lambda \) produces good results.

5.7.4 Experimental Conclusions

The experiments above illustrate ways in which DB can exploit effectively decompositions on problems with high tree-width. When Algorithm 6 is run on the RLFAP benchmarks, \( P^D.\text{Obj} \) contains most of the expected cost of the objectives, while the induced-width is substantially lower than \( P \). The Spot5+Global benchmarks illustrate that DB is effective on problems with no decomposition in their original form. Although DB posts more modest improvements over competitor algorithms on high tree-width RLFAP instances, it is still more effective than other algorithms when using the DOM/DEG heuristic.
5.8 Applicability

The experiments in this chapter demonstrate that DB is not effective on problems with low tree-width. On such problems, traditional decomposition techniques such as AND/OR and OR-Decomposition outperform DB. On problems with high tree-width, on the other hand, DB has proven effective. The tree-width of instances can be quickly estimated by heuristics as a preprocessing step, making it simple to identify which algorithm is likely to solve the given problem most quickly. If the width of the tree-decomposition is low, traditional decomposition algorithms can be used instead of DB.

DB relies on good bounds being generated by a subset of objectives. If every objective in the original problem has equal expected cost, then there is no way to predict which objectives will have the greatest impact on the lower-bound. Thus, DB is more effective on problems with a large range of expected costs of objectives.

5.9 Future Work

Algorithm 6 is a greedy heuristic for determining which objectives to add to the decomposable set. Although Algorithm 6 places objectives with high expected costs into $P^D.\text{Obj}$, there is no guarantee that the tree-width of $P^D$ is small. A natural area for future research would be to devise different methods for selecting objectives for $P^D$.

Exploiting decomposition information for a larger number of sub-problems could provide a second avenue for future work. Given an original problem, $P$, DB maintains bound information for only two sub-problems; $P^D$ and $P^C$. The original problem, $P$, can instead be partitioned into $n$ sub-problems, $P_1, ..., P_n$, where $\bigcup P_i.\text{Obj} = P.\text{Obj}$, and $\bigcap P_i.\text{Obj} = \emptyset$. If decomposition information is maintained for all of these sub-problems, then search can backtrack if $\text{getBounds}(P_1) + ... + \text{getBounds}(P_n)$ exceeds the global upper-bound.
Finally, it would be interesting to see if Algorithm 6 might improve the performance of the related work described in Section 5.6, such as CLONE and dynamic mini-buckets.

5.10 Conclusions

WCSPs can benefit greatly from decomposition, but traditional decomposition techniques are ineffective on problems with high tree-width. This chapter introduces a way of exploiting decomposition on such problems. This method is tested on a number of benchmarks, showing that it can yield improvements over current competitive algorithms.
Chapter 6

Symmetric Templating

6.1 Introduction

The previous Chapters, 3-5, examined a number of techniques for exploiting decomposition in COPs. Now we turn to some techniques for exploiting symmetries.

In Chapter 3, a technique called Component Templating was introduced. Component Templates represent the constraint graph of components encountered during search. As demonstrated in Chapter 3, given a set of assignments to the variables of a problem, templates provide an efficient method of determining which components exist. They also provide a useful tool for storing information about similar components. Since the work presented in this chapter relies heavily on Component Templates, the definitions relating to Component Templates are reviewed here.

A component, \( \kappa = \langle \kappa.\text{Vars}, \kappa.\text{Obj}, \kappa.\text{A} \rangle \), contains a set of variables, \( \kappa.\text{Vars} \), and all of the objectives, \( \kappa.\text{Obj} \), over these variables. The key feature of a component is that its objectives, \( \kappa.\text{Obj} \), are isolated from the rest of the problem by its set of assignments, \( \kappa.\text{A} \). All of the variables in the scope of the objectives in \( \kappa.\text{Obj} \) are either variables of the component (i.e., in \( \kappa.\text{Vars} \)) or are instantiated in \( \kappa.\text{A} \). Furthermore, none of the variables in \( \kappa.\text{Vars} \) are assigned in \( \kappa.\text{A} \).
Component Templates represent a set of components that all have an identical set of variables. Formally, a component template, $\mathcal{T} = \langle \mathcal{T}.Vars, \mathcal{T}.Obj, \mathcal{T}.D \rangle$, is a set of variables, $\mathcal{T}.Vars$; objectives, $\mathcal{T}.Obj$; and, another set of variables, $\mathcal{T}.D$, that are disjoint from $\mathcal{T}.Vars$ called the dependency variables. Every set of assignments $\mathcal{A}$ to the variables in $\mathcal{T}.D$ generates an instance of the template, $\mathcal{T}(\mathcal{A})$, that is a component: $\mathcal{T}(\mathcal{A}) = \langle \mathcal{T}.Vars, \mathcal{T}.Obj, \mathcal{A} \rangle$. Each instance of the template is a component with the same variables and objectives as the template. $\mathcal{A}$ is the set of assignments defining the instance, and also serving to disconnect the component from the rest of the problem. Since each instance is defined by a particular assignment to the dependency variables, the set of assignments to these dependency variables can be used as an index into a value cache associated with each Component Template. This value cache can be used to remember bounds on each template instance.

In this chapter, we present a technique called Symmetric Templating, that exploits symmetries between templates. As we will demonstrate, Symmetric Templating facilitates the efficient exploitation of symmetries in COPs by identifying symmetries that hold for an entire set of components, namely the set of instances of a component template. Symmetric Templating efficiently exploits variable symmetries in COPs by storing symmetry information in Component Templates.

Now we discuss more precisely what is meant by symmetries in COPs. If $\sigma$ is a permutation of the variables of the problem, we can apply $\sigma$ to an assignment $\mathcal{A}$ by permuting the variables in the assignment equations. For example, if the symmetry $\sigma$ exchanges the variables $X$ and $Y$, then applying $\sigma$ to the assignment $\{X=a, Y=b\}$ yields the assignment $\{Y=a, X=b\}$.

In CSPs (where complete assignments either satisfy or falsify the CSP), a permutation, $\sigma$, is a symmetry for the CSP if, whenever $\sigma$ is applied to a complete assignment it leaves the satisfying/falsifying status of the assignment unchanged. That is, a variable symmetry preserves the set of solutions. We can extend the idea of symmetries to COPs
by simply requiring that symmetries leave the cost of every complete assignment, \( A \), unchanged. That is, a permutation, \( \sigma \), is a symmetry for a COP, \( \mathcal{P} \), if for every complete assignment, \( A \), 
\[
\text{cost}(A, \mathcal{P}) = \text{cost}(\sigma(A), \mathcal{P})
\]
(\text{cost} and \( \text{mincost} \) are defined in Section 2.3.3 on page 10). Note that symmetries can also be applied to partial assignment. An assignment, \( A \), creates a reduced problem \( \mathcal{P}|_A \). The definition of a symmetry implies that for every reduced problem \( \mathcal{P}|_A \), 
\[
\text{mincost}(\mathcal{P}|_A) = \text{mincost}(\mathcal{P}|_{\sigma(A)}).
\]

A permutation, \( \sigma \), of a reduced problem, \( \mathcal{P}|_A \), is called a conditional symmetry if it is a symmetry on \( \mathcal{P}|_A \) but is not a symmetry on \( \mathcal{P} \). That is, it holds only after the assignments \( A \) have been made. The detection of conditional symmetries is called dynamic symmetry detection as it must be performed during search after assignments have been made. Dynamic symmetry detection has proven to be very costly since symmetries must be computed during search, adding considerable overhead to the search process.

In this chapter, a technique is proposed which efficiently detects and exploits conditional symmetries in COPs, by utilizing Component Templates. Basically, the technique detects symmetries between Component Templates so that information that has been computed and stored in one template can be re-used in all symmetric templates. Since templates are only generated after some variables have been instantiated, the detected symmetries are conditional symmetries. Furthermore, as demonstrated below, the template symmetries hold for all template instances. Hence, by doing symmetry detection at the template level, the computational cost of detecting symmetries can be amortized over all instances of the template. This makes dynamic symmetry detection during search feasible. Furthermore, because template symmetries apply to all of the instances of the templates, the technique can lead to large reductions in the search space.

In previous work, symmetries have been exploited by modifying the search algorithm to avoid exploring symmetric sub-trees during search [10, 44, 43, 42, 39, 96, 51], includ-

\(^1\)Other finer-grain symmetries can also be defined, e.g., permutations of the variable values that leaves the solutions invariant.
ing SBDS and SBDD (see Section 2.8.2 for details). Although symmetries have been extensively studied in the context of CSPs, there has been far less work done for COPs. Walsh proposes a method that breaks symmetries in optimization problems by adding symmetry-breaking constraints to the problem [110], although the ideas presented in the paper have not yet been implemented. Prestwich and Beck used dominance relationships to prune the search space, although the dominance relationships are hand-encoded for each problem [88].

6.2 Template Symmetries

As pointed out in the previous section, detecting symmetries at the level of Component Templates allows us to amortize the computational effort over all instances of the template. The technique we propose in this chapter involves the automatic detection of symmetry. That is, no user specified problem specific knowledge need be provided to the algorithm. Automatic symmetry detection is performed during search, so to further reduce its cost, the methods proposed in this chapter consider only variable symmetries, rather than finer-grain symmetries defined over variable-values. We first define formally what a template symmetry is.

**Definition 6.2.1** A symmetry between two templates, $\mathcal{T}_1$ and $\mathcal{T}_2$, is a one-to-one and onto mapping $\sigma$ between the variables $\mathcal{T}_1.\mathcal{D} \cup \mathcal{T}_1.\text{Vars}$ and $\mathcal{T}_2.\mathcal{D} \cup \mathcal{T}_2.\text{Vars}$ such that

1. $\mathcal{T}_2.\mathcal{D} = \sigma(\mathcal{T}_1.\mathcal{D})$ and $\mathcal{T}_2.\text{Vars} = \sigma(\mathcal{T}_1.\text{Vars})$, where $\sigma$ applied to a set $S$ is $\sigma(S) = \{\sigma(V) | V \in S\}$;

2. For any assignment $A$ to all of the variables in $\mathcal{T}_1.\mathcal{D} \cup \mathcal{T}_1.\text{Vars}$, the value of the objectives $\mathcal{T}_1.\text{Obj}$ evaluated at $A$ is identical to the value of the objectives of $\mathcal{T}_2.\text{Obj}$ evaluated at $\sigma(A)$. 

In other words, $\sigma$ keeps the dependency variables and template variables separated, and it preserves the value of the template objectives. The minimum cost of a template instance, $T_1(A)$, is the minimum of the sum of the objectives $T_1.Obj$, under the fixed assignment, $A$, and any assignment to $T_1.Vars$. Thus, the following observation holds:

**Observation 6.2.2** If $T_1$ and $T_2$ are symmetric under the mapping, $\sigma$, then for any instance of $T_1$, $T_1(A)$, it holds that $\text{mincost}(T_1(A)) = \text{mincost}(T_2(\sigma(A)))$.

Observation 6.2.2 means that any bounds computed for the component $T_1(\sigma(A))$ can be re-used for the component $T_2(A)$.

### 6.3 Automatically Detecting Symmetries

In order to automatically detect symmetries between templates, the symmetry detection problem is mapped to a graph-isomorphism problem. This mapping is done by constructing a graph representation for each template. Two templates’ graph representations are graph isomorphic if and only if the templates are symmetric in the sense of Definition 6.2.1. The graph isomorphism, which maps the vertices of one graph to the other, provides the variable-to-variable symmetry-mapping between the two templates.

We use NAUTY [80] to compute graph isomorphisms. As shown in [90], such software can be surprisingly efficient even though graph isomorphism is not known to be of polynomial complexity.

In the techniques proposed in this chapter, symmetries are detected as follows. When a template, $T$, is first created, its graph representation, $G_T$, is constructed. NAUTY is then used to compute $\text{iso}(G_T)$, a canonical isomorph of $G_T$. $T$ is symmetric to some previously cached template, $T'$, if and only if their canonical graph isomorphs are equal; i.e., $\text{iso}(G_T) = \text{iso}(G_{T'})$. By using hashing techniques on $\text{iso}(G_T)$, any isomorphic template can be found in near constant time. If an isomorphic template $T'$ is found, the invertible mappings $\sigma_1 = G_T \mapsto \text{iso}(G_T)$ and $\sigma_2 = G_T \mapsto \text{iso}(G_T)$ produced by NAUTY
are used to compute a symmetry mapping $\sigma$ from $T$ to $T'$. That is, $\sigma = \sigma_2^{-1}(\sigma_1)$ since $iso(G_T) = iso(G_T')$.

NAUTY is able to find isomorphisms between coloured graphs. An isomorphism exists between coloured graphs if and only if there is an isomorphism that maps vertices of the same colour to one another. In order to find valid symmetries between templates, a coloured graph representation is created for each template. Although there are many choices for representing the template graph, one possible way to construct a graphical representation is given below. Each template graph will have five different types of vertices; member variables vertices at step 1, dependency variables vertices at step 2, objective vertices at step 3, assignment vertices at step 4, and objective tuples vertices at step 5.

1 Symmetry Graph $(\mathcal{J} = \langle \mathcal{J}.Vars, \mathcal{J}.Obj, \mathcal{J}.D \rangle)$

\begin{algorithm}
begin
Step 1: A vertex is added for each variable $V \in \mathcal{J}.Vars$.
Step 2: A vertex is added for each variable $V \in \mathcal{J}.D$.
Step 3: A vertex is added for each objective $o \in \mathcal{J}.Obj$.
Step 4: A vertex is added for each assignment in the set 
\[ \{ V = a \mid a \in \text{Dom}[V], V \in \mathcal{J}.Vars \cup \mathcal{J}.D \} \].
Step 5: For each objective $o \in \mathcal{J}.Obj$, let $S_o$ be the set of tuples of assignments to the variables in $\text{scope}(o)$ such that $\vec{t} \in S_o \rightarrow o(\vec{t}) > 0$. A vertex is added for each tuple $\vec{t}$ in each $S_o$.
end
\end{algorithm}

Algorithm 7: Steps to Create Graphical Representation of Template $\mathcal{J}$

A fixed set of distinct colours are selected and used to colour every template graph. Different colours are needed for (a) the member variable vertices added in Step 1, (b) the dependency variable vertices added in Step 2, and (c) the objective vertices added in Step 3. In addition to these three colours, a distinct colour is needed for each value in
the union of the variable domains. Finally, a distinct colour (different from all previously used colours) is needed for each different non-zero cost in the union of the range of the problem's objectives.

With these colours, we proceed to colour the vertices of the template's graph as follows. Each member variable vertex, dependency variable vertex, and objective vertex is coloured with the colour chosen for that class of vertices. Then each assignment vertex (added in Step 4) is coloured with the colour selected for the assigned value. For example, the assignment vertices \( X = 1, Y = 1, \) and \( Z = 1 \) will all be coloured with the colour selected for the value 1, while the assignment vertices \( X = 2 \) and \( Y = 2 \) will be coloured with the (different) colour selected for the value 2. Finally, each tuple vertex, \( \vec{t} \) associated with objective \( o \), is coloured with the colour selected for the cost \( o(\vec{t}) \). That is, if the objective associated with tuple vertex \( \vec{t} \) is \( o \), and \( o(\vec{t}) = r \), then the tuple vertex \( \vec{t} \) will be coloured with the colour selected for the (non-zero) cost \( r \). For example, the two tuple vertices \( \vec{t} \) and \( \vec{s} \) with associated objectives \( o_1 \) and \( o_2 \), will be coloured with the same colour if \( o_1(\vec{t}) = o_2(\vec{s}) \). Furthermore, all other tuple vertices that yield the same objective cost will be coloured with this colour.

Note that the set of colours is selected before search, and is used to colour every template graph constructed during search. Thus, the member variables vertices of every template graph will be coloured with the same colour. Similarly, in all template graphs every tuple vertex whose associated objective assigns cost \( r \) to the tuple will be coloured with the same colour. The uniform colour scheme used across all template graphs is what allows isomorphism detection to work.

Finally, after the vertices have been added and coloured, the graph edges are added. First, an edge \((X, X = i)\) is added between each assignment vertex \( X = i \) and its associated (member or dependency) variable vertex \( X \). Second, for each tuple vertex \( \vec{t} = (X = x, Y = y, ..., Z = z) \) an edge is added between the tuple vertex and the associated assignment vertex for every assignment contained in the tuple, i.e., \((\vec{t}, X = x)\),
(\vec{t}, Y = y), \ldots, (\vec{t}, Z = z). Third, an edge \((a, \vec{t})\) is added between each objective vertex, \(a\), and each tuple vertex, \(\vec{t}\), associated with that objective (i.e., each \(\vec{t} \in S_a\) added to the graph in Step 5).

**Example 6.3.1** Consider a WCSP with variables \(\{A, B, M, N, X, Y\}\) and objectives \(o_1(A, B, M), o_2(X, Y, N),\) and \(o_3(M, N)\). \(o_1\) maps the assignment \(\langle A = 0, B = 0, M = 1 \rangle\) to cost 1, \(\langle A = 1, B = 1, M = 1 \rangle\) to cost 10, and all other tuples to cost 0. \(o_2\) maps the assignment \(\langle X = 0, Y = 0, N = 1 \rangle\) to cost 1, \(\langle X = 1, Y = 1, N = 1 \rangle\) to cost 10, and all other tuples to cost 0.

When the assignment \(M = a\) is made for the first time, a new template \(\mathcal{T}_{ab} = \langle \mathcal{T}_{ab}.\text{Vars} = \{A, B\}, \mathcal{T}_{ab}.\text{Obj} = \{o_1\}, \mathcal{T}_{ab}.D = \{M\} \rangle\) is created. When the assignment \(N = b\) is made for the first time, a new template \(\mathcal{T}_{xy} = \langle \mathcal{T}_{xy}.\text{Vars} = \{X, Y\}, \mathcal{T}_{xy}.\text{Obj} = \{o_2\}, \mathcal{T}_{xy}.D = \{N\} \rangle\) is created.

When template \(\mathcal{T}_{ab}\) is created, Algorithm 7 builds the graph representation of \(\mathcal{T}_{ab}\) as follows:

- Vertices are added for variables \(A, B,\) and \(M\). The member variables, \(A\) and \(B\), are assigned one colour, while the dependency variable, \(M\), is assigned a second colour.

- A vertex is added for the objective \(o_1\), which is a third colour.

- For each variables \(A, B,\) and \(M\), vertices are added for values 0,1. All vertices representing the assignment of value 0 will be coloured one colour, and all vertices representing the assignment of value 1 will be coloured a different colour.

- A vertex is added for each of the two tuples \(\langle A = 0, B = 0, M = 1 \rangle\) and \(\langle A = 1, B = 1, M = 1 \rangle\). These two vertices are assigned different colours, since their costs are different.
The complete graph is seen in Figure 6.1.

When template $T_{xy}$ is created, Algorithm 7 builds the graph seen in Figure 6.2.

A graph isomorphism exists between the graphs in Figure 6.1 and Figure 6.2, which maps variable $X$ to $A$, $Y$ to $B$, and $N$ to $M$.

### 6.4 Using Symmetries

When a symmetry is detected between template, $T$, and a previously created template, $T'$, no value cache is created for template $T$. Instead, $T$ is marked as being symmetric to $T'$ and the symmetry map, $\sigma$, is stored. $T$ is still used to detect the point at which any instances of $T$ appear during search (see Chapter 3); however, whenever information
about one of $T$’s instances is stored or retrieved from the cache, that access is mapped to the symmetric instance of $T'$. Thus, all instances of $T$ are able to utilize bounds computed for instances of $T'$. Furthermore, any bound computed for instances of $T$ are stored as information about instances of $T'$. Since there may be many different templates symmetric to $T'$ (symmetries always map to the earliest created template), information computed for an instance of $T$ can then be utilized by many other symmetric components.

Consider the WCSP described in Example 6.3.1. When the assignment $M = a$ is made, the problem splits into two components, $\kappa_{ab} = \langle \kappa_{ab}.Vars = \{A,B\}, \kappa_{ab}.Obj = \{o_1\}, \kappa_{ab}.A = \{M = a\}\rangle$ and $\kappa_{xyn} = \langle \kappa_{xyn}.Vars = \{X,Y,N\}, \kappa_{xyn}.Obj = \{o_2,o_3\}, \kappa_{xyn}.A = \{M = a\}\rangle$. If a template defined by the same variables and objectives as $\kappa_{ab}$ does not yet exist, the template is created. In particular, a new template $T_{ab} = \langle T_{ab}.Vars = \{A,B\}, T_{ab}.Obj = \{o_1\}, T_{ab}.D = \{M = a\}\rangle$, is created. Similarly, when the additional assignment $N = b$ is made, $\kappa_{xyn}$ is reduced to $\kappa_{xy} = \langle \kappa_{xy}.Vars = \{X,Y\}, \kappa_{xy}.Obj = \{o_2\}, \kappa_{xyn}.A = \{N = b\}\rangle$. A new template $T_{xy}$ can be created if it does not already exist.

A symmetry between the templates $T_{ab}$ and $T_{xy}$ will be the found, in this case mapping variable $N$ to $M$. The cache of template $T_{xy}$ will point to the cache of template $T_{ab}$ under the mapping $N \mapsto M$. If an instance $T_{xy}(N = a)$ is created, the cached results $T_{ab}(M = a).lb$ and $T_{ab}(M = a).ub$ can be used as bounds for $T_{xy}(N = a)$.

Note that in the original WCSP, if $N$ and $M$ are not exchangeable in $o_3(N,M)$, then there is no symmetry until both $M$ and $N$ are assigned. This is an example of a conditional symmetry that can be detected by Symmetric Templating.

### 6.5 Automorphism

The same technique used to detect isomorphisms between templates also can be used to detect automorphisms within a single template (in other words, conditional variable
interchangeabilities). An automorphism exists in template, \( T \), if there is a symmetry, \( \sigma \), between \( T \) and itself. Automorphisms can be detected by running NAUTY on the coloured graph representation presented above. If an automorphism, \( \sigma \), is detected, \( \sigma \) is stored in the template’s data structure. For example, suppose an instance of \( T \) is found, with assignments \( A \) to \( T \). When information about one of \( T \)’s instances is retrieved from the cache, the cache look-up attempts to find a solution by querying the cache indexed by the assignment \( A \), and then the assignment \( iso(A) \) (since both bounds are valid, the algorithm will use the tighter of the two bounds). Many automorphism can be detected in a single template, in which case, the access attempts to find a solution from every automorphic assignment.

6.6 Symmetric Component Caching Search

This section formally defines a decomposition search algorithm that uses Symmetric Component Caching. Symmetric Component Caching Search is the same as AND/OR Branch and Bound search with Component Caching with only minor modifications. In Chapter 3, AND/OR Branch and Bound search with Component Caching was introduced. The algorithm is reproduced in Algorithm 8 for convenience.

In Line 7, Algorithm 8 determines the way in which the current component will decompose when a variable is assigned. This determination is made by a combination of template triggering and connected component analysis (see Chapter 3 for details). Once the triggered templates are removed from the constraint graph, each remaining connected component forms new templates based on the connectivity of the remaining constraint graph. For each new template, \( T \), a coloured symmetry graph is created, and isomorphisms are calculated between the new template and all other existing templates. If a symmetry is detected between \( T \) and \( T' \), \( T \) references the cache of \( T' \) in Line 12, Line 13, and Line 20.
Chapter 6. Symmetric Templating

1 \textbf{AND/OR+CC} \((T\langle A \rangle, UB)\)

/* On entry \(T\langle A \rangle\) must have valid bounds \((T\langle A \rangle).lb, T\langle A \rangle.ub)\). If 
\[\text{mincost}(T\langle A \rangle) < UB\] then compute \(T\langle A \rangle).lb = \text{mincost}(\kappa) = T\langle A \rangle.ub\). Else compute valid bounds on \(T\langle A \rangle\) such that \(UB < T\langle A \rangle.lb \leq T\langle A \rangle.ub\) */

begin

2 \textbf{if} \((T\langle A \rangle.lb < UB \land T\langle A \rangle.lb \neq T\langle A \rangle.ub)\) \textbf{then}

3 \textbf{if} UB > \(T\langle A \rangle.ub\) \textbf{then}

4 \hspace{1em} UB := \(T\langle A \rangle.ub + 1\)

5 \hspace{1em} \(V := \text{select variable from } T.\text{Vars to branch on}\)

6 \hspace{1em} \(K := \text{Find the templates contained in the constraint graph consisting of}\)

7 \hspace{1em} \(T.\text{Obj} \text{ and } T.\text{Vars} - \{V\}\)

8 \textbf{foreach} \(d \in \text{Dom}[V]\) \textbf{do}

9 \hspace{1em} \(\Delta d = \sum_{o \in T\langle A \rangle | \text{scope}(o) \cap T.\text{Vars} = \{V\}} o_i(V = d, A)\)

10 \textbf{foreach} \(T_i \in K\) \textbf{do}

11 \hspace{1em} \(A_i := (A \cup \{V = d\})|_{T_i.d}\)

12 \hspace{1em} \(T_i(A_i).lb = \text{MAX}(\text{getBounds}(T_i(A_i)), \text{getCache}(T_i(A_i)).lb)\)

13 \hspace{1em} \(T_i(A_i).ub = \text{MIN}(\text{getBounds}(T_i(A_i)), \text{getCache}(T_i(A_i)).ub)\)

14 \textbf{foreach} \(T_i \in K\) (while \(\sum_i T_i(A_i).lb + \Delta d < UB\)) \textbf{do}

15 \hspace{1em} UB_i := UB - (\Delta d + \sum_{j \neq i} T_j(A_j).lb)

16 \hspace{1em} \textbf{AND/OR+CC} \((T_i(A_i), UB_i)\)

17 \hspace{1em} \((1b^d, ub^d) = (\sum_i T_i(A_i).lb + \Delta d, \sum_i T_i(A_i).ub + \Delta d)\)

18 \hspace{1em} UB := \text{min}(UB, ub^d)

19 \hspace{1em} \((T(A).lb, T(A).ub) = (\text{min}_d(1b^d), \text{min}_d(ub^d))\)


21 end

Algorithm 8: AND/OR Branch and Bound search with Component Caching.
6.7 Empirical Results

In this section we examine the empirical performance of Symmetry Templating. Symmetric Templating is tested on the Maximum Density Still Life problem. Since Symmetric Templates are closely related to Component Templates presented in Chapter 3, this experimental section repeats some of the results already presented in Chapter 3. The repetition is needed to properly evaluate the benefits provided from exploiting symmetry.

The following specific algorithms are tested:

1. Branch and Bound (BB) (see Algorithm 1 on page 20).

2. AND/OR (AND/OR), Algorithm 2. This version searches for components and solves them separately, i.e., it performs search with decomposition. However, AND/OR does not use templates, nor does it cache already solved components.

3. AND/OR with templates (AND/OR+T), Algorithm 3, is a template version of AND/OR. Its only improvement over AND/OR is the use of templates to improve component detection. There is no template cache to store bounds for the template instances.

4. AND/OR with Component Caching (AND/OR+CC), extends AND/OR+T by activating the template cache to store computed bounds for the template instances.

5. Symmetric Component Caching Search + Branch and Bound (AND/OR+SC) extends AND/OR+CC by performing automated symmetric detection between templates.

6. AND/OR+ASC which extends AND/OR+SC to exploit automorphisms as well as symmetries.

BB uses Dynamic Variable-Ordering, while the remaining algorithms use Partial Variable-Ordering (see Section 2.9 on page 37 for details). These six different algorithms...
all use the semantic heuristic Domain/Degree, which minimizes the size of the domain divided by the degree of the variable. All algorithms perform GAC constraint propagation on the hard objectives to prune domain values.

All experiments were run on a 2.2 GHz Pentium IV with 6GB of memory. The time limit for each run was set to 10,000 seconds. In the experiments conducted, the space used in caching never exceeded available memory, so it was not necessary to prune the cache during search.

6.7.1 Problem Set

The Maximum Density Still Life (MDSL) problem was introduced in Chapter 3 of this thesis. The problem involves finding maximum density stable configurations in Conway’s game of Life [66]. MDSL has an $N \times N$ grid of cells that can be either dead or alive, implicitly surrounded by a boundary of dead cells. A live cell is stable if and only if it is surrounded by 2 or 3 live cells, and a dead and stable if and only if it is surrounded by 0, 1, 4, 5, 6, 7, or 8 live cells. The goal is to find a stable configuration of the cells that has a maximum number of live cells. Details of the problem specification can be found in Section 3.5.1 on page 55.

6.7.2 MDSL results

Table 6.1 shows the performance of the six tested algorithms in terms of time in seconds, while Table 6.2 shows the performance of the six algorithms in terms of nodes expanded.

The experimental section of Chapter 3 discussed how using search with decomposition improves performance over B&B. AND/OR+SC further improves performance over simple decomposition, by conducting symmetry detection between templates. AND/OR+SC also provides a significant decrease in the size of the search space. AND/OR+ASC detects automorphisms within a single template, as well as detecting symmetries between templates, and provides another useful performance gain.
## Chapter 6. Symmetric Templating

On MDSL, automated symmetry detection does not add a significant overhead. The nodes/second search rate when search uses symmetry detection is only 5% lower than the search rate without symmetry detection on the largest problem.

### Table 6.1: Time taken in CPU seconds, fastest time in bold

<table>
<thead>
<tr>
<th>Size</th>
<th>AND/OR+ASC</th>
<th>AND/OR+SC</th>
<th>AND/OR+CC</th>
<th>AND/OR+T</th>
<th>AND/OR</th>
<th>BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.24</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0.2</td>
<td>0.5</td>
<td>1.1</td>
<td>2.0</td>
<td>2.4</td>
<td>2.0</td>
</tr>
<tr>
<td>7</td>
<td>3.4</td>
<td>6.5</td>
<td>8.5</td>
<td>19.8</td>
<td>41.8</td>
<td>55.1</td>
</tr>
<tr>
<td>8</td>
<td>13.8</td>
<td>29.3</td>
<td>82.3</td>
<td>316.6</td>
<td>408.8</td>
<td>2207</td>
</tr>
<tr>
<td>9</td>
<td>408.7</td>
<td>704.8</td>
<td>4662.4</td>
<td>&gt;10000</td>
<td>&gt;10000</td>
<td>&gt;10000</td>
</tr>
<tr>
<td>10</td>
<td>1540.5</td>
<td>3003</td>
<td>7845.4</td>
<td>&gt;10000</td>
<td>&gt;10000</td>
<td>&gt;10000</td>
</tr>
</tbody>
</table>

### Table 6.2: Nodes Expanded, fewest nodes in bold

<table>
<thead>
<tr>
<th>Size</th>
<th>AND/OR+ASC</th>
<th>AND/OR+SC</th>
<th>AND/OR+CC</th>
<th>AND/OR+T</th>
<th>AND/OR</th>
<th>BB</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>486</td>
<td>1076</td>
<td>1646</td>
<td>1646</td>
<td>1646</td>
<td>767</td>
</tr>
<tr>
<td>5</td>
<td>3991</td>
<td>7744</td>
<td>7744</td>
<td>8146</td>
<td>8146</td>
<td>5364</td>
</tr>
<tr>
<td>6</td>
<td>18299</td>
<td>41346</td>
<td>93335</td>
<td>206141</td>
<td>177540</td>
<td>177522</td>
</tr>
<tr>
<td>7</td>
<td>251522</td>
<td>492939</td>
<td>644175</td>
<td>1.5*10^6</td>
<td>2.1*10^6</td>
<td>4.7*10^6</td>
</tr>
<tr>
<td>8</td>
<td>9.7*10^6</td>
<td>2.0*10^6</td>
<td>6.1*10^6</td>
<td>2.4*10^7</td>
<td>2.3*10^7</td>
<td>3.1*10^8</td>
</tr>
<tr>
<td>9</td>
<td>1.5*10^7</td>
<td>3.1*10^7</td>
<td>9.7*10^7</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>10</td>
<td>5.8*10^7</td>
<td>1.2*10^8</td>
<td>5.6*10^8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
6.8 Related Work

The techniques presented in this chapter build on Puget’s technique for automatically detecting variable and value symmetries in CSPs [90]. Puget’s method creates a large, coloured graph representation of the constraint graph (see Section 2.8.1 on page 33 for details). This chapter modifies Puget’s algorithm to account for the special structure of templates used in COPs.

The method of detecting symmetries proposed in this chapter is related to a class of CSP algorithms that modify the search algorithm to avoid exploring symmetric sub-trees [43, 42, 39, 96, 51]. Typically, however, these approaches have very high overhead, which makes them impractical for use on many problems.

Symmetric Templating is also related to the algorithms presented in Chapters 3-5. The templating method presented in Chapter 3 forms the basis for Symmetric Component Templating. Templating is also used in the implementation of OR-Decomposition and Decomposition Bounding.

6.9 Future Work

Symmetric Templating could be adapted for CSPs. By storing no-goods rather than bound information, search using symmetric templating could avoid exploring sub-trees that cannot extend to a solution. In addition, the assignments of solutions could be stored and re-used.

The symmetry techniques discussed in this chapter can be applied to both OR-Decomposition and DB. When a call to getCache is made, OR-Decomposition and DB can attempt to find symmetric solutions, in addition to solutions to the exact component.
6.10 Conclusion

This chapter has presented Symmetric Component Templating. This method automatically detects symmetries between components at the abstract level of templates. In particular, symmetries are computed between templates during search. A symmetry between two templates then can be applied to all instances of the two templates; thus, a single symmetry computation can be used for many different components. Effectively, the cost of symmetry detection is amortized over all instances of a template. This amortization is key in making automatic symmetry detection cost effective.
Chapter 7

Set Branching

Chapter 6 of this thesis introduced an efficient algorithm for exploiting symmetries in COPs. However, many real-world problems have no, or very few, true symmetries. This chapter investigates the notion of almost-symmetry, where values behave in a similar, but not identical manner. This chapter explains the notion of almost-symmetry, shows how almost-symmetries can be detected, and presents an algorithm for exploiting almost-symmetries.

7.1 Introduction

Branch and Bound (B&B) is a standard technique for solving WCSPs using backtracking search (See Algorithm 1 on page 20 for details). As noted in Chapter 2, B&B works by building up partial variable assignments in a depth-first manner, and uses a bounding function to prune the search space. Each recursion is passed a WCSP, \( \mathcal{P} \), (a reduction of the original WCSP by the current set of assignments) and an upper-bound \( \text{ub} \). B&B tries to compute \( \text{mincost}(\mathcal{P}) \), subject to the condition that it can abort its computation as soon as it can conclude that \( \text{mincost}(\mathcal{P}) \geq \text{ub} \). However, if the variable’s domain of values is large, the nodes in the search space will have a high branching factor. Therefore, the explored search space will grow very quickly, even when efficient bounding
techniques are used.

**Domain Splitting** is commonly employed to address the problem of high branching factor during search [38]. In this approach, the unpruned values of the selected variable, \( V \), are first split into two sets based on a lexicographical order of the values. Then, instead of branching on \( V \)’s individual unpruned values, the search branches on these two sets, and the branching factor in the search space is reduced to two. The algorithm descends down the search tree either until a bound is violated, or the domain of every variable is reduced to a single value at a leaf node, in which case these values form a new best solution.

Domain Splitting alone does not reduce the size of \( \text{B&B} \)’s search space. It simply changes the shape of the search tree, making it narrower but potentially deeper. In particular, in the absence of pruning due to bounding, each leaf node generated by Domain Splitting lies in one to one correspondence with the leaf nodes generated by branching on individual values. Domain Splitting can only be effective when bounding is able to refute entire sets of values without having to test each value explicitly.

In this chapter, an algorithm called Set Branching is presented. Like Domain Splitting, Set Branching partitions the unpruned values of a variable into sets and then branches on these sets rather than on individual values. However, instead of arbitrarily splitting the variable’s domain into two sets based on lexicographic order, a principled technique is developed for clustering the values into sets of similar, or **almost-symmetric** values. The technique aims to construct sets, such that branching on these sets will generate improvements to the current bounds, similar to those that would be found by branching on single values. Thus, Set Branching works with bounding techniques to reduce the overall size of the **explored** search tree.
7.2 Background

Set Branching draws upon a number of well-known concepts and techniques, reviewed in this section. The Set Branching algorithm is defined in terms of the WCSP problem, defined in Section 2.3.2 on page 10.

A Domain Reduced WCSP, $\mathcal{P}|_{\text{Dom}^*}$, is defined by the tuple $\langle Vars, \text{Dom}^*, \text{Obj}^* \rangle$. Given a WCSP, $\mathcal{P} = \langle Vars, \text{Dom}, \text{Obj} \rangle$, $\mathcal{P}|_{\text{Dom}^*}$ is the reduced WCSP generated by reducing the variable domains of $\mathcal{P}$ to the new sets specified in Dom*, where $\text{Dom}^*[V] \subseteq \text{Dom}[V]$ for all $V \in Vars$. The variables, $Vars$, are the same as those in $\mathcal{P}$, but $\text{Dom}$ is replaced by $\text{Dom}^*$. $\text{Obj}^*$ includes the same objectives as $\text{Obj}$, but each objective is defined only over tuples of values contained in $\text{Dom}^*$.

As described in previous chapters, B&B is a standard technique for solving WCSPs using backtracking search. A global upper-bound, (GUB), is maintained, which represents the cost of the best known solution to the problem. At each node of the search tree, B&B selects an unassigned variable $V$ and branches on each unpruned value in $\text{Dom}[V]$. Branching on the individual values of a variable is called value branching. After assigning a value to $V$, B&B calculates a lower-bound on the cost of any complete assignment extending the current assignment. If the bound is greater than GUB, B&B backtracks, since no optimal solution can be found in the sub-tree below the current assignment.

7.3 Domain Splitting and Set Branching

When the variables of the WCSP have large domains, the branching factor of B&B’s search tree becomes large. Even when efficient bounding techniques are used, the search space explored by B&B often becomes impractically large. In this section, the Set Branching algorithm and its precursor Domain Splitting are described. These algorithms aim to reduce the search space’s branching factor.

Domain Splitting is a technique for solving WCSPs based on B&B [38]. At each node
of the search tree, Domain Splitting selects a variable $V$, and partitions $V$’s unpruned values into two sets, $L$ and $R$, based on the lexicographical order of the values. That is, all values in the set $L$ are lexicographically less than any value in the set $R$. To test whether or not $L$ leads to an optimal solution, a domain reduced WCSP is created by pruning the values of $R$ from $V$’s domain. If the lower-bound of the domain reduced WCSP is larger than $gub$, then no optimal solution can exist using any value in $L$. If the bound is not exceeded, search descends down the search tree until either a bound is violated, or the domain of every variable is reduced to one, in which case a new best solution is found. After testing $L$, the algorithm tests $R$ in the same manner.

**Set Branching** (sometimes referred to as weak-assignments) is a technique developed for Constraint Satisfaction Problems (CSP) that partitions the values of a domain into sets, and branches on each set in a fashion similar to that employed by domain splitting [65, 103]. The approach presented here differs from this previous work in two ways. The most important difference is that previous work cited above has utilized problem-specific algorithms for partitioning the values into sets. Here, a generic algorithm for partitioning the values into sets is presented. The second difference is that previous work has addressed satisfaction problems (CSPs) not optimization problems (COPs). As will be described below, the adaptation from CSPs to COPs has a significant effect on the design of a generic algorithm for partitioning the variable domains into sets.

Consider the WCSP, $\mathcal{P} = \langle Vars, Dom, Obj \rangle$. The standard **B&B** algorithm first selects a variable, $V$, and then branches on each unpruned value, $a \in Dom[V]$. Suppose that instead of branching on each value, $a \in Dom[V]$, the unpruned values of $Dom[V]$ are partitioned into $k$ disjoint sets, $S = [S_1, \ldots, S_k]$, where $\bigcup_{i=1}^{k} S_i = Dom[V]$. $S$ is called a **branching set** for $V$. Set Branching branches on a set, $S_i$, by pruning the values in all other sets, $S_j | i \neq j$, from $Dom[V]$.

Algorithm 9 shows the Set Branch (SB) algorithm. **SB** is a recursive function with one parameter, CurrentDom. CurrentDom is a vector over all variables in the WCSP,
Chapter 7. Set Branching

1 \texttt{SB}\texttt{(CurrentDom)}

\begin{verbatim}
/* If mincost(\mathcal{P}|_{CurrentDom}) < GUB, then GUB will be updated to be 
mincost(\mathcal{P}|_{CurrentDom}) */
\end{verbatim}

2 \begin{algorithm}
3 \begin{align*}
4 & \text{if getBounds(C currentDom) } \geq \text{ GUB then} \\
5 & \quad \text{return} \\
6 & \text{if } \forall V, |\text{CurrentDom}[V]| = 1 \text{ then} \\
7 & \quad \text{GUB} := \text{getBounds(C currentDom)} \\
8 & \quad \text{return} \\
9 & \text{choose (a variable } V \text{ where } |\text{CurrentDom}[V]| \geq 2) \\
10 & \text{S } = \text{ partition CurrentDom}[V] \text{ into } k \text{ sets, } k \geq 2 \\
11 & \text{foreach } S_i \in S \text{ do} \\
12 & \quad \text{CurrentDom}[V] = S_i \\
13 & \quad \text{SB(C currentDom)}
\end{align*}
\end{algorithm}

Algorithm 9: Set Branch Algorithm

where \text{CurrentDom}[V] is the current domain of variable \text{V}. If \text{|CurrentDom}[V]| = 1, \text{then V is considered to be assigned the single value in CurrentDom}[V], otherwise \text{V} is unassigned. The vector \text{CurrentDom} defines a domain reduced WCSP, \mathcal{P}|_{\text{CurrentDom}}. \text{SB} immediately backtracks if it determines that \mathcal{P}|_{\text{CurrentDom}} has no solution with cost lower than \text{GUB}. This condition is detected by a call to the function getBounds(\text{CurrentDom}), which calculates a lower-bound for \mathcal{P}|_{\text{CurrentDom}}.

If the domain size for every \text{V} \in \text{Vars} is 1, then every variable has been assigned a value, and the search is at a leaf node. \text{SB} then updates the \text{GUB} and backtracks. In this case, getBounds is called on singleton domains, and will return the exact value of the corresponding complete assignment. Otherwise, a variable is selected from the set of
unassigned variables; that is, from amongst those variables $V$ with more than one value in $\text{CurrentDom}[V]$.

Line 9 partitions the values in $\text{CurrentDom}[V]$ into $k$ disjoint sets, where $k \geq 2$. By forcing $k$ to be greater than one, $\text{SB}$ ensures that the domain of at least one variable is reduced in each recursive call; thus, $\text{SB}$ will eventually exceed the GUB or reach a leaf. In Lines 10–12, $\text{SB}$ loops through the $k$ disjoint sets, and recursively calls $\text{SB}$ for each. The loop is iterated $k$ times, where $k$ is the number of sets created in Line 9. In standard B&B, there is an equivalent loop which is iterated $|\text{Dom}[V]|$ times (see Algorithm 1 on page 20). Thus, $\text{SB}$ reduces the branching factor of the node from $|\text{Dom}[V]|$ to $k$.

In order to attain a net reduction in the search space it is not sufficient simply to reduce the branching factor. Search must also be able to backtrack from a set of values, $S_i$, without having to value branch on each value in $S_i$ at a deeper node in search. Backtracking from a set of values is only possible if the lower-bound returned by $\text{getBounds}$ is greater than or equal to GUB.

### 7.4 Obtaining Good Bounds

To obtain bounds of sufficient strength to backtrack from a set of values, it is important to examine in detail the way in which set branching affects the bounds of a reduced problem. Consider two values, $a$ and $b$ in $\text{CurrentDom}[V]$. The assignments $V \leftarrow a$ and $V \leftarrow b$ generate two different reduced WCSPs, $P_a$ and $P_b$. When standard value branching is used, $\text{getBounds}$ will be applied to compute bounds for these two reduced WCSPs. If $P_a$ and $P_b$ are structurally similar, then any well behaved $\text{getBounds}$ function will compute similar bounds for both problems irrespective of the specific bounding technique the function uses.

Intuitively, if $P_a$ and $P_b$ are similar, and thus, generate similar bounds, then $a$ and $b$ should not be distinguished. Both values are almost-symmetric and hence, should
be placed in the same set. When \textbf{SB} branches on $S_i$, the resultant reduced WCSP, $P_{(a,b)}$, likely will be structurally similar to both $P_a$ and $P_b$. Therefore, \textit{getBounds} should generate similar bounds for $P_{(a,b)}$ as would have been generated for either of $P_a$ and $P_b$. Thus, similar bounds can be obtained after branching on $S_i$, compared to the two value branches on $a$ and $b$. Therefore, the approach identifies values in $V$’s domain that would be treated similarly by \textit{getBounds}, and places these similar values into the same set during \textbf{SB}.

In order to formalize these intuitions into a specific algorithm, we must focus on the details of a particular technique for implementing \textit{getBounds}. Here, the technique of Existential Directional Arc Consistency (EDAC) \cite{28}, a powerful technique often employed to implement \textit{getBounds} when solving WCSPs, is examined. The EDAC technique, described fully in Section 2.6.1 on page 21, is reprized below.

Let $o_{ij}$ denote a binary objective between the variables $V_i$ and $V_j$, where $o_{ij}(a,b)$ is the cost of assigning $a$ to $V_i$ and $b$ to $V_j$. Let $o_i$ denote a unary objective for variable $V_i$, where $o_i(a)$ is the cost of assigning $a$ to $V_i$. If the WCSP does not initially specify a unary objective, $o_i$, for variable $V_i$, then a unary objective, $o_i$, is added to the WCSP mapping all values of $V_i$ to zero. Finally, a zero arity lower-bound objective, $O_{\emptyset}$, is also added to the WCSP with initial value zero. When using EDAC for lower-bounding a WCSP, \textit{getBounds} simply returns the current value of $O_{\emptyset}$.

The EDAC algorithm modifies a WCSP into an equivalent problem by flowing value towards $O_{\emptyset}$. The algorithm ensures that $O_{\emptyset}$ always remains a lower-bound on the minimal value that the WCSP can achieve. In addition to generating a lower-bound, EDAC also can remove infeasible values from the variable domains, thus improving the efficiency of \textbf{B&B}’s search.

The method uses two atomic transformations, both of which are guaranteed to preserve equivalence. The first transformation is a unary transformation. Let $\alpha$ be the minimum cost $o_i(a)$ for every $a \in \text{Dom}[V_i]$. If $\alpha > 0$, then $\alpha$ can be added to $O_{\emptyset}$, and
subtracted from every $o_i(a), a \in Dom[V_i]$.

The second transformation is a binary transformation. Consider two objectives, $o_{ij}$ and $o_i$ and a value, $a \in Dom[V_i]$. Let $\alpha$ be the minimum cost of $o_{ij}(a, b)$ over all $b \in Dom[V_j]$. If $\alpha > 0$, then $\alpha$ can be added to $o_i(a)$ and subtracted from $o_{ij}(a, b)$ for all $b \in Dom[V_j]$. This transformation is called a flow into $o_i(a)$. If $\alpha < 0$, then $\alpha$ can be subtracted from $o_i(a)$ and added to $o_{ij}(a, b)$ for all $b \in Dom[V_j]$. This transformation is called a flow out of $o_i(a)$.

The EDAC algorithm can be viewed as a two stage process. The first stage, called simplification, occurs when the domain of a variable, $V_i$, is reduced at a node by branching. During simplification, there is a unary transformation from $o_i$ to $O_\emptyset$. In addition, for every objective $o_{ij}$, $\forall b \in Dom[V_j]$, there is a binary transformation from $o_{ij}(a, b)$ into $o_j(b)$. The second stage, called propagation, repeatedly applies unary and binary transformations to the simplified problem, with the goal of flowing cost into $O_\emptyset$ where it can returned to the search by getBounds.

When value assignments are used, the simplification stage often significantly increasing the bounds of the reduced problem. When the assignment $V_i \leftarrow a$ is made, the domain of $V_i$ is reduced to a single value. In this case, the entire cost of $o_i(a)$ is immediately added to $O_\emptyset$. In addition, the entire cost of the edge $o_{ij}(a, b)$ is added to $o_j(b)$ for each $b \in Dom[V_j]$. These large binary transformations often allow additional binary and unary transformations that further increase $O_\emptyset$ during the propagation stage.

However, suppose that instead of reducing a variable domain to a single value, as when value branching is employed, the domain of a variable is simply reduced, as when set branching is employed. In this case, the simplification stage may not increase the bounds on the reduced problem in the same way that value branching does. For example, consider the case where the domain of $V_i$ is reduced to two values, $a$ and $b$, with unary costs $o_i(a) = 10$ and $o_i(b) = 0$. In this case, no unary transformation can occur during simplification, since $\alpha$, the minimum cost in $o_i$, is not greater than zero. In this case,
Chapter 7. Set Branching

Figure 7.1: Objectives $o_i$, $o_j$, and $o_{ij}$.

$getBounds$ would compute two very different values if value assignments had been made, rather than set branching.

Since analyzing the effects of propagation on different values is complex, and likely to be computationally expensive, we focus instead on analyzing the effects of simplification on the reduced WCSP. This analysis will give us partial information about how different values would be treated by $getBounds$.

**Simplification and Set Branching.** Suppose variable $V_i$ is selected for set branching, $Dom[V_i] = \{a, b, c, d\}$, and the WCSP, $P$, has the objective functions represented in Figure 7.1. In the diagram, edges between two values indicate the cost of $o_{ij}$ on that pair of values, while omitted edges indicate that $o_{ij}$ has zero cost on that pair. The unary objectives are represented as the costs listed with each value.

Figure 7.2 shows the WCSPs that are created after making value assignments $V_i \leftarrow a$, $V_i \leftarrow b$, $V_i \leftarrow c$, and $V_i \leftarrow d$, and then applying the simplification stage of EDAC to the problem. Simplification applies a unary transformation, which will increase the lower-bound objective, $O_\emptyset$ by 10 (in the case of $V_i \leftarrow a$), 11 (in the case of $V_i \leftarrow b$), 0 (in the case of $V_i \leftarrow c$), and 1 (in the case of $V_i \leftarrow d$). In addition, a binary transformation is made during simplification, which increases $o_j(a)$ by 5 (in the case of $V_i \leftarrow c$), and 6 (in the case of $V_i \leftarrow d$). Figure 7.2 shows that the WCSPs created after assigning $V_i \leftarrow a$ and $V_i \leftarrow b$ and then performing simplification are similar. In addition, the WCSPs created
after assigning $V_i \leftarrow c$ and $V_i \leftarrow d$ and then performing simplification are similar.

The fact that the WCSPs created after assigning $V_i \leftarrow a$ and $V_i \leftarrow b$ are similar suggests that set branching on \{a, b\} will generate a WCSP similar to the two WCSPs created after assigning $V_i \leftarrow a$ and $V_i \leftarrow b$. The result of simplification after branching on these two sets, $S_{i1} = \{a, b\}$ and $S_{i2} = \{c, d\}$, is shown in Figure 7.3. The figure shows that in this example, the intuition is correct, and set branching on \{a, b\} generated a WCSP similar to the two WCSPs created after assigning $V_i \leftarrow a$ and $V_i \leftarrow b$.

Because the propagation phase of EDAC operates on a problem after simplification, propagation is expected to generate similar increases to the lower-bound, $O_\emptyset$, on similar simplified problems. However, small difference in the simplified problem can turn out to be profound differences after propagation. Unfortunately, analyzing the effects of propagation on different values is likely to be computationally expensive. Therefore, we restrict our attention to the similarity of problems after simplification, which is a reasonable heuristic that is validated by the experimental results in Section 7.5.

Sets of values are similar if the transformations allowed during the simplification
Stage are similar, regardless of whether set branching or value branching is used on the values in a set. The **set branching ratio** is the ratio of the number of partitions to the number of values in the domain of the variables. **SB** is most effective when the set branching ratio at a node is low while the similarity is high.

### 7.4.1 Calculating Effective Branching Sets

Calculating a branching set that has a large similarity score and a small set branch ratio is extremely important for set branching to be effective. This section develops an heuristic algorithm that clusters values into sets while attempting to achieve high similarity.

For each variable, $V_i$, the clustering algorithm builds a dissimilarity matrix, $\text{Dis}$, where $\text{Dis}(a, b)$ represents the dissimilarity between value $a$ and value $b$, after simplification on the respective reduced problems takes place. **SB** uses this matrix to cluster the values of the variable into sets by trying to place highly dissimilar values into different sets. Each cell, $\text{Dis}(a, b)$, of the dissimilarity matrix is initialized to zero and then is updated with the following rules:

**Rule 1.**

\[
\text{difference} := \text{abs}(o_i(a) - o_i(b))
\]

\[
\text{Dis}(a,b) := \text{Dis}(a,b) + \text{difference}
\]

\[
\text{Dis}(b,a) := \text{Dis}(b,a) + \text{difference}
\]
Rule 1 updates $\text{Dis}(a, b)$ if $o_i(a) \neq o_i(b)$. Consider the following example: let $o_i(a) > o_i(b)$, let $\alpha = o_i(a) - o_i(b)$, and suppose the clustering algorithm places $a$ in $S_q$ and $b$ in $S_r$. Suppose also that $SB$ branches on $S_q$. If $q = r$ (i.e., $a$ and $b$ are placed in the same set), then the maximum unary transformation from $o_i$ into $O_\emptyset$ is $o_i(b)$. If $q \neq r$, then the maximum unary transformation from $o_i$ into $O_\emptyset$ is $o_i(a)$. Thus, the unary transformation allowed if $q \neq r$ may be up to $\alpha$ greater than if $q = r$. This unary transformation is not assured, however. Even if $q \neq r$ and $a$ and $b$ are placed in different sets, a third value, $c \in S_q$, may stop any unary transformation from $o_i$.

**Rule 2.**

\begin{verbatim}
foreach $V_j \in \text{Vars}$ such that $o_{ij}$ is a binary objective

foreach $k \in \text{Dom}[V_j]$

difference := $\text{abs}(o_{ij}(a, k) - o_{ij}(b, k))$

$\text{Dis}(a, b) = \text{Dis}(a, b) + \text{difference}/|\text{Dom}[V_j]|$

$\text{Dis}(b, a) = \text{Dis}(b, a) + \text{difference}/|\text{Dom}[V_j]|$

\end{verbatim}

Rule 2 updates the dissimilarity matrix of $V_i$ for each variable, $V_j$, that appears in a binary objective, $o_{ij}$. For each value $k \in \text{Dom}[V_j]$, it may be possible to perform a binary transformation into $o_j(k)$. Rule 2 evaluates the difference between this transformation for two values, $a$ and $b$, in $\text{Dom}[V_j]$.

Consider the following example: let $o_{ij}(a, k) > o_{ij}(b, k)$; Let $\alpha = o_{ij}(a, k) - o_{ij}(b, k)$; and, suppose that the clustering algorithm places $a$ in $S_q$ and $b$ in $S_r$. Suppose also that $SB$ branches on $S_q$. Consider the binary transformations made by the simplification stage of EDAC. If $q = r$ (i.e., $a$ and $b$ are placed in the same set), then the maximum binary transformation from $o_{ij}$ into $o_j(k)$ is $o_{ij}(b, k)$. If $q \neq r$, then the maximum binary transformation from $o_{ij}$ into $o_j(k)$ is $o_{ij}(a, k)$. Thus, the binary transformation allowed when $q \neq r$ may be up to $\alpha$ greater than the transformation allowed when $q = r$. As with rule 1, this binary transformation is not assured even if $a$ and $b$ are placed in different sets.
Figure 7.4: Dissimilarity matrix of Figure 7.2

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.0</td>
<td>1</td>
<td>12.5</td>
<td>12</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0.0</td>
<td>13.5</td>
<td>13</td>
</tr>
<tr>
<td>c</td>
<td>12.5</td>
<td>13.5</td>
<td>0.0</td>
<td>1.5</td>
</tr>
<tr>
<td>d</td>
<td>12</td>
<td>13</td>
<td>1.5</td>
<td>0.0</td>
</tr>
</tbody>
</table>

The total value of $\text{Dis}(a,b)$ is computed by applying rule 1 to every pair of values in $\text{Dom}[V_i]$, and rule 2 to every pair of values in $\text{Dom}[V_i]$, and to every value of $V_j$ for all $V_j$ that appear in a binary objective with $V_i$. For example, consider the WCSP represented by Figure 7.1. The value of $\text{Dis}(a,c)$ is 12.5 (rule 1 increments $\text{Dis}(a,c)$ by 10 and rule 2 increments $\text{Dis}(a,c)$ by 2.5). The complete dissimilarity matrix generated is shown in Figure 7.4.

Rules 1 and 2 are for WCSPs with unary and binary objectives. Rule 3 is used for WCSPs with ternary objectives. Under EDAC, ternary objectives do not contribute to bound calculations until one of the variables of the objective is assigned a value. When a variable is assigned a value, the ternary objective is projected to a binary objective, and EDAC is enforced on this binary objective. Based on this behavior, the following rule is used to update the dissimilarity matrix of $V_i$ for every ternary constraint with $V_i$ in its scope.

Rule 3.

\begin{verbatim}
foreach $V_j, V_k \in \text{Vars}$ such that $o_{ijk}$ is a ternary objective

foreach $x \in \text{Dom}[V_j]$

foreach $y \in \text{Dom}[V_k]$

difference := $abs(o_{ijk}(a, x, y) - o_{ijk}(b, x, y))$

$\text{Dis}(a,b) = \text{Dis}(a,b) + \text{difference/}(|\text{Dom}[V_j]| \cdot |\text{Dom}[V_k]|)$

$\text{Dis}(b,a) = \text{Dis}(b,a) + \text{difference/}(|\text{Dom}[V_j]| \cdot |\text{Dom}[V_k]|)$
\end{verbatim}
Rule 3 updates the dissimilarity matrix of $V_i$ for each pair of variables $V_j, V_k$ that appears in a ternary objective $o_{ijk}$. Suppose $V_j$ is assigned the value $x$ during search. A binary objective, $o_{ik}$, is created by projecting the objective $o_{ijk}$ based on the assignment $V_j = x$. Then, similar to rule 2, for each value $y \in \text{Dom}[V_k]$, it may be possible to perform a binary transformation into $o_k(y)$. The rule evaluates the difference between this transformation for two values $a$ and $b$ in $\text{Dom}[V_i]$.

Algorithm 10, MakeCluster, uses the computed dissimilarity matrix to cluster the values into disjoint sets. MakeCluster uses a global parameter, $\lambda$, which is a threshold parameter. The input to the MakeCluster is the set, ValueSet, initially including all values in CurrentDom[$V_i$].

```
1 MakeCluster (ValueSet)
2 begin
3     if |ValueSet| = 1 then
4         addSetToFinalList(ValueSet) return
5     (Set1, Set2, score) := valuecluster(ValueSet)
6     if score > $\lambda$ then
7         addSetToFinalList(ValueSet)
8         return
9     MakeCluster (Set1)
10    MakeCluster (Set2)
11 end
```

Algorithm 10: Make Cluster Algorithm

MakeCluster calls the valuecluster function, which splits ValueSet into two sets, Set1 and Set2 and returns a score. If the score is above $\lambda$, then the algorithm assumes the values in Set1 and Set2 are similar to one another and adds ValueSet to the final list of sets that SB will use to set branch. If the score is below $\lambda$, MakeCluster is called
recursively on Set1 and Set2 in order to further subdivide the sets until they become singleton sets or are added to the final list of sets.

Each row and each column in the dissimilarity matrix represents a single value in $Dom[V_i]$. The valuecluster function first eliminates all rows and columns from the dissimilarity matrix that do not represent a value in ValueSet.

Valuecluster then calculates a row-score for each row, $s$, as follows. The cells of the row are partitioned into two clusters (high and low) based on the cell-score (since the matrix is symmetric, using columns instead of rows would generate the same results). The clustering is done using a bimodal thresholding technique [84]. The mean of the high cluster is the average cell-score of cells in the high cluster and the mean of the low cluster is the average cell-score of cells in the low cluster. The row-score for row $s$ is equal to the mean of the low cluster divided by the mean of the high cluster. As the ratio approaches one, the values are presumed to be more similar.

After calculating the row-score for every row, valuecluster selects the row with the lowest row-score. It then returns two sets; the first set contains the values represented by the cells in the high cluster of the selected row, while the second set contains the values represented by the cells in the low cluster of the selected row. Valuecluster also returns the row-score of the selected row. If the row-score is less than some threshold percentage $\lambda$, then the partition is accepted and the algorithm is called recursively. Otherwise, the partition is not accepted (meaning that the values in the set are deemed to be similar to one another), ValueSet is added to the completed set, and MakeCluster returns.

When only unary and binary constraints are considered, MakeCluster takes time $O(nd^3)$ at every node where $n$ is the number of variables, and $d$ is the maximum domain size over all variables. This high cost makes it disadvantageous to run MakeCluster at every node. In the SB implementation, the algorithm is run as a preprocessing step for every variable of the problem, and the resulting sets are stored. After a variable is selected in Line 8 of SB, if the variable’s domain is greater than some threshold size, the
Chapter 7. Set Branching

stored set is used to partition the remaining domain. If the variable’s domain is lower than this threshold, value branching is used.

7.5 Experimental Results

This section reviews the results of testing Branch and Bound, Domain Splitting, and Set Branching on WCSP, MPE, and MAX-CSP problems.

7.5.1 Anti-MakeCluster

As part of the empirical testing of the effectiveness of this method for computing branching sets, an anti-algorithm was developed. The Anti-MakeCluster reverses the intuitions used in developing MakeCluster. Specifically, the anti-algorithm computes its branching sets by first running MakeCluster to compute a branching set \( S = S_1, ..., S_k \); then, it splits each set, \( S_i \in S \), in half, which results in sets \( T = T_1, ..., T_{2k} \). Finally, Anti-MakeCluster creates sets \( S' = S'_1, ..., S'_k \), where each set, \( S'_i \), is the union of two randomly selected sets, \( T_a, T_b \in T \) (each set \( T_i \) is included in exactly one of the newly defined sets \( S'_i \)).

Intuitively, the anti-algorithm will create branching sets where half of the values are dissimilar to the other half. If the intuition behind MakeCluster is correct, then Set Branching will not be able to find strong bounds early in the search if Anti-MakeCluster is used.

7.5.2 Algorithms Tested

All of the algorithms tested in this chapter were implemented on top of the Toolbar system [16], and all employ the EDAC. The following specific algorithms are tested:

1. BB — Branch and Bound (Algorithm 1 on page 20);
2. **DS** — Domain Splitting, which splits the variable domains in half based on the lexicographical order of the values in the original encoding;

3. **SB** — Set Branching, that utilizes **MakeCluster** to cluster the variable values into sets;

4. **SB-Anti** — Set Branching, that utilizes **Anti-MakeCluster** to cluster the variable values into sets.

The algorithms are tested with the Jersolow heuristic, in which the next variable chosen is the uninstantiated variable with highest current domain size, divided by its weighted degree. The weighted degree is the weighted sum of the objectives that the variable appears, where each objective is weighted by its dynamic expected value.

Once a variable is chosen, each algorithm attempts to solve its values in an order determined by the variable’s unary objective. That is, the values for variable $V_i$ are ordered by lowest unary cost, $o_i(V_i)$, as computed by local propagation. In the case of **DS**, **SB**, and **SB-Anti**, the sets are ordered by the minimum unary cost of the values in the set. Ties are broken by selecting the smallest set first. **DS**, **SB**, and **SB-Anti** revert to standard value branching when the domain size of the variable is smaller than 1/6 the size of the original variable domain.

All experiments were run with 1200 second timeouts, and were conducted on 2.66GHz machines with 8GB of memory. If a problem was not solved to optimality in 1200 seconds, the term **DNF** (Did Not Finished) is used. If a problem was not solved in 1200 seconds by any of the algorithms, no results are reported.

### 7.5.3 Problem Sets

The algorithms are tested on the following problem sets:

The **Radio Link Frequency Assignment Problem** (RLFAP) problems involve assigning frequencies to a set of radio links in such a way that the links can operate to-
gethether without noticeable interference. The RLFAP instances are cast as binary WCSPs [18]. The domain size of the variables typically range from 20 to 44. The benchmark family includes 46 problems.

The Pedigree problems involve determining if a Mendelian error exists in the pedigree. The Pedigree instances are cast as WCSPs [99]. The instances contain both binary and ternary objectives. The domain size of variables range from 10 to 20. The benchmark family includes 19 problems.

Random MAX-CSP are randomly generated binary MAX-CSPs cast as WCSPs. The problems consist of 15 variables, each with a domain size of 10. Each problem is a clique of binary objectives. For every objective, \( o_{ij} \), seven randomly selected tuples have cost zero, while the remaining tuples have cost one.

7.5.4 RLFAP results

Table 7.1 shows the time results of four different algorithms on the RLFAP problems. The total time includes all preprocessing time. The preprocessing time is negligible for SB, totaling only 6.3 seconds over all 21 instances.

The results show that of the four algorithms, SB performs the best, solving 14 out of 21 problems faster than the other algorithms. SB is never significantly slower than B&B or DS, except on problem CELAR7-SUB3, where SB takes over 100 seconds longer than DS. Finally, SB is able to solve problem CELAR6-SUB4 on which the other algorithms failed. SB-Anti was the slowest algorithm on all but four of the 21 instances. In sum, on these problems, B&B ran for a total of 3894 seconds and solved 19 problems; DS ran for a total of 4567 seconds and solved 19 problems; SB-Anti ran for 4953 seconds and solved 19 problems; and, SB ran for a total of 2382 seconds and solved 21 problems.

Table 7.2 shows the nodes expanded for the three different algorithms on the RLFAP problems. The results are similar to the results from Table 7.1, except on the smaller
<table>
<thead>
<tr>
<th>Instance</th>
<th>PreBB</th>
<th>PreSB</th>
<th>B&amp;B</th>
<th>DS</th>
<th>SB-Anti</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELAR6-SUB0</td>
<td>0</td>
<td>0.07</td>
<td>0.16</td>
<td>0.22</td>
<td>0.58</td>
<td>0.22</td>
</tr>
<tr>
<td>CELAR6-SUB1-24</td>
<td>0</td>
<td>0.02</td>
<td>2.66</td>
<td>2.02</td>
<td>5.83</td>
<td>3.18</td>
</tr>
<tr>
<td>CELAR6-SUB1</td>
<td>0</td>
<td>0.11</td>
<td>42.47</td>
<td>35.92</td>
<td>47.22</td>
<td>16.5</td>
</tr>
<tr>
<td>CELAR6-SUB2</td>
<td>0.01</td>
<td>0.13</td>
<td>16.05</td>
<td>36.39</td>
<td>32.54</td>
<td>13.49</td>
</tr>
<tr>
<td>CELAR6-SUB3</td>
<td>0</td>
<td>0.14</td>
<td>229.4</td>
<td>203.76</td>
<td>492.49</td>
<td>63.68</td>
</tr>
<tr>
<td>CELAR6-SUB4-20</td>
<td>0</td>
<td>0.02</td>
<td>10.44</td>
<td>3.3</td>
<td>21.67</td>
<td>2.75</td>
</tr>
<tr>
<td>CELAR6-SUB4reduc</td>
<td>0.01</td>
<td>0.15</td>
<td>DNF</td>
<td>791.28</td>
<td>DNF</td>
<td>695.22</td>
</tr>
<tr>
<td>CELAR6-SUB4</td>
<td>0</td>
<td>0.18</td>
<td>DNF</td>
<td>DNF</td>
<td>DNF</td>
<td>519.37</td>
</tr>
<tr>
<td>CELAR7-SUB0</td>
<td>0.01</td>
<td>0.09</td>
<td>0.06</td>
<td>0.13</td>
<td>0.27</td>
<td>0.13</td>
</tr>
<tr>
<td>CELAR7-SUB1-20</td>
<td>0</td>
<td>0.02</td>
<td>0.14</td>
<td>0.6</td>
<td>0.3</td>
<td>0.14</td>
</tr>
<tr>
<td>CELAR7-SUB1</td>
<td>0</td>
<td>0.11</td>
<td>4.96</td>
<td>3.91</td>
<td>10.44</td>
<td>3.71</td>
</tr>
<tr>
<td>CELAR7-SUB2</td>
<td>0</td>
<td>0.12</td>
<td>42.46</td>
<td>36.01</td>
<td>82.78</td>
<td>33.65</td>
</tr>
<tr>
<td>CELAR7-SUB3</td>
<td>0</td>
<td>0.16</td>
<td>460.28</td>
<td>283.93</td>
<td>594.13</td>
<td>393.37</td>
</tr>
<tr>
<td>CELAR7-SUB4-22</td>
<td>0</td>
<td>0.03</td>
<td>6.71</td>
<td>7.72</td>
<td>11.1</td>
<td>5.17</td>
</tr>
<tr>
<td>graph07</td>
<td>0.01</td>
<td>0.46</td>
<td>24.99</td>
<td>24.99</td>
<td>21.6</td>
<td>13.96</td>
</tr>
<tr>
<td>graph12</td>
<td>0.03</td>
<td>0.85</td>
<td>11.02</td>
<td>29.04</td>
<td>21.31</td>
<td>9.26</td>
</tr>
<tr>
<td>graph13reducmore</td>
<td>0.09</td>
<td>1.25</td>
<td>212.19</td>
<td>248.92</td>
<td>375.47</td>
<td>258.2</td>
</tr>
<tr>
<td>graph13reduc</td>
<td>0.09</td>
<td>1.24</td>
<td>210.57</td>
<td>242.55</td>
<td>371.49</td>
<td>255.72</td>
</tr>
<tr>
<td>scen06-30reduc</td>
<td>0</td>
<td>0.01</td>
<td>211.36</td>
<td>120.3</td>
<td>451.81</td>
<td>89.04</td>
</tr>
<tr>
<td>scen09</td>
<td>0.03</td>
<td>0.57</td>
<td>7.45</td>
<td>96.62</td>
<td>9.39</td>
<td>5.14</td>
</tr>
<tr>
<td>scen10</td>
<td>0.02</td>
<td>0.57</td>
<td>0.69</td>
<td>DNF</td>
<td>2.7</td>
<td>1.31</td>
</tr>
<tr>
<td>Total</td>
<td>0.3</td>
<td>6.3</td>
<td>&gt;3894</td>
<td>&gt;4567</td>
<td>&gt;4953</td>
<td>2383</td>
</tr>
</tbody>
</table>

Table 7.1: RLFAP Problems Preprocessing Time and Total Time in Seconds, best time in bold.
<table>
<thead>
<tr>
<th>Instance</th>
<th>B&amp;B</th>
<th>DS</th>
<th>SB-Anti</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELAR6-SUB0</td>
<td>6819</td>
<td>9159</td>
<td>1612</td>
<td>4659</td>
</tr>
<tr>
<td>CELAR6-SUB1-24</td>
<td>38256</td>
<td>20434</td>
<td>42910</td>
<td>35198</td>
</tr>
<tr>
<td>CELAR6-SUB1</td>
<td>290457</td>
<td>225403</td>
<td>124090</td>
<td>77113</td>
</tr>
<tr>
<td>CELAR6-SUB2</td>
<td>56103</td>
<td>137776</td>
<td>56904</td>
<td>34287</td>
</tr>
<tr>
<td>CELAR6-SUB3</td>
<td>913064</td>
<td>896552</td>
<td>1136575</td>
<td>168918</td>
</tr>
<tr>
<td>CELAR6-SUB4-20</td>
<td>158892</td>
<td>29264</td>
<td>180359</td>
<td>27881</td>
</tr>
<tr>
<td>CELAR6-SUB4reduc</td>
<td>3625398</td>
<td>1380223</td>
<td>1843528</td>
<td>1500872</td>
</tr>
<tr>
<td>CELAR6-SUB4</td>
<td>3351195</td>
<td>5798018</td>
<td>3627138</td>
<td>1304320</td>
</tr>
<tr>
<td>CELAR7-SUB0</td>
<td>2355</td>
<td>2516</td>
<td>2628</td>
<td>925</td>
</tr>
<tr>
<td>CELAR7-SUB1-20</td>
<td>3988</td>
<td>15319</td>
<td>6226</td>
<td>2944</td>
</tr>
<tr>
<td>CELAR7-SUB1</td>
<td>21558</td>
<td>13390</td>
<td>33668</td>
<td>13995</td>
</tr>
<tr>
<td>CELAR7-SUB2</td>
<td>140424</td>
<td>68872</td>
<td>152274</td>
<td>80468</td>
</tr>
<tr>
<td>CELAR7-SUB3</td>
<td>1832809</td>
<td>597584</td>
<td>922004</td>
<td>1210541</td>
</tr>
<tr>
<td>CELAR7-SUB4-22</td>
<td>83468</td>
<td>65414</td>
<td>69786</td>
<td>55588</td>
</tr>
<tr>
<td>graph07</td>
<td>615917</td>
<td>3243509</td>
<td>249615</td>
<td>111204</td>
</tr>
<tr>
<td>graph12</td>
<td>372122</td>
<td>771185</td>
<td>258552</td>
<td>176522</td>
</tr>
<tr>
<td>graph13reducmore</td>
<td>51694</td>
<td>50210</td>
<td>52388</td>
<td>57915</td>
</tr>
<tr>
<td>graph13reduc</td>
<td>51696</td>
<td>50212</td>
<td>52390</td>
<td>57917</td>
</tr>
<tr>
<td>scen06-30reduc</td>
<td>1897300</td>
<td>570362</td>
<td>1644211</td>
<td>536472</td>
</tr>
<tr>
<td>scen09</td>
<td>441349</td>
<td>5021050</td>
<td>109347</td>
<td>136539</td>
</tr>
<tr>
<td>scen10</td>
<td>16897</td>
<td>5823647</td>
<td>6463</td>
<td>3040</td>
</tr>
</tbody>
</table>

Table 7.2: RLFAP Problems: Nodes Expanded, fewest nodes in bold
<table>
<thead>
<tr>
<th>Instance</th>
<th>PreBB</th>
<th>PreSB</th>
<th>B&amp;B</th>
<th>DS</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CELAR6-SUB0</td>
<td>0</td>
<td>0.07</td>
<td>0.06</td>
<td>0.23</td>
<td>0.24</td>
</tr>
<tr>
<td>CELAR6-SUB1-24</td>
<td>0</td>
<td>0.02</td>
<td>2.59</td>
<td>6.21</td>
<td>3.35</td>
</tr>
<tr>
<td>CELAR6-SUB1</td>
<td>0</td>
<td>0.11</td>
<td>66.79</td>
<td>78.23</td>
<td>24.7</td>
</tr>
<tr>
<td>CELAR6-SUB2</td>
<td>0.01</td>
<td>0.13</td>
<td>71.91</td>
<td>438.67</td>
<td>11.65</td>
</tr>
<tr>
<td>CELAR6-SUB3</td>
<td>0</td>
<td>0.14</td>
<td>551.25</td>
<td>DNF</td>
<td>97.01</td>
</tr>
<tr>
<td>CELAR6-SUB4-20</td>
<td>0</td>
<td>0.02</td>
<td>7.71</td>
<td>8.9</td>
<td>1.85</td>
</tr>
<tr>
<td>CELAR6-SUB4reduc</td>
<td>0.01</td>
<td>0.15</td>
<td>DNF</td>
<td>DNF</td>
<td>701.43</td>
</tr>
<tr>
<td>CELAR6-SUB4</td>
<td>0</td>
<td>0.18</td>
<td>DNF</td>
<td>DNF</td>
<td>575.71</td>
</tr>
<tr>
<td>CELAR7-SUB0</td>
<td>0.01</td>
<td>0.09</td>
<td>0.11</td>
<td>0.58</td>
<td>0.17</td>
</tr>
<tr>
<td>CELAR7-SUB1-20</td>
<td>0</td>
<td>0.02</td>
<td>0.11</td>
<td>0.78</td>
<td>0.14</td>
</tr>
<tr>
<td>CELAR7-SUB1</td>
<td>0</td>
<td>0.11</td>
<td>5.48</td>
<td>7.36</td>
<td>4.6</td>
</tr>
<tr>
<td>CELAR7-SUB2</td>
<td>0</td>
<td>0.12</td>
<td>71.91</td>
<td>77.92</td>
<td>31.19</td>
</tr>
<tr>
<td>CELAR7-SUB3</td>
<td>0</td>
<td>0.16</td>
<td>301.16</td>
<td>636.09</td>
<td>1039.52</td>
</tr>
<tr>
<td>CELAR7-SUB4-22</td>
<td>0</td>
<td>0.03</td>
<td>5.65</td>
<td>10.75</td>
<td>5.25</td>
</tr>
<tr>
<td>graph07</td>
<td>0.01</td>
<td>0.46</td>
<td>75.01</td>
<td>41.18</td>
<td>15.68</td>
</tr>
<tr>
<td>graph12</td>
<td>0.03</td>
<td>0.85</td>
<td>8.57</td>
<td>62.6</td>
<td>8.44</td>
</tr>
<tr>
<td>graph13reducmore</td>
<td>0.09</td>
<td>1.25</td>
<td>196.66</td>
<td>285.4</td>
<td>279.1</td>
</tr>
<tr>
<td>graph13reduc</td>
<td>0.09</td>
<td>1.24</td>
<td>195.76</td>
<td>277.79</td>
<td>277.96</td>
</tr>
<tr>
<td>scen06-30reduc</td>
<td>0</td>
<td>0.01</td>
<td>208.84</td>
<td>322.38</td>
<td>90.5</td>
</tr>
<tr>
<td>scen09</td>
<td>0.03</td>
<td>0.57</td>
<td>10.9</td>
<td>106.19</td>
<td>5.52</td>
</tr>
<tr>
<td>scen10</td>
<td>0.02</td>
<td>0.57</td>
<td>0.7</td>
<td>932.01</td>
<td>1.33</td>
</tr>
<tr>
<td>Total</td>
<td>0.3</td>
<td>6.47</td>
<td>&gt;4261</td>
<td>&gt;6893</td>
<td>3175</td>
</tr>
</tbody>
</table>

Table 7.3: Permuted RLFAP Problems Preprocessing Time and Total Time in Seconds, best time in bold
instances such as CELAR6-SUB0 and CELAR7-SUB0. For these instances, SB expands fewer nodes than the number expanded by B&B, although the running time is higher. This discrepancy is largely due to the preprocessing time used by SB to run Algorithm 10.

To test the robustness of the clustering algorithm, the RLFAP benchmarks are tested a second time with the values for each variable randomly permuted before the beginning of search. The results are shown in Table 7.3. The results for DS are significantly slower than the unmodified problems, since the initial ordering of the values embedded structural information that was lost after permutation. However, B&B and SB both displayed only minor changes in their performance (due to some differences in tie breaking).

7.5.5 Pedigree results

Table 7.4 shows the results of B&B, DS, and SB on the Pedigree problems. DS performs very poorly on these problems, and is unable to solve any of the larger pedck instances. SB is able to solve two problems not solvable by B&B.

7.5.6 Random MAX-CSP results

Table 7.5 summarizes the results of B&B, DS, and SB, on 100 randomly generated MAX-CSPs. The instances are small, so the preprocessing time totaled less than 10 milliseconds for all instances. Randomly generated problems contain very few similar values. Thus, Set Branching is unable to create sets of values that are able to generate strong bounds early in the search space. The results show that on random MAX-CSPs, B&B performed very well on the problem set, solving the instances faster than both DS and SB. In fact, B&B expanded fewer nodes than DS. SB expands the fewest nodes, and is able to solve the set of instances faster than DS. However, SB takes almost 50% more time than B&B to solve the instances. If a problem has no similar values, as is the case in randomly generated problems, Set Branching is not an effective tool.
### Table 7.4: Pedigree problems: Preprocessing Time and Total Time in Seconds, best time in bold

<table>
<thead>
<tr>
<th>Instance</th>
<th>PreBB</th>
<th>PreSB</th>
<th>B&amp;B</th>
<th>DS</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>connell</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>eye</td>
<td>0</td>
<td>0.13</td>
<td>0</td>
<td>6.52</td>
<td>0.13</td>
</tr>
<tr>
<td>pedck350l2</td>
<td>0.02</td>
<td>0.16</td>
<td>DNF</td>
<td>DNF</td>
<td>0.61</td>
</tr>
<tr>
<td>pedck350l3</td>
<td>0.03</td>
<td>0.16</td>
<td>493.98</td>
<td>DNF</td>
<td>0.63</td>
</tr>
<tr>
<td>pedck350</td>
<td>0.03</td>
<td>0.15</td>
<td>DNF</td>
<td>DNF</td>
<td>0.85</td>
</tr>
<tr>
<td>pedck60-L12</td>
<td>0</td>
<td>0.03</td>
<td>0.76</td>
<td>11.23</td>
<td>1.01</td>
</tr>
<tr>
<td>pedck60-L1</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
<td>0.09</td>
<td>0.03</td>
</tr>
<tr>
<td>pedck60-L2</td>
<td>0</td>
<td>0.02</td>
<td>0</td>
<td>97.43</td>
<td>0.04</td>
</tr>
<tr>
<td>saudiarabia</td>
<td>0</td>
<td>0.05</td>
<td>1.69</td>
<td>3.1</td>
<td>160.73</td>
</tr>
<tr>
<td>simple</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sobel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>wijsmanguo</td>
<td>0</td>
<td>1.16</td>
<td>139.31</td>
<td>0.07</td>
<td>91.86</td>
</tr>
</tbody>
</table>

### Table 7.5: 100 Random MAX-CSP problems: Time in Seconds

<table>
<thead>
<tr>
<th>B&amp;B</th>
<th>DS</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>#s</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>----</td>
</tr>
<tr>
<td></td>
<td>5810038</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>5921777</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td><strong>4312543</strong></td>
<td>100</td>
</tr>
</tbody>
</table>

Table 7.5: 100 Random MAX-CSP problems: Time in Seconds
7.5.7 Experimental Conclusions

The RLFAP and Pedigree instances show that on problems with large domains, Set Branching can show a dramatic improvement over both B&B, and Domain Splitting. The improvement over Domain Splitting becomes even more pronounced when the values of the variable are randomly permuted. This illustrates that the effectiveness of domain splitting is dependent on implicit information contained in the problem specification (i.e., on the ordering of values in the the variable domains).

7.6 Related Work

Domain Splitting [38], and Set Branching [65, 103] are the foundations of the ideas developed in this chapter. Set Branching is a technique developed for CSPs that partitions the values of a domain into sets, and branches on each set [65, 103]. Previous Set Branching algorithms, however, create sets of values based on domain-specific information. For example, [65] creates an algorithm specifically for the crossword puzzle problems. The techniques introduced in this chapter apply Set Branching to optimization problems, and also improve on previous work by specifying a domain-independent algorithm for creating sets of similar values.

Set Branching, however, does not work well in conjunction with the decomposition techniques presented in Chapters 3-5. When a variable is assigned a single value, the underlying constraint graph is simplified because the variable, and each adjacent edge, is removed from the constraint graph. Decomposition relies on variable assignments simplifying the underlying constraint graph in order to decompose the problem. When Set Branching branches on a set containing more than one variable, the constraint graph is not simplified, and no new decomposition will emerge.

Similarity between values has been defined in the literature in terms of interchangeability [15]. Values $V = a$ and $V = b$ are interchangeable if, for any solution with $V = a$
in the assignment set, there is an identical solution except with $V = b$, and for any solution with $V = b$ in the assignment set, there is an identical solution except with $V = a$. If values are interchangeable, then one of the two assignments, $V = a$ or $V = b$, is eliminated from the problem specification. Unfortunately, the restrictive nature of this definition means that interchangeable values are rarely found. The definition of interchangeability was relaxed to include any pair of values for which the optimal solution is within some threshold [82, 104]. Although such a definition allows for interchangeable values to be found with greater frequency, treating these values as interchangeable means that the optimal solution is not necessarily found during search.

### 7.7 Future Work

The Set Branching algorithm developed in this chapter is presented in the context of the particular bounding technique of enforcing local consistency. Recently, three promising new bounding techniques have been developed for WCSPs. They are called Virtual Arc Consistency [23], Optimal Soft Arc Consistency [23], and Soft Global Constraints [69]. The clustering techniques introduced in this chapter may be modified to apply to these and other bounding techniques. To do so, a new method for computing the dissimilarity matrix would be required. The other aspects of Set Branching (specifically Algorithm 9) would continue to operate in the same way. In particular, the bounding technique would need to be examined so as to develop a way of estimating the dissimilarity between alternate values in terms of their effect on the bounds generated.

### 7.8 Conclusions

This chapter adapts an existing search algorithm to WCSPs, which branches on sets of values rather than individual values. The algorithm can significantly reduce the search space by clustering values into sets. The key contribution of this chapter is the devel-
opment of a domain independent algorithm that clusters values into sets. This method is shown to improve significantly the total running time of search compared to standard Branch and Bound and Domain Splitting on common benchmark problems of high domain size.
Chapter 8

Conclusions and Future Work

This final chapter reviews the contents of this thesis and discusses potential avenues for future research.

8.1 Summary

In this thesis, a variety of techniques are presented that increase the efficiency of COP solvers. The contributions of this thesis fall naturally into two main sections: decomposition and symmetry. Proving optimality for COPs is difficult; exploiting properties such as decomposition and symmetry allows us to solve otherwise unsolvable problems. The first section presented new methods of exploiting decompositions in COPs.

In Chapter 3, a method called Component Templating was introduced. Component templates are data structures that are used to store the structure of sub-problems encountered during backtracking search. Component templates represent all of the components that appear during search which have identical assigned and unassigned variables.

Component templates reduce the computational cost of dynamically detecting disjoint components, because they can be triggered by watch variables. Component Templates also offer the advantage of effective and efficient caching of component bounds. Component Templates are used to improve the efficiency of AND/OR search (see page 52) which
exploits decomposition by independently solving the separate components generated during search. The algorithms introduced in Chapter 4 and Chapter 5 also use component templates to detect decompositions quickly while solving WCSPs, and to efficiently cache optimal and partial solutions to the components generated during search.

In Chapter 4, an algorithm called OR-Decomposition (see page 63) was introduced. Previous decomposition algorithms, including AND/OR Search, relied on separate recursions to solve disjoint components, which forced search to solve a single component at a time.

OR-Decomposition can interleave the solving of the current components which can produce refined bounding information sufficient to refute a whole collection of components, without ever having to solve any component to optimality. OR-Decomposition can interleave the solving of components while still obtaining many of the benefits of decomposition. Component bounds can be cached in case the same component is encountered later in search. Chapter 4 also described ways that local propagation can be used with OR-Decomposition for effective bounding.

The experimental results in this chapter illustrate that by exploiting decomposition and solving multiple components concurrently, OR-Decomposition often outperforms algorithms such as AND/OR search.

Decomposition techniques, including OR-Decomposition, are only effective on problems that decompose frequently during search (problems with low tree-width). In order to overcome this problem, Decomposition Bounding (see page 95) is introduced in Chapter 5. First, Decomposition Bounding splits the objectives of the problem into two sets. The objectives induce two sub-problems: a decomposable problem and a complement problem. Decomposition Bounding modifies Branch and Bound to detect and exploit decompositions over the objectives of the decomposable sub-problem. Since there are fewer objectives in the decomposable sub-problem, decompositions are found in this sub-problem that do not occur in the original problem. Decompositions that occur on the
decomposable sub-problem during Branch and Bound search are used to efficiently compute bounds on this sub-problem. Branch and Bound can then use the updated bounds to prune parts of its search space that would otherwise have been explored.

The conclusions are empirically verified in the experimental section of Chapter 5. In particular, the Spot5+Global benchmarks illustrate that decomposition can be successfully exploited on problems that cannot be decomposed using existing decomposition algorithms.

The second major section of the thesis explores the use of symmetries in COP. Chapter 6 introduced Symmetric Templating, a technique that exploits dynamic symmetries in COPs [59]. This technique calculates symmetries between component templates, thereby allowing efficient symmetry detection during search. In order to locate these symmetries, a graph representation is created for every template. Graph isomorphisms are then found between the graph representation of every template. Caching is employed to re-use solutions found for symmetric components, thereby pruning parts of the search space that would otherwise have to be explored.

Empirical testing demonstrates that dynamic symmetry detection can be used efficiently in optimization problems. By reducing the overhead involved in symmetry detection, the solver not only dramatically reduces the search space, but also reduces the time taken to prove optimality.

Although Symmetric Templating is a useful tool for efficiently solving COPs that contain symmetries, often there are no perfect symmetries in a problem. Chapter 7 presents an algorithm called Set Branching (see page 142), which attempts to locate and exploit values that are similar, but not necessarily symmetric. Set Branching attempts to reduce the size of the branching factor in backtracking search by branching on sets of similar values.

In order for Set Branching to reduce the size of the explored search space, the sets must be carefully chosen so that strong bounds will exist early in the search space. A
principled method of partitioning the values of a variable’s domain into sets is developed based on the objectives of the problem (see page 151). Given these sets, Branch and Bound is extended so that it can branch on sets of values rather than on the single value assignments it typically uses. Experimentation not only shows an improvement in search time over Branch and Bound, but also over the related technique, Domain Splitting search.

8.2 Future Work

The techniques introduced throughout this thesis provide a variety of avenues for future research. In each chapter, a future work section presents some possibilities for future work specifically relating to each algorithm. This section discusses some of the possibilities for building upon the new techniques developed in this thesis.

Many of the algorithms described in this thesis use a cache to store and re-use bounds that are calculated during search. When solving intensive problems such as those seen in [79], the size of the cache could exhaust the available memory. Techniques similar to those proposed by [79] have the potential to reduce the memory requirements for the decomposition techniques developed in the thesis.

Although most of the algorithms presented in this thesis are tested with existensial directional arc-consistency bounding (EDAC) [28], the algorithms often use a generic \texttt{getBounds} function. Testing with different bounding algorithms (e.g, Virtual Arc Consistency [23], Optimal Soft Arc Consistency [24], mini-buckets [31], linear relaxation [55], and soft global constraints [69]) has the potential to yield interesting results.

All of the algorithms in this thesis are presented in the context of COPs. However, many of the algorithms could be adapted to SAT and CSP (by storing no-goods and solutions rather than bound information), or to solvers such as Mixed Integer Programming, which use a Branch and Bound or Branch and Cut approach [111].
8.3 Conclusion

This thesis presents a variety of techniques that can increase the efficiency of backtracking COP solvers. Component templates reduce the computational cost of dynamically detecting disjoint components. OR-Decomposition is able to exploit decomposition while avoiding separate recursions. Decomposition Bounding can exploit decomposition on problems with high tree-width. Symmetric Templating efficiently detects dynamic symmetries in COPs. Finally, Set Branching is able to exploit values that are similar, although not necessarily symmetric. All techniques have been successfully tested on industrial and academic benchmarks.
Bibliography


