REFORMULATING CONSTRAINT SATISFACTION PROBLEMS WITH APPLICATION TO GEOSPATIAL REASONING

by

Kenneth M. Bayer

A THESIS

Presented to the Faculty of
The Graduate College at the University of Nebraska
In Partial Fulfillment of Requirements
For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

August, 2007

REFORMULATING CONSTRAINT SATISFACTION PROBLEMS WITH APPLICATION TO GEOSPATIAL REASONING

Kenneth M. Bayer, M.S.

University of Nebraska, 2007

Advisor: Berthe Y. Choueiry

Scalability is a major obstacle in solving combinatorial problems. Reformulation techniques are often sought to overcome the complexity barrier. In this thesis, we propose four reformulation techniques that modify one or more components of a Constraint Satisfaction Problems (CSPs) in order to facilitate scalability. Those techniques are: query reformulation, domain reduction, constraint-model relaxation, and symmetry detection. We introduce the techniques and describe their use on general CSPs. Then, we study the building-identification problem (BID) introduced and modeled as a CSP by Michalowski and Knoblock [2005]. We introduce an improved constraint model for the BID that accurately reflects the inherent structure of a problem instance as well as a custom solver that exploits the properties of this structure.

We apply our reformulation techniques to the BID problem and interleave them with the various stages of our custom solver. We apply our integrated approach to real-world case studies of the BID to evaluate the benefits of our reformulation techniques. We show that our approach allows us to solve much larger problem instances than it was previously possible.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Berthe Y. Choueiry, for her support and input in this research. I am also thankful to the members of the committee, Dr. Jitender Deogun, Dr. Steve Reichenbach, and Dr. Ashok Samal, for their time and helpful advice. I would especially like to thank Dr. Jitender Deogun for his particularly helpful comments.

I would also like to thank Dr. Craig Knoblock and Mr. Martin Michalowski of the Information Sciences Institute (ISI) at the University of Southern California (USC) for their ideas and support in this research, which builds on their own earlier investigations. I would especially like to thank Martin for suggesting the use of the matching relaxation as a lookahead mechanism. I am also grateful to Ms. Medha Shewale (USC) who, with the help of Mr. Snehal Thakkar (ISI/USC), assisted in the creation of problem instances used in the experiments.

I would not have been able to perform the experiments necessary to evaluate the ideas proposed in this thesis without the patient assistance of Mr. Garhan Atterbury of the Research Computing Facility of the University of Nebraska-Lincoln.

This research is supported by NSF CAREER Award #0133568, and is based upon work supported in part by the Air Force Office of Scientific Research under grant numbers FA9550-04-1-0105 and FA9550-07-1-0416. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

This work was completed utilizing the Research Computing Facility of the University of Nebraska-Lincoln.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Scalability is a major obstacle to the automation of problem solving in real-world settings. Abstraction and reformulation techniques are commonly sought to overcome the complexity barrier. In this thesis, we propose and characterize new reformulation techniques for Constraint Satisfaction Problems (CSPs), which are commonly used to model NP-complete decision problems. Although our techniques are primarily designed for resource allocation problems modeled as a CSP, we identify how they can be exploited in relational consistency and symmetry detection. For example, we show how one of our reformulations is useful for reducing the space complexity of enforcing relational consistency on general CSPs. Further, we study the building-identification problem (BID), a geospatial reasoning problem modeled as a resource allocation problem where we assign each resource to a single task. This problem was introduced by Michalowski and Knoblock [2005]. We improve their proposed constraint model to better reflect the topology of the physical world and to allow the addition of constraints that reflect characteristics of a particular problem instance. Finally, we demonstrate the effectiveness of our reformulation techniques in the context of solving the BID problem, and show how they allow us to solve larger problem

instances than was previously possible.

Choueiry et al. [2005] characterized a reformulation process as a transformation of the encoding of a problem, where the encoding of a problem $\mathcal{P}$ is a combination of its *formulation* and a *query*, $\mathcal{P} = \langle \mathcal{F}, \mathcal{Q} \rangle$. The reformulation transforms the original encoding $\mathcal{P}_o = \langle \mathcal{F}_o, \mathcal{Q}_o \rangle$ into a reformulated one $\mathcal{P}_r = \langle \mathcal{F}_r, \mathcal{Q}_r \rangle$ by changing the original query and/or any of the components of the original formulation. The goal of the transformation is to 'simplify' problem solving, where the benefit of the 'simplification' and other effects of the transformation must be clearly articulated, specified, and evaluated in the particular problem-solving context.

The problem formulation of a CSP is given by $\mathcal{F} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a set of variables, $\mathcal{D}$ is the set of their respective domains, and $\mathcal{C}$ is a set of constraints. A constraint is a relation over a subset of the variables specifying the allowable combinations of values for the variables in its scope. A solution is an assignment to the variables such that all constraints are satisfied. The query is usually to find one consistent solution, in which case the general problem is in **NP**-complete. Alternatively, the query could be to find all possible solutions. When reformulating a CSP, we can reformulate any aspect of the problem definition, as illustrated in Figure 1.1.

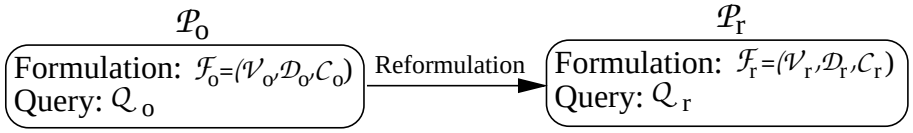

Figure 1.1: The reformulation process.

In this thesis, we present four techniques to reformulate different components of the CSP formulation and query. These techniques were also presented in [Bayer *et al.*, 2007a] and [Bayer *et al.*, 2007b]. The techniques we present are useful in many application domains, particularly for resource allocation problems. We demonstrate the effectiveness of the techniques on the building-identification problem introduced

by Michalowski and Knoblock [2005].

In this chapter, we introduce some motivating applications. We then describe the contributions of this thesis and discuss related work. Finally, we describe the organization of this document.

1.1 Motivating examples

In this section, we describe three applications that can be naturally modeled as a CSP and solved using Constraint Processing (CP) techniques. These applications are: the computer-configuration problem, the teaching-assistant assignment problem, and the building-identification problem (BID).

1.1.1 Computer configuration

The computer-configuration problem is the task of selecting and connecting components to form a computer that fulfills a given set of computational and Internet connectivity needs. Given a client's needs, a set of components along with a description of their characteristics and functionalities, a set of rules about how components can be chosen and combined, our goal is to find a satisfactory configuration of components for a computer. Examples of the rules (e.g., constraints) follow:

- We may have limits on the number of components of a certain type we may use. For example, a computer may only have a single motherboard.

- Certain components may be mutually exclusive. For example, certain processors may only be compatible with certain motherboards.

- Some components, or their features, may already be specified by the user. For example, the users may request a specific graphics card, or they may specify

only the type of the card.

- The users may specify a maximum total price or a price range.

The query may be either to find a single computer configuration that meets the requirements, or to find the set of all such configurations.

1.1.2 Teaching assistant assignment

The task in the teaching-assistant (TA) assignment problem is to assign TA's based on their qualifications, availability, and preferences, to tasks such as grading, supervising laboratory work, and conducting lectures or recitations [Glaubius and Choueiry, 2002]. Finding a satisfactory set of assignments is a difficult problem to solve. Previous work at the Constraint Systems Laboratory has investigated automated and interactive CP techniques to address this problem with much success [Glaubius, 2001; Zou, 2003; Guddeti, 2004; Thota, 2004; Lim, 2006].

1.1.3 Building identification

The building-identification problem was introduced by Michalowski and Knoblock [2005]. The task is to assign a list of postal addresses to buildings appearing in a satellite image. We can extract the buildings and streets from satellite images using techniques such as those introduced by [Agouris and Stefanidis, 1996] and [Doucette *et al.*, 1999]. However, we do not know the addresses of the buildings or, for a building located on a street corner, on which street the building's address lies. A variety of data sources, such as phone-books and gazetteers, provides a (likely incomplete) list of addresses.

One possible practical scenario is when a user opens a Google-map interface, clicks on a building and asks for the possible postal addresses of the building. Alternatively, the user gives a postal address and the system highlights the possible buildings that

may have the requested address[1]. This problem is important, as an effective solution to it could have prevented catastrophes such as the inadvertent bombing of the Chinese embassy in Belgrade, as described in [Pickering, 1999]. More generally, the information gained by data integration can be used to verify and augment geospatial databases (e.g., gazetteers[2]), and extend the capabilities of geospatial systems (e.g., Google Maps, Google Earth, and Microsoft VirtualEarth).

A variety of algorithms exists to solve the similar problem of geocoding, which is the task of determining the precise latitude and longitude of a given address using information obtained from diverse sources [Bakshi *et al.*, 2004]. This problem differs from the BID problem, where the task is to assign addresses to buildings identified in a satellite image. Michalowski and Knoblock [2005] modeled the BID problem as a CSP. Their work established the feasibility of modeling and solving this problem as a CSP and identified an important new area where CP techniques are useful for solving real-world problems. However, they were only able to demonstrate its effectiveness on a single problem instance containing 34 buildings. The techniques we propose, when applied to the BID problem, allow us to solve problems with as many as 206 buildings.

1.2 Contributions

We present four main contributions:

1. The design of four new reformulation techniques for the general Constraint Satisfaction Problem, see Chapter 3.

2. A new constraint model and a custom solver for the Building Identification

[1]Currently, Google map gives approximate results based only on geocoding coordinates.
[2]http://www.census.gov/cgi-bin/gazetteer

problem, see Chapter 4.

3. A proof the that BID problem as described in [Michalowski and Knoblock, 2005] can be solved in polynomial time.

4. The application of the proposed reformulations to the BID problem, and their integration with the designed solver, see Chapter 5. The benefits of the reformulations are evaluated and demonstrated on real-world data sets that we built, in collaboration with Martin Michalowski and Medha Shewale, from publicly available sources on the world-wide web.

Below, we briefly discuss each of these contributions.

1.2.1 Reformulation techniques for the CSP

We introduce four reformulation techniques for CSPs.

1. *Query reformulation:* We introduce a technique that reformulates the query in a problem definition, reducing the complexity class of a problem from a counting problem to a satisfaction one. We show how this reformulation yields general relational-consistency algorithms with a significantly smaller space complexity than $\mathrm{RC}_(i, m)$, the algorithm proposed by Dechter [2003].

2. *Domain reformulation:* We introduce a technique that reformulates the domains of CSP variables to reduce their size in the presence of a new global constraint that we specify. We argue that this new global constraint and its reformulation are particularly useful in resource allocation problems.

3. *Constraint relaxation:* We argue that many resource allocation problems have, at their core, a problem that can be modeled as a matching problem on a bipartite graph. Thus, for a problem that may be in **NP**-complete, we find

a relaxation that is in **P**. We exploit this relaxation in a number of ways to improve the performance of the search process used to solve the problem.

4. *Reformulation via symmetry detection:* We describe a way to characterize all maximum matchings in a bipartite graph as symmetric to a single maximum matching. Then, we use one maximum matching along with the symmetry description to represent all maximum matchings in a compact manner.

1.2.2 Modeling and solving the BID problem as a CSP

We introduce a new CSP model for the BID problem, and a 'custom' backtrack-search solver for this problem. Our model builds on the work of [Michalowski and Knoblock, 2005], improving their model to better reflect the underlying structure of the problem. Further, it is also allow the addition, as a plug in, of new constraints that reflect characteristics of a particular problem instance. Our solver is designed to take advantage of the structure of the BID problem. For example, it focuses on those variables that, when instantiated, reduce the CSP into a tractable structure[3].

1.2.3 The tractability of the BID problem

We show that the BID problem as described in [Michalowski and Knoblock, 2005] can be solved in polynomial time. We introduce a reduction of the BID problem into the problem of finding a maximum matching in a bipartite graph. While we show that the specific BID problem they described is tractable, we also characterize versions of the problem that may not be tractable.

[3]Such variables are called backdoor variables [Kilby *et al.*, 2005].

1.2.4 Reformulating the BID problem

We discuss the application of our reformulation techniques to the BID problem and the integration of these techniques in the solver we designed for the BID problem. We provide empirical results to show the value of our reformulation techniques in terms of improved runtime performance for problem solving. Importantly, we show that under certain conditions the BID problem can be solved in polynomial time, a result that was not previously known.

1.3 Related work

In this section, we give a broad discussion of related work. Specific approaches are discussed in more detail in the relevant sections of the remaining chapters.

Although, since the early 1990's, Constraint Processing has been particularly successful in industrial and other real-world settings, the fact that the satisfiability of a CSP is likely intractable hinders the scalability of this approach to large problems. Consequently, developing techniques the improve the performance of problem solving is an important area of research. In this thesis, we focus on techniques based on abstraction and reformulation. Abstraction and reformulation, although ubiquitous in Science and Engineering in general, have received a particular attention in the field of Artificial Intelligence where they have been studied for their own sake since the inception of the field. Simon [1969] has characterized the entire endeavor of problem solving as a change of representation. Holte and Choueiry [2003] provide a general discussion of approaches to abstraction and reformulation in Artificial Intelligence, including CSPs. In CSPs, as in other disciplines, most work can be roughly categorized into two main categories, modeling and reformulation, although *a clear boundary between those categories is often difficult to draw.*

- *Modeling:* Alternative, manually crafted, models of a given problem are built, often manually, and compared. Often times, additional constraints (also called redundant constraints) are added to the problem to speed up constraint propagation and search. In general, such techniques do *not* modify the set of solutions of the problem. Modeling remains an art, and has so far resisted automation.

  Nadel [1990] used the n-Queens problem as a case study and considered 8 different modelings of the problem, some of which were much easier to solve than others. Simonis [2005] investigated the addition of various redundant constraints to the model of a Sudoku puzzle to speed up constraint propagation. A series of workshops at the the Constraint Programming Conference, known as the Workshop on Modeling and Reformulating Constraint Satisfaction Problems, is devoted to this topic.

- *Reformulation:* Transformation techniques are designed that *automatically* modify a problem encoding to facilitate problem solving. The key issue here is that an initial encoding of the problem exists, and the reformulation is a *computational mechanism* that transforms the original encoding into a reformulated one. While reformulations are sometimes applied manually, the idea is that their application can be, and often times is, automated. In general, reformulation techniques may or may not change the set of solutions to the problem. For example, Ellman [1993] studies both necessary and sufficient abstractions of CSPs.

  Reformulation techniques have been proposed for reformulating CSPs, modifying the variables themselves (e.g., by aggregation), their domains (e.g., by interchangeability [Freuder, 1991]), and/or the constraints (e.g., by relaxation). The Symposium on Abstraction, Reformulation and Approximation (SARA),

with proceedings published by Springer Verlag, is devoted to discussing and unifying views on abstraction and reformulation across Artificial Intelligence[4], including CP.

Problem relaxation by constraint removal, which we discuss in Section 3.3, is the basis for many approximation techniques in Science and Engineering. In AI, it is the basis for generating tractable admissible heuristics for A* search, one of the oldest and most famous AI algorithms, see page 107 in [Russell and Norvig, 2003]. In mathematical programming, examples of common relaxation techniques include Lagrangian relaxation and the relaxation of an integer program into linear program [Milano, 2004]. In this context too, a common reformulation technique is the cutting-plane method, which consists in adding, instead of removing, constraints. A more complete approach is to generate two tractable problems that sandwich the original problem, one that is less constrained than the original problem, thus providing a sufficient approximation, and another that is more constrained, thus providing a necessary approximation. Such an approach is discussed for Propositional Logic in [Selman and Kautz, 1996]. The challenge is to keep both problems as close as possible to the original problem, thus providing tighter approximations, while guaranteeing that both problems are tractable.

Another approach to reformulation is based on (detecting and) exploiting symmetry to improve search performance. Such symmetries have been studied as far back as 1874 [Glaisher, 1874], and have been a topic of great interest recently [Brown *et al.*, 1988; Freuder, 1991; Ellman, 1993; Puget, 1993; Backofen and Will, 1999]. When symmetries are known, symmetry breaking constraints are added to the problem to prevent search from exploring symmetrical solutions. In [Freuder, 1991; Haselböck, 1993; Choueiry and Davis, 2002; Lal *et al.*, 2005], symmetries are detected automatically

[4] And to some extent Databases and Software Engineering.

and used to aggregate equivalent values in the domains of variables. Since 2001, a new workshop on Symmetry in Constraint Satisfaction Problems has been organized in conjunction with the Constraint Programming Conference. In Section 3.4, we discuss how to use symmetry as a reformulation to generate all maximum matchings in a bipartite graph from a single maximum matching.

Régin [1994] introduced an important global constraint known as the AllDiff constraint. An AllDiff constraint, also known as constraint of mutual exclusion, restricts given variables from being assigned the same value. Régin reformulated the problem of enforcing generalized arc-consistency (GAC) on an AllDiff into a problem involving finding a maximum matching in a bipartite graph. We use the construction of bipartite-graph matching in several ways in our work, specifically in Sections 3.3 and 3.4, where we discuss Régin's work in more detail.

Razgon et al. [2006] studied a class of problems that is similar to the one we investigate, and which they call Two Families of Sets constraints (TFOS). They introduced a technique for reformulating TFOS problems into network flow problems. The matchings we study in Section 3.3 constitute a special case of the TFOS problem.

1.4 Overview of this thesis

This thesis is organized as follows. Chapter 2 provides background information on CSPs and the BID problem. Chapter 3 presents our reformulation techniques. Chapter 4 describes our CSP model for the BID problem and our custom solver. Chapter 5 applies the reformulation techniques to the BID problem and discusses our experimental results, which demonstrate the benefits of the techniques in this context. Chapter 6 discusses directions for future work. Finally, Appendix A discusses the new XML file formats that we designed for storing and exchanging instances of the

BID problem; and Appendix B documents the Java code that implements our model and solver for the BID problem.

Chapter 2

Background

In this chapter, we review some background information about reformulation, Constraint Satisfaction Problems (CSPs), constraint propagation, backtrack (BT) search (which is the basis of our solver), and the building-identification problem (BID), to which we apply our reformulation techniques.

2.1 Abstraction and reformulation

As stated in Chapter 1, abstraction and reformulation are ubiquitous and aim at solving a problem by reformulating it into a 'simpler' problem. It is often difficult to define what it means for a problem to be 'simpler' to solve than the original one. Computationally, this may mean that the reformulated problem is in a 'smaller' complexity class, or that it can be solved with an algorithm whose asymptotic running time is lower. However, 'simpler' may also mean that the reformulated problem has properties that we can exploit heuristically to improve runtime, while the worst-case runtime remains the same for both algorithms. Choueiry et al. [2005] discuss the definition of simplicity in greater detail. In our work, we consider both reformulations that change the complexity class of the underlying problem and reformulations that

improve the average runtime performance.

For this purpose, we include reformulations that modify the problem by approximating it, potentially increasing or decreasing the set of solutions the problem has. Ellman described two types of approximations for CSPs in [1993], *necessary* and *sufficient* approximations[1].

Definition 1 (Necessary approximation, Ellman [1993]) $\mathcal{P}_\mathrm{r}$ is a *necessary approximation* of problem $\mathcal{P}_\mathrm{o}$ when the following holds: $\mathcal{P}_\mathrm{o}$ is solvable only when $\mathcal{P}_\mathrm{r}$ is solvable.

Conversely, when $\mathcal{P}_\mathrm{r}$ has no solution, then $\mathcal{P}_\mathrm{o}$ does not have a solution. If we define a function Φ to map $\textsc{Solutions}(\mathcal{P}_\mathrm{o})$, which is the set of solutions of $\mathcal{P}_\mathrm{o}$, to $\textsc{Solutions}(\mathcal{P}_\mathrm{r})$, which is the set of solutions to $\mathcal{P}_\mathrm{r}$, necessary approximations verify the following condition: $\Phi(\textsc{Solutions}(\mathcal{P}_\mathrm{o})) \subseteq \textsc{Solutions}(\mathcal{P}_\mathrm{r})$, as illustrated in Figure 2.1:

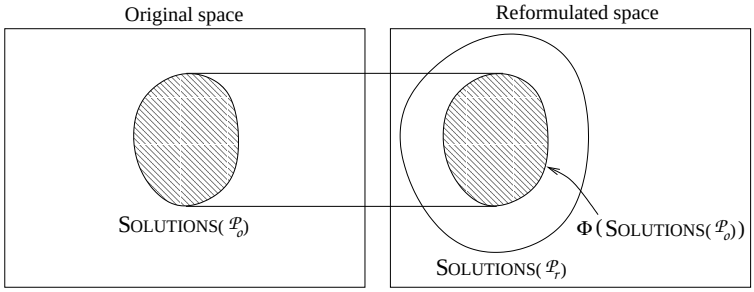

Figure 2.1: Necessary approximations as mappings between sets of solutions.

Definition 2 (Sufficient approximation, Ellman [1993]) $\mathcal{P}_\mathrm{r}$ is a *sufficient approximation* of problem $\mathcal{P}_\mathrm{o}$ when the following holds: $\mathcal{P}_\mathrm{r}$ is solvable only when $\mathcal{P}_\mathrm{o}$.

Conversely, when $\mathcal{P}_\mathrm{o}$ has no solution, then $\mathcal{P}_\mathrm{r}$ does not have a solution. If we define a function Φ as a mapping between the two sets of solutions, sufficient approximations

[1]In Software Engineering, the terms 'over-approximation' and 'under-approximation' are used.

verify the following condition: $\Phi^{-1}(\text{SOLUTIONS}(\mathcal{P}_\text{r})) \subseteq \text{SOLUTIONS}(\mathcal{P}_\text{o})$, as illustrated in Figure 2.1:

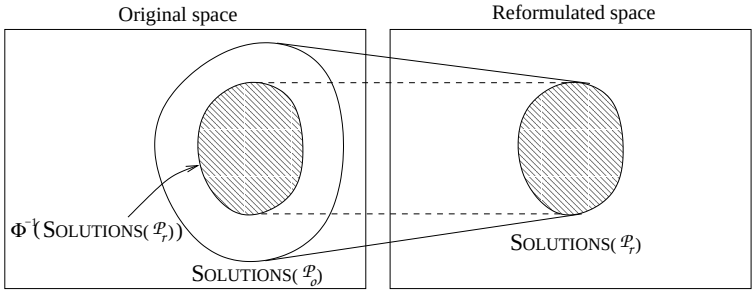

Figure 2.2: Sufficient approximations as mappings between sets of solutions.

In summary, we can use a necessary approximation to conclude the non-solvability of the original problem when the reformulated (assumingly simpler) one is not solvable. Similarly, we can use a sufficient approximation to infer that the original problem is solvable when we have determined the solvability of the reformulated problem. *Exact approximations* enjoy both properties. For example, alternative models [Nadel, 1990] and redundant models [Cheng *et al.*, 1996; Simonis, 2005] of constraint problems explore semantically equivalent models of a constraint problem to enhance the performance of constraint propagation and search.

Similar concepts were also introduced by Giunchiglia and Walsh in [1992] with a terminology based on formal logic. The above approximation are called 'theorem decreasing,' 'theorem increasing,' and 'theorem constant' abstractions, respectively.

The reformulations we introduce are described in Chapter 3. The query and domain reformulations and the reformulation by symmetry detection are exact approximations, where as the reformulation by constraint relaxation is a necessary approximation.

2.2 The constraint satisfaction problem (CSP)

Definition 3 (Constraint Satisfaction Problem) A Constraint Satisfaction Problem (CSP) is given by a tuple $\mathcal{F} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, defined as follows:

- $\mathcal{V} = \{V_1, V_2, ..., V_n\}$ is a set of variables.
- $\mathcal{D} = \{D_1, D_2, ..., D_n\}$ are their respective domains.
- $\mathcal{C} = \{C_1, C_2, ..., C_k\}$ is a set of constraints the restrict the possible combinations of values assigned to each variable.

A variable-value pair is an assignment of a value to a variable taken from its domain. We say that the variable is *instantiated*. A solution is an assignment of a value to each variable. The task can be to determine whether the problem has a solution, find one solution, or find all solutions.

Each constraint is defined over a subset of the variables, called the *scope* of the constraint. The *arity* of a constraint is the size of its scope. A *unary* constraint has arity 1, a *binary* constraint has arity 2, and a *non-binary* constraint has an arity greater than 2. A binary CSP has constraints of only arity 1 or 2. A *universal constraint* is a constraint that allows all possible combinations of values. These constraints are trivially satisfied, so we usually ignore them except when constraint propagation results in removing value tuples from them, see Section 2.3.

We represent a constraint network by a graph whose vertices correspond to the variables and whose edges correspond to the constraints. The vertices are labeled with the domain of the variables. We represent each binary constraint as an edge between the vertices in its scope, and we represent a non-binary constraint as a hyperedge across the vertices in its scope. Figure 2.3 shows an example of a binary constraint network.

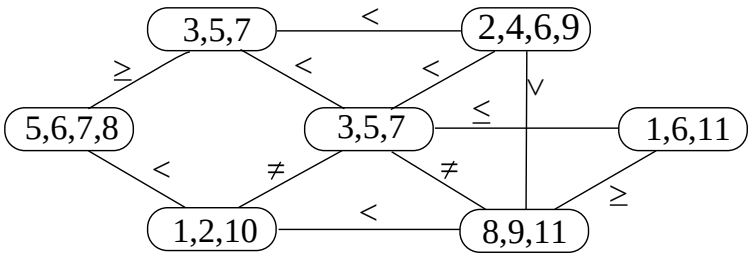

Figure 2.3: An example of a binary constraint network.

Constraints can be defined either in *extension* or in *intension*. We define a constraint in extension by explicitly listing the set of all allowed tuples that represent consistent assignments to the variables in its scope. This set can be implemented as a table, as in a relational database. We define a constraint in intension by defining a predicate function that determines whether a set variable-values pairs satisfy, are allowed by, the constraint. Constraints defined in intension can save the space necessary for storing the table of tuples allowed by the constraint. However, constraints defined in extension can be more easily used to join constraints and create higher arity constraints, which allow better propagation and greater filtering, as shown in [Dechter and van Beek, 1996], while consuming more space.

When modeling a problem as a CSP, it may be useful to include in the CSP model additional variables that do not correspond to decision variables in the problem. Such these variables are called called *hidden variables*. They are used to simply store information internal to the CSP model and facilitate propagation.

2.3 Constraint propagation

The concept of constraint propagation is perhaps the most important contribution of the CP community to the scientific field. It consists in applying relatively efficient algorithms to remove values, or combination of values, from the model of a CSP. The

key issue to keep these algorithms efficient, typically with a quadratic or cubic asymptotic complexity, or at least with an exponent bounded by a (constant) parameter. These algorithms can be broadly classified as follows:

- Algorithms that remove values from the domain of the variables do domain filtering.
- Algorithms that remove combinations of values typically add new constraints to the CSP, which is less desirable as the space necessary for storing the CSP is increased and may explode.

The basic operation of a domain filtering algorithm consists in *revising* the domain of a variable V_i given the constraint that links it to another variable V_j. During this revise operation, values in D_{V_i} that are not consistent with any value in D_{V_j} are removed from D_{V_i}. When this operation is repeated for V_j, we say that the constraint C_{V_i,V_j} is made arc-consistent. An arc-consistency algorithm consists in repeating that revise operation over all the constraints in the CSP until reaching a fixed point, in which case the CSP is said to be arc-consistent. Figure 2.4 shows the constraint network from Figure 2.3 after performing arc-consistency.

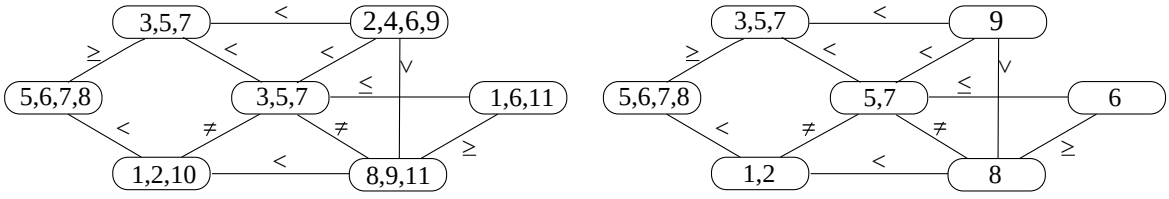

Figure 2.4: A binary CSP before and after arc-consistency.

Arc-consistency filters only the domains of the variables and considers, at each step, individually each constraint, and does not modify the constraints in the CSP. Mechanisms for enforcing higher-consistency levels exist. They typically operate by

composing multiple constraints, using the join operator of relational algebra, and either projecting the results on the domains of the variables (for domain filtering) or on a subset of the variables, thus generating new constraints.

2.4 Backtrack search

Backtrack search (BT) is a systematic algorithm that we can use to solve CSPs. In this section we first describe the BT algorithm. We then describe techniques that improve the performance of the basic algorithm. These techniques are the following:

1. Lookahead, which employs a simple form of constraint propagation.
2. Backjumping, which reduces the backtracking effort during search.
3. Exploiting tree-structured constraint networks. And,
4. Symmetry detection techniques.

2.4.1 BT algorithm

Backtrack (BT) search tentatively assigns values to variables in some order. After performing an assignment, the algorithm determines whether any constraints have been violated. (This operation is called back-checking.) If any constraint is violated, the algorithm undoes the last assignment and tries the next value for the un-assigned variable. If all values for the current variable result in broken constraints, the algorithm moves back to the last variable, undoes its assignment, and proceeds as described above.

BT is both sound and complete. It returns a solution if no constraints are violated, and it tries every possible set of assignments before returning that no solution exists. However, it is not an efficient algorithm. In the worst case it may have to try every

possible set of assignments. Thus, the algorithm requires $O(d^n)$ time, where n is the number of variables in the CSP and d is the size of their domains.

The problem of determining the satisfiability of a CSP is **NP**-complete. Thus, it is reasonable that any sound and complete algorithm to determine satisfiability runs in exponential time. However, there are a variety of techniques that we can use to improve the performance of search. These techniques fall primarily into two categories: lookahead and backjumping.

2.4.2 Lookahead

Lookahead techniques attempt to prune the search space by applying constraint propagation to remove values from the domains of future variables given the instantiation of the variables in the path currently expanded in the tree.

Forward checking is the most basic lookahead strategy [Haralick and Elliott, 1980]. At every variable instantiation during search, it revises the domains of the un-instantiated variables adjacent to the current variable by removing from these domains the values that are not consistent with the current instantiation. In doing so, it considers only the constraints existing between the current variable and the future variables. If this process annihilates the domain of any future variable, we know that the current search path cannot extend to a complete solution, and we undo the current instantiation. Thus, lookahead may allow us to backtrack earlier and avoid exploring unnecessary portions of the search space. With some adaptation, we can use forward checking with non-binary constraints. Several techniques for lookahead with non-binary constraints were introduced in [Bessière *et al.*, 1999].

We can perform even more filtering if we check more constraints. For example, we can apply an arc-consistency algorithm over all the future variables in the CSP, thus revising also all the constraints between the future variables. MAC is a looka-

head technique introduced by [Gaschnig, 1974] and [Sabin and Freuder, 1997] that makes the entire problem arc consistent after each instantiation. MAC performs more filtering than FC, but it is also more costly.

2.4.3 Backjumping

Backjumping avoids unnecessary search by making more informed decisions when we encounter an inconsistent partial solution. When it determines that there are no consistent values for the current variable during search, BT always backtracks to the previous variable instantiated. However, the true source of conflict may lie much earlier in the search. Conflict-Directed Backjumping (CBJ) jumps back to the variable that was the source of the current inconsistency [Prosser, 1993].

One can use both lookahead and backjumping in the search procedure. Lookahead techniques filter the domains of future variables, whereas backjumping techniques only affect the behavior of backtracking. For example, we could perform backtrack search using both MAC and CBJ. This algorithm is MAC-CBJ, introduced in [Prosser, 1995].

2.4.4 Tree-structured CSPs

Whenever possible, it is useful to exploit the structure of the CSP network (or the semantic of the constraints) to improve the performance of search. For example, Freuder [1982] showed that any CSP with a tree-structured constraint graph can be solved in polynomial time[2]. Dechter and Pearl [1987] proposed a technique that takes advantage of this fact in search. Their technique, called the *cycle-cutset* technique, identifies a set of variables whose removal leaves only a forest of trees. Once the cycle cutset is instantiated and all the remaining trees are filtered by arc-consistency

[2]More precisely, in linear time after arc-consistency, which is quadratic.

against the instantiated variables, we can find a solution to the remaining part of the CSP in linear time.

2.4.5 Symmetry detection

Symmetry detection techniques take advantage of structures in the search space. If we can detect that multiple portions of the solution space are symmetrical, we can take advantage of the symmetry to improve the performance of search. Puget [1993] introduced techniques to take advantage of exact symmetries, while Ellman [1993] considered approximations of symmetry relations. Freuder [1991] introduced the concept of interchangeability among the values of CSP variables as special form of symmetry.

2.5 The building-identification problem (BID)

Our case study targets the building-identification problem described by [Michalowski and Knoblock, 2005]. In this section, we define the problem and specify the components of a problem instance.

2.5.1 Problem definition

We define the BID problem as follows. Given a map of streets and buildings positioned on the streets, and a phone book that has a list of addresses, determine which address (in the phone book) corresponds to which building on the map. There are restrictions on matching phone-book addresses and buildings. Our work addresses four kinds of restrictions:

1. Buildings on the same side of the street must have the same parity (e.g., odd or even).

2. Buildings on a street, or a given street segment, must appear in increasing or decreasing order.

3. All addresses in the phone book must be used.

4. Across certain gridlines, addresses increase or decrease to the next increment of some value k. For example, in many cities in the United States, addresses increase to the next increment of 100 when crossing intersections. Gridlines can correspond to physical street intersections or correspond to virtual grids [Michalowski, 2006].

In [Michalowski and Knoblock, 2005], the problem is modeled as a CSP and solved using the CPlan solver of [van Beek and Chen, 1999]. Our model, described in Sections 4.1 and 4.2, uses slightly different variables and constraints; and our solver, described in Section 4.4, is a custom backtrack search, with lookahead and backjumping, tailored to improve the performance of problem solving.

2.5.2 Problem instance specification

A problem instance is composed of four pieces of data:

1. The layout of buildings and streets.

2. A list of phone-book addresses.

3. The layout of the gridlines.

4. A set of known landmarks, which are buildings with known addresses.

The first two items are required in every instance specification, while the last two are optional. We store problem instances as a set of XML files of a format that we designed and articulated in collaboration with Martin Michalowski. Appendix A

describes the schema for each XML file. Figure 2.5 shows a simple problem instance, with 2 streets and 2 buildings. Solid squares represent non-corner buildings, and dashed squares represent corner buildings.

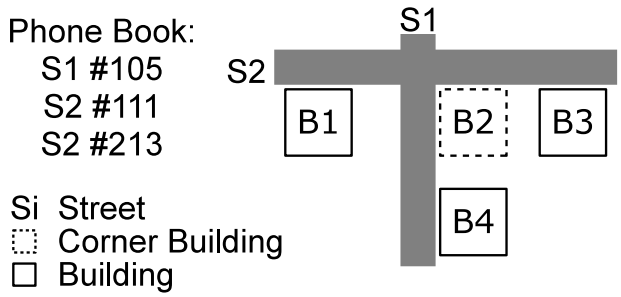

Figure 2.5: A two-street example.

2.5.2.1 Layout

The *layout* data describes the streets and buildings on the map. It must provide information about the order in which buildings appear along streets and the set of streets to which each building is adjacent.

2.5.2.2 Phone-book

The *phone-book* data lists the known addresses in the region, typically collected from online white-pages. We consider that the buildings corresponding to these addresses must exist. Thus, any solution to a BID instance must use all the addresses in the phone-book. Note that it is possible to have fewer phone-book addresses than we have buildings in the layout. In this case, we have an *incomplete* phone-book. If we have exactly as many addresses as we do buildings, we have a *complete* phone-book. When the phone-book is incomplete, some buildings will have to be assigned addresses that may or may not be their addresses in the real world. In Figure 2.5 the map has four buildings and the phone-book lists three addresses. Thus, the

phone-book is incomplete. One possible solution to the example in Figure 2.5 is B1←S2#111, B2←S2#205, B3← S2#213, and B4←S1#105. However, because the phone-book is incomplete, this problem has many other solutions. In addition to the above solution, the assignments B1←S2#111, B2←S1#119, B3←S2#213, and B4←S1#105 also form a solution.

The layout and phone-book information are required for every problem instance. The remaining information, gridlines and landmarks, is optional. When it is available, it is used to further constrain the problem.

2.5.2.3 Gridlines

The *gridline* data about a region describes virtual lines across which addresses increment to the next increment of some value. For example, in many cities in the United States, addresses increase to the next increment of 100 across intersections. Thus, in these regions there are grid lines along the streets with increments of 100.

2.5.2.4 Landmarks

The *landmark* data provides an exact address for a specific building that is already known. For example, it is often possible to determine the addresses of some buildings using gazetteers.

2.5.2.5 Additional numbering rules

Above, we described the four types of data that we consider in modeling and solving the BID problem. However, it is possible that in different regions, different rules may apply. For example, in Italy, a red-black numbering scheme can be used, and it is not uncommon to increase numbers by a half number or by adding 'A' and 'B' suffices.

In this thesis we only consider the above listed rules. The doctoral research

of Martin Michalowski is concerned with determining which rules apply in which context. Our new CSP model provides the flexibility of easily integrating such rules as new constraints, and our solver is designed to handle them with relatively few modifications.

Summary

In this chapter we gave background information on reformulation, CSPs, and the backtrack search algorithm that we can use to solve them. We also described the BID problem, which we use as a case study for the techniques we develop.

Chapter 3

Reformulation techniques for CSPs

In this chapter, we propose four reformulation techniques to improve the performance of solving CSPs. As described in Section 2.1, reformulation techniques may change any aspect of a problem's formulation or query. In this chapter, we introduce techniques that modify the query (Section 3.1), the domains of the CSP variables (Section 3.2), and the constraints (Section 3.3). We also introduce a technique for generating all the maximum matchings in a bipartite graph by symmetry from a single maximum matching in the graph (Section 3.4).

3.1 Query reformulation

In a CSP, the query is usually to find a single solution (i.e., satisfiability problem) or all solutions (i.e., enumeration problem). However, in some applications, we may be interested in *all* the values that a given variable (alternatively, all variables) can possibly take. One way to compute this exact set is to generate all solutions to the problem, and collect the values taken by the variables in those solutions, which corresponds to solving an enumeration problem. In this section, we propose to look at this task as a reformulation of an enumeration problem into a set of satisfiabil-

ity problems. Below, we motivate the reformulation, provide a simple algorithm to implement it, and describe an improvement suggested by an anonymous reviewer. Finally, we discuss how this simple reformulation is useful for computing relational (i, m)-consistency with a lower space complexity than was previously proposed.

3.1.1 Per-variable solution

Example 1 Consider the computer configuration problem described in Section 1.1.1. Assume that the users have chosen a specific video card and a processor, and they want to know which motherboards and which monitors they may choose. Here, the users are not looking for a complete configuration. Rather, they are looking for all possible motherboards and monitors they can choose from, that is, the set of all possible values for the variables corresponding to the motherboard and the monitor.

One way to solve the above problem is to find all solutions[1], then, for each variable, compute the union of the values it takes in all those solutions. We propose to reformulate the problem as follows. For each of the variables of interest to the users, consider every value in the domain of the variable, and determine whether the CSP with that variable-value pair assignment is solvable. If it is, then the value is kept in the answer to the query; otherwise, it is discarded. On average, it is significantly less costly to establish satisfiability (i.e., stop searching after finding the first solution), then to count all solutions (i.e., traverse the entire search space). We express this new problem, which we call a *per-variable solution*, as a reformulation of the counting problem by a change of the query. That is, we reformulate the query of the counting problem from $\mathcal{Q}_{\text{o}}$=enumerating all solutions to $\mathcal{Q}_{\text{r}}$=finding a per-variable solution,

[1] As it is done in [Michalowski and Knoblock, 2005].

where $\mathcal{Q}_{\mathrm{r}}$ is defined as:

$$\forall\, V_i, x \in D_{V_i}, \text{find if } \mathcal{F}_{\mathrm{o}} \wedge (V_i \leftarrow x) \text{ is satisfiable.} \tag{3.1}$$

Figure 3.1 illustrates this reformulation.

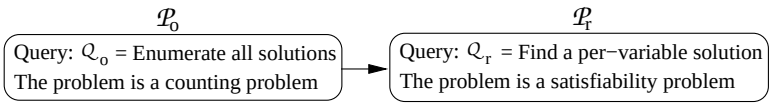

Figure 3.1: Query reformulation.

In CP terminology, the 'per-variable solution' corresponds to finding the minimal network of a CSP as described in [Montanari, 1974]. It is also equivalent to the inverse consistency property introduced in [Freuder and Elfe, 1996], and to relational (1,$|\mathcal{C}|$)-consistency defined in [Dechter and van Beek, 1996].

In terms of asymptotic complexity, we are replacing a counting problem, whose complexity is $O(d^n)$, where n is the number of variables and d is the maximum domain size, by a polynomial number of satisfiability problems $O(n \cdot d \cdot d^{n-1})$=$O(n \cdot d^n)$, which may appear to be bad idea. However, the worst-case asymptotic complexity is misleading as, in practice, proving satisfiability is significantly cheaper than enumerating all solutions. In some cases, particularly those with a large number of solutions, the reformulated problem is significantly easier to solve than the original one.

3.1.2 Algorithm for finding per-variable solutions

Given a CSP $\mathcal{F}$ =($\mathcal{V}, \mathcal{D}, \mathcal{C}$), Algorithm 1 tests (Line 10), for every variable-value pair (V_i, x), where $V_i \in \mathcal{V}$ and $x \in D_{V_i}$, whether the CSP with V_i←x is soluble (Line 11). When a solution exists, x is added to the data structure returned by the algorithm (Line 12). The algorithm returns the set of variables along with all their values

Input: $\mathcal{F} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$
Output: S, or a message indicating that no solution exists

```
1  foreach V_i ∈ 𝒱 do
2  |  S[V_i] ← ∅
3  end
4  vvps ← ∅
5  foreach V_i ∈ 𝒱 do
6  |  foreach x ∈ D_{V_i} do
7  |  |  vvps ← vvps ∪{(V_i, x)}
8  |  end
9  end
10 foreach (V_i, x) ∈ vvps do
11 |  if F with V_i←x has a solution then
12 |  |  S[V_i] ← S[V_i] ∪ {x}
13 |  end
14 end
15 if |S[v]| = 0 then
16 |  return F has no solutions
17 end
18 return S
```

Algorithm 1: Finding the per-variable solutions.

that appear in a solution. The loop of the algorithm, Line 10, runs $O(nd)$ times. Each iteration requires determining the satisfiability of a CSP. This operation appears costly, but in cases where the original CSP has significantly more than nd solutions, Algorithm 1 performs in practice significantly better than enumerating all solutions to the CSP.

We introduce here an improvement proposed to us by an anonymous reviewer. When the test in Line 11 is executed by finding a solution to the CSP, the values for the variables in the solution found can be collected, and excluded from future calls in the loops at Line 10, because we already know that they appear in at least one solution. Thus, we would execute fewer iterations of the loop, improving performance. As we apply the query reformulation to the BID problem, however, Line 11 is such that we do *not* generate a solution to the problem to establish satisfiability. Thus,

we cannot exploit the improvement proposed by the reviewer. We explain in greater detail why our BID solver does not use this improvement in Section 5.2.

3.1.3 Relational consistency by query reformulation

In Section 2.3 we discussed the arc-consistency algorithm that considers each constraint individually and filters, from the domains of variables in its scope, any values that violate the constraint. We can perform even more filtering by enforcing higher levels of consistency that consider combinations of constraints.

3.1.3.1 Relational (i, m)-consistency

Dechter and van Beek introduced *relational (i,m)-consistency* as a consistency property for CSPs [Dechter and van Beek, 1996]. Dechter defines the relational (i, m)-consistency property as follows.

Definition 4 (Relational (i, m)-consistency, Dechter [2003]) A set of relations $\{R_{S_1}, ..., R_{S_m}\}$ is *relationally (i,m)-consistent* iff for every subset of variables $\mathcal{A}$ of size i, $\mathcal{A} \subseteq \cup_{j=1}^{m} S_j$, any consistent assignment to $\mathcal{A}$ can be extended to an assignment to $\cup_{j=1}^{m} S_j - \mathcal{A}$ that satisfies $R_{S_1}, ..., R_{S_m}$ and the respective domain constraints simultaneously. A network is relationally (i, m)-consistent iff every set of m relations is relationally (i, m)-consistent. A network is strong relationally (i, m)-consistent iff it is relationally (j, m)-consistent for every $j \leq i$.

Dechter [2003] also proposed the algorithm $\mathrm{RC}_{(i,m)}$ for enforcing relational (i, m)-consistency in a constraint network. $\mathrm{RC}_{(i,m)}$ takes as input a constraint network $\mathcal{F} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$. For every set of constraints $\mathcal{C}_m = \{C_{S_1}, ..., C_{S_m}\} \subseteq \mathcal{C}$, where C_{S_m} is a constraint over the scope S_m, the algorithm computes the join of the constraints in $\mathcal{C}_m$ and projects the result onto each subset of variables $\mathcal{A} = \{V_1, ..., V_i\} \subseteq \cup_{j=1}^{m} S_j$.

Figure 3.2 illustrates a step in the algorithm for $\mathrm{RC}_{(2,2)}$. At this step, $\mathcal{C}_m = \{C_{S_1}, C_{S_2}\}$ and $\mathcal{A} = \{V_3, V_4\}$. The algorithm will compute the join of of C_{S_1} and C_{S_2}, and then project the result onto the constraint between V_3 and V_4. When $m = |\mathcal{C}|$, $\mathrm{RC}_{(i,m)}$

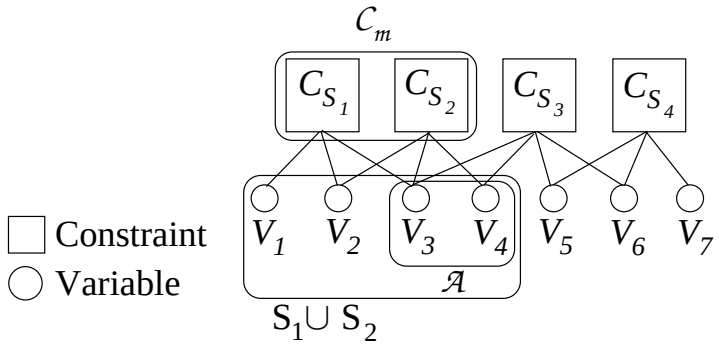

Figure 3.2: A step in the execution of $\mathrm{RC}_{(i,m)}$.

is not useful, because the algorithm would compute the join of all relations in the CSP. Computing this join completely solves the problem by enumerating all solutions, eliminating the value of enforcing higher levels of consistency.

On the other hand, Algorithm 1 enforces relational $(1, |\mathcal{C}|)$-consistency in a CSP, where $\mathcal{C}$ is the set of constraints, without enumerating all solutions. After executing Algorithm 1, the domain of each variable only contains the values that appear in at least one solution, which is equivalent to computing the join of all constraints and projecting the result onto each variable's domain.

3.1.3.2 A new algorithm for Relational (i, m)-consistency

We can easily *adapt* Algorithm 1 to enforce relational $(1, m)$-consistency for $m \leq |\mathcal{C}|$. For every subset $\mathcal{C}_m = \{C_{S_1}, ..., C_{S_m}\} \subseteq \mathcal{C}$, generate the induced CSP containing only the constraints in $\mathcal{C}_m$ and execute Algorithm 1 on the induced CSP. This algorithm, like $\mathrm{RC}_{(1,m)}$, has a runtime complexity that is exponential in $|\cup_{j=1}^{m} S_j|$. However, $\mathrm{RC}_{(1,m)}$ must compute and *store* the join of all constraints in $\mathcal{C}_m$. The number of

tuples in the join is exponential in $|\cup_{j=1}^{m} S_j|$. The adapted version of Algorithm 1 we described enforces the same level of consistency, but requires only polynomial space.

Algorithm 2 generalizes Algorithm 1 to enforce relational (i, m)-consistency for any $i \leq |\mathcal{V}|$ and $m \leq |\mathcal{C}|$. The algorithm loops over every subset of m constraints and generates the CSP induced by those constraints (Line 3 and 4). For each subset of constraints, the algorithm loops over every subset $\mathcal{A}$ of i variables in the union of their scopes (Line 5). For each subset of variables, the algorithm loops over every tuple in the cross product of their domains (Line 7). For each tuple, the algorithm tests whether or not the assignment of the values in the tuple to the variables in $\mathcal{A}$ appears in at least one solution to the induced CSP (Line 8). If it does, we add it to the set of allowed tuples (Line 9). After all tuples have been tested, the algorithm intersects the set of allowed tuples with the existing constraint over the scope $\mathcal{A}$ (Line 12). If no such constraint already exists, then $\mathcal{C}_{\mathcal{A}}$ is the universal constraint over $\mathcal{A}$.

Algorithm 2 returns a network that is relationally (i, m)-consistent. Note that Algorithm 1 is a special case of Algorithm 2 where $i = 1$ and $m = |\mathcal{C}|$.

Line 8 in Algorithm 2 must determine the satisfiability of CSPs with s variables, where $s = |\cup_{j=1}^{m} S_j|$. Thus, Algorithm 2 has a worst-case time complexity of $O(d^s)$, where d is the maximum domain size. $\mathrm{RC}_{(i,m)}$ also has a worst-case time complexity of $O(d^s)$, because it must compute the join of s variables with domain size d. However, Algorithm 2, which finds one solution, is in practice significantly less costly than $\mathrm{RC}_{(i,m)}$, which must enumerate all solutions to the same induced CSP.

However, $\mathrm{RC}_{(i,m)}$ requires computing and storing the join of each subset of m constraints, which requires $O(d^s)$ space. Algorithm 2 only stores the tuples for each set of i variables, and thus only requires $O(d^i)$ space, where $i \leq s$, for each subset of variables in of size i in $\cup_{j=1}^{m} S_j$. The total space required is then $O(s^2 d^i)$. Thus, Algorithm 2 enforces the same level of consistency as $\mathrm{RC}_{(i,m)}$, but requires less space.

Input: $\mathcal{F} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, i, m
Output: A relational (i, m)-consistent network equivalent to $\mathcal{F}$, or a message indicating that $\mathcal{F}$ has no solutions.

```
1  repeat
2      F_p ← F
3      for every C_m = {C_S1, ..., C_Sm} ⊆ C do
4          F_i ← the CSP induced by C_m on F
5          for every A = {V_1, ..., V_i} ⊆ ∪_{j=1}^{m} S_j do
6              R = ∅
7              foreach tuple r ∈ V_1 × ... × V_i do
8                  if F_i with (V_1 ← r[1]) ∧ ... ∧ (V_i ← r[i]) has a solution then
9                      R ← R ∪ {r}
10                 end
11             end
12             C_A ← C_A ∩ R
13             if |C_A| = 0 then
14                 return F has no solution
15             end
16         end
17     end
18 until F_p = F ;
19 return F
```

Algorithm 2: Enforcing relational (i, m)-consistency.

3.2 Domain reformulation

In this section, we introduce AllDiff-Atmost, a global constraint useful for formulating resource allocation problems. We define AllDiff-Atmost and describe a reformulation procedure that modifies the domains of the variables in its scope. Then we show how to apply this reformulation when the variable domains are totally ordered. In Section 5.3, we use AllDiff-Atmost to model the BID problem and illustrate the use of the reformulation in that context.

3.2.1 AllDiff-Atmost

An AllDiff-Atmost constraint is useful in a resource allocation problem to restrict the variables in its scope from using more than a certain number of values from a given set. Consider the simple resource allocation problem of Example 2.

Example 2 An emerging country received an aid to build 7 hospitals on its territory, but does not want to put more than 2 hospitals in areas with high volcanic activity.

We propose the constraint AllDiff-Atmost to model this situation. Formally, we define the AllDiff-Atmost constraint as follows.

Definition 5 (AllDiff-Atmost) Given a set of variables $\mathcal{A} = \{V_1, \ldots, V_i, \ldots, V_n\}$ with continuous or finite domains D_{V_i}, AllDiff-Atmost$(\mathcal{A}, k, d)$, where $d \subseteq D_{V_i}$, $k \in \mathbb{N}$, and $k \leq |d|$, requires that the following conditions hold:

1. All variables are different.

2. At most k variables in $\mathcal{A}$ can take values from d. When $k \geq |d|$, then $k \leftarrow |d|$.

Example 3 is another example of the AllDiff-Atmost constraint.

Example 3 Consider the variables $\mathcal{A}$=$\{V_1,V_2,V_3,V_4\}$ of a CSP, with D_i=$\{1,2,\ldots,8\}$ and the constraint AllDiff-Atmost($\mathcal{A}$,2,$\{1, 3, 4, 5, 8\}$). The assignment V_1←5, V_2←2, V_3←7, and V_4←4 satisfies the constraint.

The AllDiff-Atmost constraint is useful in real-world settings; we use it in Section 5.3 in our constraint model for the BID problem. Example 4 illustrates the use of this constraint in the computer configuration problem of Section 1.1.1.

Example 4 Consider the configuration of a computer that has 3 expansion-card slots, represented by the variables s_1, s_2, and s_3. The domain of each s_i is $D_{s_i} = \mathcal{S}$,

the set of all expansion cards. Suppose we want to restrict the configuration to have only a single *network* card, where $\mathcal{N} \subseteq \mathcal{S}$ is the set of all expansion cards that are network cards. We can model this restriction with the constraint AllDiff-Atmost($\{S_1, S_2, S_3\}, 1, \mathcal{N}$).

3.2.2 Reformulation of AllDiff-Atmost

We now introduce a reformulation that reduces the domain size of the variables in the scope of an AllDiff-Atmost constraint. We replace the original domain $D^o_{V_i}$ of a variable V_i in the scope of the constraint AllDiff-Atmost($\mathcal{A}, k, d$) with the reformulated domain $D^{ref}_{V_i}$ by introducing k values s_i that we call *symbolic values* as follows:

$$\forall\, V_i \in \mathcal{A}\ D^{\text{ref}}_{V_i} = \{s_1, s_2, \ldots, s_k\} \cup (D_{V_i} \setminus d) \tag{3.2}$$

where the symbolic values s_j $(1 \leq j \leq k)$ can take any distinct values in d. We then replace the AllDiff-Atmost($\mathcal{A}, k, d$) constraint with AllDiff($\mathcal{A}$).

Strictly speaking, the reformulation procedure affects both the AllDiff-Atmost constraint, which is replace with an AllDiff constraint, and the set $\mathcal{D}_{\text{o}}$ of the domains of the variables in the scope of the AllDiff-Atmost constraint, as shown in Figure 3.3. However the most significant modification is the domain reformulation. The set of domains in the original CSP $\mathcal{D}_{\text{o}}$ is replace with the set of reformulated domains $\mathcal{D}_{\text{r}}$, where the domains of variables in $\mathcal{A}$ have been reformulated according to Equation (3.2). Replacing d in the original domains with k symbolic values reduces the domains sizes by $|d| - k$, which is useful when d is large or infinite.

In Example 2, the domains become $\{s_1,\ s_2\} \cup \{\text{sites in non-volcanic areas}\}$ where s_1, s_2 are different and range over sites with volcanic activities. Applying this reformulation to Example 3 yields the following domains for all four variables:

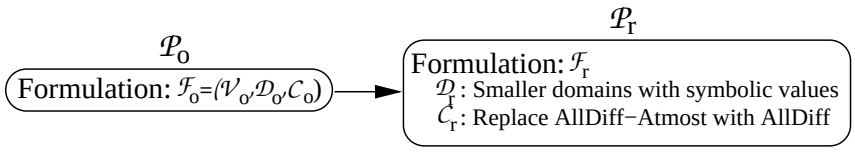

Figure 3.3: The reformulation of AllDiff-Atmost.

D_{V_i}=$\{s_1,s_2,2,6,7\}$, where s_1,s_2 can take any different values in $\{1,3,4,5,8\}$.

This reformulation is an exact approximation in the sense that solutions to the reformulated problem map to solutions to the original problem [Ellman, 1993]. The benefit of this reformulation is the reduction of the domain sizes. Because the complexity of many CP techniques depends on the sizes of the domains, the reformulation improves the performance of the solver.

Indeed, this operation is particularly useful during backtrack search where the domain values are enumerated. If we want to assign 'ground' values to each symbolic value, we can do so as a post-processing step while ensuring that two symbolic values are always mapped back to distinct ground values. While a solution to the reformulated problem does not map to a unique solution to the original problem, we can generate any solution to the original problem from some solution to the reformulated problem.

Of particular concern is the interaction between this reformulation and the other constraints in the problem. When all the constraints in a problem can be checked on the symbolic values, as in the case of the BID problem, the reformulation is sound. When one or more constraints in a problem must be checked on the 'ground' values, then propagation must run on the appropriate representation for each constraint and, as soon as domain filtering causes $|d| \leq k$, then reformulated domains should be dropped and AllDiff-Atmost replaced with a AllDiff constraint, as is the case in a BID instance with a complete phone-book. While this double representation works for constraint propagation, using it during backtrack search requires further

investigation.

3.2.3 Reformulating totally ordered domains

When the values in the domains of the variables follow a total order, as in numeric domains, it is common to represent the domains as intervals. It is also common to restrict propagation of algebraic constraints defined over such variables to the endpoints of the intervals in the domains, as in box- or bound-consistency algorithms [Benhamou *et al.*, 1994]. The reformulation of an AllDiff-Atmost obviously remains valid in the presence of totally ordered domains. However, in order to *restrict propagation to the endpoints of the intervals* representing the domains, we need to enforce the following:

1. We require the values in d to form a convex interval.

2. We must add total ordering constraints between the s_k symbolic values: $s_1 < s_2 < \ldots < s_k$.

3. We must add total ordering constraints between the two extreme symbolic values, s_1 and s_k, and their closest neighbors in the reformulated domains, which is accomplished as follows. Let $D_{V_i}^{\text{ref,l}}$ and $D_{V_i}^{\text{ref,r}}$ be the intervals of $D_{V_i}^{o} \setminus d$ respectively to the left and right of, and adjacent to, d. The right endpoint of $D_{V_i}^{\text{ref,l}}$ must be less than s_1, and the left endpoint of $D_{V_i}^{\text{ref,r}}$ must be greater than s_k. Figure 3.4 illustrates the effects of the reformulation on the domains.

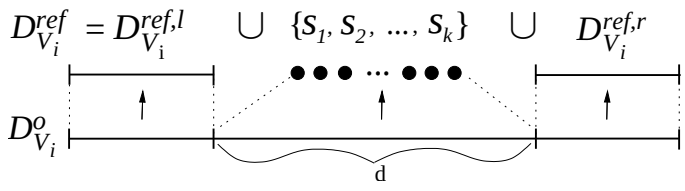

Figure 3.4: Reformulation of a totally ordered domain.

4. When mapping the symbolic values back to the ground values at the post-processing step, the ground values must respect the total ordering imposed on the symbolic values.

Example 5 illustrates the domain reformulation for a problem where we maintain the total ordering for the symbolic values.

Example 5 Consider the variables $\mathcal{A}$=$\{V_1,V_2,V_3,V_4\}$ of a CSP, with D_i=$\{1,2,\ldots,8\}$ and the constraint AllDiff-Atmost($\mathcal{A}$,2,$\{2,3,4,5\}$). The set $\{2,3,4,5\}$ is a convex interval, so we can reformulate the domains as follows. D_{V_i}=$\{1,s_1,s_2,6,7,8\}$, where $1 < s_1 < s_2 < 6$.

Thus, the reformulation of the AllDiff-Atmost constraint reduces the sizes of domains while maintaining a total ordering among the values. We apply this reformulation techniques to the BID problem in Section 5.3.

3.3 Problem reformulation by constraint relaxation

As we argued in Section 1.3, relaxing a problem by removing a constraint yielding a necessary approximation that is tractable is a common approach in many fields including Artificial Intelligence, Mathematical Programming, and Constraint Programming. In this section we describe one such approximation. In particular, we show how to relax a likely intractable resource allocation problem into a matching problem in a bipartite graph, which is tractable. We then show how to use the relaxation, both as a pre-processing mechanism and as a lookahead mechanism, during search process we use to solve the original problem.

3.3.1 Bipartite matching as a relaxation

At the core of many resource allocation problems lies the problem of matching between the elements of two sets: the tasks and the resources. In general, the problem may be complex (and likely intractable). However, in some cases, we may be able to identify some constraints that, when removed, reduce the original problem into the problem of finding a matching in a bipartite graph that saturates one of the two partitions, as defined below.

First, we introduce some graph theoretic terms, which can be found in [West, 2001]. Let $G = (X \cup Y, E)$ be a bipartite graph with edge set E, vertex set $V = X \cup Y$, and partitions X and Y, which are disjoint sets of vertices. We define a *match count* for each vertex in $v \in V$, which we denote $m(v)$, to be a positive (non-null) integer. A *matching* in G is a set of edges $M \subseteq E$ such that for every $v \in V$ there exists at most one edge $e \in M$ incident to v. In this thesis we consider a matching in G to be a set of edges $M \subseteq E$ such that for every $v \in V$ there exists at most $m(v)$ edges $e \in M$ incident to v. Further, we say that a matching M saturates vertex v iff there are $m(v)$ edges in M incident to v, and a matching M saturates a set of vertices S iff M saturates all vertices in S. Figure 3.5 shows a bipartite graph with partitions X and Y. The match count of each vertex is shown in parenthesis. The matching $M = \{(x_1, y_1), (x_3, y_2), (x_4, y_2)\}$, shown as darkened edges, saturates Y.

Finding a saturating matching in a graph where vertices have specified match counts has the same complexity as finding a saturating matching in a bipartite graph without match counts. We simply construct a graph G' where every vertex is replicated as many times as its match count. There exists a matching saturating vertex set S in G if and only if there exists a matching saturating the corresponding set of replicated vertices in G'. Figure 3.6 shows the graph from Figure 3.5 after replicating vertices.

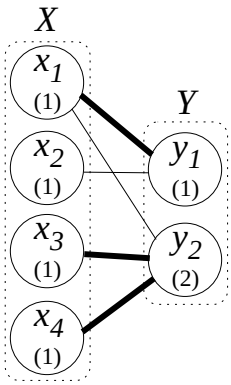

Figure 3.5: A bipartite graph with a matching saturating Y.

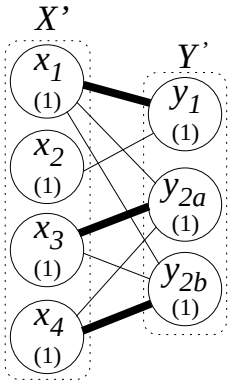

Figure 3.6: The graph from Figure 3.5 after vertex replication.

When S is the partition of the bipartite graph (i.e., X or Y) that has the smallest number of vertices, we can find a matching that saturates S by finding a maximum matching in G'. Thus, for our purposes, finding a maximum matching and finding a saturating matching are the same problem. We can find a maximum matching in a bipartite graph in polynomial time, using an algorithm such as the one introduced by [Hopcroft and Karp, 1973].

We propose to reformulate a general resource allocation problem by relaxing it into a matching problem in a bipartite graph that saturates one of the graph's two partitions (obviously, the one with the smallest number of vertices). Specifically, the variables in the problem correspond to vertices in partition X, and values correspond to vertices in partition Y. Thus, and edge (x, y) in the matching corresponds to the

assignment of value y to variable x. We relax the problem by removing constraints until we can model the remaining problem as a matching. Figure 3.7 illustrates this relaxation. The challenge is to identify those constraints whose removal yields the relaxation.

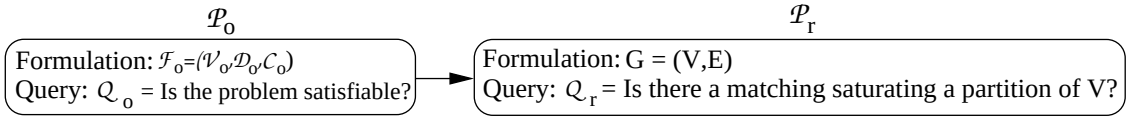

Figure 3.7: Relaxation of a CSP as a matching problem.

While the original problem may be intractable, the reformulated one can be efficiently solved (i.e., in polynomial time). When the reformulated problem is not soluble, the more constrained original problem is not soluble. However, the solubility of the reformulated problem does not guarantee that of the original problem. Our reformulation is thus a necessary approximation in the sense of [Ellman, 1993].

When modeling an assignment problem as a CSP and solving it with backtrack search, we can take advantage of the relaxed problem in two ways:

1. As a *pre-processing step* prior to search.

2. As *a lookahead mechanism* during search to filter out, from the domains of the future variables, those values that cannot yield a solution.

Next, we describe how to exploit the matching relaxation in those two ways.

3.3.2 Matching as a pre-processing step

When solving a resource allocation problem, modeled as a CSP, with backtrack search, we can use the matching relaxation as a pre-processing step prior to search in two ways. The first technique is useful prior to solving a CSP using search. The second technique is useful when finding a per-variable solution using Algorithm 1.

We can use a matching relaxation as a pre-processing technique prior to search as follows. If we determine, using the matching relaxation, that the relaxed problem is not soluble, we can safely avoid running search at all. If the relaxed problem is soluble, we must still proceed with the search to determine whether or not the CSP has a solution. The reformulation of a resource allocation problem into a matching problem is especially useful when finding per-variable solutions, because Algorithm 1 solves a large number of CSPs.

When solving a problem using Algorithm 1, we can use a matching relaxation to filter some Variable-Value Pairs (VVPs) from being considered in the loop at Line 10. Before Line 10, we enumerate the set of all edges that appear in any maximum matching. We can identify this set of edges using the algorithm proposed by [Régin, 1994]. If a vvp does not appear in any maximum matching in the original problem, then it cannot appear in any solution. Thus, we do not have to include the corresponding vvp's in the loop at Line 10.

We propose Algorithm 3 that integrates both of these pre-processing mechanisms in Algorithm 1. Instead of initializing *vvps* to contain all possible vvps, Line 4 initializes the *vvps* with only the set of vvps that appear in at least one maximum matching. Line 6 tests whether the relaxed problem is soluble, and Line 7 proceeds with the backtrack search only when the test has succeeded.

3.3.3 Matching as a lookahead mechanism

We use the matching *during* the backtrack search to solve the CSP as follows. An assignment (i.e., variable-value pair) appears in a solution if and only if its corresponding edge appears in a matching that saturates Y. Hence, along a given path in the search tree, we can apply the matching algorithm to the remaining CSP, not only to determine its satisfiability, but also to filter from the domains of the future

Input: $\mathcal{F} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$
Output: S, or a message indicating that no solution exists
foreach $V_i \in \mathcal{V}$ **do**
  $S[V_i] \leftarrow \emptyset$
end
vvps $\leftarrow$ the set of all vvps whose corresponding edges appear in at least one maximum matching
foreach $(V_i, x) \in$ *vvps* **do**
  if *the matching problem with* $V_i \leftarrow x$ *is solvable* **then**
    if $\mathcal{F}$ *with* $V_i \leftarrow x$ *has a solution* **then**
      $S[V_i] \leftarrow S[V_i] \cup \{x\}$
    end
  end
end
if $|S[v]| = 0$ **then**
  return $\mathcal{F}$ *has no solutions*
end
return S

Algorithm 3: Integrating matching in Algorithm 1.

variables values that may not appear in any maximum matching. To implement this filtering, we adapt the algorithm of [Régin, 1994] for generalized arc-consistency on an All-Diff constraint, which finds all edges of the bipartite graph that do not participate in *any* saturating matching, to identify, in one step, all values in the domains of all future variables that do not participate in any solution. This single operation allows us to filter the domains of all future variables in one step.

This additional filtering does not come without a cost. It is possible that the cost of calculating the matching may exceed the benefits gained by the additional filtering or prevented searches. In Chapter 5 we provide experimental results to evaluate these techniques on the BID problem.

Note that using the matching as a pre-processing step and using the matching as a lookahead mechanism require different algorithms, and thus cannot easily be combined. Pre-processing uses the Hopcroft-Karp algorithm [1973] to determine the existence of a *single* maximum matching. For lookahead to filter a vvp, we must

determine that the vvp does not appear in *any* maximum matching, and thus must run Régin's algorithm [1994] to enumerate the set of all edges that appear in at least one maximum matching.

3.4 Generating solutions by symmetry

Given a bipartite graph and one maximum matching, we can enumerate all all maximum matchings using an algorithm such as the one proposed by Uno [1997]. In this section, we describe a way to characterize all maximum matchings in a bipartite graph as symmetric to a single 'base' matching. We then use this symmetry detection to enumerate all maximum matchings. First, we describe some properties of maximum matchings, then, we show how to use these properties to identify symmetric maximum matchings.

3.4.1 Matching terminology

As in Section 3.3, we consider a bipartite graph $G = (X \cup Y, E)$ with partitions X and Y. Assume that $|X| \geq |Y|$. Our goal is to find a matching that saturates Y. Consider the bipartite graph $G = (X \cup Y, E)$ in Figure 3.8. G has two possible matchings that saturate Y:

$$M_1 = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$$

and

$$M_2 = \{(x_2, y_1), (x_3, y_2), (x_4, y_3)\}.$$

Figure 3.9 shows these matchings, where darkened edges indicate the edges included in the matching.

If we are simply trying to find a solution, it is sufficient to find *any* such matching.

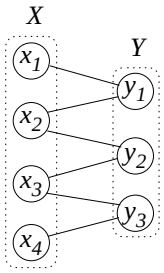

Figure 3.8: Bipartite graph with multiple maximum matchings.

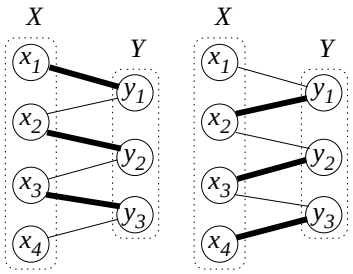

Figure 3.9: Matchings M_1 and M_2 for Figure 3.8.

However, if we want to find all solutions, we need an algorithm that finds *all* such matchings. Berge [1973] addressed the issue and showed that, given a maximum matching M, an edge e appears in some maximum matching *iff* e appears in M or some *alternating cycle* or *even alternating path beginning at a free vertex* relative to M as defined below.

Definition 6 (Free vertex, Berge [1973]) For a bipartite graph $G = (X \cup Y, E)$ with matching M, a *free vertex* is a vertex in G that is not incident to any edge in M.

Definition 7 (Alternating cycle, Berge [1973]) For a bipartite graph $G = (X \cup Y, E)$ with a matching M, an *alternating cycle* is a cycle in G that alternates between edges in M and edges that are not in M.

Definition 8 (Alternating path, Berge [1973]) For a bipartite graph $G = (X \cup Y, E)$ with a matching M, an *alternating path* is a path in G that alternates between

edges in M and edges that are not in M.

Hopcroft and Karp [1973] proposed a technique for finding alternating cycles for a graph G and a matching M. We orient the graph G such that all edges in M are oriented from X to Y, and all edges not in M are oriented from Y to X. Any cycle in the oriented graph is an alternating cycle.

To take advantage of alternating paths and cycles, we use the *symmetric difference* operation for graphs. The symmetric difference operation for graphs is an extension of the symmetric difference operation for sets.

Definition 9 (Symmetric difference, West [2001]) For graphs G and H, the *symmetric difference* $G\Delta H$ is the sub-graph of $G \cup H$ whose edges are the edges of $G \cup H$ appearing in exactly one of G and H. We also use this notation for sets of edges.

Clearly, if M is a maximum matching and P is an even alternating path or cycle, then $M\Delta P$ is also a maximum matching.

Definition 10 (Disjoint subgraphs) Two sub-graphs of a graph are said to be *disjoint* if and only if the sets of their edges are disjoint.

3.4.2 Symmetric maximum matchings

West [2001] showed that given two matchings M_1 and M_2, $M_1\Delta M_2$ is a set of alternating paths and cycles. As a corollary to this theorem, given M, a maximum matching, and $\mathcal{S}$, a set of all even alternating paths and cycles relative to M, we can enumerate all maximum matchings by taking the symmetric difference between M and every set of disjoint alternating paths and cycles in $\mathcal{S}$. Example 6 illustrates the construction of alternative maximum matchings by finding the symmetric difference between a base matching M_0 and three sets of disjoint alternating paths and cycles.

Example 6 Consider the graph G shown in Figure 3.10. Figure 3.11 shows the matching M_0, which saturates Y. We can construct graph G', shown in Figure 3.12, by orienting the edges in G such that edges in M_1 are oriented from the X to Y and edges not in M_1 are oriented from Y to X. Any cycle in G' corresponds to an alternating cycle in G relative to M_0. Figures 3.13 and 3.14 show the alternating path P and the alternating cycle C, respectively. P and C are disjoint. Thus, we can generate the following maximum matchings: $M_1 = M_0 \Delta P$, $M_2 = M_0 \Delta C$, and $M_3 = M_0 \Delta (P \cup C)$, where $P \cup C$ is the set of edges that appear in either P or C. Figures 3.15, 3.16, and 3.17 show M_1, M_2, and M_3.

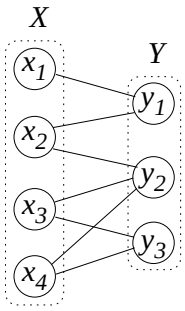

Figure 3.10: Bipartite graph G.

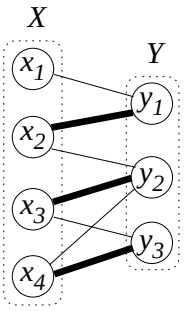

Figure 3.11: The matching M_0 of G.

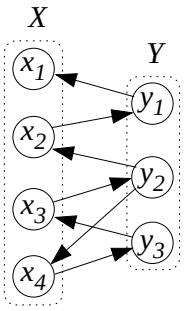

Figure 3.12: G', oriented relative to M_0.

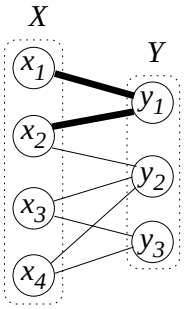

Figure 3.13: Alternating path P in G'.

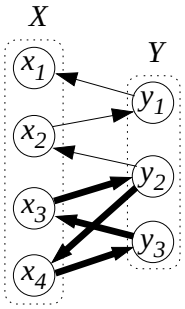

Figure 3.14: Alternating cycle C in G'.

We can enumerate all even alternating paths starting at free vertices from the set of free vertices [Berge, 1973]. We can orient the graph using the technique described by

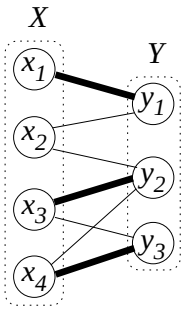

Figure 3.15: $M_1 = M_0 \Delta P$.

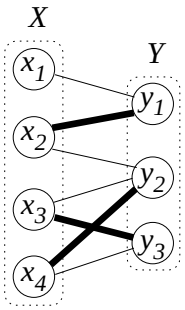

Figure 3.16: $M_2 = M_0 \Delta C$.

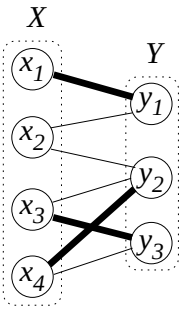

Figure 3.17: $M_3 = M_0 \Delta (P \cup C)$.

Hopcroft and Karp [1973] and then enumerate all alternating cycles from the strongly connected components in the oriented graph, as described by Régin [1994]. Thus, to store the information necessary to enumerate all alternating paths and cycles, and therefore all maximum matchings, we only need to store a single base matching, the set of free vertices, and the set of strongly connected components[2].

3.4.3 Symmetric solutions as a reformulation

Figure 3.18 illustrates the two reformulations of $\mathcal{P}_o$, the problem of enumerating all maximum matchings. We can reformulate $\mathcal{P}_o$ as $\mathcal{P}_{r1}$, the set of all maximum matchings, using Uno's algorithm. Alternatively, we can reformulate the problem as $\mathcal{P}_{r2}$, a base matching and its corresponding sets of strongly connected components and free vertices. All matchings can be enumerated from $\mathcal{P}_{r2}$ as needed. Our construction has the same time complexity as Uno's, which is linear in the number of maximum matchings. However, our characterization of the solutions as symmetries is valuable:

1. It provides a more compact representation of the set of solutions. Rather than storing all matchings, we store a single matching, a set of strongly connected components, and a set of free vertices.

[2]An improvement suggested by a anonymous reviewer.

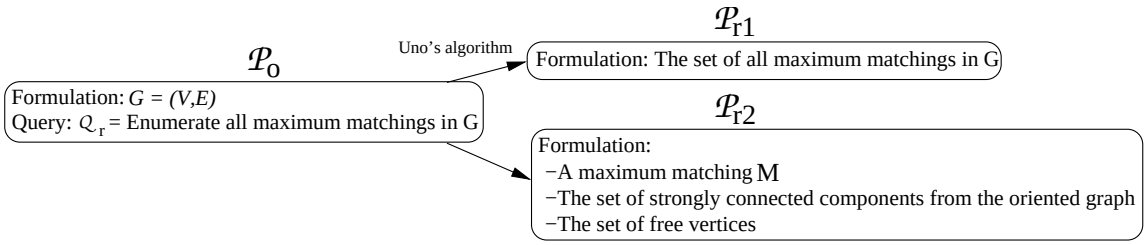

Figure 3.18: Finding all maximum matchings.

2. In case one is indeed seeking *all*, or a given number of, the solutions to the BID problem (similarly, to a resource allocation problem that has a maximum matching relaxation), we can generate every matching symmetric to that known single matching and test whether or not it satisfies the additional constraints of the non-relaxed problem. When it does, the matching is a solution to the non-relaxed problem. Note that it is found without search. Naturally, the number of maximum matchings can be large.

We do not currently exploit those features for solving the BID problem, but they deserve further investigations.

Summary

In this chapter, we introduced four reformulation techniques for CSPs. First, we described a query reformulation that reduces a counting problem into a polynomial number of satisfiability problems. Using this reformulation, we designed relational-consistency algorithms with significantly reduced space complexity than existing ones. Second, we introduced the AllDiff-Atmost constraint and defined a reformulation to reduce the domain sizes of the variables in its scope. Third, we argues that a general resource allocation problem can be relaxed into a maximum matching problem, which is tractable. We then showed how to use this necessary approximation before and

during search. Finally, we showed that all maximum matchings in a bipartite graph can be derived by symmetry, providing a more compact representation of the set of all maximum matchings, and sketched a strategy for exploiting this knowledge in practice.

Chapter 4

A Constraint Model and Solver for the BID

In this chapter, we describe the constraint model we designed for the BID problem and the custom solver we built for solving it. Our model is based on the one described in [Michalowski and Knoblock, 2005], but we introduce various improvements that allow our model to accurately reflect the inherent structure of a BID instance, which is then exploited in our custom solver to improve the performance of problem solving. We first specify the constraint model: the variables, their domains, and the constraint that apply to the variables. Then, we discuss a number of special layout configurations that our model can naturally handle. Finally, we discuss our custom solver and highlight its advantages.

To illustrate the various components of our model, we use the simple BID instance shown in Figure 4.1 and give the corresponding constraint network in Figure 4.17.

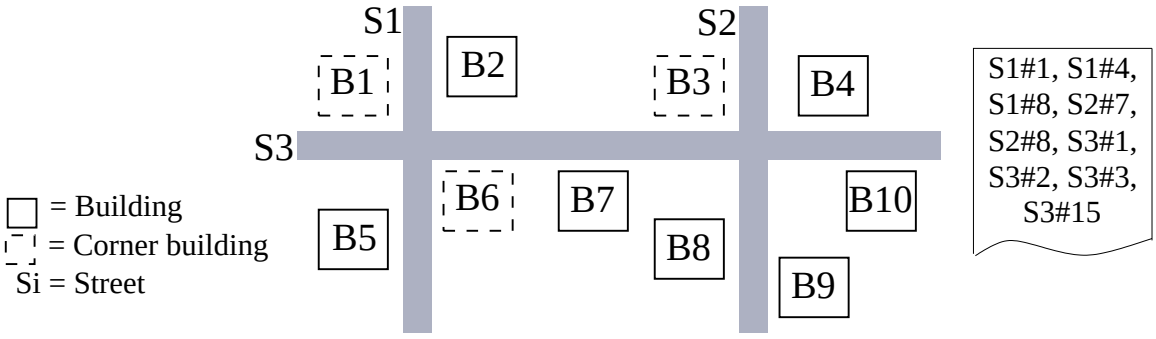

Figure 4.1: A simple instance of the BID problem.

4.1 Variables

Our constraint model has three types of variables: *orientation* variables (called global variables in [Michalowski and Knoblock, 2005]), *building* variables, and *corner* variables.

1. The *orientation* variables describe the overall layout of the map, such as the direction in which the numbers appear.

2. The *building* variables store the numeric address assignments to a specific building.

3. The *corner* variables store the street assignments to corner buildings.

4.1.1 Orientation variables

There are four orientation variables, exactly like the ones described by Michalowski and Knoblock [2005]. We use different names for them to make them more descriptive. Each of these variables is a Boolean variable:

1. *OddOnNorthSide*: If this variable is true, then odd addresses lie on the north side of the street; otherwise, they lie on the south side of the street.

2. *OddOnEastSide*: If this variable is true, then odd addresses lie on the east side of the street; otherwise, they lie on the west side of the street.

3. *IncreasingNorth*: If this variable is true, then numbers increase when moving toward the north on north-south running streets; otherwise, they increase when moving toward the south.

4. *IncreasingEast*: If this variable is true, then numbers increase when moving toward the east on east-west running streets; otherwise, they increase when moving toward the west.

Normally, the model has exactly one of each of the four orientation variables. This fact reflects that street numbering schemas are homogeneous over the considered geographical area. However, there are real-world situations where the number schemas differ between streets, as is the case for the city of Belgrade. Our model can easily accommodate such non-homogeneous numbering schemas by generating additional orientation variables for those streets that do not follow the regular pattern.

We assume that the street numbering schema for the BID instance shown in Figure 4.1 is homogeneous across the streets, and include only four orientation variables in the corresponding constraint model.

4.1.2 Building variables

The model includes a building variable for every building on the map. The domain of each such variable is the set of all addresses (a combination of a street and a number) that the building can take.

To populate the domain for a building on a given street, Michalowski and Knoblock [2005] enumerated all addresses from 1 to the largest number that appears in the phone-book for that street. However, choosing the largest phone-book address is

an arbitrary limit, and leads to incorrect results when the correct solution contains addresses larger than the largest phone-book address. To address this issue, in our model the range of possible addresses is specified as the $(0, \infty)$, unless the largest value is known. Note that the domains contain values that are not in the phone-book, and thus the solutions that we find contain inferred values for the missing addresses.

The BID instance of Figure 4.1 has 10 buildings. Our model includes the building variables {B1, B2, B3, B4, B5, B6, B7, B8, B9, B10}. The domains for the variables are as follows:

$$D_{\text{B1}} = \{\text{S1\#1, S1\#2, \ldots, S1\#}\infty\text{, S3\#1, S3\#2, \ldots, S3\#}\infty\},$$
$$D_{\text{B2}} = \{\text{S1\#1, S1\#2,\ldots, S1\#}\infty\},$$
$$D_{\text{B3}} = \{\text{S2\#1, S2\#2, \ldots, S2\#}\infty\text{, S3\#1, S3\#2, \ldots, S3\#}\infty\},$$
$$D_{\text{B5}} = \{\text{S1\#1, S1\#2, \ldots, S1\#}\infty\},$$
$$D_{\text{B4}} = \{\text{S3\#1, S3\#2, \ldots, S3\#}\infty\},$$
$$D_{\text{B6}} = \{\text{S1\#1, S1\#2, \ldots, S1\#}\infty\text{, S3\#1, S3\#2, \ldots, S3\#}\infty\},$$
$$D_{\text{B7}} = \{\text{S3\#1, S3\#2,\ldots, S3\#}\infty\},$$
$$D_{\text{B8}} = \{\text{S2\#1, S2\#2, \ldots, S2\#}\infty\},$$
$$D_{\text{B9}} = \{\text{S2\#1, S2\#2, \ldots, S2\#}\infty\}\text{, and}$$
$$D_{\text{B10}} = \{\text{S3\#1, S3\#2, \ldots, S3\#}\infty\}.$$

Section 4.4.3 describes how we implement the domains as sets of intervals, and Section 5.3.1 discusses how we reformulate these intervals to reduce their size.

4.1.3 Corner variables

We include a corner variable in the constraint model for every corner building on the map. The domain of a corner variable is the set of streets to which the building is adjacent.

We use separate variables for the assignment of an address and a street, where an

address is a combination of a street and a number, for the following reason. Determining the streets on which corner buildings lie before we assign numbers to them decomposes the constraint network into chains, as we discuss in Section 4.4. Note that the corner variables are *hidden* variables, in the sense that they are not part of the solution reported to the user. They are variables that only exist to facilitate the decomposition of the problem.

The BID instance in Figure 4.1 has 3 corner buildings. Our model has the following set of corner variables {B1-corner, B3-corner, B6-corner}. The domains for these variables are as follows:

$$D_{\text{B1-corner}} = \{\text{S1,S3}\},$$
$$D_{\text{B3-corner}} = \{\text{S2, S3}\}, \text{ and}$$
$$D_{\text{B6-corner}} = \{\text{S1, S3}\}.$$

4.2 Constraints

Our current model contains five types of constraints, which can be enriched with constraints to reflect street-numbering strategies used around the world [Michalowski *et al.*, 2007a]:

1. *Parity* constraints.
2. *Ordering* constraints.
3. *Phone-book* constraints.
4. *Corner* constraints.
5. *Grid* constraints.

The first four constraints already existed in the original model proposed in [Michalowski and Knoblock, 2005], although they were implemented *and* propagated differently.

The grid constraint is a new constraint that was not accounted for in the original model. Interestingly, we show, in Corollary 1 in Section 5.4.1, that the BID problem defined in [Michalowski and Knoblock, 2005] is tractable and can efficiently be solved without search. The tractability of the BID problem in the presence of grid constraints remains an open question. Thus, modeling the BID problem as a CSP as advocated in [Michalowski and Knoblock, 2005] is a pertinent approach because it gives us the flexibility to represent arbitrary constraints such as grid constraints and other street-addressing schemas used around the world.

Below we discuss and illustrate each of the above five constraints.

4.2.1 Parity constraints

A parity constraint is a binary constraint that exists between each building variable b and the appropriate orientation variable (i.e., *OddOnEastSide* or *OddOnNorthSide*). This constraint ensures that the parity of the building's number matches what the orientation variable dictates.

If b is a corner-building on streets $s_1, s_2, ..., s_k$, we create a parity constraint between b and the orientation variables corresponding to each street s_i. The constraint for each street s_i requires that if we assign an address on street s_i to building b, the parity of that address must match that of the specified orientation variable. In general, $k \leq 2$. However, it is possible for a corner building to be adjacent to more than two buildings, such as the example shown in Figure 4.2, where building B3 is adjacent to streets S1, S2, and S3. In Figure 4.2, parity constraints exist between the following pairs of variables: (B1, *OddOnNorthSide*), (B1, *OddOnEastSide*), (B2, *OddOnNorthSide*), (B2, *OddOnEastSide*), (B3, *OddOnNorthSide*), (B3, *OddOnEastSide*), and (B4, *OddOnEastSide*). Figure 4.3 shows these parity constraints in the constraint network.

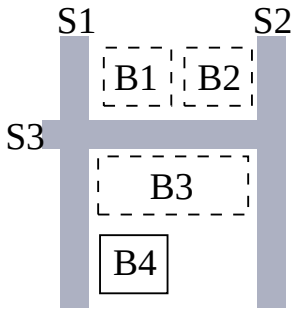

Figure 4.2: Simple BID instance.

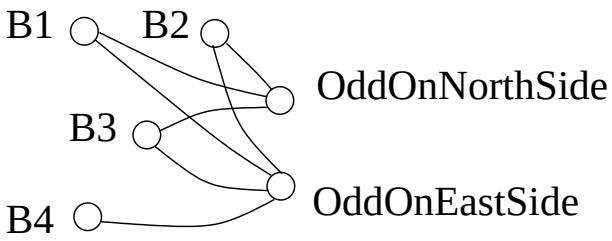

Figure 4.3: Parity constraints for Figure 4.2.

4.2.2 Ordering constraints

An ordering constraint is a ternary constraint over an orientation variable (i.e., *IncreasingNorth* or *IncreasingEast*) and pairs of adjacent buildings whose number must respect a numeric ordering (increasing or decreasing). We distinguish four interesting configurations of increasing complexity for imposing a set of ordering constraints:

1. Between adjacent buildings.
2. Around a corner building.
3. Around multiple corner buildings.
4. Across multiple blocks.

After describing each of these situations, we describe the general rule that defines where ordering constraints exist. To improve the readability of the diagrams in this section, we simply draw an edge between any two buildings that are subject to an ordering constraint, thus abstracting away the orientation variable from the representation.

4.2.2.1 Adjacent buildings

The first situation where an ordering constraint exists is the simplest and most common one. There exists an ordering constraint between any two adjacent non-corner

buildings. Thus, in Figure 4.4, there are two ordering constraints: one between B1 and B2, and one between B2 and B3.

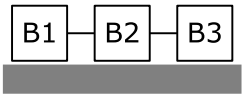

Figure 4.4: Ordering constraints for 3 non-corner buildings.

4.2.2.2 Around a corner building

When creating ordering constraints for a corner building, we do not know a priori on which street the corner building lies. If the corner building is on a different street than the two buildings adjacent to it on the map, then we need to make sure that the adjacent buildings still respect their ordering. Consequently, we add an additional ordering constraint between those two buildings. In the example of Figure 4.5, we add an ordering constraint between B1 and B3. The two ordering constraints between {B1,B2} and {B2,B3} are declared as activation constraints [Mittal and Falkenhainer, 1990]: If B2 is on street S1, then they are deactivated; otherwise, they are active. When B2 is on street S1, the constraint {B1,B3} maintains the correct sequencing of the buildings. If B2 is on street S2, then the constraint {B1,B3} becomes redundant in the sense that it can be inferred from the two existing ones. Generally speaking, redundant constraints are useful to speed-up search when partial look-ahead is used, although they do not affect the number of solutions.

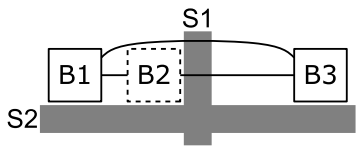

Figure 4.5: Ordering constraints around a corner building.

4.2.2.3 Around multiple corner buildings

When there are two corner buildings in a row, we add an ordering constraint between every pair of buildings, because we cannot predict which of the buildings on the corners will end up on different streets. Figure 4.6 illustrates this situation. Note that the (simplified) graph is a clique.

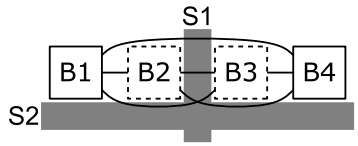

Figure 4.6: Ordering constraints around 2 corner buildings.

In the extreme case of this, we have four corner buildings, and end up with a clique of constraints for *each* row of buildings, as illustrated in Figure 4.7.

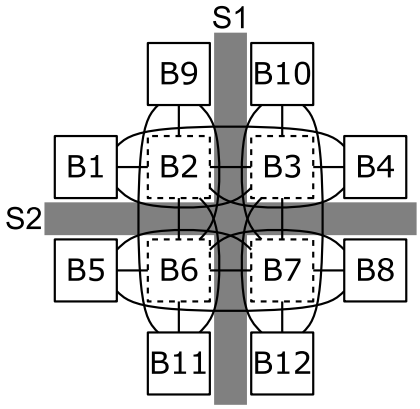

Figure 4.7: Ordering constraints around 4 corner buildings.

4.2.2.4 Across multiple blocks

When a block along a street S has no buildings between two corner buildings, it is possible that neither building on the block appears on street S. Thus, we must have an ordering constraint between the buildings on the adjacent blocks, across the block in question. Figure 4.8 shows such a situation. In this case, we have a clique

of constraints containing each corner building and the non-corner buildings at an extremity. If we have multiple consecutive blocks with only corner buildings, this clique extends across the buildings along all the blocks in the sequence.

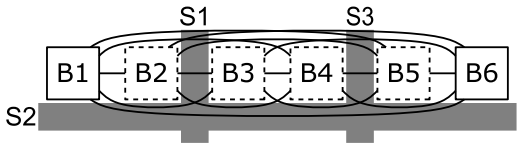

Figure 4.8: Ordering constraints across multiple blocks.

4.2.2.5 General case for ordering constraints

The general rule for creating ordering constraints is as follows. For any sequence of buildings $\langle B_1, B_2, \ldots, B_k \rangle$ along a street where B_1 and B_k are not corner building, but B_i, with $1 < i < k$, is a corner building, every pair of buildings in the sequence must be connected by an ordering constraint.

Thus, the model may have a large number of ordering constraints. However, once we know on which street a building lies, most of the ordering constraints are deactivated or become redundant, and thus can be ignored. Figure 4.9 shows a situation where we have determined that B2, B3, B6, and B7 are on street S1. As a result, the constraint network is decomposed into four independent chains.

4.2.3 Phone-book constraint

For each street S, there exists a phone-book constraint $C_{ph,s}$ whose scope is all buildings adjacent to street S. The constraint requires that we assign each address on S appearing in the phone-book to exactly one building in the scope of $C_{ph,s}$.

In the example Figure 4.10, a phone-book constraint $C_{ph,S1}$ exists for street S1 where the scope of $C_{ph,S1}$ is the set of variables {B1, B2, B4, B6, B7}. $C_{ph,S1}$ requires

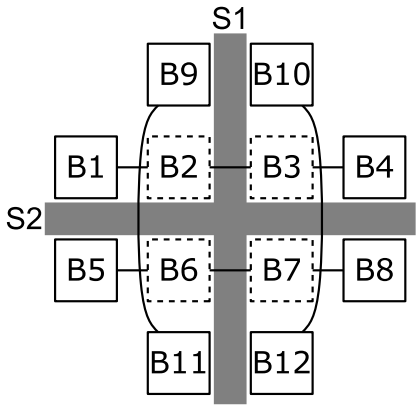

Figure 4.9: Constraints deactivated after street side is known for corner buildings.

that the buildings in its scope use all of the addresses 5, 6, and 11. Similarly, $C_{ph,S2}$ exists for street S2 whose scope is {B1, B2, B3, B4, B5}. $C_{ph,S2}$ requires that the buildings in its scope use all of the addresses 2, 4, and 6. Figure 4.11 shows the constraint graph for this example.

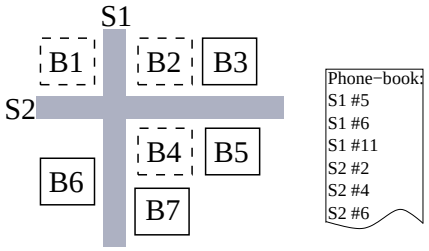

Figure 4.10: Simple BID instance.

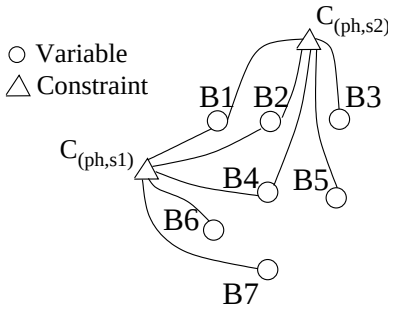

Figure 4.11: Phone-books constraints for Figure 4.10.

4.2.4 Corner constraint

The constraint model has one corner constraint for each corner building on the map. The corner constraint is a binary constraint and is declared over the corner variable Bi-corner and building variable Bi corresponding to a given corner building. The corner constraint is different from the other constraints in that it does not model a property of the underlying building-identification problem. Instead, it is an internal constraint

that ensures that the values assigned to the building variables in the solution are consistent with the values assigned to the hidden corner variables.

In Figure 4.12, three corner constraints exist with the following scopes, where *Bi* is a building variable and *Bi-corner* is the corresponding corner variable: {B1, B1-corner}, {B2, B2-corner}, and {B4, B4-corner}. Figure 4.13 shows the constraint graph for this example.

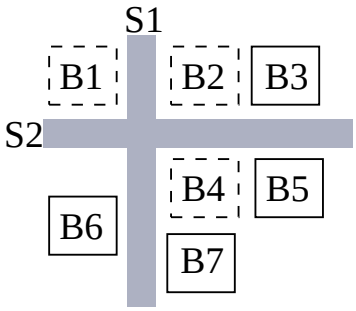

Figure 4.12: A simple BID instance.

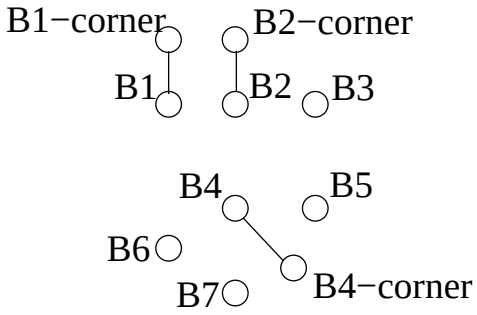

Figure 4.13: Corner constraints for Figure 4.12.

4.2.5 Grid constraint

A grid constraint is a binary constraint that ensures the buildings on different sides of certain grid lines lie in different increments. If the region contains gridlines, we must create a grid constraint for any adjacent buildings that are on opposite sides of a gridline. The grid constraint requires that we assign addresses to the buildings that are in different numeric increments of some value k.

There are two levels of grid constraints: *un-instantiated* gridlines and *instantiated* gridlines. An un-instantiated grid constraint is less restrictive and simply states that addresses increment across gridlines. However, we may known in advance to which numeric increment each gridline corresponds. This information may be available from street-vector data, for example. When we have this information that further specializes the constraint, we call it an instantiated gridline.

Note that if a building adjacent to the gridline is a corner building, we must create redundant grid constraints following the same technique described for ordering constraints in Section 4.2.2.

Grid constraints do not exist in the constraint model of every BID instance. If the region that the CSP models does not contain gridlines, we do not create these constraints.

Assume that there are two grid lines in Figure 4.14 across streets S1 and S2. The constraint model will include four grid constraints with the respective scopes: (B1,B2, IncreasingEast), (B1, B3, IncreasingEast), (B4, B2, IncreasingNorth), and (B5, B2, IncreasingNorth). Figure 4.15 shows the constraint graph for this example.

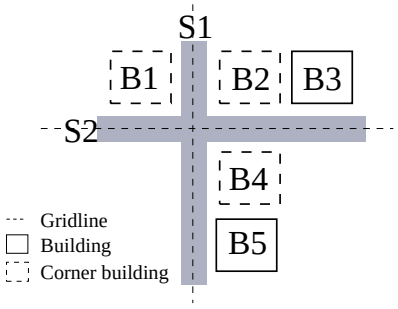

Figure 4.14: Simple BID instance.

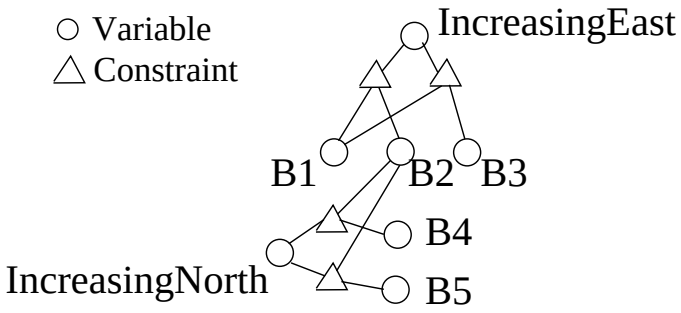

Figure 4.15: Grid constraint for Figure 4.14.

4.2.6 Example constraint network

Figure 4.16 shows a simple instance of the BID problem, and Figure 4.17 shows the corresponding constraint network.

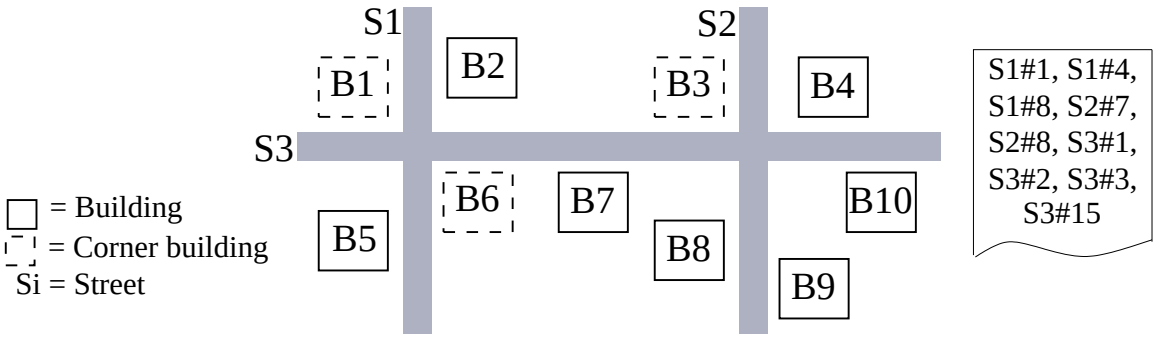

Figure 4.16: A simple instance of the BID problem.

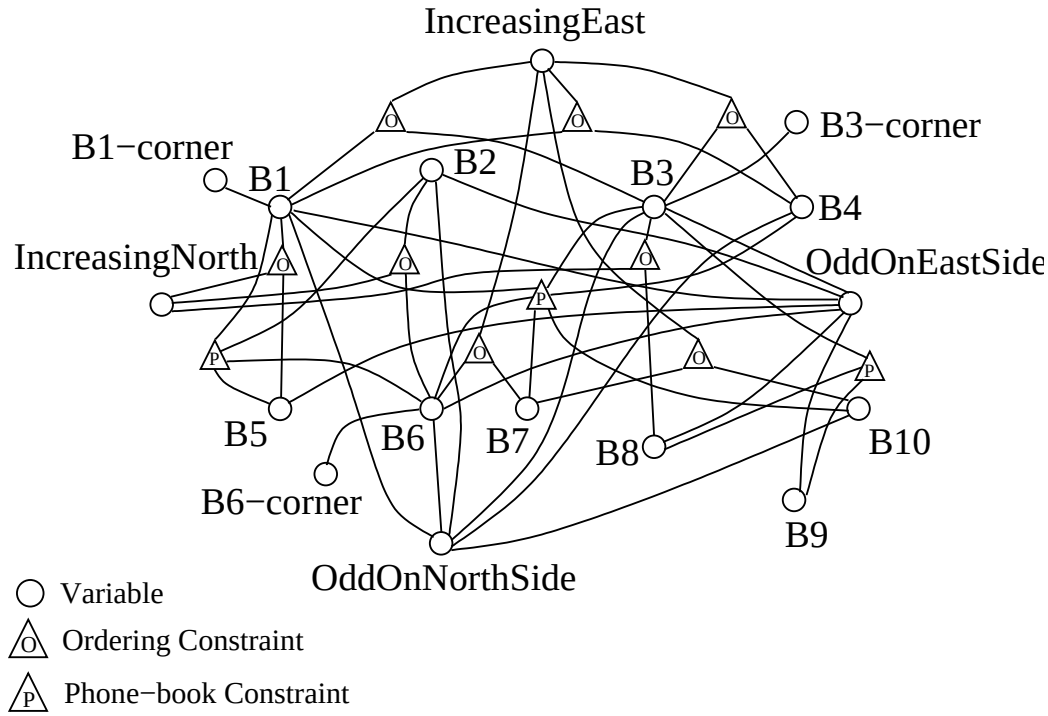

Figure 4.17: Constraint network for Figure 4.1.

4.3 Special configurations

In addition to the basic cases presented above, there are other interesting configuration that occur in real-world problems. It is important for our model to represent those situations correctly. In this section, we identify and describe three such situations. The first situation occurs when a building spans the width of a block, the second situation occurs when a building is adjacent to more than two streets, and the third situation occurs when orientation varies from street to street.

Figure 4.18 illustrates the first case we consider. In this situation, building B5

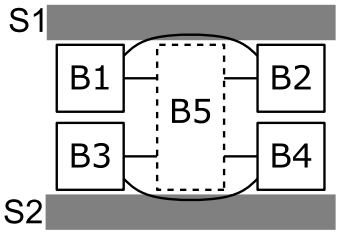

Figure 4.18: Building B5 spans the width of a block.

is not a corner building in the proper sense, because it is not at the intersection of two streets. However, the building is adjacent to multiple streets. Thus, we must determine on which street its address lies, and, in our model, we handle it as a corner building. We use the rules described in Section 4.2.2 for generating the ordering constraints, shown as lines between buildings in Figure 4.18.

Figure 4.19 illustrates the second configuration we identified. In this case, building B6 is adjacent to street S1, S2, and S3, showing how a corner building can be adjacent to more than two streets. This situation is gracefully modeled by including the addresses on all three streets in the domain of the corner variable for building B6.

The third situation that arises is when numbering schemas are not homogeneous across the considered geographical region, as is the case in Belgrade. In most regions, all street sides in a given cardinal direction have the same parity. That is, if odd

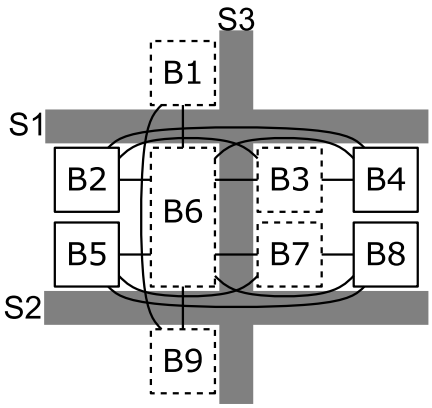

Figure 4.19: Building B6 is adjacent to 3 streets.

numbers appear on the north side of one street, it is likely that they appear on the north side of all streets. Exceptions occur, especially in old-world cities. For example, in Belgrade, each street has its own parity values. Thus, we cannot use a single orientation variable for the parity orientation for the whole region. Rather, we must create a parity orientation variable for each street.

We encountered all cases described in this section in the real-world case studies we investigated. Our solver, which we describe next, is capable of handling each of these situations.

Our preliminary investigations of cities across the US (e.g., Boulder, San Francisco, and Los Angeles) indicate that our model is able to handle most situations. However, as we validate it in other contexts across the world[1], the 'basic' constraint model will have to be enhanced with additional constraints that accurately reflect the numbering schemas adopted around the world. Identifying those constraints and dynamically compiling them into an appropriate constraint model are discussed in [Michalowski *et al.*, 2007a], and constitute the topic of the doctoral research of Martin Michalowski.

[1]For example, to handle the red-black numbering schemas used in Italy [Michalowski, 2006].

4.4 A custom backtrack-search solver (BT)

We implement a custom backtrack-search solver in Java. This solver uses backtrack search (BT) with nFC3, a look-ahead strategy for non-binary CSPs [Bessière *et al.*, 1999], and Conflict-Directed Backjumping [Prosser, 1995]. We call this combination nFC3-CBJ.

Our solver exploits the topology of the problem to improve the performance of search. After we assign streets to the corner buildings, any ordering constraint between a corner building and a building on a different street are deactivated. Thus, once the orientation and corner variables have been assigned, the constraint network becomes a forest. Figure 4.20 illustrates a BID instance after assigning the orientation variables, but before assigning the corner variables. Figure 4.21 shows the same instance after assigning the following streets to corner variables: B1 ← S1, B3 ← S2, B4 ← S3, B6 ← S3, B8 ← S2, and B10 ← S3. The arcs in Figures 4.21 and 4.20 correspond to the ordering constraints, which are binary because the orientation variables have been instantiated.

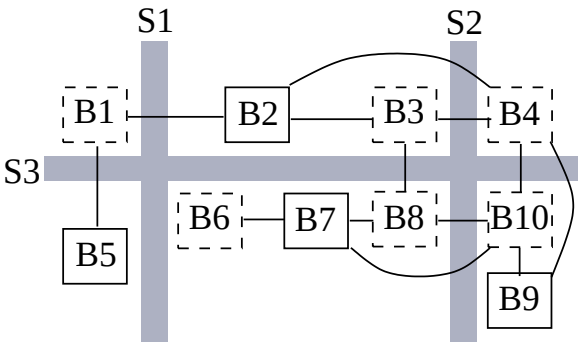

Figure 4.20: BID instance before assigning corner variables.

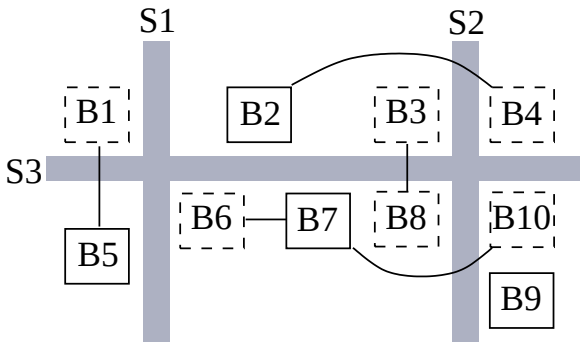

Figure 4.21: BID instance after assigning corner variables.

Freuder [1982] showed that tree-structured constraint graphs can be solved in polynomial time. We first apply arc-consistency directionally from the leaves of the tree to the root, chosen arbitrarily. Then, when no domain is wiped out by directional

arc-consistency, we can instantiate the nodes of the tree in a backtrack-free manner (i.e., in linear time in the number of the nodes). Consequently, in our solver, as long as nFC3 does not cause a domain wipe-out, we can stop search and guarantee solvability without instantiating the remaining variables. In this sense, the corner variables are backdoor variables of the CSP[2] as defined by [Kilby *et al.*, 2005].

In our experiments in Chapter 5, we extend this solver in different ways to test the effectiveness of the different techniques we propose.

4.4.1 Variable ordering

The variables are instantiated in the following order: variables corresponding to landmark buildings, orientation variables, corner variables, and building variables.

When the problem has one or more landmarks (with correct addresses), the corresponding corner and building variables are instantiated immediately and the effect is propagated.

The next step in the search is to instantiate the orientation variables. These variables determine the numbering scheme of the region and their assignment significantly constrains the problem. This intuition is supported by the popular heuristic known as the domain-degree variable-ordering heuristic [Tsang, 1993], which consists in instantiating first the variable whose ratio of domain size to degree (in the graph) is smallest. Indeed, the orientation variables are Boolean. Further, each of the ordering and parity variable pairs is connected to all the building variables. Thus, the degrees of the orientation variables are high.

The third set of variables to be instantiated are the corner variables. They are instantiated before the building variables because their assignment decomposes the

[2]Backdoor variables are those variables whose instantiation reduce the CSP into a tractable problem [Kilby *et al.*, 2005].

problem into independent sub-problems each of which is a chain, one for each side of each street.

Finally, the last stage of search assigns values to each building variable. These values are the numeric addresses assigned to each building. Note that, for the reasons discussed above, we can determine the satisfiability of the problem without instantiating any of these variables.

4.4.2 Implementing and checking the constraints

We implement all the constraints in intension. This implementation makes constraint checking more efficient. However, it prevents us from easily enforcing higher levels of consistency in the network. Our implementation of nFC3, which is standard, operates on each constraint individually and does not propagation the join multiple constraints.

Below we describe how we propagate each constraint. We discuss constraint checking in general, regardless of the variable ordering chosen during search. Note that the filtering mechanism depends on which variables in the scope, if any, are instantiated.

Parity constraint: If search has instantiated the orientation variable, we filter all values from the building variable's domain that do not match the parity specified by the orientation variable.

If neither the orientation variable nor the building variable in the scope of a parity constraint is instantiated, we iterate through the domain of the building variable to determine whether all remaining values have the same parity. If they all have the same parity, we force the orientation variable to the corresponding value. Otherwise, we perform no filtering.

Ordering constraint: If the orientation variable has been instantiated, the ordering constraint becomes a GREATER-THAN constraint: $B_i > B_j$ where B_i, B_j are two building variables in the scope of an ordering constraint. We propagate this constraint only on the boundaries of each interval, as in bound or box consistency [Benhamou *et al.*, 1994]. Thus, we remove all values d_i from the domain of B_i where $d_i \leq min(B_j)$ and all values d_j from the domain of B_j where $d_j \geq max(B_i)$.

If the orientation variable is not instantiated, we check whether all values in the domain of B_i are uniformly either less than or greater than all values in the domain of B_j. If they are, then we can filter the ordering variable's domain to only contain the orientation reflecting the ordering relation that holds between the domains of B_i and B_j.

Phone-book constraint: The phone-book constraint for street S computes the following parameters:

$p_u =$ the number of phone-book addresses assigned to building variables along street S

$p_r =$ the number of addresses for street S in the phone book

$b_a =$ the total number of un-instantiated buildings along street S.

We distinguish three possibilities:

1. $b_a < p_r - p_s$*:* There are not enough buildings to use all required phone-book addresses, and search must backtrack.

2. $b_a = p_r - p_s$*:* There are exactly enough buildings left along street S to use the required addresses. All remaining buildings along street S must have an address from the phone-book. We filter the domains of these buildings to only contain phone-book addresses.

3. $b_a > p_r - p_s$: There are more than enough buildings to use all required phone-book addresses. The constraint cannot perform any filtering.

Corner constraint: When a corner variable is instantiated, we remove all values from the domain of the building variables that do not correspond to addresses on the street specified by the corner variable.

When a building variable is instantiated, we filter the domain of the corresponding corner variable to contain only the value of the street assigned to the building variable.

Grid constraint: The implementation of the grid constraint is similar to that of the ordering constraint, except that we must account for the numeric increment k across grid lines. The street addresses a_i, a_j of two building variables B_i, B_j connected by a grid constraint must satisfy exactly one of the two relations:

$$a_i - k_i < a_j - k_j \tag{4.1}$$

$$a_i - k_i > a_j - k_j \tag{4.2}$$

where $a_i \equiv k_i \pmod{k}$, and $a_j \equiv k_j \pmod{k}$.

4.4.3 Domain representation

The domains for the building variables are stored as follows:

- Every domain is a set of intervals, one interval for each street to which the building is adjacent.
- Every interval is stored as a sorted list of values.
- We propagate ordering and grid constraints only on the bounds of an interval, as in box consistency. This operation is possible because each interval is stored

in a sorted list.

- If a constraint does not operate on the endpoints of an interval, such as the phone-book constraints, we filter the set of values directly. Any filtering must maintain the ordering of the remaining values. That operation does not require any extra effort, because the values are already sorted, and removing values does not disrupt the sorting of the remaining values.

- In Section 3.2 we introduced a new constraint, the AllDiff-Atmost constraint. In Section 5.3, we argue that each phone-book constraint entails a set of AllDiff-Atmost constraints, which we add to the constraint model. We reformulate the domains of the variables in the scope of these new constraints to reduce their size. This domain reformulation has no effect on the way we store and propagate domains, because the reformulated domains still respect the total ordering of the numeric values.

4.5 Building the constraint model of the BID problem

The solver reads problem instances from a set of XML files. These XML files describe the basic description of a problem instance and information about landmarks or additional constraints that hold in the region. A problem instance is described with the following XML files:

1. *Layout:* The layout file describes layout of the buildings and streets on the map.

2. *Phone-book:* The phone-book file lists the known phone-book addresses for the problem instance.

3. *Grid:* The grid XML file describes any gridlines in the problem instance.

4. *Landmark:* The landmark XML file describes any landmarks with known addresses.

Appendix A describes the schema for each XML file. Note that the landmark files and grid files are optional. For example, depending on the information available, a problem instance may or may not have grid constraints.

4.6 Improvements over the previous model

Our new model improves the original one proposed in [Michalowski and Knoblock, 2005] as follows.

- The number of variables for non-corner buildings is reduced by half, reducing number of variables between 37% and 43% in our test cases.

- Domains of the building variables in [Michalowski and Knoblock, 2005] were enumerated and upper bounds chosen arbitrary. We represent them as intervals with potentially infinite bounds in the new model.

- We reduced constraint arity from four to two for parity constraints, and from six to three for ordering constraints.

- Corner constraints are new and allow early decomposition of the problem.

- Grid constraints are also new and allow a more precise modeling of the real world. Interestingly, we show, in Section 3.3, that in the absence of grid constraints, the BID problem is tractable. The tractability of the BID problem in the presence of grid constraints remains an open question.

Summary

In this chapter, we presented a CSP model for the BID problem. We described our custom solver that uses backtrack search to find solutions to BID instances modeled as CSPs. We also described how our model improves over the model described in [Michalowski and Knoblock, 2005]. We use this model and solver as the basis for our experiments in Chapter 5

Chapter 5

Reformulating the BID problem

In this chapter, we discuss the application of the reformulation techniques introduced in Chapter 3 to the BID problem defined in Chapter 4. First, we describe the BID case studies. Then, we describe the application of each of the three reformulations to the BID problem, and report experimental results that demonstrate the benefits of each reformulation in terms of improved runtime performance. We also briefly discuss the use of the fourth technique in this context. Finally, we discuss the quality of the solutions we found to the BID instances.

5.1 Description of the case studies of the BID problem

We test and validate our approach on several different regions of the city of El Segundo, CA. These cases vary in size and which regions of the city they cover. Table 5.1 describes the properties of the regions on which we ran our experiments, listed in increasing number of corner buildings. The largest region tested by [Michalowski and Knoblock, 2005] contained 34 buildings and a single city block. All of the areas we

test represent an increased problem size over their work. The completeness of the phone book indicates what percent of the buildings on the map have a corresponding address in the phone book. We created the complete phone books using property-tax data, and the incomplete phone books using the real-world phone-book. The number

Table 5.1: Case studies used in experiments.

Case study	**Phone-book completeness**	**Number of...** bldgs	corner bldgs	blocks	building-address combinations
NSeg125-c	100.0%	125	17	4	4160
NSeg125-i	45.6%				1857
NSeg206-c	100.0%	206	28	7	4879
NSeg206-i	50.5%				10009
SSeg131-c	100.0%	131	36	8	3833
SSeg131-i	60.3%				2375
SSeg178-c	100.0%	178	46	12	4852
SSeg178-i	65.6%				2477

of building-address combinations is the number of possible combinations of buildings and phone-book addresses and corresponds to the number of vvps tested in the loop at Line 10 in Algorithm 1. Note that this number is smaller when the phone book is incomplete than when it is complete. Each call to Line 11 was timed out after one hour. We report the number of timed out executions in our results.

In our experiments, we assume that we know the values of the orientation variables in advance. If we do not know these values, the number of vvps we must test would be multiplied by 16, because we would have to test each building-address combination for each possible combination of assignments to the orientation variables. When grid constraints are not present, we do not need to test all values for the ordering orientation variables, because any solution with one set of assignments to the ordering variables has a symmetric solution with the opposite assignments to the ordering

variables.

5.2 Query reformulation in the BID problem

The straightforward approach taken by [Michalowski and Knoblock, 2005] consisted in enumerating all possible solutions to the CSP, which is prohibitively expensive in practice. After all solutions are enumerated, they iterated through the set of solutions to collect, for each building, the set of addresses to which it is assigned in some solution. A significant challenge when solving the BID problem is the large number of possible solutions. The problem is often under-constrained, leading to a large number of consistent solutions. This fact is true especially in cases where the phone book is incomplete. The large number of solutions makes enumeration impractical in most cases. By reformulating the query as proposed in Section 3.1, we can use Algorithm 1 to obtain the same result at a much cheaper cost. In summary, we replace the query: "Enumerate all solutions and collect the addresses taken by the buildings in these solutions" with the query "Find all the addresses that a given building can take."

For a CSP with with n variables of domain size d, Algorithm 1 tests $O(nd)$ CSPs for solvability. This operation appears costly, but in cases where the original CSP has significantly more than nd solutions, Algorithm 1 can perform significantly better than enumerating all solutions to the CSP. Further, because the corner variables are hidden, Line 10 iterates over *only* all vvps for the building variables. Thus, n is the number of buildings, which is smaller than the number of variables in the CSP, and d is the number of phone book addresses.

Note that, for the reasons described in Section 4.4, our BT solver needs to instantiate only the corner variables to determine satisfiability. However, the vvps we

consider in the loop at Line 10 in Algorithm 1 are the vvps for the building variables. Our solver never instantiates any vvp that we could remove from the vvp list for future iterations, and thus we cannot take advantage of the improvement to Algorithm 1 described in Section 3.1.

Table 5.2 shows the results of experiments comparing an exhaustive BT solver (exh-BT), which enumerates all solutions, with Algorithm 1 using our BT solver in Line 11 (Alg1+BT). The $*$ values indicate that the results could not be obtained because exh-BT did not terminate within a week of CPU time. We report results on instances both with and without grid constraints.

Table 5.2: Exhaustive BT (exh-BT) vs. Algorithm 1 using BT in Line 11 (Alg1+BT). Note that the runtimes would be higher by a constant factor if we did not assume that we knew the correct values for the orientation variables.

Case study	Model without grid constraints			Model with grid constraints		
	Runtime [sec]		**#CSP**	**Runtime** [sec]		**#CSP**
	exh-BT	**Alg1+BT**	**Solutions**	**exh-BT**	**Alg1+BT**	**Solutions**
Case studies with an incomplete phone book						
NSeg125-i	>1 week	744.7	$*$	>1 week	1232.5	$*$
NSeg206-i	>1 week	14818.9	$*$	>1 week	4052.8	$*$
SSeg131-i	>1 week	66901.1	$*$	>1 week	114405.9	$*$
SSeg178-i	>1 week	119002.4	$*$	>1 week	138404.2	$*$
Case studies with a complete phone book						
NSeg125-c	1.5	139.2	1	1.1	100.8	1
NSeg206-c	20.2	4971.2	7	4.3	2277.5	1
SSeg131-c	1123.4	38618.4	2117	220.9	17063.3	299
SSeg178-c	3291.2	117279.1	3751	115.2	78528.6	19

In cases with incomplete phone books, Alg1+BT allows us to solve instances that exh-BT cannot handle in a reasonable amount of time. However, with a complete phone book, the CSP has only a small number of solutions, and the overhead of the additional backtrack searches performed by Algorithm 1 causes an increase in runtime. Obviously, adding grid constraints to a case study reduces the number of

solutions. Interestingly, this reduction directly translates into runtime improvement for both exh-BT and Alg1+BT when the phone book is complete (e.g., the CSP is highly constrained), however, the effect is unpredictable when the phone book is incomplete. While in the case studies with a complete phone book Alg1+BT performed significantly worse than exh-BT, the performance improvement in the incomplete phone-book cases is substantial. Unless otherwise noted, in all remaining experiments, we use solvers based on Algorithm 1.

5.3 Using AllDiff-Atmost in the BID problem

When the number of buildings on the map is equal to the number of addresses in the phone book, the phone book is complete. If the phone book is incomplete, we must infer the missing numbers to add to the variables' domains. [Michalowski and Knoblock, 2005] proposed to enumerate all numbers between 1 and the largest address that appears on the street. Their approach has two problems:

1. The choice of the upper limit is arbitrary. When the largest address is not in the phone book, this approach may yield incorrect solutions. As a result of this problem, Michalowski and Knoblock reported the elimination of correct solutions as a drop in recall, where recall is the percent of buildings whose solution contains the correct address.

2. The size of the domains becomes prohibitively large on real-world data. We propose to infer from the phone-book constraint a set of AllDiff-Atmost constraints, which we add to the constraint model. We then reformulate the domains of the variables in the scope of the AllDiff-Atmost constraints to reduce their size by using symbolic values as explained in Section 3.2.

In this section we first describe how to add AllDiff-Atmost constraints to the constraint model of a BID instance. Then, we present results that show the benefits of the domain reformulation on AllDiff-Atmost constraints in terms of both domain-size reduction and improved runtime performance.

5.3.1 Modeling the BID problem with AllDiff-Atmost

First, we assume that the BID instance does not have any grid constraints. We then describe how to adapt the reformulation in the presence of grid constraints.

Let S be a street, $P_S = \{a_1, \ldots, a_n\}$ its set of phone-book addresses of a given parity, B_S the set of buildings on the side of S of that parity, and [min,max] the range of address numbers on that side of S. The address numbers in P_S partition [min,max] into consecutive convex intervals. For any pair of addresses a_i,$a_j \in P_S$, we cannot use more than $|B_S|-|P_S|$ addresses in the interval (a_i,a_j), which we express as AllDiff-Atmost(B_S,k_a,(a_i,a_j)) with k_a=minimum($|B_S|$-$|P_S|$,$\lfloor\frac{(a_j-a_i)-1}{2}\rfloor$). We also create the corresponding constraints for the boundary intervals [min, a_1) and ($a_{|P_S|}$, max]. We reduce the variables' domains using the reformulation of Section 3.2 on each of these intervals.

For example, assume we have, on the even side of S, B_S={B_1, B_2, ..., B_6}, P_S={S#112, S#118, S#316}, min=2, and max=624. An assignment cannot use more than 3 numbers in each of the intervals:

[2,112), (112,118), (118,316), and (316,624],

yielding four AllDiff-Atmost constraints with the following arguments:

(B_S,3,[2,112)), (B_S,2,(112,118)), (B_S,3,(118,316)), and (B_S,3,(316,624]).

We replace the domain [2,624] of each variable with the significantly smaller set:

$$\{s_1, s_2, s_3, 112, s_4, s_5, 118, s_6, s_7, s_8, 316, s_9, s_{10}, s_{11}\},$$

where s_1,s_2,$s_3 \in$ [2,112), s_4,$s_5 \in$ (112,118), s_6,s_7,$s_8 \in$ (118,316), s_9,s_{10},$s_{11} \in$ (316,624],

and $s_i<s_j$ for $i<j$. This process allows us to choose an arbitrarily large, even infinite, upper bound (max) on a given street. Figure 5.1 illustrates this reformulation.

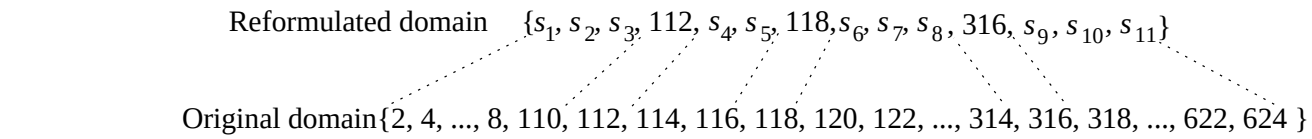

Figure 5.1: Domain reformulation for the BID problem.

When grid constraints are present, the reformulation is slightly different. We define S, P_S, B_S and $[min, max]$ as above. Additionally, we define G as the set of numbers corresponding to the increments of the grid lines. The elements of G are defined as follows. Let k be the increment of address values and n be the numbers of elements of G. We have:

$$n = \lfloor \frac{max}{k} \rfloor - \lfloor \frac{min}{k} \rfloor \tag{5.1}$$

and

$$G = \{(i + \lfloor \frac{min}{k} \rfloor)k \mid 1 \leq i \leq n\} \tag{5.2}$$

The numbers in $P_S \cup G$ partition [min,max] into consecutive convex intervals. The reformulation continues as above, with one change. For a pair of consecutive addresses $a_i, a_j \in (P_S \cup G)$ with $a_i \in G$, we create an AllDiff-Atmost constraint for the interval $[a_i, a_j)$ instead of using the interval (a_i, a_j). We close the lower bound of the interval in this case, because a_i does not correspond to a phone-book address, and should be included in the set of numeric values replaced by symbolic values.

For example, suppose our example above has grid lines with an increment value k=100. We have n=6 and G={100,200,300,400,500,600}. We specify the domain as the set of intervals:

[2,100), [100,112), (112,118), (118, 200), [200,300),

[300,316), (316, 400), [400,500), [500,600), and [600,624],

and generate an AllDiff-Atmost constraint for each interval.

5.3.2 Evaluating the effects of domain reformulation

Table 5.3 shows the benefit of domain reformulation by comparing the use of original enumerated domains and the reformulated ones. The experiment uses the search solver (BT) of Section 4.4. We report the results on problems with (top) and without (bottom) grid constraints. The assumption that we know the correct values for the orientation variables does not affect the domain size reduction. When the phone book

Table 5.3: BT runtime using numeric and symbolic values.

Case study	**Avg. domain size**			**Runtime** [sec]			**Timeouts**	
	Orig.	Ref.	% Improved	Orig.	Ref.	% Improved	Orig.	Ref.
Model without grid constraints								
NSeg125-i	1103.1	236.1	78.6%	2943.7	744.7	74.7%	0	0
NSeg206-i	1102.0	438.8	60.2%	14818.9	5533.8	62.7%	0	0
SSeg131-i	792.9	192.9	75.7%	67910.1	66901.1	1.5%	18	17
SSeg178-i	785.5	186.3	76.3%	119002.4	117826.7	1.0%	32	29
Model with grid constraints								
NSeg125-i	1103.1	266.8	75.8%	2176.5	1232.5	43.3%	0	0
NSeg206-i	1102.0	490.8	55.5%	19948.3	4052.8	79.7%	2	0
SSeg131-i	792.9	222.3	72.0%	131008.5	114405.9	12.7%	36	30
SSeg178-i	785.5	223.0	71.6%	153140.0	138404.2	9.6%	42	35

is complete, no AllDiff-Atmost constraints exist, and the original and reformulated representations are identical. We report results on the incomplete phone-book cases in terms of CPU time, number of timeouts, and domain size. The number of timeouts indicates the number of times the call to Line 11 in Algorithm 1 could not be completed within an hour. In the case of the incomplete phone-book, the advantage of the reformulation is clear, and increases with the incompleteness of the phone-book. We use the domain reformulation in all the experiments reported in the remainder of this chapter.

5.4 Reformulating the BID problem as a matching problem

The constraint model of Michalowski and Knoblock [2005] included the constraints listed in Chapter 4 except for the grid constraint. We show below that, without the grid constraint, the BID problem can be modeled as a matching in a bipartite graph, and is thus tractable. *Note that the CSP approach remains relevant in that it allows one to represent arbitrary street-addressing schemas used around the world, such as the grid constraints.* We propose the removal of grid constraints as a tractable relaxation of the BID problem.

In this section, we first describe the reformulation of the BID problem as a matching problem. We then present results that use this reformulation both to solve the problem directly (when no grid constraints exist) and as a tool before and during search (when grid constraints exist).

5.4.1 Ignoring the grid constraints in the BID problem

Given an instance of this problem without grid constraints, we construct a bipartite graph $G = (B \cup S, E)$ as follows. First, assume an assignment to the orientation variables (there are 2^4 such assignments). For each building β in the problem, add a vertex b to B. For each street σ in the problem, add two vertices s_{odd} and s_{even} to S, one for each side of the street. For each building β, add an edge between vertex b and the street vertex corresponding to the street side on which β may be. (Note that corner buildings are on two streets.) Assuming odd numbers appear on the North and West sides of the street, Figure 5.3 shows the construction of G for the map in Figure 5.2. Theorem 1 uses this construction to determine solvability of the problem.

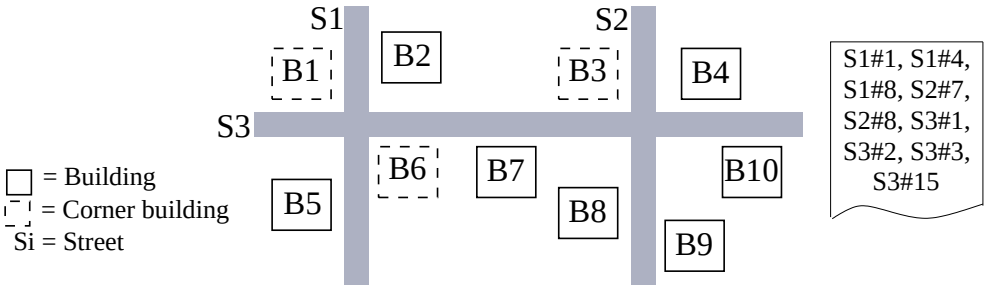

Figure 5.2: A BID instance .

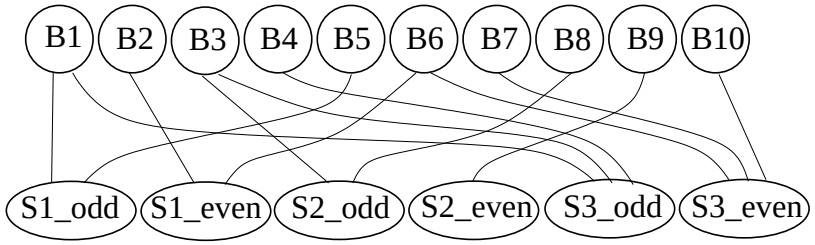

Figure 5.3: Graph construction for the example of Figure 5.2.

Theorem 1 *$\forall b \in B$, let $m(b) = 1$. $\forall s \in S$, let $m(s) = |P_\sigma|$, where P_σ is the set of phone-book addresses that correspond to σ's street and parity. There exists a solution to the relaxed BID problem represented by G iff there exists a matching that saturates S.*

Proof 1 ($\Rightarrow$) Given a solution to the relaxed BID problem, we construct a matching M as follows. $\forall \beta, \alpha$ such that the solution assigns address α to building β, let $b \in B$ be the vertex corresponding to building β, and let $s_i \in S$ be the vertex corresponding the street and parity of α. Add edge $e = (b, s)$ to M. A solution to the problem assigns exactly 1 address to every building, therefore e is the only edge incident to b in the matching M. Because a solution assigns all the addresses in the phone book to some building, the number of edges incident to s_i in M is equal to $m(s_i)$. Therefore, M is a matching that saturates S.

($\Leftarrow$) Given a matching M, we will construct a solution to the relaxed BID problem. We can represent each edge $e \in M$ as $e = (b, s)$, where $b \in B$ and $s \in S$. $\forall e \in M$,

assign the building β corresponding to b to the street and parity specified by s. When the phone book is incomplete, M does not saturate B. In this case, for any remaining vertex $b' \in B$ assign the street and parity to which the b' is adjacent to the building β' represented by b' in G. Every building now has a street and parity, it only needs a street number. Starting from the street's end with smaller addresses (depending on the orientation chosen), assign numbers to buildings by taking the numbers from the phone book for that street and parity in increasing order. After assigning all numbers from the phone book, continue assigning arbitrary numbers of increasing value. The addresses are thus assigned to satisfy the ordering and parity constraints. The match counts of vertices in S guarantee that each street has at least as many buildings assigned to it as it has phone-book addresses. Hence, the phone-book constraint is satisfied. □

Figure 5.4 shows a satisfying matching for the graph from Figure 5.3, where the edges included in the matching are darkened. The numbers in parentheses indicate the match count of the vertex.

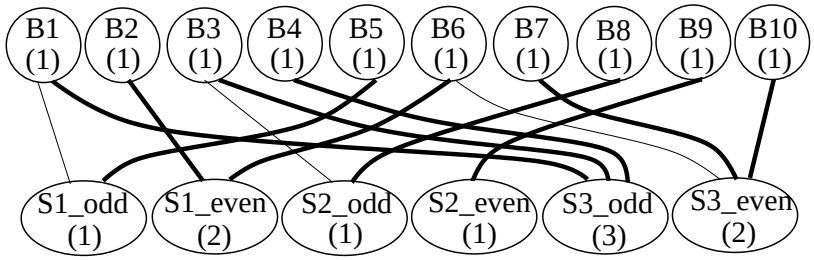

Figure 5.4: Satisfying matching for Figure 5.2.

A maximum matching in the graph described above corresponds to an assignment of streets to buildings that is guaranteed to satisfy the phone-book constraint. Given an assignment of streets to all buildings, we can assign numbers to the buildings as a post-processing step. Thus, while the matching reformulation does not provide a

complete solution to the BID problem, we can easily generate a complete solution using a construction such as the one described in Proof 1.

Note that the set of vertices that the matching must saturate corresponds to the phone-book addresses. In the matching relaxation, the 'tasks' are the phone-book addresses, and the goal is to assign a building to each address, whereas in the CSP model we assign addresses to buildings. Thus, this reformulation constitutes a dual viewpoint of the problem, as described in [Geelen, 1992].

While the matching reformulation is powerful, it does not take into account grid constraints. Nevertheless, Theorem 1 allows us to state that the problem proposed in [Michalowski and Knoblock, 2005] is tractable.

Corollary 1 *The BID problem with parity, ordering, corner, and phone-book constraints, but without the grid constraints, can be solved in polynomial time.*

To compute a maximum matching, we use the $O(n^{5/2})$ algorithm by Hopcroft and Karp [1973]. Whether or not the BID problem with the grid constraints can be solved efficiently remains an open question.

5.4.2 Exploiting the matching reformulations in the BID

We propose to use the matching relaxation in four different ways: one when a problem instance has no grid constraints, and three when it does.

5.4.2.1 In the absence of grid constraints

When a problem instance has no grid constraints, we use the matching as an exact approximation of the BID problem. We efficiently solve the problem with Algorithm 1 while using the bipartite-matching algorithm (which we call the *matching solver*) in

Line 11, instead of backtrack search. Figure 5.5 shows the flowchart for the matching solver, which implements Line 11 in Algorithm 1.

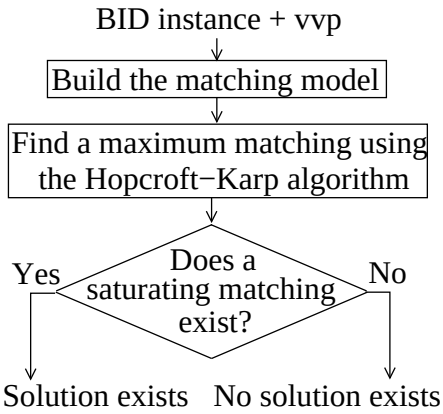

Figure 5.5: Flowchart for the matching solver.

To illustrate the effectiveness of this reformulation, we evaluate the performance improvement obtained by replacing the BT solver in Line 11 of Algorithm 1 (Alg1+BT) with the matching solver (Alg1+matching). Because Line 11 is called $O(nd)$ times, and each execution of the matching solver runs in $O(n^{5/2})$ time, Alg1+matching runs in polynomial time while Alg1+BT runs in exponential time. Table 5.4 gives the runtime for both algorithms. As expected, the polynomial-time matching solver has significantly better performance than the exponential-time BT solver. Note that Alg1+matching is quicker when the phone book is incomplete than when it is complete because fewer vvps (i.e., building-address combinations) are tested.

5.4.2.2 In the presence of grid constraints

When an instance has grid constraints, we can use the matching as a sufficient approximation of the relaxed problem (i.e., ignoring the grid constraints). This relaxation can in turn be used in three different ways: two for pre-processing and one for lookahead. Below, we explain these mechanisms and evaluate them on our case studies.

Table 5.4: In the absence of grid constraints, Alg1+matching efficiently solves the BID problem.

Case study	**Runtime** [sec]	
	Alg1+BT	Alg1+matching
NSeg125-c	139.2	4.8
NSeg125-i	744.7	2.5
NSeg206-c	4971.2	16.3
NSeg206-i	5533.8	8.5
SSeg131-c	38618.3	7.3
SSeg131-i	66901.1	3.1
SSeg178-c	117279.1	22.5
SSeg178-i	117826.7	4.9

Preproc1. Line 3 in Algorithm 3 is a pre-processing step, which we call Preproc1. It eliminates some vvps that cannot possibly appear in a solution. We implement this pre-processing step by adapting the Régin [1994] algorithm for all-diff constraints to operate on our matching model. Figure 5.6 shows the flowchart for Preproc1.

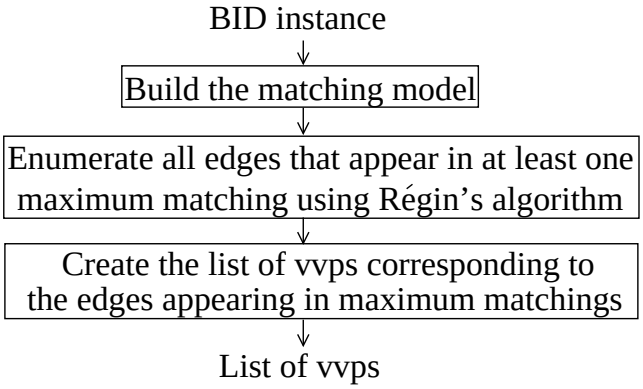

Figure 5.6: Flowchart for Preproc1.

Preproc2. Line 6 in Algorithm 3 is another pre-processing step, which we call Preproc2. It determines whether or not the sufficient approximation of a BID problem where a vvp has been assigned is solvable. We implement this line using the matching solver shown in Figure 5.5. If no solution exists to the relaxed problem, we can avoid

performing the costly operation at Line 7 of Algorithm 3, which must be implemented by backtrack search as the problem has grid constraints.

Lookahead. Line 7 of Algorithm 3 executes our backtrack-search solver introduced in Section 4.4, which is enhanced with domain reformulation. Figure 5.7 shows the flowchart for the BT solver.

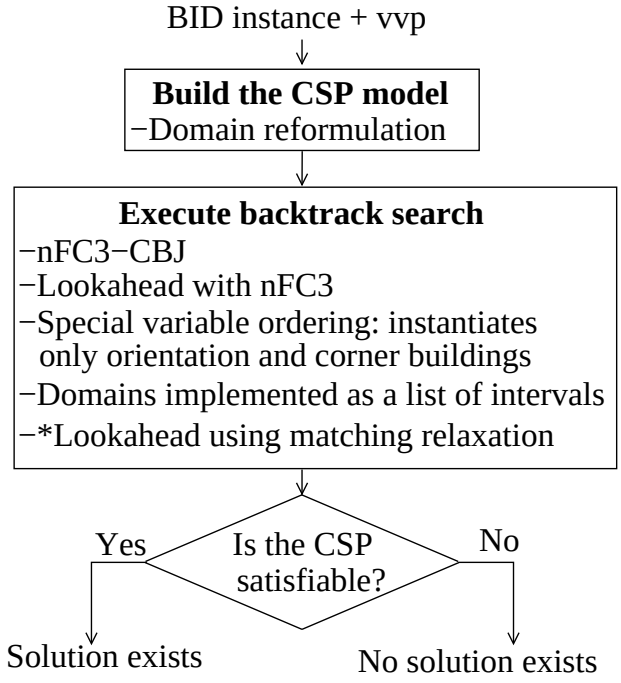

Figure 5.7: Flowchart for the BT solver.

In addition to nFC3, Figure 5.7 includes a second lookahead mechanism[1] that is based on the matching relaxation of the BID problem, as suggested in Section 3.3.3. In the CSP model, a partial solution corresponds to a set of assignments to orientation and corner variables, but not to building variables. Thus, using the matching as a lookahead mechanism filters the domains of the corner variables, but not those of the building variables.

[1]Marked with an * in Figure 5.7.

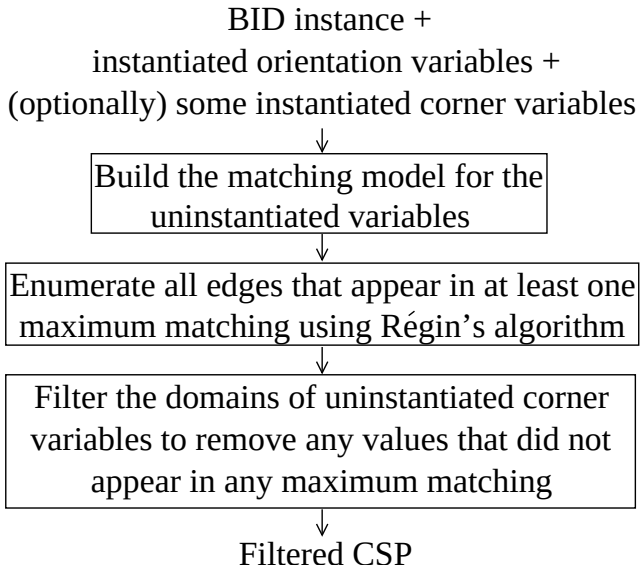

Figure 5.8: Flowchart for the matching relaxation as a lookahead mechanism.

Evaluation of matching relaxation on the BID problem The experiments in this section are conducted over case studies that include grid constraints. Consequently, the reformulation of the BID problem as a matching is only a necessary approximation. The goal of the evaluation is to assess the benefits of Preproc1, Preproc2, and the lookahead based on the matching relaxation when integrated in Algorithm 3. Figure 5.9 shows the flowchart for our implementation of Algorithm 3, and asterisks indicate components that were included or removed for sake of comparison. We consider the following combinations of techniques, all of which yield exactly the same solutions: BT, BT+Lookahead, Preproc2+BT, Preproc2+BT+Lookahead, and Preproc1&2+BT+Lookahead.

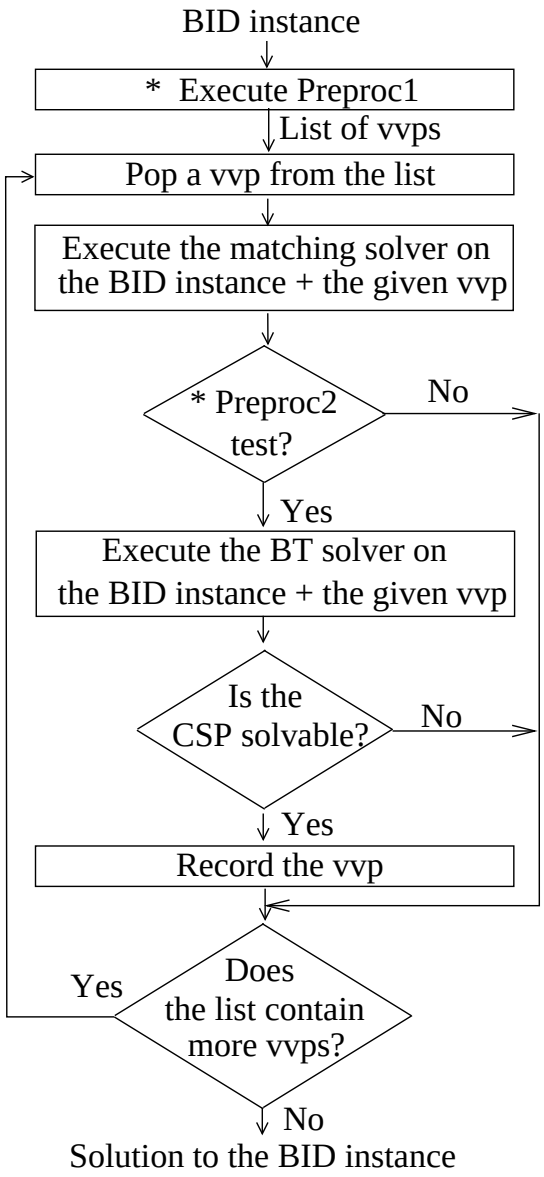

Figure 5.9: Flowchart for Algorithm 3 adapted to BID.

Note that the matching reformulation requires an assignment to the orientation variables. Thus, all experiments that use the matching reformulation assume we know the correct values for the orientation variables. Table 5.5 shows the results of these experiments. These results indicate that, *in general*, the integration of the matching with BT improves performance. There are some exceptions.

- While Preproc2 avoids a large number of calls to BT, in cases such as SSeg131-i, marked with a *, those calls corresponded to searches that terminated quickly. Thus, the overhead of computing the matching for the pre-processing step exceeded the time saved by avoiding these searches.

- We see a similar problem in all cases for Preproc1, except those marked with **, where the pre-processing either did not filter any vvps or the benefit did not significantly exceed the overhead. Preproc1 can filter an assignment to a corner variable only when corresponding building cannot be placed on a street to which it is adjacent in any solution. This situation is not common, so Preproc1 improved performance significantly only in the two cases marked with **.

- Similarly, when using the matching as a lookahead mechanism, there were cases such as NSeg125-c, marked with ***, where the overhead of running the algorithm of [Régin, 1994] exceeded the performance improvement gained by the additional filtering.

Table 5.5: Improvements due to pre-processing and lookahead.

NSeg125-c + grid	**CPU [sec]**	**#Timeouts**	**%Calls saved**
BT	100.8	0	-
Preproc2+BT	33.2	0	97.0%
BT+Lookahead	*** 140.2	0	-
Preproc2+BT+Lookahead	39.6	0	97.0%
Preproc1&2+BT+Lookahead	** 29.8	0	99.3%
NSeg125-i + grid	**CPU [sec]**	**#Timeouts**	**%Calls saved**
BT	1232.5	0	-
Preproc2+BT	1159.1	0	62.6%
BT+Lookahead	726.6	0	-
Preproc2+BT+Lookahead	701.1	0	62.6%
Preproc1&2+BT+Lookahead	701.2	0	62.6%
NSeg206-c + grid	**CPU [sec]**	**#Timeouts**	**%Calls saved**
BT	2277.5	0	-
Preproc2+BT	614.2	0	97.0%
BT+Lookahead	1559.2	0	-
Preproc2+BT+Lookahead	443.8	0	97.0%
Preproc1&2+BT+Lookahead	** 309.9	0	97.9%
NSeg206-i + grid	**CPU [sec]**	**#Timeouts**	**%Calls saved**
BT	4052.8	0	-
Preproc2+BT	3806.7	0	87.8%
BT+Lookahead	3499.5	0	-
Preproc2+BT+Lookahead	3510.0	0	87.8%
Preproc1&2+BT+Lookahead	3509.9	0	87.7%
SSeg131-c + grid	**CPU [sec]**	**#Timeouts**	**%Calls saved**
BT	17063.3	0	-
Preproc2+BT	5997.9	0	92.5%
BT+Lookahead	9745.8	0	-
Preproc2+BT+Lookahead	4256.0	0	92.5%
Preproc1&2+BT+Lookahead	4256.1	0	92.5%
SSeg131-i + grid	**CPU [sec]**	**#Timeouts**	**%Calls saved**
BT	114405.9	30	-
Preproc2+BT	* 114141.3	29	74.2%
BT+Lookahead	107896.3	30	-
Preproc2+BT+Lookahead	108646.5	30	74.2%
Preproc1&2+BT+Lookahead	108646.6	30	74.2%
SSeg178-c + grid	**CPU [sec]**	**#Timeouts**	**%Calls saved**
BT	78528.6	14	-
Preproc2+BT	15717.9	1	91.9%
BT+Lookahead	74172.0	14	-
Preproc2+BT+Lookahead	13961.1	1	91.9%
Preproc1&2+BT+Lookahead	13961.6	1	91.9%
SSeg178-i + grid	**CPU [sec]**	**#Timeouts**	**%Calls saved**
BT	138404.2	35	-
Preproc2+BT	103244.7	25	72.7%
BT+Lookahead	121492.4	32	-
Preproc2+BT+Lookahead	85185.9	22	72.7%
Preproc1&2+BT+Lookahead	85185.7	22	71.4%

5.5 Symmetric maximum matchings in the BID problem

We identify two types of symmetries in the BID problem. The first symmetry is present when the problem has no grid constraints. In this case, any solution with one set of assignments to the ordering orientation variables has a symmetric solution with the opposite set of assignments. The second type of symmetry uses the technique discussed in Section 3.4 that identifies all maximum matchings in a bipartite graph as symmetries of a single base matching. Using this technique, we can identify such symmetries in the matching relaxation of the BID problem. A maximum matching in the relaxed BID problem corresponds to a set of assignments of streets to buildings. Thus, a symmetric maximum matching corresponds to a different set of assignments of streets to buildings.

Figure 5.10 shows an example BID instance with four corner buildings B1, B2, B3, and B4. Each corner building is assigned to a street: B1 $\leftarrow$ S1, B2 $\leftarrow$ S2, B3 $\leftarrow$ S2, and B4 $\leftarrow$ S1. Assume that the phone book contains exactly one odd and one even address for each street. After vertex replication, the corresponding bipartite graph, shown in Figure 5.11, has a symmetric maximum matching, shown in Figure 5.13. The second maximum matching corresponds to the set of corner building assignments B1 $\leftarrow$ S2, B2 $\leftarrow$ S1, B3 $\leftarrow$ S1, and B4 $\leftarrow$ S2 shown in Figure 5.12.

While we can easily use the matching construction to identify symmetric sets of assignments to the corner variables, it is not clear how we can take advantage of this symmetry when finding complete solutions to the BID problem where we still need to instantiate the building variables corresponding to the corner buildings. Further, while some of these symmetries, such as the one shown in Figure 5.10 and 5.13, hold when the model includes grid constraints, the grid constraints may break others.

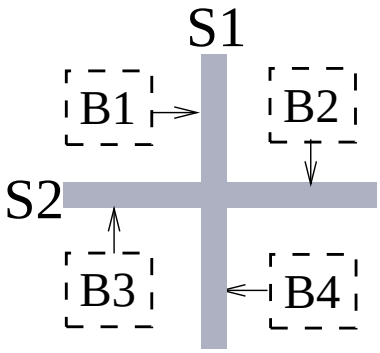

Figure 5.10: BID instance with streets assigned to corner buildings.

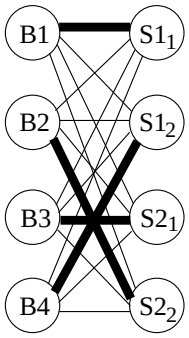

Figure 5.11: Bipartite graph for Figure 5.10.

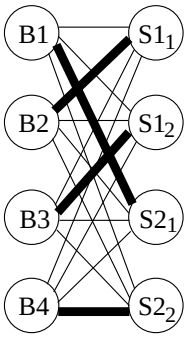

Figure 5.12: Symmetric maximum matching to Figure 5.11.

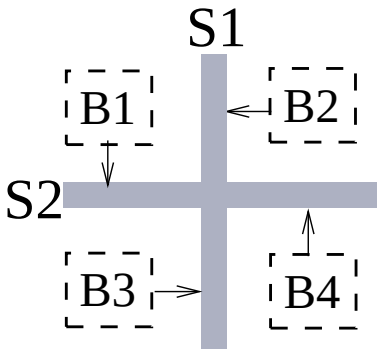

Figure 5.13: Street assignments corresponding to the matching in Figure 5.12.

Identifying symmetries that are not broken by grid constraints and determining how to exploit such symmetries is an interesting area for future investigations.

5.6 Solution quality

While the primary goal of our work is to enhance the scalability of problem solving, it is important to discuss the quality of the solutions we find. Michalowski and Knoblock [2005] proposed a CSP model for the BID problem and tested it on small instances, thus establishing the feasibility of the approach. Further, they measured the quality of solutions using precision and recall with respect to the ground truth. Precision is defined as the percentage of solutions returned by the solver that are correct. Recall

is defined as the percentage of building that were assigned the correct address in at least one solution.

Table 5.6 shows the precision and recall for the case studies used in our experiments[2]. The values for precision would be lower if we did not assume that we knew the correct values for the orientation variables.

Table 5.6: Solution quality.

Case study	**Phone-book completeness**	**Precision**		**Recall**	
		W/o grid	With grid	W/o grid	With grid
Case studies with an complete phone book					
NSeg125-c	100.0%	100.0%	100.0%	100.0%	100.0%
NSeg206-c	100.0%	73.6%	**100.0**%	100.0%	100.0%
SSeg131-c	100.0%	60.8%	78.6%	100.0%	100.0%
SSeg178-c	100.0%	59.8%	81.7%	100.0%	100.0%
Case studies with an incomplete phone book					
NSeg125-i	45.6%	13.9%	14.5%	100.0%	100.0%
NSeg206-i	50.5%	7.0%	11.2%	100.0%	100.0%
SSeg131-i	60.3%	10.3%	11.1%	100.0%	100.0%
SSeg178-i	65.6%	13.3%	15.3%	100.0%	100.0%

Our approach is 'conservative' in that it never rejects an assignment that satisfy the constraints included in the model. Thus, the value of recall is always 100%. As far as precision is concerned, the situation is more complex and deserves some discussion.

Even when the phone book is complete, there may be several solutions that are consistent with the numbering schema such as the symmetric solutions discussed in Section 5.5. The knowledge of known landmarks, such as 10 Downing Street in London or 1600 Pennsylvania Avenue in Washington D.C., allow the solver to eliminate solutions that do not hold in the real world. When the phone book is incomplete or when some numbers are skipped in addresses along a street, the solver will find multiple solutions. Such situations naturally yield a drop of precision values.

[2]All our solvers return exactly the same solutions.

Generally speaking, the way precision is measured in our context causes its value to deteriorate as the range of addresses increases, which, in turn, increases with the size of the problem. The drop of precision in the experiment of Michalowski and Knoblock in [2005] is smaller than the drop in our experiments because our experiments are significantly larger.

The addition of further constraints, such as the grid constraint, improves precision in all cases. Of particular interest is NSeg206-c, shown in bold in Table 5.6, where the addition of grid constraints increased the precision to 100%. Thus, the more information we can infer about a region from various data sources, the more we can improve the precision and quality of the solutions to the BID problem. Michalowski and Knoblock are currently investigating the development of constraint inference techniques from traditional and non-traditional data sources for the BID problem.

Summary

In this chapter, we applied three of the reformulation techniques introduced in Chapter 3, and evaluated their benefit on real-world case studies. We briefly discussed the application of the fourth reformulation to the BID problem, and argued that it deserves deeper investigations. Finally, we discussed the quality of the solutions to the BID problem produced by our approach.

Chapter 6

Concluding Notes

In this thesis, we introduced four reformulation techniques that operate on Constraint Satisfaction Problems (CSPs) and discussed the advantages of these techniques on general CSPs. Our investigations were initially motivated by our study of the Building Identification Problem (BID) introduced by Michalowski and Knoblock in [2005]. We then applied our reformulation techniques to the BID problem and evaluated their benefits on real-world case studies.

This chapter concludes this document by summarizing our contributions and discussing possible directions for future work.

6.1 Summary of contributions

We introduced four reformulations that improve the performance of solving constraint satisfaction problems:

1. We described a query reformulation technique that reduces a counting problem into a polynomial number of satisfiability problems. We also discussed how to exploit the query reformulation to reduce the space complexity of enforcing

higher levels of relational consistency in a constraint network.

2. We introduced the new AllDiff-Atmost global constraint, and illustrated its usefulness in general resource allocation problems. We also described a reformulation technique that reduces the sizes of the domains the variables in the scope this global constraint.

3. We argued that many resource allocation problems can be relaxed into is a matching problem in a bipartite graph. We described multiple opportunities to exploit this relaxation both an exact and a necessary approximation.

4. We described how to characterize all maximum matchings in a bipartite graph as symmetries of a single base matching.

We introduced a constraint for the BID problem that improves on the model given in [Michalowski and Knoblock, 2005] and allows the local addition of new constraints to represent numbering schemas used around the word. We also designed and implemented a custom backtrack-search solver that exploits the structure of the constraint graph to improve the performance of search.

Finally, we applied our first three of our reformulation techniques to the BID problem, and discussed how the fourth reformulation could be used. We evaluated the three reformulation techniques on real-world instances of the BID problem and demonstrated their benefits in terms of scalability.

6.2 Directions for future research

There remain multiple directions for research that builds on our results. We describe a few such directions below:

1. *Relational consistency:* Algorithm 2 for enforcing relational (i, m)-consistency described in Section 3.1.3 improves the space complexity of $\mathrm{RC}_{(i,m)}$, the only known algorithm for enforcing this property. It is to empirically compare the runtime and space performance of $\mathrm{RC}_{(i,m)}$ and our Algorithm 2.

2. *Constraint-model relaxation:* The constraint-model relaxation described in Section 3.3 is powerful when we can relax the problem into a matching problem. We believe that many resource allocation problems may be amenable to such a relaxation. It would be useful to identify such situations to test the value of our technique in other application domains.

3. *Symmetry detection:* The symmetry-detection techniques described in Section 3.4 are particularly promising. Most current work on symmetry in CSPs has focused on breaking symmetries during search and ignored the difficult task of *detecting* them. Exceptions exist and include notably interchangeability detection [Freuder, 1991; Choueiry and Noubir, 1998] and using graph isomorphism [Puget, 2005]. We strongly suspect that our technique for identifying all solutions to a maximum matching problem as symmetric of a single maximum matching can be exploited to improve the performance of problem solving. We believe that this issue deserves more careful and deeper investigations.

4. *Solution probability:* Currently, we return lists of buildings or addresses as solutions to BID problem instances. Thus, for a single building, we return a set of addresses that it might have. However, by counting the number of solutions in which each vvp appears, we could associate with each vvp a probability that it is the correct solution. This approach would allow to better 'qualify' the solutions returned for the BID problem.

5. *Solution precision:* The results on solution quality discussed in Section 5.6 show that it is important to include in the constraint model as much information about the particular numbering schema in an area (e.g., existence of grid constraints) in to order to improve the precision of the results. An important characteristic of our constraint model is that it can incorporate additional constraints (both locally on particular streets or subareas, and globally across the problem) in order to seamlessly accommodate variations of numbering schemas around the world. The work on constraint inference is a parallel branch of research being conducted by Michalowski et al. [2007b] at the Information Sciences Institute of the University of Southern California.

As a final note, our investigations are one more demonstration of the power of abstraction and reformulation as effective tools for breaking down the complexity barrier.

Appendix A

Problem Definition: XML Schema

This appendix describes the XML schema for all XML files that compose a problem instance. We show the XSD schema definition, followed by an example XML file for the problem instance shown in Figure A.1.

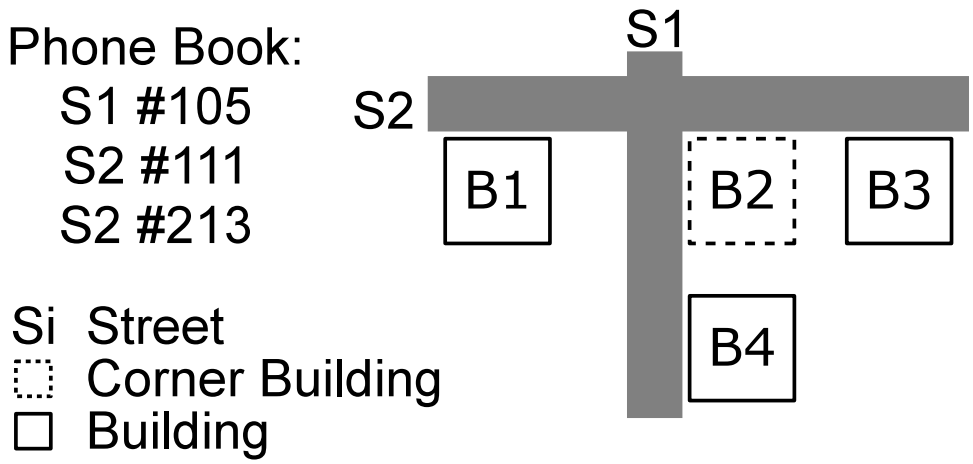

Figure A.1: Two-street example.

Section A.1 describes the layout file. Section A.2 describes the phone-book file. Section A.3 describes the grid file. Section A.4 describes the landmark file.

A.1 Layout XML file

The schema for the layout XML file is the following.

```
<xs:schema xmlns:xs=''http://www.w3.org/2001/XMLSchema''>
  <xs:simpleType name=''commaSeparatedList''>
    <xs:restriction base=''xs:string''>
      <xs:whiteSpace value=''collapse'' />
      <xs:pattern value=''[0-9a-zA-z ]+(,[0-9a-zA-z ]+)*'' />
      <xs:pattern value='''' />
    </xs:restriction>
  </xs:simpleType>
  <xs:element name=''layout''>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=''building'' minOccurs=''0'' maxOccurs=
        ''unbounded''>
          <xs:complexType>
            <xs:sequence>
              <xs:element name=''street'' minOccurs=''0'' maxOccurs=
              ''unbounded''>
                <xs:complexType>
                  <xs:attribute name=''streetname'' type=''xs:string''
                  use=''required'' />
                  <xs:attribute name=''side'' use=''required''>
                    <xs:simpleType>
                      <xs:restriction base=''xs:string''>
                        <xs:enumeration value=''N'' />
                        <xs:enumeration value=''S'' />
                        <xs:enumeration value=''E'' />
                        <xs:enumeration value=''W'' />
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:attribute>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name=''buildingid'' type=''xs:integer''
            use=''required'' />
            <xs:attribute name=''lat'' type=''xs:double'' use=''required'' />
            <xs:attribute name=''lon'' type=''xs:double'' use=''required'' />
          </xs:complexType>
        </xs:element>
```

```
        <xs:element name=''street'' minOccurs=''0'' maxOccurs=''unbounded''>
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base=''commaSeparatedList''>
                <xs:attribute name=''streetname'' type=''xs:string''
                use=''required'' />
                <xs:attribute name=''orientation'' type=''xs:string''
                use=''required'' />
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The layout file has two elements:

1. *building*: The file contains a *building* element for each building on the map. Every building has a *buildingid* attribute that provides a unique identifier for the building. Every building node has one or more children that are *street* elements. Each street element represents a street to which the building is adjacent. Each street element contains a *streetname*, which is the name of the street, and a *side*, which is a single character that indicates on which side of the street the building lies (N,S,E, or W).

2. *street*: The file contains a *street* element for each street on the map. Each street element has a *streetname* attribute, which stores the name of the street, and an *orientation* attribute, which indicates the orientation of the street (NS or EW).

   The content of the street element is a comma separated list of the buildings on the street. The order of the buildings in this list indicate the order in which the buildings lie along the street, starting with the southmost (respectively, westmost) building and ending with the northmost (respectively, eastmost) building.

The sequence contains buildings from both sides of the street, even though the addresses for the opposite sides of the street may not be interdependent.

The layout XML file for Figure A.1 is the following.

```
<boundarylayout districtid=''1''>
   <building buildingid=''B1''>
      <street streetname=''S2'' side=''S'' />
   </building>
   <building buildingid=''B2''>
      <street streetname=''S1'' side=''E'' />
      <street streetname=''S2'' side=''S'' />
   </building>
   <building buildingid=''B3''>
      <street streetname=''S2'' side=''S'' />
   </building>
   <building buildingid=''B4''>
      <street streetname=''S1'' side=''E'' />
   </building>
   <street streetname=''S1'' orientation=''NS''>B4, B2</street>
   <street streetname=''S2'' orientation=''EW''>B1, B2, B3</street>
</boundarylayout>
```

A.2 Phone-book XML file

The schema for the phone-book XML file is the following.

```
<xs:schema xmlns:xs=''http://www.w3.org/2001/XMLSchema''>

  <xs:simpleType name=''commaSeparatedList''>
    <xs:restriction base=''xs:string''>
    <xs:whiteSpace value=''collapse''/>
      <xs:pattern value=''[0-9]+(,[0-9]+)*'' />
      <xs:pattern value='''' />
    </xs:restriction>
  </xs:simpleType>

  <xs:element name=''phonebook''>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=''street'' minOccurs=''0'' maxOccurs=''unbounded''>
```

```
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base=''commaSeparatedList''>
                <xs:attribute name=''streetname'' type=''xs:string'' />
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name=''districtid'' type=''xs:int''
      use=''required'' />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The phone-book xml file contains multiple *street* elements. Each street element has a *streetname* attribute which indicates the name of the street corresponding to the phone-book entries. The content of the street element is a comma separated list of all phone-book addresses for that street.

The phone-book XML file for Figure A.1 is the following.

```
<phonebook districtid=''1''>
   <street streetname=''S1''>105</street>
   <street streetname=''S2''>111, 213</street>
</phonebook>
```

A.3 Grid XML file

The schema for the grid XML file is the following.

```
<xs:schema xmlns:xs=''http://www.w3.org/2001/XMLSchema''>
  <xs:element name=''constraint''>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=''grid''>
          <xs:complexType>
            <xs:sequence>
              <xs:element name=''street'' minOccurs=''0'' maxOccurs=
              ''unbounded''>
```

```
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name=''incrementalpoint'' minOccurs=''0''
                    maxOccurs=''unbounded''>
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name=''buildingPreceding'' type=
                          ''xs:int'' />
                          <xs:element name=''buildingFollowing'' type=
                          ''xs:int'' />
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name=''streetname'' type=''xs:string''
                  use=''required'' />
                  <xs:attribute name=''value'' type=''xs:int''
                  use= ''required'' />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name=''districtid'' type=''xs:int'' use=''required''/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The grid xml file contains multiple *street* elements. Each street element has a *streetname* attribute, which indicates the name of the street along which the gridlines occur, and a *value* attribute, which indicates the increment size for the gridlines. Each street element has multiple *incrementalpoint* children, which each have both a *buildingpreceding* and a *buildingfollowing* child. Each *incrementalpoint* node corresponds to a point where a grid line crosses the street. The *buildingpreceding* and *buildingfollowing* nodes indicate the buildings that lie on either side of the grid line.

A possible grid XML file for Figure A.1 is the following, assuming there are grid lines at each cross street.

```
<constraint districtid=''1''>
  <grid>
    <street streetname=''S2'' value=''100''>
      <incrementalpoint>
        <buildingPreceding>B1</buildingPreceding>
        <buildingFollowing>B2</buildingFollowing>
      </incrementalpoint>
    </street>
  </grid>
</constraint>
```

A.4 Landmark XML file

The schema for the landmark XML file is the following.

```
<xs:schema xmlns:xs=''http://www.w3.org/2001/XMLSchema''>
  <xs:element name=''constraint''>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=''landmarks''>
          <xs:complexType>
            <xs:sequence>
              <xs:element name=''point'' minOccurs=''0'' maxOccurs=
              ''unbounded''>
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name=''address'' type=''xs:int'' />
                    <xs:element name=''street'' type=''xs:string'' />
                  </xs:sequence>
                  <xs:attribute name=''name'' type=''xs:string''
                  use=''required'' />
                  <xs:attribute name=''buildingid'' type=''xs:int''
                  use=''required'' />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
```

```
      </xs:sequence>
      <xs:attribute name=''districtid'' type=''xs:int'' use=''required'' />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

In a landmark XML file, the *landmarks* node contains multiple *point* children. Each *point* has a *name* attribute, which indicates the common name for the building, and a *buildingid* attribute, which indicates the specific identifier from the layout that corresponds to this building. The *point* node has an *address* and *street* child, which indicate the known number and street assignment for the landmark.

A possible landmark XML file for Figure A.1 is the following, assuming we know that the address of building B4 is S1#105.

```
<constraint districtid=''1''>
  <landmarks>
    <point name=''Apartment Building'' buildingid=''B4''>
      <address>105</address>
      <street>S1</street>
    </point>
  </landmarks>
</constraint>
```

Appendix B

Java Documentation

This appendix has the code documenation for the Java implementation of our CSP model and solver for the BID problem.

B.1 Package Matching

Package Contents *Page*

Classes

B.2 Classes

B.2.1 Class **BipartiteGraph**

This is a bipartite graph with 2 partitions - U and V.

B.2.1.1 Declaration

public class BipartiteGraph

extends Object

B.2.1.2 Fields

- public HashMap U
  - The first partition of the graph
- public HashMap V
  - The second partition of the graph

B.2.1.3 Constructors

- `public` **BipartiteGraph**`( )`

B.2.1.4 Methods

- `public void` **addVertexToU**`( Vertex` **v** `)`
  - **Usage**
    - Adds a vertex to partition U
  - **Parameters**
    - `v` - The vertex to add
- `public void` **addVertexToV**`( Vertex` **v** `)`
  - **Usage**
    - Adds a vertex to partition V
  - **Parameters**
    - `v` - The vertex to add
- `public static void` **addVerticesToHashMap**`( HashMap` **vertices**`, LinkedList` **allNewVertices** `)`
  - **Usage**
    - Adds vertices to a hashmap indexed by the vertex label
  - **Parameters**
    - `vertices` - Hashmap to add to
    - `allNewVertices` - Vertices to add to the hashmap

- `public BipartiteGraph` **createDigraphForHopcroftKarp(** `LinkedList` **matching)**
  - **Usage**
    - This clones the graph and directs all of the edges: If e is in the matching, then it goes from v to u. If it is not in the matching, it goes from u to v. From Hopcroft, Karp (1973)
  - **Parameters**
    - `matching` - Matching to direct the graph with
  - **Returns** - The graph, directed as described above
- `public BipartiteGraph` **DeepClone(** **)**
  - **Usage**
    - Performs a deep clone of the graph
  - **Returns** - A deep copy of the graph
- `public void` **duplicateVertices(** **)**
  - **Usage**
    - This function takes the matchcount for each vertex in the graph and removes it if it is 0, and creates multiple copies if it is >1
- `public static void` **duplicateVertices(** `HashMap` **vertices)**
  - **Usage**
    - This function takes the matchcount for each vertex in vertices and removes it if it is 0, and creates multiple copies if it is >1
  - **Parameters**
    - `vertices` - The set of vertices to duplicate
- `public Collection` **GetEdgesInAllMatchings(** **)**

  - **Usage**
    * Returns the set of edges that exist in at least one matching. Uses the algorithm from Regin 94
  - **Returns** - Collection of edges that appear in at least one maximum matching

- `public LinkedList` **getMaximalSetOfDisjointAugmentingPaths(** `LinkedList` **matching)**
  - **Usage**
    * Gets a maximal set of disjoint augmenting paths in G, relative to a given matching. Also from Hopcroft-Karp 1973
  - **Parameters**
    * `matching` - The matching relative to which we find augmenting paths
  - **Returns** - A set of augmenting paths

- `public LinkedList` **GetMaximumMatching(**)
  - **Usage**
    * Finds a maximum matching in the graph This is the algorithm described in Hopcraft, Karp (1973)
  - **Returns** - A maximum matching in the graph

- `public Vertex` **GetVertexInUWithLabel(** `VertexLabel` **label)**
  - **Usage**
    * Gets the vertex in partition U that matches the given label
  - **Parameters**
    * `label` - The label to match
  - **Returns** - The desired vertex

- `public Vertex` **GetVertexInVWithLabel**`( VertexLabel` **label** `)`
  - **Usage**
    - Gets the vertex in partition V that matches the given label
  - **Parameters**
    - `label` - The label to match
  - **Returns** - The desired vertex
- `public Vertex` **GetVertexWithLabel**`( VertexLabel` **label** `)`
  - **Usage**
    - Gets the vertex in either partition that matches the given label
  - **Parameters**
    - `label` - The label to match
  - **Returns** - The desired vertex
- `public boolean` **HasMatchingCoveringV**`( )`
  - **Usage**
    - Determines whether the graph has a matching saturating partition V
  - **Returns** - True, if the graph has a matching that saturates V
- `public boolean` **IsValidMatching**`( LinkedList` **matching** `)`
  - **Usage**
    - Used internally to verify whether a matching is valid; ie, whether it contains the same vertex multiple times
  - **Parameters**
    - `matching` - The matching to check
  - **Returns** - True, if the matching is valid

- `public boolean` **MatchingCoversV**`( LinkedList` **matching** `)`
  - **Usage**
    - Given a matching, determines whether it covers partition V
  - **Parameters**
    - `matching` - The matching to check
  - **Returns** - True, if the matching covers V
- `public String` **toString**`( )`
- `public void` **Transpose**`( )`
  - **Usage**
    - Transposes the graph

B.2.2 Class **Component**

Represents a component (set of vertices) in a graph

B.2.2.1 Declaration

public class Component

extends Object

B.2.2.2 Fields

- public LinkedList vertices
  - The vertices in the component

B.2.2.3 Constructors

- `public` **Component**`( )`

B.2.3 Class **Edge**

An edge in a graph

B.2.3.1 Declaration

public class Edge

extends Object

B.2.3.2 Fields

- public Vertex End1
  - The start point of the edge
- public Vertex End2
  - The end point of the edge

B.2.3.3 Constructors

- `public` **Edge**`( )`
  - **Usage**
    - Create an edge

- `public` **Edge**`( Vertex` **e1**`, Vertex` **e2** `)`
  - **Usage**
    - Create an edge with the specified endpoints
  - **Parameters**
    - `e1` - First endpoint
    - `e2` - Second endpoint

B.2.3.4 Methods

- `public static boolean` **ContainsEdgeMatchingLabels_Directed**`(`
  `Collection` **edges**`, Edge` **targetEdge** `)`
  - **Usage**
    - Returns true if edges contains edge
  - **Parameters**
    - `edges` - The set of edges to search through
    - `targetEdge` - The edge to look for
  - **Returns** - True, if the list of edges contains edge
- `public static boolean` **ContainsEdgeMatchingLabels_Undirected**`(`
  `LinkedList` **edges**`, VertexLabel` **label1**`,`
  `VertexLabel` **label2** `)`
  - **Parameters**
    - `edges` - The set of edges to search through
    - `label1` - Label of one end of the edge
    - `label2` - Label of other end of the edge
  - **Returns** - True, if the list of edges contains a matching edge

- `public static boolean` **ContainsVertex**`( LinkedList` **edges**`, Vertex` **v** `)`
  - **Usage**
    - Determines whether the set of edges contains the given vertex
  - **Parameters**
    - `edges` - The set of edges to search through
    - `v` - The target vertex
  - **Returns** - True, if some edge in edges contains vertex v
- `public static boolean` **ContainsVertexWithLabel**`( LinkedList` **edges**`, VertexLabel` **label** `)`
  - **Usage**
    - Determines whether the set of edges contains a vertex matching the givel label
  - **Parameters**
    - `edges` - The set of edges to search through
    - `label` - The label to search for
  - **Returns** - True, if some edge in edges contains a vertex with label
- `public boolean` **equals**`( Object` **target** `)`
- `public int` **hashCode**`( )`
- `public void` **Reverse**`( )`
  - **Usage**
    - Reverses the direction of an edge
- `public static LinkedList` **SymmetricDifference**`( LinkedList` **matching1**`, LinkedList` **matching2** `)`

  - **Usage**
    * Finds the symmetric difference of two matchings
  - **Parameters**
    * `matching1` - First matching
    * `matching2` - Second matching
  - **Returns** - The symmetric difference of the two matchings

- `public String` **toString**`( )`

B.2.4 Class MatchingSolver

Solve an address assignment problem by reformulating it as a matching

B.2.4.1 Declaration

public class MatchingSolver
extends Object

B.2.4.2 Constructors

- `public` **MatchingSolver**`( )`

B.2.4.3 Methods

- `public static void` **addBuildingVerticesToGraph**`(`

  `BipartiteGraph` **graph**, `Map` **map**, `LinkedList` **buildings** `)`

  - **Usage**
    * Adds vertices for the buildings to the graph

  - **Parameters**
    * `graph` - Target graph
    * `map` - Problem instance
    * `buildings` - Known buildings

- `public static void` **addEdgesToGraph**`( BipartiteGraph` **graph**`, Map` **map**`, Orientation` **orientation**`, LinkedList` **buildings**`, LinkedList` **addresses** `)`
  - **Usage**
    * Adds the necessary edges to the graph
  - **Parameters**
    * `graph` - The graph of the problem
    * `map` - The problem instance
    * `orientation` - The values of the orientation varables
    * `addresses` - The set of known addresses
    * `buildings` - The set of known buildings
- `public static void` **addStreetVerticesToGraph**`(` `BipartiteGraph` **graph**`, Map` **map**`, LinkedList` **addresses** `)`
  - **Usage**
    * Adds vertices for the streets to the graph
  - **Parameters**
    * `graph` - Target graph
    * `map` - Problem instance
    * `addresses` - Known addresses
- `public static BipartiteGraph` **GenerateGraph**`( Map` **map**`, Orientation` **orientation**`, LinkedList` **buildings**`, LinkedList` **addresses** `)`

  - **Usage**
    * Generates a bipartite graph for an address-assignment problem instance
  - **Parameters**
    * `map` - The problem instance
    * `orientation` - The values of the orientation variables
    * `buildings` - The set of known buidlings
    * `addresses` - The set of known addresses
  - **Returns** - A bipartite graph generated using the construction from my thesis

- `public static Parity` **getBuildingParity**`( Building` **building**`, String` **street**`, Orientation` **orientation** `)`
  - **Usage**
    * Gets the parity of a building for a given orientation and street assignment
  - **Parameters**
    * `building` - The target building
    * `street` - The assigned street for the building
    * `orientation` - The current orientation
  - **Returns** - The parity of the building

- `public static void` **setMatchingCountsInGraph**`(`
  `BipartiteGraph` **graph**`, Map` **map**`, Orientation` **orientation**`,`
  `LinkedList` **addresses** `)`
  - **Usage**

    * Sets the matching counts for the street vertices based on the number of phone book addresses for the corresponding street.

  - **Parameters**

    * `graph` - The graph of the problem
    * `map` - The problem instance
    * `orientation` - The values of the orientation varables
    * `addresses` - The set of known addresses

B.2.5 Class **Orientation**

Represents an assignment to the orientation variables

B.2.5.1 Declaration

public class Orientation

extends Object

B.2.5.2 Fields

- public boolean IncreasingEast
  - If true, addresses increase when moving east along a street
- public boolean IncreasingNorth
  - If true, addresses increase when moving north along a street
- public boolean OddOnNorthSide
  - If true, odd numbers appear on the north sides of streets

- public boolean OddOnEastSide
  - If true, odd numbers appear on the east sides of streets

B.2.5.3 Constructors

- `public` **Orientation**`( )`
  - **Usage**
    - Creates an orientation
- `public` **Orientation**`( boolean` **increasingEast**`, boolean` **increasingNorth**`, boolean` **oddOnEastSide**`, boolean` **oddOnNorthSide** `)`
  - **Usage**
    - Creates an orientation
  - **Parameters**
    - `increasingEast` - Addresses increase to the east
    - `increasingNorth` - Addresses increase to the north
    - `oddOnEastSide` - Odd addresses are on the east side
    - `oddOnNorthSide` - Odd addresses are on the north side

B.2.5.4 Methods

- `public boolean` **equals**`( Object` **target** `)`
- `public static LinkedList` **GenerateAllOrientations**`( boolean` **assumeDirections** `)`
  - **Usage**

    * Generates a list of all possible orientations. If assumeDirections is true, only one value is generated for each directional variable
  - **Parameters**
    * `assumeDirections` - If this is true, assume some value for the increasing variables
  - **Returns** - The set of all possible orientations (at most, 16)
- `public static LinkedList` **GenerateSegundoOrientations**`( )`
  - **Usage**
    * Generates the orientation for the city of El Segundo
  - **Returns** - The orientation for the city of El Segundo
- `public String` **toString**`( )`

B.2.6 Class **Path**

A path in a graph

B.2.6.1 Declaration

public class Path

extends Object

B.2.6.2 Fields

- public LinkedList Edges
  - The set of edges in the path

B.2.6.3 Constructors

- `public` **Path( )**

B.2.7 Class **Vertex**

A vertex in a graph

B.2.7.1 Declaration

public class Vertex

extends Object

B.2.7.2 Fields

- public LinkedList Neighbors
  - The set of vertices adjacent to this vertex
- public int MatchCount
  - This is the minimum number of edges that must be adjacent to this vertex in order for a matching to saturate it
- public VertexLabel Label
  - Used to identify problem specific data for a vertex
- public boolean Visited
  - Used by searches to avoid revisiting vertices

- public int FinishTime
  - Used by searches to track the order we finish vertices
- public int Partition
  - Which partitition in a bipartite graph contains this vertex
- public Vertex Parent
  - Used by searches to track the parent of each node
- public LinkedList ChildrenAlreadyVisited
  - Used by searches to track which children have been visited

B.2.7.3 Constructors

- `public` **Vertex**`( )`
  - **Usage**
    - Create a vertex
- `public` **Vertex**`( String` **streetName** `)`
  - **Usage**
    - Create a vertex
  - **Parameters**
    - `streetName` - The streetname for the vertex's label
- `public` **Vertex**`( String` **streetName**`, Map.Parity` **parity** `)`
  - **Usage**
    - Create a vertex
  - **Parameters**

    * `streetName` - The streetname for the vertex's label
    * `parity` - The parity for the vertex's label

- `public` **Vertex**`( String` **streetName**`, Map.Parity` **parity**`, VertexLabel.IncDec` **incOrDec** `)`
  - **Usage**
    * Create a vertex
  - **Parameters**
    * `streetName` - The streetname for the vertex's label
    * `parity` - The parity for the vertex's label
    * `incOrDec` - The incOrDec for the vertex's label
- `public` **Vertex**`( VertexLabel` **label** `)`
  - **Usage**
    * Create a vertex
  - **Parameters**
    * `label` - The label for the vertex

B.2.7.4 Methods

- `public boolean` **ContainsNeighborWithLabel**`( VertexLabel` **label** `)`
  - **Usage**
    * Determines whether this vertex has a neighbor with a given label
  - **Parameters**
    * `label` - Label to search for
  - **Returns** - True, if vertex has a neighbor with the specified label

- `public static boolean` **ContainsVertexWithLabel**`( LinkedList` **list**`, VertexLabel` **label** `)`
  - **Usage**
    - Determines whether any vertex in a list has a given label
  - **Parameters**
    - `list` - List of vertices
    - `label` - Label to search for
  - **Returns** - True, if list contains a vertex with the given label
- `public boolean` **equals**`( Object` **target** `)`
- `public Vertex` **GetNextUnvisitedChild**`( )`
- `public static Vertex` **GetVertexWithHighestFinishTime**`( LinkedList` **vertices** `)`
  - **Usage**
    - Finds the vertex in the list with the highest finish time
  - **Parameters**
    - `vertices` - Vertices to look through
  - **Returns** - Vertex with highest finish time
- `public int` **hashCode**`( )`
- `public boolean` **MostlyMatches**`( Vertex` **vertex** `)`
  - **Usage**
    - Two vertices mostly match if they have the same label, except for the duplicateId
  - **Parameters**
    - `vertex` - Target vertex

  - **Returns** - True, if this vertex label and the target vertex label match on all values other than duplicateId

- `public static void` **RemoveVertexWithLabel**`( LinkedList` **list**`, VertexLabel` **label** `)`
  - **Usage**
    - Removes that has the given label from the given list
  - **Parameters**
    - `list` - List of vertices
    - `label` - Label of vertex to remove
- `public void` **SortNeighborsByFinishTime_Increasing**`( )`
  - **Usage**
    - Sorts this vertex's neighbors in order of increasing finish time
- `public String` **toString**`( )`

B.2.8 Class **VertexLabel**

A label for a vertex; stores problem specific information

B.2.8.1 Declaration

public class VertexLabel
extends Object

B.2.8.2 Fields

- public String Name

  - The name of the problem element that this vertex represents - a building name, or a street name

- public Map.Parity Parity
  - The parity assigned to this problem element
- public VertexLabel.IncDec IncOrDec
  - The numeric range assigned to this vertex
- public int DuplicateId
  - When we create multiple copies of the same vertex, we increment the duplicateId to differentiate between these vertices

B.2.8.3 Constructors

- `public` **VertexLabel**`( String` **street** `)`
  - **Usage**
    - Creates a VertexLabel
  - **Parameters**
    - `street` - Street for the vertex
- `public` **VertexLabel**`( String` **streetName**`, int` **duplicateId** `)`
  - **Usage**
    - Creates a VertexLabel
  - **Parameters**
    - `streetName` - Street for label
    - `duplicateId` - duplicateId for label

- `public` **VertexLabel**`( String` **street**`, Map.Parity` **parity** `)`
  - **Usage**
    - Creates a VertexLabel
  - **Parameters**
    - `street` - Street for the vertex
    - `parity` - Parity of the vertex
- `public` **VertexLabel**`( String` **street**`, M ap.Parity` **parity**`, VertexLabel.IncDec` **incOrDec** `)`
  - **Usage**
    - Creates a VertexLabel
  - **Parameters**
    - `street` - Street for the vertex
    - `parity` - Parity for the vertex
    - `incOrDec` - IncOrDec range for the vertex
- `public` **VertexLabel**`( VertexLabel` **originalLabel**`, int` **duplicateId** `)`
  - **Usage**
    - Creates a duplicated copy of a vertex with the given duplicateId
  - **Parameters**
    - `originalLabel` - Label of the vertex to copy
    - `duplicateId` - DuplicateId of the new vertex

B.2.8.4 Methods

- `public boolean` **equals**`( Object` **target** `)`

- `public int` **hashCode**`( )`
- `public String` **toString**`( )`

B.2.9 Class VertexLabel.IncDec

In the address assignment problem, a vertex can correspond to values on the increasing or decreasing side of some known value. This enumeration indicates which category this vertex belongs to, if any

B.2.9.1 Declaration

public static final class VertexLabel.IncDec
extends Enum

B.2.9.2 Fields

- public static final VertexLabel.IncDec Increasing
  - Vertex corresponds to values greater than some fixed address
- public static final VertexLabel.IncDec Decreasing
  - Vertex corresponds to values less than some fixed address
- public static final VertexLabel.IncDec Neither
  - This vertex does not correspond to a specific numeric range

B.2.9.3 Methods

- `public static VertexLabel.IncDec` **valueOf**`( String` **name** `)`

- `public static final VertexLabel.IncDec` **values**`( )`

B.3 Package AddressCsp

Package Contents *Page*

Classes

B.4 Classes

B.4.1 Class **AddressInterval**

An AddressInterval is a domain for a building variable. It is an ordered set of values, both numeric and symbolic. Thus, certain constraints can operate on the endpoints of the interval. This class manages that dual definition: discrete set and interval.

B.4.1.1 Declaration

public class AddressInterval

extends Object

implements Iterable

B.4.1.2 Fields

- public LinkedList values
  - This is a SORTED (by PositionInOrdering) list of all values allowed in the street
- public String Street

- Street that corresponds to this AddressInterval

B.4.1.3 Constructors

- `public` **AddressInterval**`( )`
  - **Usage**
    - Create an AddressInterval

B.4.1.4 Methods

- `public void` **ClearAllValues**`( )`
  - **Usage**
    - Removes all values from the interval
- `public AddressInterval` **Clone**`( )`
  - **Usage**
    - Clones the interval
  - **Returns** - A clone of the interval
- `public boolean` **equals**`( Object` **target** `)`
- `public Map.Parity` **GetAllowedParity**`( )`
  - **Usage**
    - Gets the parity allowed in this interval
  - **Returns** - This interval's parity
- `public Value` **GetMax**`( )`
  - **Usage**

    * Gets the greatest value in the interval
  - **Returns** - The greatest value in the interval
- `public Value` **GetMin**`( )`
  - **Usage**
    * Gets the least value in the interval
  - **Returns** - The least value in the interval
- `public int` **GetNumValues**`( )`
  - **Usage**
    * Gets the number of values in the interval
  - **Returns** - The number of values in the interval
- `public boolean` **IsEmpty**`( )`
  - **Usage**
    * Determines whether the interval is empty
  - **Returns** - True, if the interval is empty
- `public Iterator` **iterator**`( )`
- `public boolean` **SetAllowedAddresses**`( LinkedList` **unusedAddresses** `)`
  - **Usage**
    * Explicitly sets the list of addresses allowed on this street. Removes all other addresses from the interval
  - **Parameters**
    * `unusedAddresses` - Address to keep
  - **Returns** - True, if any values are removed

- `public boolean` **SetAllowedParity**`( Map.Parity` **parity** `)`
  - **Usage**
    - Sets the parity of this interval and filters the values accordingly
  - **Parameters**
    - `parity` - The parity for the interval
  - **Returns** - True, if any values are filtered
- `public boolean` **SetMaxExclusive**`( Value` **newMax** `)`
  - **Usage**
    - We set the minimum value by removing any value from the end of the list that are greater than newMax's position. Returns true if any values were removed.
  - **Parameters**
    - `newMax` - The new maximum value for the interval
  - **Returns** - True, if any values were removed
- `public boolean` **SetMaxInclusive**`( Value` **newMax** `)`
  - **Usage**
    - We set the minimum value by removing any value from the end of the list that are greater than newMax's position. Returns true if any values were removed.
  - **Parameters**
    - `newMax` - The new maximum value for the interval
  - **Returns** - True, if any values were removed
- `public boolean` **SetMaxIncrementExclusive**`( Value` **newMax** `)`
  - **Usage**

        * Sets the maximumincrement value. This is used to enforce the grid constraints; we remove all values from increments greater than the increment of the given value.

    - **Parameters**

        * `newMax` - The minim increment value

    - **Returns** - True, if any values were removed

- `public boolean` **SetMinExclusive**`( Value` **newMin** `)`

    - **Usage**

        * We set the minimum value by removing any value from the beginning of the list that are less than newMin's position. Returns true if any values were removed.

    - **Parameters**

        * `newMin` - The new minimum value for the interval

    - **Returns** - True, if any values were removed

- `public boolean` **SetMinInclusive**`( Value` **newMin** `)`

    - **Usage**

        * We set the minimum value by removing any value from the beginning of the list that are less than newMin's position. Returns true if any values were removed.

    - **Parameters**

        * `newMin` - The new minimum value for the interval

    - **Returns** - True, if any values were removed

- `public boolean` **SetMinIncrementExclusive**`( Value` **newMin** `)`

    - **Usage**

    * Sets the minimum increment value. This is used to enforce the grid constraints; we remove all values from increments less than the increment of the given value.

  - **Parameters**
    * `newMin` - The minim increment value
  - **Returns** - True, if any values were removed

- `public String` **toString**`( )`

B.4.2 Class CornerConstraint

The corner constraint for our CSP model of the address assignment problem

B.4.2.1 Declaration

public class CornerConstraint
extends Csp.Constraint

B.4.2.2 Fields

- public String BuildingVar
  - The building variable for the constraint
- public String CornerVar
  - The corner variable for the building

B.4.2.3 Constructors

- `public` **CornerConstraint**`( )`

B.4.3 Class **CornerVariable**

A corner variable. Represents the street assignment for a building

B.4.3.1 Declaration

public class CornerVariable

extends DiscreteVariable

B.4.3.2 Constructors

- `public` **CornerVariable**`( )`

B.4.3.3 Methods

- `public Variable` **CloneWithoutConflictSet**`( )`

B.4.4 Class **CSPMapLoader**

Creates a CSP instance from a Map problem instance

B.4.4.1 Declaration

public class CSPMapLoader

extends Object

B.4.4.2 Fields

- public static boolean EnforceCrossStreetOrdering
  - Set this to true if you want to enforce building ordering across the two sides of the street. In reality, ordering is often only enforced along each side of the street independantly

B.4.4.3 Constructors

- `public` **CSPMapLoader**`( )`

B.4.4.4 Methods

- `public static Csp` **CreateIntervalCspFromFile**`( String` **folder**`, String` **mapFile**`, String` **phoneBookFile**`,`
  `String` **constraintFile** `)`
  - **Usage**
    - Creates an interval CSP for an address-assignment problem defined by a set of XML files
  - **Parameters**
    - `folder` - Folder containing problem files
    - `mapFile` - Layout XML file
    - `phoneBookFile` - Phone-book XML file
    - `constraintFile` - Additional constraint XML file
  - **Returns** - A CSP instance for the problem specified in the files

- `public static Csp` **CreateIntervalCspFromMap**`( Map` **map** `)`
  - **Parameters**
    - `map` -
  - **Returns** -

B.4.5 Class DiscreteVariable

A DiscreteVariable is a variable whose domain is a set of discrete values, rather than an interval

B.4.5.1 Declaration

public class DiscreteVariable
extends Csp.Variable

B.4.5.2 Constructors

- `public` **DiscreteVariable**`( )`
  - **Usage**
    - Creates a variable
- `public` **DiscreteVariable**`( String` **name**`, Object []` **domain** `)`
  - **Usage**
    - Creates a variable
  - **Parameters**
    - `name` - Variable name
    - `domain` - Variable domain

B.4.5.3 Methods

- `public void` **AddFilteredValue**`( Csp.Constraint` **filteringConstraint**`,`
  `String` **filterVar**`, Object` **value**`,`
  `HashMap` **vars**`, HashMap` **solution** `)`
  - **Usage**
    - Filters a value from the domain of the variable
  - **Parameters**
    - `filteringConstraint` - Constraint that caused the filtering
    - `filterVar` - Variable that caused the filtering
    - `value` - The value to be filtered
    - `vars` - The variables in the problem
    - `solution` - The current partial solution
- `public void` **AddFilteredValues**`( Csp.Constraint` **filteringConstraint**`,`
  `String` **filterVar**`, LinkedList` **values**`,`
  `HashMap` **vars**`, HashMap` **solution** `)`
  - **Usage**
    - Filters values from the domain of the variable
  - **Parameters**
    - `filteringConstraint` - Constraint that caused the filtering
    - `filterVar` - Variable that caused the filtering
    - `values` - The values to be filtered
    - `vars` - The variables in the problem
    - `solution` - The current partial solution
- `public Variable` **CloneWithoutConflictSet**`( )`

- `public Iterator` **GetDomainIterator**`( )`
- `public void` **InitializeFilteredValues**`( HashMap` **vars** `)`
- `public void` **RestoreFilteredValues**`( String` **varInstantiated** `)`

B.4.6 Class **GridConstraint**

Represents a grid constraint in the address-assignment problem

B.4.6.1 Declaration

public class GridConstraint

extends Csp.Constraint

B.4.6.2 Fields

- public String booleanOrderingVariable
  - The orientation variable that determines the increasing direction
- public String Buildings
  - The buildings on either side of the grid line
- public String Street
  - The street on which the buildings lie

B.4.7 Class **IntervalIterator**

An iterator over an AddressInterval

B.4.7.1 Declaration

public class IntervalIterator

extends Object

implements Iterator

B.4.7.2 Constructors

- `public` **IntervalIterator**`( LinkedList` **domain** `)`
  - **Usage**
    - Initialize an IntervalIterator
  - **Parameters**
    - `domain` - The AddressIntervals to iterate over

B.4.7.3 Methods

- `public boolean` **hasNext**`( )`
- `public Object` **next**`( )`
- `public void` **remove**`( )`

B.4.8 Class IntervalVariable

An IntervalVariable is a variable whose domain is a set of AddressIntervals, such as a building variable

B.4.8.1 Declaration

public class IntervalVariable

extends Csp.Variable

B.4.8.2 Constructors

- `public` **IntervalVariable**`( )`
  - **Usage**
    - Creates an interval variable
- `public` **IntervalVariable**`( String` **name**`, Object [] ` **domain** `)`
  - **Usage**
    - Creates an interval variable
  - **Parameters**
    - `name` - Variable Name
    - `domain` - Set of AddressIntervals in the domain

B.4.8.3 Methods

- `public boolean` **AllDomainsEmpty**`( )`
  - **Usage**
    - Determines whether the domain is empty
  - **Returns** - True, if all intervals in the domain are empty
- `public Variable` **CloneWithoutConflictSet**`( )`

- `public boolean` **DomainContainsAddress**`( Address` **address** `)`
  - **Usage**
    - This checks to see if a domain contains a given address (the address does not contain position information, so we have this function that ignores the position.)
  - **Parameters**
    - `address` - The address to search for
  - **Returns** - True, if some value in the domain matches the address
- `public boolean` **DomainContainsValue**`( Value` **value** `)`
  - **Usage**
    - Determines whether the domain of this variable contains a given value
  - **Parameters**
    - `value` - Value to search for
  - **Returns** - True, if the domain contains the value
- `public Iterator` **GetDomainIterator**`( )`
- `public AddressInterval` **GetIntervalForStreet**`( String` **street** `)`
  - **Usage**
    - Gets the AddressInterval corresponding to the given street
  - **Parameters**
    - `street` - Target street
  - **Returns** - The AddressInterval for a given street.
- `public void` **InitializeFilteredValues**`( HashMap` **vars** `)`
- `public boolean` **OnlyContainsAddressesOnStreet**`( String` **street** `)`
  - **Usage**

    * Determines whether this variable's domain only contains addresses on a single given street
  - **Parameters**
    * `street` - Target street
  - **Returns** - True, if all values in the domain are on the given street

- `public void` **RestoreFilteredValues**`( String` **varInstantiated** `)`
- `public void` **SetFilteredInterval**`( Csp.Constraint` **filteringConstraint**`, String` **filterVar**`, AddressInterval` **interval**`, HashMap` **vars**`, HashMap` **solution** `)`
  - **Usage**
    * Replaces an AddressInterval in the domain with a filtered version.
  - **Parameters**
    * `filteringConstraint` - Constraint causing the filtering
    * `filterVar` - Variable causing the filtering
    * `interval` - Filtered interval
    * `vars` - All variables in the CSP
    * `solution` - Current partial solution
- `public HashMap` **testGetfilteredValues**`( )`
  - **Returns** -

B.4.9 Class MatchingConstraint

This constraint implements lookahead using the matching relaxation

B.4.9.1 Declaration

public class MatchingConstraint

extends Csp.Constraint

B.4.9.2 Constructors

- `public` **MatchingConstraint**`(` `Map` **map**`,` `LinkedList` **cornerBuildings** `)`
  - **Usage**
    - Initializes that matching lookahead mechanism
  - **Parameters**
    - `map` - The problem instance
    - `cornerBuildings` - The set of corner buildings in the problem instance

B.4.10 Class **OrderingConstraint**

The ordering constraint for the address assignment problem

B.4.10.1 Declaration

public class OrderingConstraint

extends Csp.Constraint

B.4.10.2 Fields

- public String booleanOrderingVariable
  - The orientation variable that determines the direction addresses increase in
- public String Buildings
  - The building ordered by this constraint
- public String Street
  - The street along which this buildings lie

B.4.10.3 Constructors

- `public` **OrderingConstraint**`( )`

B.4.11 Class ParityConstraint

The parity constraint for the address assignment problem

B.4.11.1 Declaration

public class ParityConstraint
extends Csp.Constraint

B.4.11.2 Fields

- public String booleanVariable
  - The orientation variable that determines parity for this building

- public String BuildingVariable
  - The building whose parity is determined by this constraint
- public boolean MustBeOddIfTrue
  - If this value is true, then the building variable must be odd
- public String ExpectedStreet
  - The building's street on which this constraint applies

B.4.12 Class **StreetConstraint**

The phone-book constraint for the address assignment problem

B.4.12.1 Declaration

public class StreetConstraint
extends Csp.Constraint

B.4.12.2 Fields

- public Street TargetStreet
  - The street for which this constraint applies
- public LinkedList RequiredAddresses
  - The phone-book addresses for this street
- public static Map map
  - The problem instance

B.4.13 Class **Value**

A value for a building variable. This may be either a numeric or symbolic value

B.4.13.1 Declaration

public class Value

extends Object

B.4.13.2 Fields

- public int PositionInOrdering
  - The position of this value in the total ordering of all values
- public String Street
  - The street to which this value corresponds
- public int Increment
  - The increment corresponding to this value. If a street has incremental points, we create a new increment for each such point. this stores which increment this value belongs to, so we can make constraints that jump to the next increment.

B.4.13.3 Constructors

- `public` **Value**`( int` **position** `)`
  - **Usage**

        * Creates a value
    - **Parameters**
        * `position` - The position in the relative ordering for this value
- `public` **Value**`( int` **position**`, int` **number**`, String` **street** `)`
    - **Usage**
        * Creates a value
    - **Parameters**
        * `position` - The position in the relative ordering for this value
        * `number` - The numeric value
        * `street` - The street on which this value lies
- `public` **Value**`( int` **position**`, int` **number**`, String` **street**`, Map.Parity` **parity** `)`
    - **Usage**
        * Creates a value
    - **Parameters**
        * `position` - The position in the relative ordering for this value
        * `number` - The numeric value
        * `street` - The street on which this value lies
        * `parity` - The parity of the value
- `public` **Value**`( int` **position**`, int` **number**`, String` **street**`, Map.Parity` **parity**`, int` **increment** `)`
    - **Usage**
        * Creates a value
    - **Parameters**

    * `position` - The position in the relative ordering for this value
    * `number` - The numeric value
    * `street` - The street on which this value lies
    * `parity` - The parity of the value
    * `increment` - The increment of the value

B.4.13.4 Methods

- `public boolean` **equals**`( Object` **target** `)`
- `public int` **getNumber**`( )`
  - **Usage**
    * The numeric value of this value; -1 means symbolic value
  - **Returns** - The numeric value of this Value
- `public Parity` **GetParity**`( )`
  - **Usage**
    * Gets the parity of this value
  - **Returns** - The parity of the value
- `public static boolean` **GreaterThan**`( Value` **v1**`, Value` **v2** `)`
  - **Usage**
    * Compares two values
  - **Parameters**
    * `v1` - First value
    * `v2` - Second value
  - **Returns** - True, if v1 >v2
- `public static boolean` **GTE**`( Value` **v1**`, Value` **v2** `)`

  - **Usage**
    - Compares two values
  - **Parameters**
    - `v1` - First value
    - `v2` - Second value
  - **Returns** - True, if v1 >= v2

- `public int` **hashCode**`( )`
- `public static boolean` **LessThan**`( Value` **v1**`, Value` **v2** `)`
  - **Usage**
    - Compares two values
  - **Parameters**
    - `v1` - First value
    - `v2` - Second value
  - **Returns** - True, if v1 <v2
- `public static boolean` **LTE**`( Value` **v1**`, Value` **v2** `)`
  - **Usage**
    - Compares two values
  - **Parameters**
    - `v1` - First value
    - `v2` - Second value
  - **Returns** - True, if v1 <= v2
- `public boolean` **MatchesAddress**`( Address` **address** `)`
  - **Usage**
    - Determines whether an address matches this value

  - **Parameters**
    - `address` - The address to check
  - **Returns** - True, if the address matches the value's street and number

- `public void` **setNumber**`( int  value )`
  - **Usage**
    - Sets the numeric value
  - **Parameters**
    - `value` - Numeric value

- `public Address` **ToAddress**`( )`
  - **Usage**
    - Converts a value to an address
  - **Returns** - The equivalent address

- `public String` **toString**`( )`

B.5 Package Csp

Package Contents *Page*

Classes

A variable in the CSP.

B.6 Classes

B.6.1 Class **Assignment**

An assignment of a value to a variable

B.6.1.1 Declaration

public class Assignment

extends Object

B.6.1.2 Fields

- public String Name
  - Variable name
- public Object Value
  - Value to assign to the variable

B.6.1.3 Constructors

- `public` **Assignment**`( )`
  - **Usage**
    - Create an assignment

- `public` **Assignment**`( String` **name**`, Object` **value** `)`
  - **Usage**
    - Create an assignment
  - **Parameters**
    - `name` - Variable name
    - `value` - Value to assign to the variable

B.6.1.4 Methods

- `public boolean` **equals**`( Object` **target** `)`
- `public int` **hashCode**`( )`
- `public String` **toString**`( )`

B.6.2 Class **Constraint**

All constraints subclass this class. It provides some common constraint functionality, as well as acting as an interface of required methods for all constraints

B.6.2.1 Declaration

---

public abstract class Constraint

extends Object

B.6.2.2 Fields

- public static int ConsistencyChecks

  - The number of consistency checks performed by search

- public static boolean ChangeWasMadeByLastFiltering
  - Used by the BT solver. True, if the last filtering removed some value from some variable's domain

- public LinkedList Scope
  - The scope of this constraint

B.6.2.3 Constructors

- `public` **Constraint**`( )`

B.6.2.4 Methods

- `public abstract boolean` **Check**`( HashMap` **vars** `)`
  - **Usage**
    - Checks a constraint against a partial solution
  - **Parameters**
    - `vars` - A partial solution
  - **Returns** - True, if the partial solution does not violate the constraint
- `public abstract boolean` **FilterDomains**`( HashMap` **solution**`, HashMap` **vars**`, String` **filterVar**`, Set` **outVarsWhoseDomainsChanged** `)`
  - **Usage**
    - Filters domains of variables in this constraint's scope

  - **Parameters**
    - `solution` - Current partial solution
    - `vars` - All variables in the CSP
    - `filterVar` - Variable whose instantiation triggered this filtering
    - `outVarsWhoseDomainsChanged` - The set of variables whose domains were changed by this filtering
  - **Returns** - True, if no domain wipeout occrs

- `public Set` **GetInstantiatedVarsInScope**`( HashMap` **solution** `)`
  - **Usage**
    - Gets the set of variables in this constraint's scope that are already instantiated.
  - **Parameters**
    - `solution` - Current partial solution
  - **Returns** - The set of variables in this constraint's scope that are instantiated in the given partial solution

- `public boolean` **IntersectsWith**`( Set` **vars** `)`
  - **Usage**
    - Determines whether this constraint's scope intersects a set of variables
  - **Parameters**
    - `vars` - The variables to check against
  - **Returns** - True, if this constraint's scope intersects with the given set of variables

B.6.3 Class Csp

A CSP problem instance

B.6.3.1 Declaration

public class Csp

extends Object

B.6.3.2 Fields

- public HashMap Variables
  - The variables in the CSP
- public LinkedList Constraints
  - The constraints in the CSP
- public LinkedList CornerBuildings
  - The corner buildings in the underlying address assignment problem
- public LinkedList PhoneBook
  - The phone-book for the underlying address assignment problem
- public LinkedList KnownBuildings
  - The known buildings in the problem
- public Map SourceMap
  - The base problem instance
- public SymbolicIntervalGenerator IntervalGenerator
  - The interval generator that creates the domains for this problem

B.6.3.3 Constructors

- `public` **Csp**`( )`

B.6.3.4 Methods

- `public float` **GetAverageDomainSizeForBuildings**`( )`
  - **Usage**
    - Used to measure the statistics of the problem. Returns the average domain sizes of the building variables
  - **Returns** - The average domain sizes of the building variables
- `public void` **PrintConstraintCounts**`( )`
  - **Usage**
    - For debug purposes, it prints the number of constraints generated, by type.
- `public void` **SetAddressForBuilding**`( String` **buildingName**`, Address` **address** `)`
  - **Usage**
    - Sets the domain of a known building to a given address
  - **Parameters**
    - `buildingName` - Building with the known address
    - `address` - The address
- `public void` **SetOrientationValues**`( Orientation` **orientation** `)`
  - **Usage**

        * Sets the orientation variables in the CSP to given values
    - **Parameters**
        * `orientation` - The orientation to use
- `public void` **SetStreetForCornerBuilding**`( String` **buildingName**`, String` **streetName** `)`
    - **Usage**
        * Sets the domain of a known corner building to a given street
    - **Parameters**
        * `buildingName` - Building with the known street
        * `streetName` - Name of street
- `public String` **toString**`( )`

B.6.4 Class **Variable**

A variable in the CSP. All variables must subclass this variable

B.6.4.1 Declaration

public abstract class Variable

extends Object

B.6.4.2 Fields

- public String Name
    - The name of the variable.

- public LinkedList Domain
  - The domain of the variable
- public Set ConflictSet
  - Used by CBJ; stores the conflict set of the variable
- public Set PastFC
  - Used by CBJ; stores the past variables that filtered this variable

B.6.4.3 Constructors

- `public` **Variable**`( )`
  - **Usage**
    - Creates a variable
- `public` **Variable**`( String` **name**`, Object []` **domain** `)`
  - **Usage**
    - Creates a variable
  - **Parameters**
    - `name` - Name of the variable
    - `domain` - Domain of the variable

B.6.4.4 Methods

- `public abstract Variable` **CloneWithoutConflictSet**`( )`
  - **Usage**

    * Clones the variable, but resets the conflict set. Used when creating a subproblem for a cSP
  - **Returns** - A clone of the variable, without the conflict sets.
- `public abstract Iterator` **GetDomainIterator**`( )`
  - **Usage**
    * Gets an iterator over this variable's domain
  - **Returns** - An iterator over this variable's domain
- `public abstract void` **InitializeFilteredValues**`( HashMap` **vars** `)`
  - **Usage**
    * Initializes the sets filtered values to nothing
  - **Parameters**
    * `vars` - The set of all variables in the csp
- `public abstract void` **RestoreFilteredValues**`( String` **varInstantiated** `)`
  - **Usage**
    * Restores values filtered by the instantiation of the given variable
  - **Parameters**
    * `varInstantiated` - Variable whose filtering should be undone
- `public String` **toString**`( )`

B.7 Package Map

Package Contents *Page*

Classes

B.8 Classes

B.8.1 Class **Address**

Stores an instance of an address

B.8.1.1 Declaration

public class Address

extends Object

implements Comparable

B.8.1.2 Fields

- public String Street
  - The street portion of the address
- public int Number
  - The numeric portion of the address

B.8.1.3 Constructors

- `public` **Address**`( )`
  - **Usage**
    - Creates an address
- `public` **Address**`( String` **street**`, int` **number** `)`
  - **Usage**
    - Creates an address from a given street and number
  - **Parameters**
    - `street` - Street Name
    - `number` - Number

B.8.1.4 Methods

- `public Object` **clone**`( )`
- `public int` **compareTo**`( Address` **target** `)`
- `public boolean` **equals**`( Object` **target** `)`
- `public int` **hashCode**`( )`
- `public boolean` **MatchesParity**`( Map.Parity` **parity** `)`
  - **Usage**
    - Returns true if the address matches the parity passed in (Odd or Even)
  - **Parameters**
    - `parity` - The parity to check against
  - **Returns** - True, if the address matches the given parity. False otherwise
- `public String` **toString**`( )`

B.8.2 Class **Building**

Stores an instance of a building

B.8.2.1 Declaration

public class Building

extends Object

B.8.2.2 Fields

- public String Name

  - The identifying name of the building

- public LinkedList Street
  - The list of streets to which the building is adjacent
- public LinkedList Side
  - The list of which side of the street the building is on for each street to which it is adjacent. Possible values are N,S,E, or W

B.8.2.3 Constructors

- `public` **Building**`( )`
  - **Usage**
    - Create a building
- `public` **Building**`( String` **name** `)`
  - **Usage**
    - Create a building
  - **Parameters**
    - `name` - Building's name

B.8.2.4 Methods

- `public Object` **clone**`( )`
- `public boolean` **equals**`( Object` **target** `)`
- `public Character` **GetSideOnStreet**`( String` **streetName** `)`
  - **Usage**

    * Returns which side of a given street this building is on
  - **Parameters**
    * `streetName` - Target street
  - **Returns** - The side of the target street on which this building lies
- `public int` **hashCode**`( )`
- `public boolean` **IsAdjacentToStreet**`( String` **street** `)`
  - **Usage**
    * Determines whether a building is adjacent to a street
  - **Parameters**
    * `street` - Target street name
  - **Returns** - True, if the building is adjacent to street
- `public boolean` **IsCorner**`( )`
  - **Returns** - True, if the building is a corner building
- `public String` **toString**`( )`

B.8.3 Class **GridPoint**

Stores information necessary to identify a grid point, which is simply the buildings that precede and follow the point

B.8.3.1 Declaration

public class GridPoint
extends Object

B.8.3.2 Fields

- public Building BuildingPreceding
  - Building preceding this grid point
- public Building BuildingFollowing
  - Building following this grid point

B.8.3.3 Constructors

- `public` **GridPoint**`( )`
  - **Usage**
    - Create a new grid point
- `public` **GridPoint**`( Building` **buildingPreceding**`, Building` **buildingFollowing** `)`
  - **Usage**
    - Create a new grid point
  - **Parameters**
    - `buildingPreceding` - Building preceding grid point
    - `buildingFollowing` - Building following grid point

B.8.3.4 Methods

- `public boolean` **equals**`( Object` **target** `)`
- `public int` **hashCode**`( )`

B.8.4 Class Map

Contains an instance of the map and phone book for the address assignment problem

B.8.4.1 Declaration

public class Map

extends Object

B.8.4.2 Fields

- public LinkedList Buildings
  - All buildings on the map
- public LinkedList PhoneBook
  - All known addresses in the phone-book
- public LinkedList Streets
  - All streets on the map
- public LinkedList StreetNames
  - The names of all streets on the map
- public HashMap KnownBuildings
  - All known buildings (landmarks)
- public boolean ParityConstraintsIndependentForEachStreet
  - If true, then parities are different for each street, rather than being global to the entire problem

B.8.4.3 Constructors

- `public` **Map**`( )`

B.8.4.4 Methods

- `public void` **AddGridPoint**`( String` **street**`, String` **buildingPreceding**`, String` **buildingFollowing** `)`
  - **Usage**
    - Adds a grid point for the given street, between the two given buildings
  - **Parameters**
    - `street` - The street on which to add the grid point
    - `buildingPreceding` - The building preceding the grid point
    - `buildingFollowing` - The building following the grid point
- `public static boolean` **BuildingsAreOnSameSideOfStreet**`(`
  `String` **streetName**`, Building` **b1**`, Building` **b2** `)`
  - **Usage**
    - Determines whether two buildings are on the same side of the street
  - **Parameters**
    - `streetName` - The street to check
    - `b1` - First building to check
    - `b2` - Second building to check
  - **Returns** - True if b1 and b2 are on the same side of street streetName
- `public Map` **Clone**`( )`
  - **Returns** - A copy of the problem instance

- `public int` **GetAddressCount_Greater(** `String` **streetName,** `int` **value,** `Map.Parity` **parity)**
  - **Usage**
    - Gets the number of addresses on a given street and parity greater than a given value
  - **Parameters**
    - `streetName` - Target street
    - `parity` - Target parity
    - `value` - Minimum address to count
  - **Returns** - Number of phone-book addresses matching street and parity
- `public int` **GetAddressCount_Less(** `String` **streetName,** `int` **value,** `Map.Parity` **parity)**
  - **Usage**
    - Gets the number of addresses on a given street and parity less than a given value
  - **Parameters**
    - `streetName` - Target street
    - `parity` - Target parity
    - `value` - Maximum address to count
  - **Returns** - Number of phone-book addresses matching street and parity
- `public int` **GetAddressCount(** `String` **streetName,** `Map.Parity` **parity)**
  - **Usage**
    - Gets the number of addresses on a given street and parity

  - **Parameters**
    - `streetName` - Target street
    - `parity` - Target parity
  - **Returns** - Number of phone-book addresses matching street and parity
- `public Building` **GetBuildingByName**`( String` **buildingName** `)`
  - **Usage**
    - Given a name, finds the corresponding building
  - **Parameters**
    - `buildingName` - Name of building to find
  - **Returns** - The desired building
- `public int` **GetHighestAddress**`( )`
  - **Usage**
    - Finds the highest address appearing in the phone-book
  - **Returns** - The highest address in the phone-book
- `public int` **GetHighestAddress**`( String` **streetName** `)`
  - **Usage**
    - Searches all known addresses on a street to find the lowest one
  - **Parameters**
    - `streetName` - Target street
  - **Returns** - The highest number of any address on that street
- `public int` **GetLowestAddress**`( String` **streetName** `)`
  - **Usage**
    - Searches all known addresses on a street to find the lowest one

  - **Parameters**
    - `streetName` - Target street
  - **Returns** - The lowest number of any address on that street
- `public static Map.Parity` **GetParity**`( int` **num** `)`
  - **Usage**
    - Gets the parity of a given number
  - **Parameters**
    - `num` - Number to get the parity of
  - **Returns** - The parity of num
- `public LinkedList` **GetPhoneBookEntriesForStreet**`( String` **street** `)`
  - **Usage**
    - Gets the phone book entries for a given street
  - **Parameters**
    - `street` - Street to get phone-book entries for
  - **Returns** - The phone book entries for street
- `public Street` **GetStreetByName**`( String` **streetName** `)`
  - **Usage**
    - Given a name, finds the corresponding street
  - **Parameters**
    - `streetName` - Name of the street to find
  - **Returns** - The desired street
- `public static Map.Parity` **InvertParity**`( Map.Parity` **original** `)`

  - **Usage**
    * Inverts the parity given; Odd becomes Even, Even becomes Odd, and Unknown remains unknown
  - **Parameters**
    * `original` - The original parity
  - **Returns** - The opposite parity
- `public String` **toString**`( )`

B.8.5 Class Map.Parity

Possible number parities

B.8.5.1 Declaration

public static final class Map.Parity
extends Enum

B.8.5.2 Fields

- public static final Map.Parity Odd
  - Odd
- public static final Map.Parity Even
  - Even
- public static final Map.Parity Unknown
  - Unknown Parity

B.8.5.3 Methods

- `public static Map.Parity` **valueOf**`( String` **name** `)`
- `public static final Map.Parity` **values**`( )`

B.8.6 Class MapReader

Reads a problem instance from XML files into a Map object

B.8.6.1 Declaration

public class MapReader

extends Object

B.8.6.2 Methods

- `public static Map` **ReadMap**`( String` **folder**`, String` **mapFile**`, String` **phoneBookFile**`, String` **constraintFile** `)`
  - **Usage**
    - Reads a problem instance from xml files into a Map object
  - **Parameters**
    - `folder` - The folder containing the problem files
    - `mapFile` - The name of the map file
    - `phoneBookFile` - The name of the phone-book file
    - `constraintFile` - The name of the additional constraints file
  - **Returns** - The map object for the problem instance

B.8.7 Class **Street**

A street, defined by the set of buildings and gridpoints along it

B.8.7.1 Declaration

public class Street

extends Object

B.8.7.2 Fields

- public String Name
  - The identifying name of the street
- public Building Buildings
  - The buildings along the street
- public Street.StreetDirection Orientation
  - The orientation of the street
- public LinkedList GridPoints
  - The set of grid points along the street

B.8.7.3 Constructors

- `public` **Street**`( )`
  - **Usage**

    * Create a street

- `public` **Street**`( String` **name**`, Map.Building []` **buildings** `)`
  - **Usage**
    * Create a street
  - **Parameters**
    * `name` - Street name
    * `buildings` - Buildings along the street

B.8.7.4 Methods

- `public void` **AddGridPoint**`( Map.Building` **buildingPreceding**`, Building` **buildingFollowing** `)`
  - **Usage**
    * Adds a grid point between the specified buildings
  - **Parameters**
    * `buildingPreceding` - Building preceding the grid point
    * `buildingFollowing` - Building following the grid point
- `public boolean` **Building1IsBeforeBuilding2**`( Building` **building1**`, Building` **building2** `)`
  - **Usage**
    * Used internally; returns true if building1 is north or east of building 2
  - **Parameters**
    * `building1` - First building to consider
    * `building2` - Second building to consider
  - **Returns** - True, if building1 is north or east of building2

- `public Object` **clone**`( )`
- `public boolean` **ContainMatchingIncrementalPoint**`( Building` **firstBuilding**`, Building` **secondBuilding** `)`
  - **Usage**
    - Determines whether the street contains a grid point between two given buildings
  - **Parameters**
    - `firstBuilding` - Building preceding the grid point
    - `secondBuilding` - Building following the grid point
  - **Returns** - True, if the street contains a matching grid point
- `public boolean` **ContainsIncrementalPoint**`( )`
  - **Usage**
    - Determines whether the street contains any grid points
  - **Returns** - True, if the street contains any grid points
- `public String` **toString**`( )`

B.8.8 Class Street.StreetDirection

Possible directions which a street may run

B.8.8.1 Declaration

public static final class Street.StreetDirection

extends Enum

B.8.8.2 Fields

- public static final Street.StreetDirection NorthSouth
  - Indicates a street that runs from north to south
- public static final Street.StreetDirection EastWest
  - Indicates a street that runs from east to west

B.8.8.3 Methods

- `public static Street.StreetDirection` **valueOf**`( String` **name** `)`
- `public static final Street.StreetDirection` **values**`( )`

B.8.9 Class SymbolicIntervalGenerator

Generates the intervals used as domains for the CSP variables. These intervals may contain symbolic values, if we are using the AllDiff-Atmost reformulation

B.8.9.1 Declaration

public class SymbolicIntervalGenerator
extends Object

B.8.9.2 Fields

- public static boolean EnumerateAllValues

- If this flag is true, we always enumerate all values in the domain. If it is false, we use the AllDiff-Atmost reformulation and generate symbolic values where necessary

B.8.9.3 Constructors

- `public` **SymbolicIntervalGenerator**`( Map` **map** `)`
  - **Usage**
    - Initializes the generator
  - **Parameters**
    - `map` - The map that the generator creates intervals for

B.8.9.4 Methods

- `public AddressInterval` **GetIntervalContainingSingleAddress**`( int` **number**`, String` **street** `)`
  - **Usage**
    - Gets an interval containing just a single address. Useful when we want to assign a specific value to a building, but still want to be able to order it relative to other symbolic values
  - **Parameters**
    - `number` - Number portion of the address
    - `street` - Street portion of the address
  - **Returns** - An addressinterval just containing the value corresponding to street and number

- `public AddressInterval` **GetIntervalForStreet**`( String` **street** `)`
  - **Usage**
    - Gets the interval (domain) for a given street
  - **Parameters**
    - `street` - Target street
  - **Returns** - The interval for the target street
- `public Value` **GetValueFromAddress**`( Address` **address** `)`
  - **Usage**
    - Gets the Value for a given Address. Useful, because the value contains information that allows us to order an address relative to symbolic values
  - **Parameters**
    - `address` - Target address
  - **Returns** - The value corresponding to address

B.9 Package Solvers

Package Contents *Page*

Classes

B.10 Classes

B.10.1 Class **AggregateSolver**

CSP solver that enumerates all solutions and stores the per-variable solution

B.10.1.1 Declaration

public class AggregateSolver

extends Solvers.BaseSolver

B.10.1.2 Constructors

- `public` **AggregateSolver**`( Csp.Csp` **csp** `)`
  - **Usage**
    - Create the solver
  - **Parameters**
    - `csp` - Csp to solve

B.10.1.3 Methods

- `public HashMap` **SolveAggregate**`( )`
  - **Usage**

    * Solve the problem by enumerating all solutions

  - **Returns** - The per-variable solution

B.10.2 Class **BaseSolver**

This base BT solver provides functionality used by all BT solvers

B.10.2.1 Declaration

public class BaseSolver

extends Solvers.Solver

B.10.2.2 Fields

- public int TotalSolutions
  - The total number of solutions found

B.10.2.3 Constructors

- `public` **BaseSolver**`( Csp.Csp` **csp** `)`
  - **Usage**
    * Initialize the solver
  - **Parameters**
    * `csp` - Csp instance to solve

B.10.2.4 Methods

- `public void` **findSolutions**`( )`
  - **Usage**
    - Performs the actual BT search to find solutions
- `public static Orientation` **GetCurrentOrientation**`( HashMap` **solution** `)`
  - **Usage**
    - Returns the currently instantiated orientation, or null if the orientation is not completely instantiated
  - **Parameters**
    - `solution` - Current partial solution
  - **Returns** - Instantiated orientation

B.10.3 Class **Solver**

This base solver provides functionality used by all solvers

B.10.3.1 Declaration

public abstract class Solver

extends Object

B.10.3.2 Fields

- public static boolean PerformGAC
  - Determine how to perform the backtrack search; if true, we perform GAC at certain stages
- public static boolean MaintainGac
  - Determine how to perform the backtrack search; if true, we perform GAC at all instantiations
- public static boolean EnableCBJ
  - Determine how to perform the backtrack search; if true, we use conflict directed backjumping
- public static boolean UseMatchingConstraint
  - Determine how to perform the backtrack search; if true, we use the matching as a lookahead mechanism
- public static boolean PreFilterUsingMatching
  - Use the matching only as a prefilter mechanism; only makes sense if UseMatchingConstraint == true
- public boolean UseSpecialVariableOrder
  - Determine how to perform the backtrack search; if true, we use a static variable order that instantiates corner buildings first
- public LinkedList VariableOrder
  - The order to instantiate variables in
- public HashMap VariableDepths
  - Stores what depth each variable is instatiated at

- public static int NumberOfBackTracks
  - The number of backtracks performed by search
- public double RunTime
  - Runtime of search, in ms
- public Csp TargetCsp
  - Csp instance to solve

B.10.3.3 Constructors

- `public` **Solver**`( )`

B.10.3.4 Methods

- `public boolean` **ForwardCheck**`( HashMap` **solution**`, Csp.Variable` **varToCheckAgainst** `)`
  - **Usage**
    - Performs nFC3
  - **Parameters**
    - `solution` - Current partial solution
    - `varToCheckAgainst` - Variable to filter against
  - **Returns** - True, if no domain wipeout occurs
- `public boolean` **GAC1**`( HashMap` **solution**`, Csp.Variable` **filteringVariable** `)`
  - **Usage**

    * Performs GAC1
  - **Parameters**
    * `solution` - Current partial solution
    * `filteringVariable` - Variable most recently instantiated
  - **Returns** - True, if no domain wipeout occurs
- `public void` **PrintVariableOrderAndWidth**`( )`
  - **Usage**
    * For debug purposes; prints the variable instantiation order and width of the ordering

B.10.4 Class VerifySolutionExistsSolver

This solver simply determines whether a solution exists. It terminates search as soon as the problem is decomposed into a consistent forest

B.10.4.1 Declaration

public class VerifySolutionExistsSolver
extends Solvers.BaseSolver

B.10.4.2 Constructors

- `public` **VerifySolutionExistsSolver**`( Csp.Csp` **csp** `)`
  - **Usage**
    * Instantiate the solver
  - **Parameters**

* `csp` - Csp problem instance

B.10.4.3 Methods

- `public boolean` **SolveVerifySolutionExists**`( )`
  - **Usage**
    * Determine the solvability of the CSP
  - **Returns** - True, if the CSP has at least 1 solution

Bibliography

[Agouris and Stefanidis, 1996] P. Agouris and A. Stefanidis. Integration of Photogrammetric and Geographic Databases. *International Archives of Photogrammetry and Remote Sensing, ISPRS XVIIIth Congress*, 31:24–29, 1996.

[Backofen and Will, 1999] R. Backofen and S. Will. Excluding Symmetries in Constraint-Based Search. In *Principles and Practice of Constraint Programming, CP'99*, volume 1713 of *LNCS*, pages 73–87. Springer Verlag, 1999.

[Bakshi *et al.*, 2004] R. Bakshi, C.A. Knoblock, and S.Thakkar. Exploiting Online Sources to Accurately Geocode Addresses. In *12th ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS'04)*, 2004.

[Bayer *et al.*, 2007a] Kenneth M. Bayer, Martin Michalowski, Berthe Y. Choueiry, and Craig A. Knoblock. Reformulating Constraint Satisfaction Problems to Improve Scalability. In *Symposium on Abstraction, Reformulation and Approximation (SARA 07)*, pages 64–79, Whistler, Canada, 2007. LNAI 4612, Springer.

[Bayer *et al.*, 2007b] Kenneth M. Bayer, Martin Michalowski, Berthe Y. Choueiry, and Craig A. Knoblock. Reformulating CSPs for Scalability with Application to Geospatial Reasoning. In *International Conference on Principles and Practice of Constraint Programming (CP 07)*, 2007.

[Benhamou *et al.*, 1994] F. Benhamou, D. McAllester, and P. van Hentenryck. CLP(Intervals) Revisited. In *Proc. of ILPS '94*, pages 124–138, 1994.

[Berge, 1973] C. Berge. *Graphs and Hypergraphs*. American Elsevier, 1973.

[Bessière *et al.*, 1999] C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa. On Forward Checking for Non-binary Constraint Satisfaction. In *CP'99*, pages 88–102, 1999.

[Brown *et al.*, 1988] C.A. Brown, L. Finkelstein, and P.W. Purdom, Jr. Backtrack Searching in the Presence of Symmetry. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *LNCS*, pages 99–110. Springer Verlag, 1988.

[Cheng *et al.*, 1996] B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. Speeding Up Constraint Propagation by Redundant Modeling. In *Second International Conference on Principles and Practice of Constraint Programming (CP 96), LNCS 1118*, pages 91–103. Springer, 1996.

[Choueiry and Davis, 2002] B.Y. Choueiry and A.M. Davis. Dynamic Bundling: Less Effort for More Solutions. In *the 5^{th} International Symposium on Abstraction, Reformulation and Approximation (SARA 2002)*, volume 2371 of *LNAI*, pages 64–82. Springer Verlag, 2002.

[Choueiry and Noubir, 1998] B.Y. Choueiry and G. Noubir. On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems. In *Proc. of AAAI-98*, pages 326–333, 1998. Revised version KSL-98-24, `ksl-web.stanford.edu/KSL_Abstracts/KSL-98-24.html`.

[Choueiry *et al.*, 2005] B.Y. Choueiry, Y. Iwasaki, and S. McIlraith. Towards a Practical Theory of Reformulation for Reasoning About Physical Systems. *Artificial Intelligence*, 162 (1–2):145–204, 2005. Special Issue on Abstraction.

[Dechter and Pearl, 1987] R. Dechter and J. Pearl. The Cycle-Cutset Method for Improving Search Performance Performance in AI Applications. In *Third IEEE Conference on AI Applications*, pages 224–330, 1987.

[Dechter and van Beek, 1996] R. Dechter and P. van Beek. Local and Global Relational Consistency. *Journal of Theoretical Computer Science*, 1996.

[Dechter, 2003] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Doucette *et al.*, 1999] P. Doucette, P. Agouris, M. Musavi, and A. Stefanidis. Automated Extraction of Linear Features from Aerial Imagery Using Kohonen Learning and GIS Data. In *Selected Papers from the International Workshop on Integrated Spatial Databases, Digital Images and GIS (ISD 99), LNCS 1737*, pages 20–33. Springer Verlag, 1999.

[Ellman, 1993] T. Ellman. Abstraction via Approximate Symmetry. In *Proc. of the 13 th IJCAI*, pages 916–921, 1993.

[Freuder and Elfe, 1996] E.C. Freuder and C.D. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proc. of AAAI-96*, pages 202–208, 1996.

[Freuder, 1982] E.C. Freuder. A Sufficient Condition for Backtrack-Free Search. *J. ACM*, 29(1):24–32, 1982.

[Freuder, 1991] E.C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proc. of AAAI-91*, pages 227–233, 1991.

[Gaschnig, 1974] J. Gaschnig. A Constraint Satisfaction Method for Inference Making. In *Proc. of 12th Annual Allerton Conference on Circuit and System Theory*, page 0, 1974.

[Geelen, 1992] P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *ECAI '92*, pages 31–35. John Wiley & Sons, Inc., 1992.

[Giunchiglia and Walsh, 1992] F. Giunchiglia and T. Walsh. A Theory of Abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.

[Glaisher, 1874] J.W.L. Glaisher. On the Problem of the Eight Queens. *Philosophical Magazine*, 4(48):457–467, 1874.

[Glaubius and Choueiry, 2002] R. Glaubius and B.Y. Choueiry. Constraint Modeling and Reformulation in the Context of Academic Task Assignment. In *Working Notes of the Workshop on Modelling and Solving Problems with Constraints, ECAI 2002*, Lyon, France, 2002.

[Glaubius, 2001] R. Glaubius. A Constraint Processing Approach to Assigning Graduate Teaching Assistants to Courses. Undergraduate Honors Thesis. Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2001.

[Guddeti, 2004] V.P.R. Guddeti. An Improved Restart Strategy for Randomized Backtrack Search. Master's thesis, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, December 2004. December 2004.

[Haralick and Elliott, 1980] R.M. Haralick and G.L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[Haselböck, 1993] A. Haselböck. Exploiting Interchangeabilities in Constraint Satisfaction Problems. In *Proc. of the 13 th IJCAI*, pages 282–287, Chambéry, France, 1993.

[Holte and Choueiry, 2003] R.C. Holte and B.Y. Choueiry. Abstraction and Reformulation in Artificial Intelligence. *Philosophical Transactions of the Royal Society Section Biological Sciences*, 358(1435):1197–1204, 2003.

[Hopcroft and Karp, 1973] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM*, 2:225–231, 1973.

[Kilby *et al.*, 2005] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and Backdoors in Satisfiability. In *AAAI 2005*, pages 1468–1373, 2005.

[Lal *et al.*, 2005] A. Lal, B.Y. Choueiry, and E.C. Freuder. Neighborhood Interchangeability and Dynamic Bundling for Non-Binary Finite CSPs. In *Proc. of AAAI-2005*, pages 387–404, 2005.

[Lim, 2006] R.W.H. Lim. GTAAP: An Online System For Managing and Assigning Graduate Teaching Assistants to Academic Tasks. Master's Project. Department of Computer Science & Engineering, University of Nebraska-Lincoln, 2006.

[Michalowski and Knoblock, 2005] M. Michalowski and C.A. Knoblock. A Constraint Satisfaction Approach to Geospatial Reasoning. In *Proc. of AAAI 2005*, pages 423–429, 2005.

[Michalowski *et al.*, 2007a] M. Michalowski, C.A. Knoblock, and B.Y. Choueiry. Reformulating Constraint Models Using Input Data. In *7th International Symposium on Abstraction, Reformulation and Approximation (SARA 2007)*, volume – of *LNAI*, pages –. Springer, 2007.

[Michalowski *et al.*, 2007b] Martin Michalowski, Craig A. Knoblock, and Berthe Y. Choueiry. Reformulating Constraint Models Using Input Data. In *Symposium on Abstraction, Reformulation and Approximation (SARA 07)*, pages 402–403, Whistler, Canada, 2007. LNAI 4612, Springer.

[Michalowski, 2006] M. Michalowski. Personal communication, 2006.

[Milano, 2004] M. Milano, editor. *Constraint and Integer Programming: Toward a Unified Methodology*. Kluwer Academic Publishers, 2004.

[Mittal and Falkenhainer, 1990] S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proc. of AAAI-90*, pages 25–32, Boston, MA, 1990.

[Montanari, 1974] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(66):95–132, 1974.

[Nadel, 1990] B. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n-Queens. *IEEE Expert*, 5(3):16–24, 1990.

[Pickering, 1999] T. Pickering. Speech by Under Secretary of State T. Pickering on June 17, 1999 to the Chinese Government Regarding the Accidental Bombing of the PRC Embassy in Belgrade, 1999.

[Prosser, 1993] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9:268–299, 1993.

[Prosser, 1995] P. Prosser. MAC-CBJ: Maintaining Arc Consistency with Conflict-Directed Backjumping. Technical Report 95/177, Univ. of Strathclyde, 1995.

[Puget, 1993] J.F. Puget. On the Satisfiability of Symmetrical Constraint Satisfaction Problems. In *ISMIS93*, pages 350–361, 1993.

[Puget, 2005] J.-F. Puget. Automatic Detection of Variable and Value Symmetries. In *11th International Conference on Principle and Practice of Constraint Programming (CP 05)*, volume 3709 of *LNCS*, pages 475–489. Springer Verlag, 2005.

[Razgon *et al.*, 2006] I. Razgon, B. O'Sullivan, and G. Provan. Generalizing Global Constraints Based on Network Flows. In *Proc. of the Fifth International Workshop on Constraint Modelling and Reformulation*, pages 74–87, 2006.

[Régin, 1994] J.C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *AAAI 1994*, pages 362–367, 1994.

[Russell and Norvig, 2003] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapter Informed Search and Exploration. Prentice Hall, 2003.

[Sabin and Freuder, 1997] D. Sabin and E. C. Freuder. Understanding and Improving the MAC Algorithm. *Foundations of Constraint Satisfaction*, pages 167–181, 1997.

[Selman and Kautz, 1996] B. Selman and H. Kautz. Knowledge Compilation and Theory Approximation. *Journal of the ACM*, 43(2):193–224, 1996.

[Simon, 1969] H.A. Simon. The Science of Design, Problem Solving as Change in Representation. In *The Sciences of the Artificial*, page 77. The M.I.T. Press, Cambridge, MA, 1969. Karl Taylor Compton Lecture, 1968.

[Simonis, 2005] H. Simonis. Sudoku as a Constraint Problem. In *Working notes of the CP 2005 Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, 2005.

[Thota, 2004] V.R. Thota. Online Interactive Problem-Solving. Masters Project. Department of Computer Science & Engineering, University of Nebraska-Lincoln, 2004.

[Tsang, 1993] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press Limited, 1993.

[Uno, 1997] T. Uno. Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs. In *the 8^{th} International Symposium on Algorithms and Computation (ISAAC 97)*, pages 92–101, London, UK, 1997. Springer-Verlag.

[van Beek and Chen, 1999] P. van Beek and X. Chen. CPlan: A Constraint Programming Approach to Planning. In *AAAI 1999*, pages 585–590, 1999.

[West, 2001] D.B. West. *Introduction to Graph Theory.* Prentice Hall, 2nd edition, 2001.

[Zou, 2003] H. Zou. Iterative Improvement Techniques for Solving Tight Constraint Satisfaction Problems. Master's thesis, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, December 2003.