A GENERAL APPROACH TO USING PROBLEM INSTANCE DATA FOR MODEL REFINEMENT IN CONSTRAINT SATISFACTION PROBLEMS

by

Martin Michalowski

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

December 2008

Dedication

To my wife Sarah, dad Wojtek, mom Margaret, and sister Maia,

for their infinite support, love and patience.

Acknowledgements

I would like to thank my thesis advisor Dr. Craig Knoblock. His guidance throughout my doctoral studies has been paramount in my maturation as a researcher. He has made me a better researcher than I had hoped to become and has provided me with the tools necessary to contribute to the research community. The hours he spent reading my drafts, discussing my ideas, and providing insights into potential problems were invaluable. On a personal level, I would like to thank Craig for being open, caring, and a good friend. Thanks for making it fun to come to work every day.

I would also like to thank Berthe Choueiry. Her sabbatical at ISI made a very positive contribution to my thesis. She helped propel my research forward when I felt it was standing still and her ideas helped drive me in the right directions. Her enthusiasm for research was contagious and helped motivate me to do better work. Her attention to grammatical detail has made me a better writer.

I would like to thank both Cyrus Shahabi and John Wilson for being on my committee. Their questions and comments helped improve this thesis. I would like to thank Yolanda Gil for her contributions to my qual and for giving me the opportunity to contribute to the research community and to the AI-Grads organization at ISI. Thank you Steve Minton for helping me during my early years as a researcher and for his glowing recommendation

letters. Thank you to Alma Nava for helping me with any and all issues I needed resolved at ISI.

I also want to thank all of my colleagues in our research group at ISI. Thank you Snehal Thakkar for helping me get through the early days and for being so easy to beat at FIFA. Thank you Matt Michelson for understanding my humor. I had a lot of fun in our offices and it was mostly due to your humor and plethora of random facts. Thank you Wesley Kerr for being a great friend and a soccer fan (Polska!). Thank you Rattapoom Tuchinda for rarely coming in and giving Matt and I more space in the office. Thank you Aram Galstyan for all our soccer talks and for your friendship. Lastly, thank you José Luis Ambite, Mark Carman, and the rest of the information integration group members.

I'd like to thank all of my friends in LA, namely Tom, Dan, Shawn and Walter. You made life fun. Thank you Matt Sterbenz for being a great friend for many years, for keeping me level headed throughout my prolonged academic career, and for making the best skis out there. Thank you Randy Cordray for all the amazing ski trips and for being a great friend. Thanks Ted, Bill, Rob, Doug, Dennis, Matt, and the rest of the Mammoth crew for some of my best days in years.

I would like to also thank each member of my family. Thank you Wojtek for caring about me more than any son could hope for. You are an amazing inspiration and the best father and the best friend I could have hoped for. Thank you Margaret for being the most caring and loving mother alive. If I could plant a flower in your garden for each time you amazed me with your strength, courage, tenderness, and love, you would live in the Amazon. Thank you Maia for being an amazing sister that has made me realize

what true compassion is about. You are a bright and amazing woman who will achieve anything you set your mind to.

Finally, thank you to my wife Sarah. The patience, love, and compassion you have shown me goes beyond words. You bring joy to my life in countless ways and the love we share is endless. I cherish all of the moments we spend together and I am lucky to have shared so many amazing memories with you already. Our life's journey will be an amazing adventure but more importantly it will be one we share together.

This research is based upon work supported in part by the National Science Foundation under award number IIS-0324955, and in part by the Air Force Office of Scientific Research under grant number FA9550-07-1-0416.

The U.S. Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. 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.

Table of Contents

Dedication ii

Acknowledgements iii

List Of Tables ix

List Of Figures xi

Abstract xiii

Chapter 1: Introduction 1
- 1.1 Problem Statement . . . 1
  - 1.1.1 Application Domains . . . 2
- 1.2 Motivating Examples . . . 4
  - 1.2.1 Building Identification (BID) Problem . . . 4
  - 1.2.2 Sudoku Puzzles . . . 6
- 1.3 Thesis Statement . . . 8
- 1.4 Proposed Approach . . . 8
- 1.5 Contributions of the Research . . . 11
- 1.6 Dissertation Outline . . . 11

Chapter 2: Constraint-Inference Framework 13
- 2.1 Generic Model . . . 14
- 2.2 Data Points . . . 15
- 2.3 Constraint Library . . . 19
- 2.4 Inference Rules . . . 23
- 2.5 Inference Engine . . . 25
- 2.6 General Framework Definition . . . 27
- 2.7 End-to-End Model Generation Process . . . 29
- 2.8 Case Study . . . 30

Chapter 3: Selecting Constraints 35
- 3.1 Constraint Inference Algorithm . . . 35
  - 3.1.1 Bucket Creation . . . 37
  - 3.1.2 Evaluating Inference Rules . . . 41
  - 3.1.3 Determining Constraint Applicability . . . 45
- 3.2 Augmenting Data Points Using Constraint Propagation . . . 49

Chapter 4: Using Support Vector Machines to Learn the Scope of Constraints 53
4.1 Support Vector Machines . . . 56
4.2 Training a SVM Model . . . 59
4.3 Assigning Data Points to a Scope . . . 65

Chapter 5: Instantiating a Constraint Model 69
5.1 Transitioning to an Instantiated Model . . . 69
5.2 Constraint Propagation . . . 72
5.3 Constraint Schema . . . 78
5.4 Instantiating a BID Problem Instance . . . 80

Chapter 6: Experimental Evaluation 84
6.1 Data Sets . . . 84
6.1.1 BID Problem . . . 85
6.1.2 Sudoku Puzzles . . . 88
6.2 Experimental Setup . . . 89
6.3 Inferring Constraint Models . . . 89
6.3.1 BID Problem . . . 91
6.3.2 Sudoku . . . 95
6.4 Learning the Scope of Constraints using SVMs . . . 100
6.5 Augmenting Input Data . . . 106
6.6 Automatic Model Inference . . . 110
6.6.1 BID Problem . . . 111
6.6.2 Sudoku . . . 118

Chapter 7: Related Work 122
7.1 Constraint Programming . . . 122
7.2 Constraint Modeling . . . 125
7.3 Learning Constraints . . . 128
7.4 Geospatial Integration and Reasoning . . . 129

Chapter 8: Discussion and Future Work 131
8.1 Contributions . . . 131
8.2 Application Areas . . . 133
8.2.1 Machine translation . . . 133
8.2.2 Genealogical Trees . . . 134
8.3 Limitations . . . 135
8.4 Directions for Future Work . . . 136
8.4.1 Learning Inference Rules . . . 136
8.4.2 Enhancing the Learning of a Constraint's Scope . . . 138
8.4.3 BID Problem . . . 138
8.4.4 Sudoku Puzzles . . . 139

Bibliography 140

Appendices 145
Appendix A
Constraint Inference Rules . . . 146
A.1 BID Problem Inference Rules . . . 146
A.1.1 *Odd on North* rules . . . 146
A.1.2 ¬*Odd on North* rules . . . 146
A.1.3 *Odd on East* rules . . . 147
A.1.4 ¬*Odd on East* rules . . . 147
A.1.5 *Increasing North* rules . . . 148
A.1.6 *Increasing South* rules . . . 148
A.1.7 *Increasing East* rules . . . 149
A.1.8 *Increasing West* rules . . . 149
A.1.9 *K-Block Numbering* rules . . . 149
A.1.10 *San Francisco Block Numbering* rules . . . 150
A.1.11 *C-Continuous Numbering* rule . . . 150
A.1.12 *Marker Distance* rule . . . 150
A.2 Sudoku Inference Rules . . . 150
Appendix B
Constraint Model: XML Schema . . . 152
B.1 Layout XML file . . . 152
B.2 Phone-book XML file . . . 154
B.3 Grid XML file . . . 155
B.4 Landmark XML file . . . 156
B.5 Inferred Ranges XML file . . . 158
B.6 Ascending/Descending Value XML file . . . 159
B.7 Parity XML file . . . 161
B.8 Continuous Numbering XML file . . . 162
B.9 District Boundaries XML file . . . 163

List Of Tables

2.1 Sample constraints stored in the BID problem constraint library. 20

2.2 Sample constraints stored in the Sudoku puzzle constraint library. 22

6.1 Homogeneous BID problem instances. 85

6.2 Non-homogenous BID problem instances. 87

6.3 BID problem instance results. 92

6.4 Other cities: inferred constraints. 93

6.5 Sudoku: accuracy and completeness of inferred constraints. 96

6.6 Bayes factor: strength of evidence. 98

6.7 Bayes factor for varying numbers of filled-in cells. 99

6.8 Sudoku: accuracy and completeness of model with a support level of 5. . . 100

6.9 Accuracy measures when applying SVMs to Sudoku puzzles. 105

6.10 Constraint Propagation: average number of new data points. 107

6.11 Iterative Propagation: accuracy and completeness of inferred constraints. 108

6.12 Automated BID problem instance results. 112

6.13 BID problem case studies used to evaluate performance. 115

6.14 BID problem results for case studies without block-numbering constraints. 116

6.15 BID problem results for case studies with block-numbering constraints. . . 116

6.16 Sudoku problem results with an inferred constraint model. 119

List Of Figures

1.1 Data integration in the BID problem. 5

1.2 Variations of Sudoku puzzles. 7

2.1 Sample BID gazetteer data points for El Segundo. 17

2.2 Sample data points for a Sudoku puzzle. 18

2.3 Three sample inference rules. 24

2.4 Three sample Sudoku inference rules. 26

2.5 General definition of the constraint-inference framework. 28

2.6 General architecture of the constraint-inference framwork. 29

2.7 End-to-end inference process. 30

2.8 The area of El Segundo for which a model must be inferred. 31

2.9 Two example data points in El Segundo. 32

2.10 The coverage of the applicable constraints in El Segundo. 33

3.1 Constraint inference algorithm. 36

3.2 The algorithm used to create buckets of data points. 38

3.3 Bucketing algorithm example. 39

3.4 Subset of data points in El Segundo. 40

3.5 The algorithm used to evaluate inference rules for a given bucket. 43

3.6 Sample BID problem inference rules providing support for *Odd on North*. 44

3.7 Example scenario where false support is provided for *Block-numbering*. . . 47

3.8 Iterative algorithm to find new data points. 50

3.9 Before and after constraint propagation using SAC for a Sudoku puzzle. . 51

4.1 The scopes of the *Increasing East* and *West* constraints in El Segundo. . 54

4.2 The two scopes of the *Parity* constraints in Belgrade. 55

4.3 Separating hyperplanes learned by SVMs. 57

4.4 Sample input vectors for conflicting constraints used to train the SVM model. 60

4.5 Labels used to learn the SVM model in El Segundo. 63

4.6 New training examples generated in the BID problem. 64

4.7 Assigning data points to a scope. 65

4.8 El Segundo data points with no support for either conflicting constraint. . 67

5.1 Algorithm used to propagate the set of applicable constraints. 74

5.2 E/W street coverage of the *Odd on North* constraint **before** propagation. 76

5.3 E/W street coverage of the *Odd on North* constraint **after** propagation. . 77

5.4 Instantiated example of the *Odd on North* constraint. 79

5.5 The process by which a constraint model is instantiated for the BID problem. 82

6.1 Results for finding contexts in El Segundo using SVMs. 102

6.2 Example Samurai Sudoku puzzle. 104

Abstract

The initial formulation, or model, of a problem greatly influences the efficiency of the problem-solving process. Hence, modeling is critical in determining the performance of the problem-solving process and the quality of the produced solution. Unfortunately, modeling remains an art and has resisted automation. Additionally, slight variations in the characteristics of a given problem instance make it difficult to represent one class of problems using a unique model. Consequently, a robust approach to modeling is required.

My thesis presents an automated approach to modeling. I exploit information contained in the *input data* in order to customize the constraint model of a given problem instance. I apply my approach to the area of Constraint Programming, focusing on a class of problems where a solution is guaranteed to exist for all problem instances. Specifically, I enhance a generic constraint model of a Constraint Satisfaction Problem (CSP) by adding new constraints to this model. These additional constraints are selected from a library of constraints by testing features of the problem instance at hand. The resulting constraint model is customized such that it best represents the problem instance given the data provided as input.

The resulting framework is applicable to problems where instances are seeded with some initial input data. The techniques present in the framework are generally defined so

they can be applied across domains. They cope with the uncertainty in the model refinement process by handling noisy data and incorrect inferences, generating additional information as needed. Furthermore, the framework uses Support Vector Machines (SVMs) to determine the scope of the inferred constraints and it automatically instantiates the constraint model in a format supported by a specialized constraint solver.

I evaluate the effectiveness of my approach in two domains, Sudoku puzzles and the building identification (BID) problem. I show how it can infer the most specific constraint model given the available public information, scale to large problem instances, and use a SVM model to efficiently determine constraint scopes. I also evaluate the effectiveness of the framework for the BID problem in areas such as New Orleans and Belgrade where non-homogenous problem structures co-exist and across different Sudoku puzzle variations, demonstrating the resulting improvement when using an inferred model over a generic one. Additionally, I evaluate the framework's ability to augment the initial set of data points with new ones when applying an iterative constraint-propagation algorithm, which leads to more accurate constraint models.

Chapter 1

Introduction

In this chapter, I start by describing the problem being addressed by my thesis work and I present two motivating examples. I provide a thesis statement and I generally describe my proposed approach to automated modeling. Finally, I summarize the contributions of my research and layout the remainder of this dissertation.

1.1 Problem Statement

Constraint Programming (CP) is an effective paradigm for modeling and solving combinatorial problems [45; 47; 51; 59]. A fundamental component of the problem-solving process is the ability to accurately model the problem instance at hand. Modeling typically requires a significant effort from a domain expert and must be carried out for every new problem encountered. Specific to CP, a domain expert must define the variables and their domains along with the applicable constraints. In application domains that exhibit numerous problem variations, the repetition of the model generation task places a considerable burden on the domain expert.

The performance of problem solving and the accuracy of the results heavily depends on the quality of the model. A 'good' constraint model consists of well chosen variables and constraints that faithfully represent the characteristics of the problem domain, allowing the solver to leverage the inherent structure of the problem. The novelty of the approach presented in this dissertation lies in exploiting the input data of a given problem instance for automatic model generation. Specifically, I show how to customize the generic model[1] of a CSP by adding constraints that best represent a given instance.

1.1.1 Application Domains

Among the various problems that are modeled and solved using CP techniques, I distinguish as a special class those that are guaranteed to always have a solution and I focus my investigations on this class of problems. Two examples of such problems are the Building Identification (BID) problem [47] and puzzles such as Sudoku [59]. For the BID problem, where all buildings on a map must be assigned addresses, a solution exists as witnessed by the physical reality. In the case of Sudoku puzzles, they are "naturally" built to have a solution. These problem classes contain variations across instances within a class that require CSP models with different sets of constraints. When solving these problems, using the same model for all instances may yield imprecise solutions because the model may enforce constraints that incorrectly reduce the problem space while ignoring applicable ones. Section 1.2 outlines the BID problem and Sudoku puzzles in more detail, providing additional motivation.

[1] A generic or vanilla model is the model with the basic set of constraints that represent the general characteristics of the problem.

Focusing on the class of problems where a solution is guaranteed to exist leads to the new technologies presented in this dissertation. The nature of these problems allows me to make assumptions that may not hold in other domains. After specializing a model for a given problem instance, if a constraint solver returns no solution I know an incorrect model for the instance was inferred. This knowledge signifies that backtracking over the inferences is required to relax the model until a solution is produced. Even though we may never know if we have inferred the most constrained model, any model that leads to a solution correctly represents the problem with a varying degree of accuracy. As such, inferred models can be seen as an improvement over generic models for all problem instances.

The techniques presented in this dissertation are general and can be applied to other domains. I note that they can be applied to any problem domain where commonalities existing across all problem instances can be leveraged and where variations in the instances makes it infeasible to generate and store all possible constraint models beforehand. The nature of these problems is that a solution exists: the proof is the real world. Additionally, the existence of a solution for all problem instances allows for the use of relaxation techniques and constraint propagation to ensure that the inferred constraint model is correct and as complete as possible.

Apart from the two problem domains studied and presented in Section 1.2, I also outline additional domains that can benefit from the application of the constraint-inference techniques. These domains are discussed in more detail in Section 8.2 and include machine translation and genealogical trees.

1.2 Motivating Examples

In this section, I present the BID problem and Sudoku puzzles in more detail. These two problem domains serve as motivating examples for my work and they are used throughout the thesis to provide examples and clarification of introduced concepts.

1.2.1 Building Identification (BID) Problem

Consider the problem of mapping postal addresses to buildings in satellite imagery using publicly available information, which I defined as the BID problem in [47]. This problem takes as input:

- A bounding box that defines the area of a satellite image
- Buildings identified in the image
- Vector information that specifies streets in the image
- A set of phone-book entries for the area.

This information is used to create a constraint model that is solved by a CSP solver that returns a set of possible address assignments for each building. The model comprises a set of constraints exploiting the geospatial characteristics of addressing in the world and is assumed to apply globally. The integration of data sources in the BID problem is illustrated in Figure 1.1.

The BID problem exhibits two main traits that make it especially difficult to solve as a CSP. First, the nature of assigning addresses to buildings leads to the existence of

Figure 1.1: Data integration in the BID problem.

numerous constraint models representing all of the addressing strategies and variations exhibited throughout the world. For example, consider the following addressing patterns:

1. "Block-numbering" occurs when numbers across city blocks increment by a fixed value and is mainly seen in some cities in the US.

2. In Europe, buildings that surround squares are usually numbered consecutively and in the clockwise direction.

3. In rural areas of Australia, the numbering system is based on tenths of kilometers.

As we can see, these patterns are not global but specific to different regions of the world and may also differ between new and older districts within a given region. Therefore, different models with different sets of constraints hold for different areas of the world. Thus a model that most accurately and faithfully represents an area has to be built. Building and storing models for every area in the world is a daunting task for a domain expert and may be unrealistic in practice.

The second difficulty encountered in the BID problem is the heterogeneity of addressing *within* small geographic areas. For example, in El Segundo California, the address numbers along east/west running streets are non-monotonic. Specifically, address numbers increase to the west direction for buildings west of Main Street and to the east when buildings are east of Main Street. The ability to solve BID problems for El Segundo relies on both the inference of constraints and the ability to determine the *scope* of these constraints. That is, constraints must be instantiated over only the buildings to which they are applicable (i.e., buildings east/west of Main Street) and the relevant set of buildings must be inferred automatically. Without this mechanism, constraint models would contain conflicting constraints and would be unsolvable.

1.2.2 Sudoku Puzzles

Another problem domain that falls within my studied class of CSPs is Sudoku puzzles. Sudoku is a logic-based placement puzzle similar to a Latin Square. It has been shown that solving basic Sudoku puzzles is NP-complete [65] and can be accomplished using CP [59]. Interestingly, slight variations of the basic Sudoku puzzle are played throughout the

world (see Figure 1.2), where each variation adds to the constraints that define the basic puzzle. For example, the diagonal Sudoku adds an 'all-different' constraint for each of the diagonals; and the Samurai Sudoku combines five different puzzle variations in a quincunx arrangement. A generic CSP model represents only weakly all these variations. Hence, to solve a given Sudoku instance, an automated system must identify the constraints applicable to a given instance and their scope.

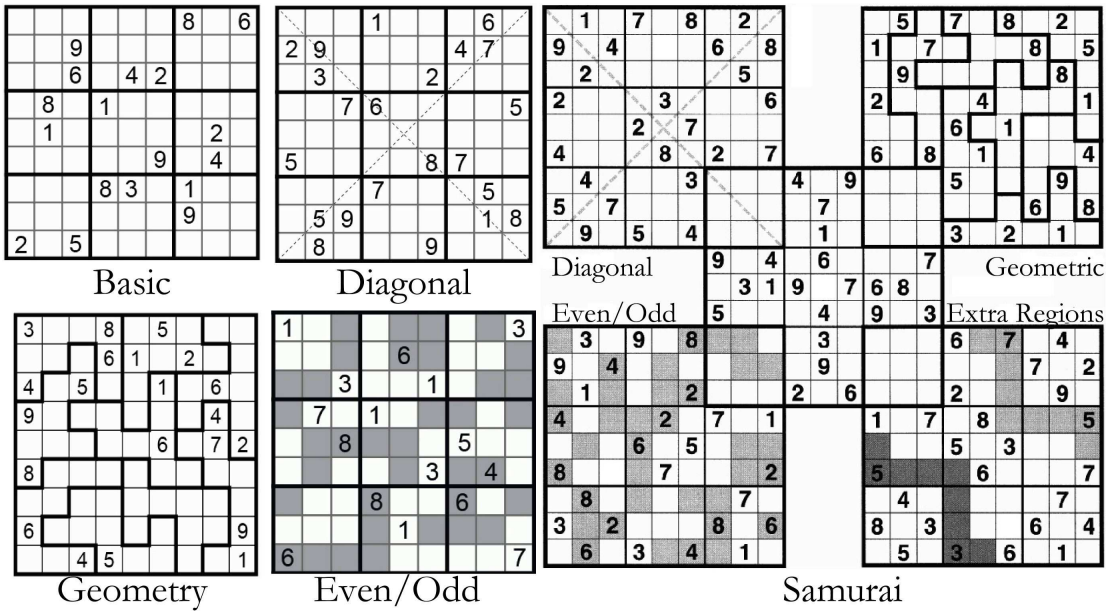

Figure 1.2: Variations of Sudoku puzzles.

In practice, a user is given a Sudoku puzzle with a set of constraints that define it. However, picture a scenario where we would like to develop an automated Sudoku solver that can easily be expanded to support variations on the basic puzzle. Given any Sudoku puzzle, the solver would model the puzzle given its filled-in cells and solve it accordingly. Rather than storing models for all possible puzzle variations, the solver has access to a set of constraints that may appear in some puzzle and this set of constraints is easily expanded to support new variations. This problem scenario is a new approach to Sudoku

and serves as a vehicle to illustrate the ideas behind my work. Although the BID problem is more complex in nature, its issues can be quite naturally mapped to the Sudoku puzzle scenario described above. Additionally, Sudoku puzzles are designed so that a solution exists for every instance and thus fall into the identified class of solvable CSPs.

As the two example problem domains show, a constraint inference engine needs to infer a set of applicable constraints *and* their scope for problem instances that vary within a given problem class. The methodology I present in this dissertation possesses this ability, and I show how the framework handles the aforementioned cases.

1.3 Thesis Statement

I describe a novel approach to model refinement in Constraint Satisfaction Problems (CSPs) that augments the basic model of a given problem instance by exploiting input data to infer additional applicable constraints and their scope, allowing for the automatic generation of a constraint model specialized for the given instance.

1.4 Proposed Approach

The proposed inference framework augments a basic model of a given problem instance with additional constraints inferred from data points defined by the instance's input data. This framework consists of three major building blocks:

- the data points
- a library of constraints

- inference rules

The data points are defined by a set of domain-specific features and used to test the inference rules. The inference rules provide support for the specific constraints contained in the constraint library. The testing of these rules results in a set of applicable constraints that best represent the problem instance given the input information.

Input Data: The input data is comprised data points (variable-value pairs), represented as variables whose values are obtained from the problem description or public online sources when available. The data points constitute a partial solution that can be used to determine the set of applicable constraints. The relationship between the variable-value pairs contains information which I exploit to determine the set of constraints that make up the constraint model for the given problem instance. In domains such as the BID problem, additional input data from sources such as vector data can be used to reduce the domains of variables in the model (i.e., limiting the set of potential streets for a given building), leading to a more specific and accurate instantiation of the constraint model.

Constraint Library: The library of constraints consists of a set of domain-expert defined constraints. These constraints capture characteristics of the problem domain that have occurred in *some* problem instance but do not necessarily apply across *all* instances. Rather than generating constraint models for all foreseeable problem instances in a given domain, the framework only requires the definition of constraints that may apply to some instances. I believe that the task of populating the constraint library once is more tractable than the generation of all constraint models. Furthermore, to handle previously unseen characteristics I simply add new constraints to the library, a

more desirable alternative to going through all constraint models and updating them as needed.

Inference Rules: The framework evaluates a set of inference rules to provide support, both positive and negative, to the constraints in the library. An inference rule is a logical expression that, when asserted, provides positive support for the corresponding constraint in the library. Conversely, negative support can also be provided. Additionally, multiple inference rules can provide support for one constraint, strengthening the framework's confidence in its inferences. Finally, the constraint inference engine uses the data points and the set of inference rules to infer the applicability of the constraints in the library based on their respective levels of support.

The selection of constraints entails a three-step process. First, the inference rules are evaluated and constraints are provided with both positive and negative supports. Based on each constraint's *support level*, the constraints are divided into one of three categories: applicable, non-applicable, or unknown. Finally, before generating a specialized model for the given problem instance, the framework must determine the *scope* of the constraints. Support Vector Machines (SVMs) are used to learn scopes, supporting overlapping and distinct scopes. The use of SVMs provides a means for dealing with the heterogeneity found within problem instances (examples of which are shown in Section 1.2). Given a set of inferred constraints and their scope, a constraint model is instantiated over the problem variables and this model is passed to a specialized constraint solver.

1.5 Contributions of the Research

The key contribution of the research is a *novel approach to Constraint Satisfaction Problem (CSP) model refinement*. The resulting framework includes the following contributions:

- A general framework for automatic model generation that supports varying problem instances within a problem class.
- The inference of a constraint model based on the problem instance at hand.
- The ability to deal with noisy data and incorrect inferences using an iterative propagation algorithm and support levels.
- The use of Support Vector Machines to automatically learn the scope of applicable constraints.

1.6 Dissertation Outline

The remainder of this proposal is organized as follows. Chapter 2 provides background information and defines the general framework. Chapter 3 describes the algorithm used to select the applicable constraints for a given problem instance. Chapter 4 introduces a Support Vector Machine based approach that automatically learns the scopes of the inferred constraints. Chapter 5 outlines the process by which the inferred constraint model is instantiated and passed to a specialized constraint solver. Chapter 6 evaluates the general framework when applied to the BID problem and Sudoku puzzles, and provides empirical results for individual components that improve the overall inference process.

Chapter 7 reviews work related to this research. Finally, Chapter 8 presents a summary of my contributions and provides directions for future work.

Chapter 2

Constraint-Inference Framework

In this chapter, I introduce the constraint-inference framework. I motivate the framework's key components by outlining their purpose, providing each concept's definition along with a concrete example from both application domains (the BID problem and Sudoku puzzles). I show how a *generic CSP model* is used to represent the characteristics defining a particular class. I introduce *data points* as a representation of the information contained within a problem instance. I describe a *constraint library* as a repository of characteristics known to exist for some but not all instances of a problem domain. Finally, I present *inference rules* as a means to map the data points to constraints in the library and outline the *inference engine* that evaluates the set of inference rules. I conclude this chapter by generally defining the constraint-inference framework and the process used to infer constraint models and by introducing a case study used as a running example for the later chapters of this thesis.

The definition of the constraint-inference frameworks lays the groundwork for the techniques used to infer instance specific models. The case study presents the BID problem for El Segundo California and helps ground and motivate the techniques presented in subsequent chapters.

2.1 Generic Model

As introduced in Section 1.1, the class of problems studied in this work is grounded in the real world. As such, all instances are solvable and a set of common characteristics exist for all problem instances. The guaranteed solvability of an instance leads to new techniques used to infer a constraint model and these techniques are detailed in later chapters. The representation of the common characteristics of a problem domain is the first step in defining the constraint-inference framework. It is achieved through the use of a *generic model*, which defines the universally applicable characteristics of a problem domain.

The generic model is composed of a set of generic constraints that apply to *all* instances for a problem class. This set of constraints represents the core characteristics that define a given problem domain. When the generic model is used to model any problem instance within a particular domain, it is guaranteed that solving this model will generate a solution. However, because a generic model tends to be under-constrained, this model is inefficient to solve and the resulting solutions can be imprecise. As such, this generic model is refined with newly inferred constraints that best represent the instance given the information contained in the problem.

Specifically, the generic model is comprised of the intersection of all constraint models for a problem class. Intuitively, the commonalities found across problem instances within a given problem class define this class. A generic model's role is to capture these commonalities and leverage them in various ways. One assumption being made for all constraints in the framework (including the ones in the generic model) is that they are monotonic. This means a constraint can only reduce the domain of a variable and it cannot re-introduce new values when it is asserted. One of the advantages of maintaining monotonicity is that the generic model can also be used to augment the information contained within a problem definition through the use of constraint propagation (see Section 3.2).

Example constraints that make up the generic model for the BID problem are: all known addresses have to be assigned to a building, corner buildings are only assigned to one street, and all buildings must be assigned an address. The full set of generic constraints was determined by globally surveying addressing characteristics and noting the ones that applied throughout the world. For Sudoku puzzles, example generic constraints are: all numbers along a row and along a column must be different. Again, these constraints characterize all of the puzzle instances considered in this work.

2.2 Data Points

Once the framework is able to represent the general characteristics that govern a particular problem domain, the challenge lies in finding characteristics *specific* to a given problem instance. Once found, these characteristics are used to refine the generic model such that

it best represents the problem instance. To extract information specific to an instance, the framework must represent the initial specifications of an instance along with any additional information that may be available. This ability is captured in the framework through the use of *data points*.

Generally speaking, data points are elements of the input data, such as information that instantiates some of the CSP variables of the generic model. These data points are described using a set of domain-specific features defined by a domain expert. These features represent characteristics specific to the problem domain. A collection of feature values describes a single data point and provides the framework with instance-specific information used to infer the characteristics governing the instance.

The availability of public online data sources, such as gazetteers for the BID problem domain, allows the framework to represent data points using a rich set of features. The feature values can be automatically extracted from online information and in turn the data points provide enough information to infer the representative characteristics of a given problem instance. For the BID problem, data points comprise buildings with known addresses obtained from online gazetteers. These data points are landmark buildings defined by the following features: ID, Address Number, Street Name and Orientation, Side of Street, Latitude, Longitude, Block Number, and Street Ordering. Vector data sources and online phone books provide information that can be used to automatically assign values to the features of all data points and to reduce the domains of variables in the constraint model. Figure 2.1 provides an example set of data points for El Segundo California.

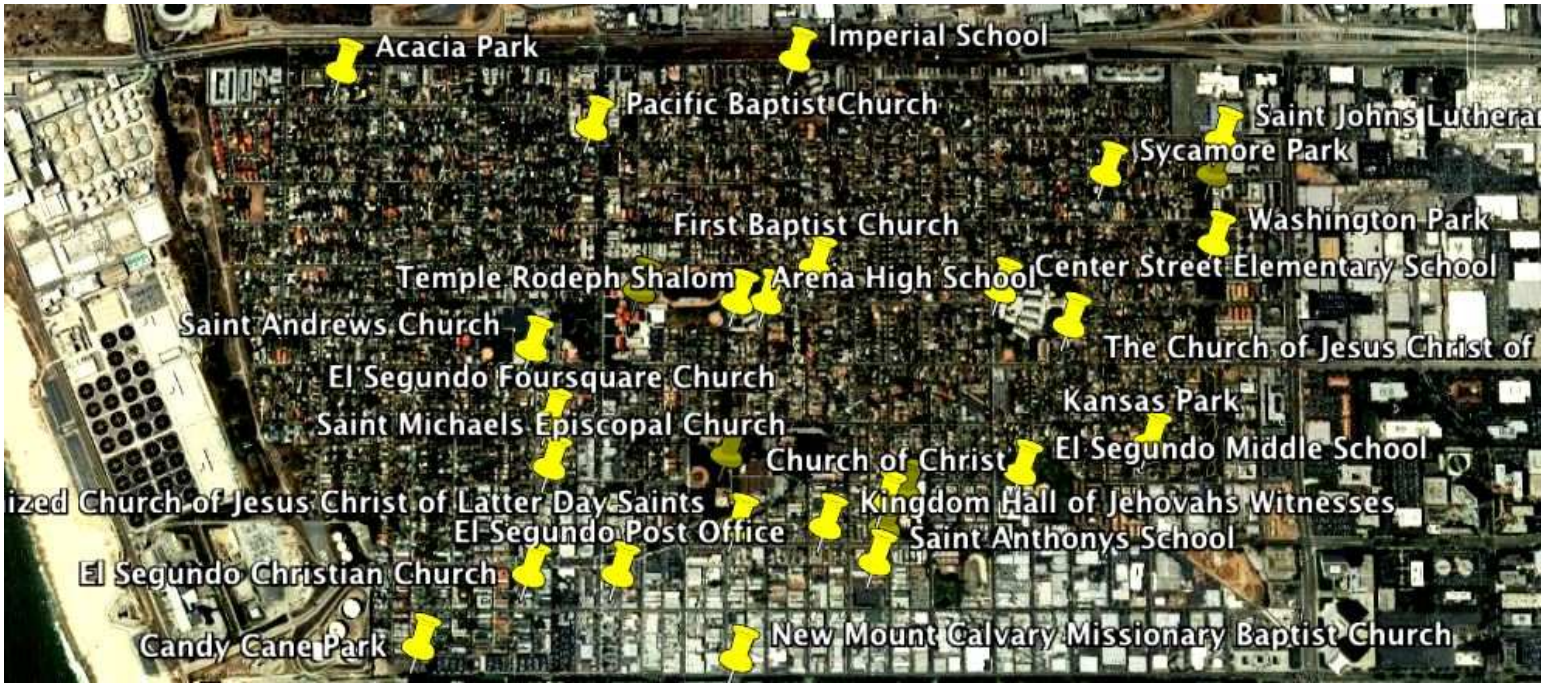

Figure 2.1: Sample BID gazetteer data points for El Segundo.

When additional information is neither available nor applicable, as in Sudoku puzzles, the data points represent all of the information provided by the problem specification. Specifically, data points in the Sudoku domain represent the cells filled with numbers in the initial puzzle definition. These data points are defined by the features: Filled-in Number, Row, Column, Region, and Cell Color. The values for each data points' feature can be derived from the problem definition. As an example, all filled-in cells on the first row have a value of 1 assigned to the *Row* feature. A sample Sudoku puzzle is presented in Figure 2.2 where the data points represent all the cells that contain a number.

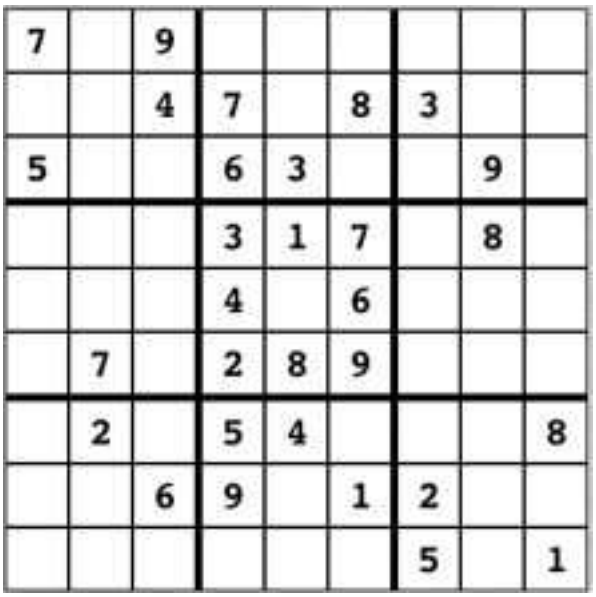

Figure 2.2: Sample data points for a Sudoku puzzle.

The framework uses the relationships between instance-specific data points to infer the applicable characteristics of the instance. This processes involves the selection of constraints from a *constraint library* (described in Section 2.3) using *inference rules* (outlined in Section 2.4) to exploit the identified relationships. The approach used to select the applicable constraints is described in Chapter 3.

2.3 Constraint Library

Having established a representation for the common characteristics of a problem domain (the generic model) and for the information specific to the instance (the data points), the framework needs to represent and store relationships that *might* exist for any given problem instance. With the goal of refining a generic constraint model and to be consistent with the representation of this generic model, each specialized characteristic is defined as a constraint. All of these constraints are stored in a *constraint library* and made available to the framework.

Specifically, a constraint library is used as a repository of problem characteristics that do not apply in all problem instances. A problem characteristic, identified by a domain expert and not universally applicable to all problem instances, is represented as an individual constraint and stored in the library. Subsequently, the automatic inference of a constraint model involves the selection (from this library) and the instantiation of applicable constraints for a given problem instance. The selection process is outlined in Chapter 3 and Chapter 5 describes the instantiation of the applicable constraints.

An alternate approach to housing individual constraints in a library is to store pre-constructed sets of constraints where a set defines one problem instance type for a given problem class. Using this approach, all pre-determined problem instance types would be represented by some set of constraints. However, treating the library as a "bag of constraints", as done in the inference framework, is more robust in its ability to handle variations across problem instances. Specifically, storing the constraints individually

rather than in sets allows the framework to support problem instance types that may not have been pre-determined by the domain expert.

For example, if a new variation of Sudoku uses a previously unseen combination of the existing Sudoku constraints, the framework would be able to support this new variation without any modification. The only scenario that would require an update to the constraint library would be when a new constraint within a supported problem class is identified. Once identified, this new constraint would be added to the library and all subsequent problem instances that exhibit the characteristics captured by this constraint could then be correctly modeled. This view of the constraint library is similar to the Local-As-View approach to data integration [28; 43].

Table 2.1: Sample constraints stored in the BID problem constraint library.

Name	**Description**
Odd on North	Addresses on the same side of a E/W running street have the same parity
Increasing North	Addresses increase monotonically to the North along a N/S running street
K-Block Numbering	Address increment by a factor of k across street blocks

Table 2.1 specifies some of the constraints found in the BID problem constraint library. The full set of constraints can be found in Appendix A.1. As an example, *Odd on North* is a unary constraint that specifies the side on which odd numbers lie for East/West running streets. For example, if the constraint *Odd on North* is asserted in an instantiated model, all buildings that are placed on the north side of E/W running streets must be assigned an odd value. The notion of addresses on one side of a street having the same parity is a common addressing characteristic that applies to most of the cities in the world.

However, in surveying addressing schemas throughout the world, I found that the side on which addresses are even/odd varies drastically. Additionally, parts of some cities, such as London, England, contain addresses with a mixed parity on the same side of the street and do not abide by this constraint. The variation of the parity in addresses is demonstrated in the experimental results presented in Chapter 6.

The *Increasing North* constraint is a binary constraint that specifies the direction in which addresses get bigger along a North/South running street. Much like the *Odd on North* constraint, the direction in which addresses increase varies greatly depending on the area of interest. As such, constraints representing the directions in which addresses increase for North/South and East/West running streets are stored in the constraint library. Furthermore, the direction in which addresses increase also varies *within* a given city, further complicating the inference process. This phenomenon is handled by learning the scope of inferred constraints and the process by which this is done is detailed in Chapter 4.

The *K-Block numbering* is a binary constraint that limits the allowable values of two corner buildings across a city block. This constraint specifics the increment value K by which addresses increase across street blocks. This is a common characteristic for North American cities but its applicability is rare in other parts of the world, as shown in Chapter 6. The applicability of the three above-mentioned constraints is not universal and as such they are part of the constraint library. For the BID problem, the constraint library represents a collection of addressing schemas that we know to exist *somewhere* but not *everywhere* in the world.

Table 2.2: Sample constraints stored in the Sudoku puzzle constraint library.

Name	**Description**
AllDiff Diagonal	All numbers for cells along a diagonal must be different
Color Small	All numbers for cells in the given color must be in the range {1...4}
Magic	All numbers for cells in the given color must be greater then or equal to the number of colored cells

Table 2.2 specifies some of the constraints found in the Sudoku puzzle constraint library. The full set of constraints can be seen in Appendix A.2. As outlined in Section 1.2, variations of the basic Sudoku puzzles are played throughout the world. The role of the constraint library in the Sudoku puzzle domain is to capture the various characteristics that are added to the basic Sudoku puzzle when creating these variants. For example, the *AllDiff Diagonal* constraint is non-binary and ensures that numbers on the diagonals are all different. This characteristic is derived from Diagonal Sudoku puzzles and it also occurs in some color-based puzzles and in Magic Sudoku variations.

Similarly, *Color Small* is a unary constraint that captures a characteristic found in Big/Small Sudoku puzzles and limits the domain of cells for a given color to the range [1,4]. Finally the *Magic* unary constraint limits the domain of cells for the given color to the number of cells of that same color in the particular region. Again, this constraint is derived from a variation of Sudoku called Magic Sudoku. In capturing the defining characteristics of Sudoku puzzles that build on the basic one, the constraint library allows the inference framework to be the foundation for a generic Sudoku puzzle solver. For a generic Sudoku solver to function, it must contend with both puzzles of varying constraints and with combinations of constraints it may not have seen previously. The latter presents a strong

case for treating the constraint library as a "bag of constraints" versus storing constraint models for all known puzzle types.

2.4 Inference Rules

The main function of the constraint-inference framework is the selection of applicable constraints for a given problem instance from the constraint library. Two difficulties need to be addressed before this process can take place. First, certain properties must hold in the problem instance for specific constraints to be applicable. Testing for these properties is imperative to determining the applicability of constraints. Second, the data points are defined by a set of features and as such reside in the *feature* space. On the other hand, the constraints in the library are defined over the constraint variables that make up a constraint model and exist in the *variable* space. These are two separate spaces and a mapping from one to the other must be established. To handle both issues, the framework uses *inference rules* to represent relationships between data points and maps these relationships to constraints in the variable space.

The inference rules in the framework are predefined by a domain expert, similar to the use of expert modules in PROVERB [45], and the rule language supports any programmable predicate expressions. The rules are separated from the constraints in the library and act as a link between the constraints and the features defining the data points. These logical expressions represent a relationship that must hold between the feature values of two or more data points if a given characteristic appears in the particular problem instance. This characteristic is represented by a unique constraint in the constraint library.

More specifically, the difference between rules and constraints is as follows: A constraint's scope is over a subset of the variables in the model. Therefore, a constraint is satisfied given a particular set of variable-value pairs (i.e., assignments of values to variables). On the other hand, each inference rule maps a logic expression x to a constraint c in the constraint library where each logic expression is defined over the features of the data points. Therefore, the inference rules act as a mapping of the features defining the data points to supports for constraints in the constraint library.

When a logical expression x (an inference rule) is asserted, a positive support is registered for the corresponding constraint c, otherwise c receives negative support. The inference framework supports mapping multiple rules, i.e. different logic expressions, to a single constraint. In Chapter 6, I show how this ability leads to higher confidence when inferring constraints. However, more general rules may lead to noisy supports for a constraint. Section 3.1.2 explains the process by which the framework deals with this noise.

IF B_1 and B_2 are on E/W-running street $\wedge$
(addr(B_1) and addr(B_2) are odd $\wedge$ B_1, B_2 are on N side of street)
THEN increment positive support of constraint 'Odd on North'
ELSE increment negative support of constraint 'Odd on North'

IF B_1 and B_2 are on the same street $\wedge$
(modulo_k(addr(B_1)) $-$ modulo_k(addr(B_2)) = blockNum(B_1) $-$ blockNum(B_2))
THEN increment positive support of constraint 'K-Block Numbering' where k=10
ELSE increment negative support of constraint 'K-Block Numbering' where k=10

IF B_1, B_2 are on E/W-running street $\wedge$
(addr(B_1)>addr(B_2) $\wedge$ latitude(B_1)>latitude(B_2))
THEN increment positive support of constraint 'Increasing North'
ELSE increment negative support of constraint 'Increasing North'

Figure 2.3: Three sample inference rules.

Figure 2.3 shows three sample rules in the BID problem domain for the constraints shown in Table 2.1. The first rule provides support for the "Odd on North" constraint that limits odd numbers to the North side of East/West running streets. The second rule provides support for the "K-Block Numbering" constraint where k=10, specifying the increment by which addresses across streets increase. Finally, the last rule provides support for the "Increasing North" constraint that ensures addresses increase to the North direction on North/South running streets. These rules represent a very small subset of the rules used in the BID problem domain. Appendix A.1 presents the full set of rules that map to all of the constraints in the BID problem constraint library.

Figure 2.4 shows three sample rules in the Sudoku puzzle domain for the constraints shown in Table 2.2. The first rule provides support for the "All-Diff Diagonal" constraint that ensures all numbers on the diagonals are different. The second rule provides support for the "Color Small" constraint that ensures all numbers for a given color are in [1,4]. Finally, the last rule ensures that the number in a colored cell is less than or equal to the numbered of colored cells in the given region (a characteristic of the Magic Sudoku puzzle). These rules represent a subset of the rules used for Sudoku puzzles. Appendix A.2 presents the full set of inference rules introduced by the Sudoku puzzle domain.

2.5 Inference Engine

The data points, inference rules, and the constraint library provide the framework with components that can be used together to infer the characteristics of a particular problem instance. However, a mechanism to evaluate the rules and select the constraints from the

IF P_1, P_2 are cells on the same diagonal $\wedge$ (Number(P_1)! =Number(P_2))
THEN increment positive support of constraint "All-Diff Diagonal"
ELSE increment negative support of constraint "All-Diff Diagonal"

IF P_1, P_2 are cells with the same color $\wedge$ (Number(P_1)$<$ 5 $\wedge$ Number(P_2)$<$ 5)
THEN increment positive support of constraint "Color Small"
ELSE increment negative support of constraint "Color Small"

IF P_1, P_2 are cells on a colored cell $\wedge$
(Number(P_1) <= #ColorsInRegion $\wedge$ Number(P_2) <= #ColorsInRegion)
THEN increment positive support of constraint "Magic"
ELSE increment negative support of constraint "Magic"

Figure 2.4: Three sample Sudoku inference rules.

library needs to be defined. The *inference engine* serves this purpose in the framework. As described below, its goal is to be general and its use domain independent.

As in a classical Expert System architecture, the inference engine is separated from the rules so that the inference framework can be applied across problem domains. The inference engine's responsibility lies in testing the inference rules. These tests determine which constraints should be selected from the library of constraints and the scope of the applicable constraints. The constraint-selection process (see Chapter 3) and the determination of scopes (see Chapter 4) are presented in detail in subsequent chapters.

The inference engine is central in taking the knowledge represented in the data points, the constraint library, and in the inference rules and combining it to produce the most representative constraint model for a specific problem instance. Furthermore, the information contained in the problem definition may be insufficient or sparse. For example, in the BID problem there may be a small number of data points available for the geographic area of interest or in a Sudoku puzzle few cells are initially filled in. For the

framework to be effective as a whole, it must contend with this lack of information. As such, the inference engine implements an iterative approach for constraint propagation (see Section 3.2) that augments the set of initial data points.

Additionally, when dealing with input data, the framework needs to be flexible enough to handle noise coming from the online data sources (when applicable) and noise introduced by the generation of incorrect supports from the data points. Towards this end, the inference engine uses *support levels* to deal with noise. A support level is a general formalism used to represent the framework's belief in the applicability of a constraint in the constraint library and it is formally defined in Section 3.1.2.

To be applicable in a large-scale application, the framework must efficiently and automatically generate constraint models. A bucketing approach (see Section 3.1.1) is used to separate the data points and improves the performance of the process. To automate the process, the inference engine uses generalized machine learning techniques to learn the scopes of the constraints (see Chapter 4). Finally, the engine uses a new XML schema to represent the generated constraint model that is passed onto a constraint solver (see Chapter 5).

2.6 General Framework Definition

Figure 2.5 informally defines the general inference framework. The generic representation of the concepts presented in this chapter allows the framework to be applied across various problem domains. When the framework is applied to a particular domain, a domain expert defines the features of the data points, inference rules and the constraints for

that domain. Even though the effort exerted by the domain expert is non-trivial, it pales in comparison to the effort required to manually defining constraint models for all foreseeable problem instances. An initial "setup" step is the only requirement of the expert and, when combined with the automated generation of constraint models by the framework, represents a significant reduction in the onus placed on the expert.

Given:

- a generic CSP model of a particular class of problems containing a set of generic constraints C_G applicable to all problem instances within the class,
- a set of data points $\{D_i\}$, where each data point D_i is described by a set of domain-expert defined features, and
- a library with a set of constraints C_L that may apply to a given instance but do not apply to all instances of this class,

a *Constraint-Inference Framework* is a set of rules $\{R_m\}$ along with an algorithm (i.e., inference engine) that operates on these rules. The rules map the features of D_i to the constraints $\{C_L\}$ of the library, indicating which constraints govern a problem instance.

Figure 2.5: General definition of the constraint-inference framework.

This definition aims to cover all of the requirements of CSPs (variables, domains, and constraints) and the algorithms used within the framework leverage existing techniques to limit the involvement of a domain expert. Using a generic CSP model allows for the use of constraint propagation to augment the input information for a problem instance. The combination of data points and a constraint library makes the automatic generation of constraint models for given problem instances feasible. More importantly, the general definition provided in Figure 2.5 allows for the automatic generation of constraint models *across* problem domains. Subsequent chapters describe algorithms that use the framework's components to select the applicable constraints (Chapter 3), find the scopes of

the inferred constraints (Chapter 4), and instantiate the constraint model (Chapter 5). Figure 2.6 illustrates the components of the general framework.

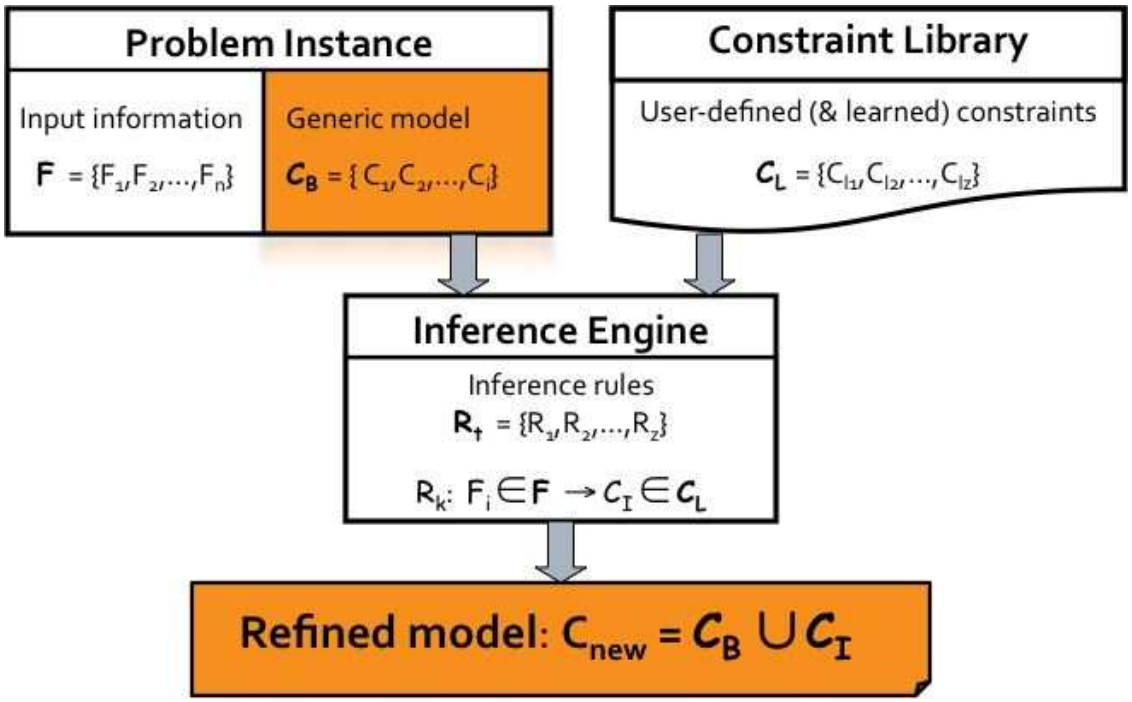

Figure 2.6: General architecture of the constraint-inference framwork.

2.7 End-to-End Model Generation Process

In this section, I outline the end-to-end process that uses the framework presented in this chapter to refine constraint models. This process is domain-independent and defines the general procedures used by the constraint-inference framework to generate a specialized model for a given problem instance. Each component in this process is presented in detail in subsequent chapters and the case study presented in Section 2.8 is used to clarify each concept. Additionally, examples from Sudoku puzzles help maintain the generality of this process.

Figure 2.7 illustrates the end-to-end model generation process. Generally speaking, the process of constructing instance-specific constraint models begins by taking data specific to the instance and inferring the set of applicable constraints. The *constraint-inference* process is outlined in detail in Chapter 3. The *Finding Scope* component is responsible for determining the scope of the applicable constraints by learning Support Vector Machine (SVM) models for the instance, the details of which are presented in Chapter 4. Finally, the inferred constraints along with their scopes are passed to a *model creation* component which instantiates a model of the problem instance. This instantiated model is then passed to a CSP solver that solves it. The process behind the instantiation of the inferred model is described in Chapter 5.

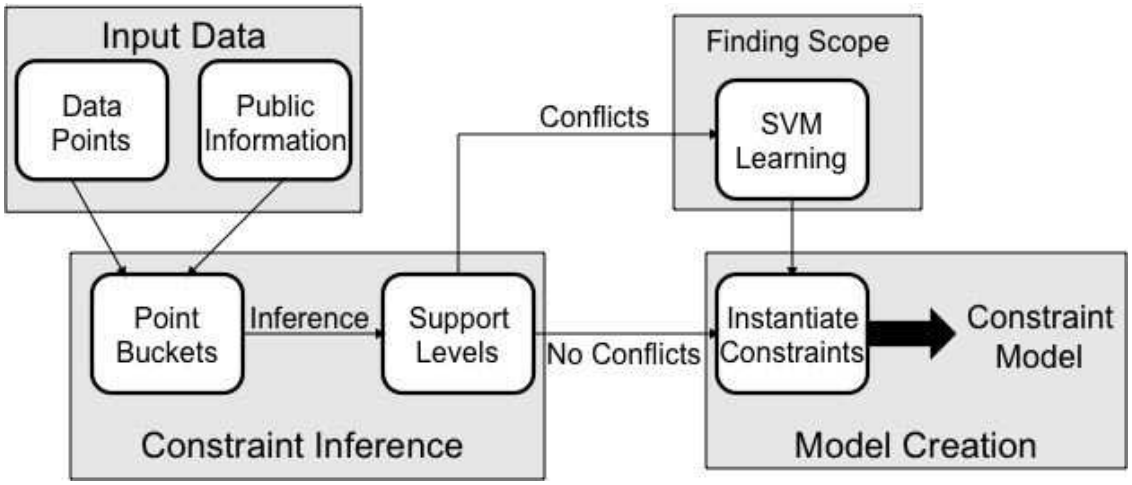

Figure 2.7: End-to-end inference process.

2.8 Case Study

In this section, I describe a specific instance of the BID problem for El Segundo California. I provide examples of data points used in this area and I define the set of addressing

characteristics that govern the city. I also outline the non-homogeneity of the area and I explain how the framework must contend with this irregularity. This specific problem instance is used in the following chapters of the thesis to clarify the process by which the inference framework refines representative constraint models.

To better illustrate the model generation process, consider the application of the inference framework to solving the BID problem for the city of El Segundo California, as shown in Figure 2.8. To extract information specific to this problem instance, the framework uses the coordinates that define the area's bounding box to retrieve the relevant vector data and phone-book entries. The vector data allows the framework to determine the set of potential street assignments for any given building, and the phone book provides a list of known addresses in the area. The data points for the area are obtained from an online gazetteer, which provides a small set of buildings identified by both an address and their latitude and longitude coordinates.

Figure 2.8: The area of El Segundo for which a model must be inferred.

The data points are defined using the features: address number, street name and orientation, side of street, block number, street ordering and latitude and longitude coordinates. The feature values for these points are either provided by the gazetteer or deduced from the vector data. Two example data points, representing the First Baptist Church and the Saint Anthony's School, are shown in Figure 2.9. The goal of the constraint inference framework is to infer the defining addressing constraints for this area from the full set of data points.

⟨*Name* | *AddressNum* | *StreetType* | *StreetName* | *StreetSide* | *Latitude* | *Longitude*⟩
First Baptist Church | 591 | EW | E Palm Ave | N | 33.925291 | -118.4100763
Saint Anthony's School | 233 | NS | Lomita St | W | 33.9183466 | -118.4084094

Figure 2.9: Two example data points in El Segundo.

The set of applicable constraints for El Segundo is illustrated in Figure 2.10. These constraints are characterized as either being applicable to the entire city or only applying to a geographical subsection. As shown in Figure 2.10, the "globally" applicable constraints are as follows: odd numbers are on the West side of North/South running streets and on the North side of East/West running streets. Secondly, the block numbering addressing scheme is present, enforcing that the addresses of buildings across city blocks increment by 100. Lastly, when traveling in the northern direction along North/South running streets, the addresses of buildings increase. Again, these constraints apply to all areas of El Segundo.

Two additional addressing constraints govern this area. The addresses of buildings get bigger when you travel to the West along East/West running streets. However, this constraint only applies to buildings that are located to the West of Main Street.

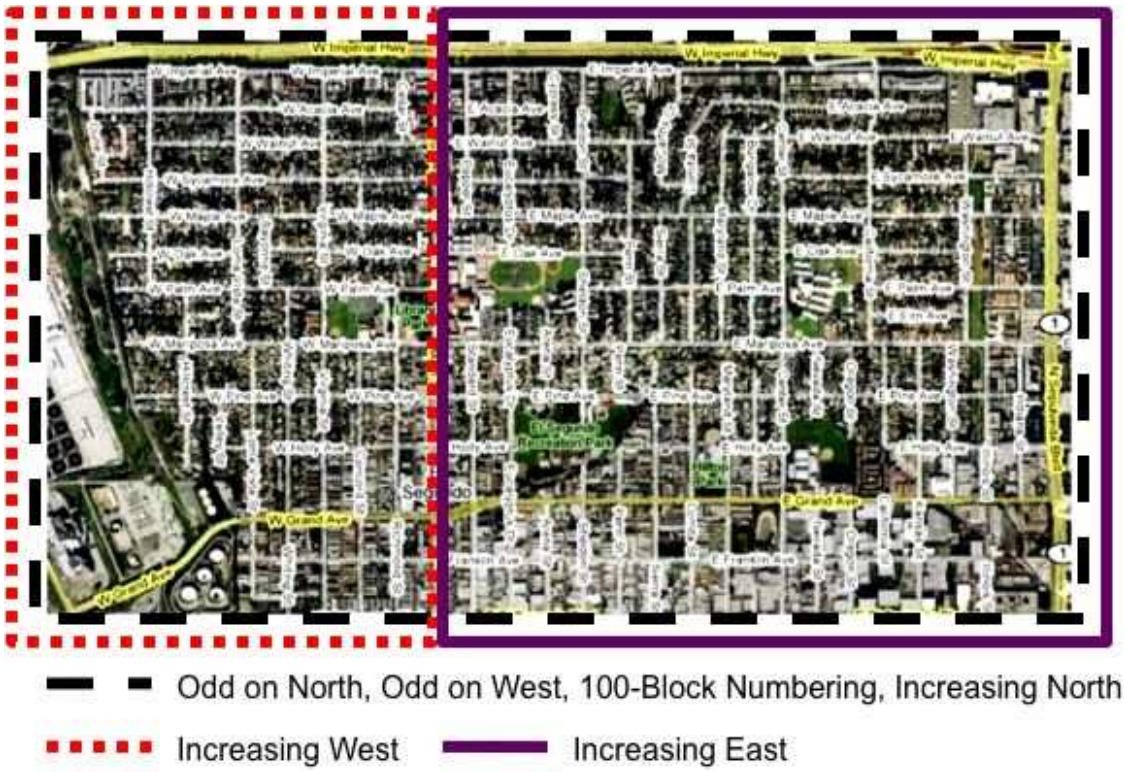

Figure 2.10: The coverage of the applicable constraints in El Segundo.

For all buildings located East of Main Street, the addresses increase when you travel East along East/West running streets. Therefore, the set of buildings to which either of these constraints apply is defined by a building's geographic location relative to the North/South running Main Street. These subsets of buildings define the *scope* of these two constraints and are highlighted in Figure 2.10.

The constraint inference framework must infer the applicable addressing constraints (along with their scopes) that represent the addressing characteristics described above. Specifically, the framework uses buildings with known addresses (data points), such as those found in the USGS gazetteer and shown in Figure 2.9, to establish the relationships that occur between the buildings. For example, it discovers that when comparing two

address on the North side of East/West running they are both odd. The discovery of such relationships signifies that the *Odd on North* constraint applies to this area and the framework infers its applicability over all buildings on East/West running streets. This process continues until all of the applicable addressing constraints and their respective scopes are inferred, and a constraint model corresponding to the addressing characteristics illustrated in Figure 2.10 is constructed.

The variables in the constraint model represent the buildings in the area. Each building has a corresponding variable to which a value (an address) must be assigned. The domain of these variables is determined using the vector data and a building's position on the map. Finally, the constraints correspond to the set of applicable constraints inferred by the framework. These constraints are instantiated such that they cover the buildings within each constraint's scope. For example, the *Increasing West* constraint is instantiated over all buildings on East/West running streets west of Main Street, such as W Pine Avenue, but not over any buildings to the east of Main Street, such as those on E Imperial Avenue. Similarly, the *Odd on North* constraint is instantiated over all buildings along East/West running streets in the area. Once the instantiated model is generated, it can be passed to the CSP solver so that addresses are assigned to buildings and a viable solution(s) is produced.

Chapter 3

Selecting Constraints

In this chapter, I introduce the algorithm used to select the applicable constraints for a given problem instance. Each component of this algorithm is motivated and detailed. The BID problem instance presented in Section 2.8 is used help ground the techniques used and further examples from the Sudoku puzzle domain are used to demonstrate the generality of these techniques but are presented in less detail. The chapter concludes by outlining how constraint propagation can be used to augment the input information of a problem instance.

3.1 Constraint Inference Algorithm

In this section I describe the algorithm used to select the applicable constraints from the constraint library for a given problem instance. I begin by presenting the algorithm in its entirety and outlining the flow of execution. Subsequent subsections delve deeper into the function of each component of the algorithm, providing detailed examples for clarification. The BID problem in the El Segundo area serves as the primary example and Sudoku puzzles provide a secondary example used to highlight the generality of the

techniques. I conclude the section by presenting an optional algorithm that can be used to augment the input information in certain domains.

```
CONSTRAINT-INFERENCE(D, finalSet)
 1  finalSet ← {}
 2  constraints ← constraintLibrary
 3  buckets ← CREATEBUCKETS(D)
 4  for i ← 0 to size[buckets]
 5        do B ← buckets[i]
 6           constraints ← EVALUATERULES(B)
 7  for i ← 0 to size[constraints]
 8        do C ← constraints[i]
 9           if POSSUPPORT(C) > NEGSUPPORT(C)
10              then finalSet ← finalSet ⋃ C
```

Figure 3.1: Constraint inference algorithm.

Figure 3.1 presents the algorithm used to infer the applicable constraints. The algorithm begins with a library of constraints relevant to the problem domain, an empty set of inferred constraints and a set of data points. The CREATEBUCKETS procedure, outlined in Section 3.1.1, divides the data points into buckets based on their feature values, resulting in a more efficient execution of the framework for instances that contain a large set of data points. Once the data points have been divided, the framework is ready to find the relationships that exist between the points.

These relationships are extracted by the EVALUATERULES procedure, analyzed in Section 3.1.2. This procedure evaluates the relevant inference rules using data points in each bucket to provide supports for the constraints in the library. The framework determines the applicability of constraints by studying the positive and negative supports generated for each constraint. Section 3.1.3 details the procedure used to make this determination. The algorithm returns the set of constraints it has classified as applicable.

The execution of this algorithm is domain independent. It assumes a set of data points is available and a domain expert has previously populated the constraint library and defined inference rules that map to these constraints. The data points are provided by an online source such as a gazetteer in the case of the BID problem for El Segundo or they are created from the initial problem definition such as the filled-in cells of a Sudoku puzzle. However, none of the procedures use techniques that are only applicable in a certain domain and as such make the algorithm applicable to all problem domains within the class of solvable CSPs.

3.1.1 Bucket Creation

The ability to infer applicable constraints for a given problem instance largely depends on how well the framework can identify relationships between data points and map these relationships to constraints in the library. However, only certain pairs of data points provide useful information about characteristics that hold for the problem instance. Take the BID problem in El Segundo as an example. Two buildings along the East/West running streets W Imperial Avenue and W Pine Avenue, when compared to each other, do not provide support for the direction in which addresses increase along the North/South running Sierra Street.

One approach to evaluating the inference rules is to do a pairwise comparison of all the data points. For instances where the number of data points is relatively small, this is an acceptable approach. However, to improve the robustness and applicability of the inference framework, the comparison of data points should be efficient and scale to large sets of points. The general technique used in the framework is to divide the full set of data

points into "relevant" subsets where the comparison of data points within each subset provides pertinent information that can be used to generate support for constraints.

```
CREATE-BUCKETS(D)
 1  buckets ← {}
 2  for each point d ∈ D
 3        do
 4           for each featureValue of d
 5                 do
 6                    if ∃b ∈ buckets where b is defined by featureValue
 7                       then b ← d
 8                       else
 9                              create bucket newB defined by featureValue
10                              buckets ← newB
11                              newB ← d
12  for each bucket b ∈ buckets
13        do
14           if |b| = 1
15              then remove b from buckets
16  return buckets
```

Figure 3.2: The algorithm used to create buckets of data points.

I developed a bucketing algorithm (represented as CREATEBUCKETS in Figure 3.1) to divide the data points into meaningful subsets. This algorithm is presented in Figure 3.2 and works as follows. The framework looks at the feature values for each data point. Given a feature-value pair, for example *street = E Grand Ave*, it checks if a bucket defined by this feature-value pair exists in the set of previously created buckets. If such a bucket exists, then the given data point is added to it. Otherwise a new bucket, defined by the feature-value pair, is created and the data point is added to this new bucket. The algorithm continues until it has analyzed all of the data points. Finally, it then removes all buckets with only one data point in them from the set of constructed buckets and

returns this reduced set. The rational behind this removal is that data point comparisons are binary and as such buckets with a single point are not useful.

This algorithm is similar in nature to the bucket approach used for data integration [41]. The policy used to create the buckets is outlined above and supports any transformation on a feature value. However, the current implementation of the framework only contains rules that require a feature value of two points to be the same and as such all buckets contain data points with a common feature value. In the BID problem, this means that buckets are created so they contain data points that are on the same street (i.e., E Grand Avenue), the same side of the street (i.e., South side), or some other common feature value (for a full set of rules, please see Appendix A.1). The bucket creation policy is linear in the set of all possible feature values.

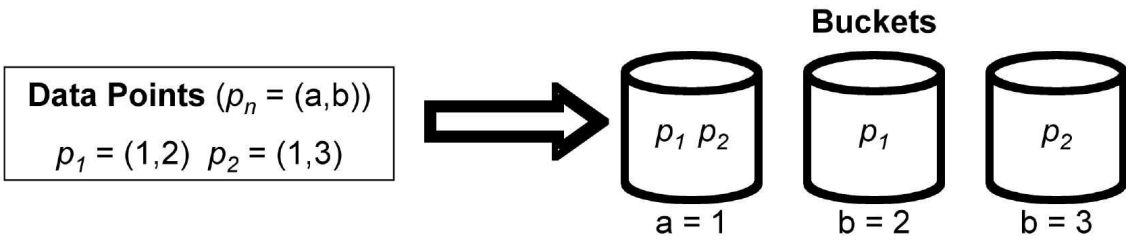

Figure 3.3: Bucketing algorithm example.

A more concrete example of the CreateBuckets algorithm is illustrated in Figure 3.3 and works as follows: As previously stated, the data points are placed into buckets that correspond to unique feature values, and a single data point may appear in multiple buckets. For example, consider two data points p_1 and p_2 defined by features (a,b) where $p_1(a,b)$=(1,2) and $p_2(a,b)$=(1,3). In this case, three buckets are created: (a=1)=$\{p_1,p_2\}$, (b=2)=$\{p_1\}$, and (b=3)=$\{p_2\}$.

Church of Christ | 717 | EW | E Grand Ave | N | 33.9197355 | -118.4081317
Kingdom Hall | 608 | EW | E Grand Ave | S | 33.91918 | -118.4097984
Imperial School | 540 | EW | E Imperial Ave | S | 33.9302909 | -118.4106319

Figure 3.4: Subset of data points in El Segundo.

More specifically, consider the subset of data points for El Segundo seen in Figure 3.4. Some example buckets created by the CreateBuckets procedure would be: a bucket with all points on *E Grand Ave* would contain the *Church of Christ* and the *Kingdom Hall* points, a bucket for all East/West running streets (*EW* feature value) would contain all three points, and a bucket for points on the North side of a street (*N* feature value) would contain only the *Church of Christ* data point. Obviously when applied to the full set of data points for El Segundo, the number and the size of buckets would increase.

To improve the efficiency of the selection algorithm, the inference rules are evaluated on every combination of data points within a bucket to provide support for a given constraint (the algorithmic details for rule evaluation are presented in Section 3.1.2). This approach eliminates a pair-wise comparison of *all* data points, rather limiting the comparison to only data points within a bucket. The reduction in the number of data point comparisons dramatically improves the algorithm's runtime, although this reduction would be less drastic given less restrictive buckets, such as those defined using the $\leq$ or $\geq$ operators.

Without this approach, the time complexity of data point comparison is $O(k2^n)$ where k is the number of inference rules and n is the number of data points. However, the bucketing algorithm described in this section greatly reduces the chances of reaching this worst case scenario as it is highly unlikely all data points would be placed in each

bucket. In practice, each bucket rarely contains more then 75% of the data points and on average contains roughly 50% of the points. Additionally, the creation of buckets is a linear process in the number of feature values. The benefits of this approach are best seen in the evaluation of the inference framework on the El Segundo region with 1650 gazetteer points (see Section 6.3.1) where 69 relevant buckets are created from ∼1720 feature values.

Applying the bucketing technique in the Sudoku puzzle domain also tackles the problem where not all data-point comparisons provide useful information for each constraint in the library. For example, two points not on the same row do not provide any insight into whether the *AllDiff Row* constraint (enforcing that all numbers in a row are different) is applicable to a particular puzzle. Therefore, buckets are created such that each one contains all the data points in the same row, the same column, and based on some other common feature values. Subsequently, only buckets containing data points in the same row would be used to generate support for a *All-Diff Row* constraint. The creation of buckets for Sudoku puzzles and all problem domains to which the constraint-inference framework is applied is accomplished using the general bucket-creation algorithm described above.

3.1.2 Evaluating Inference Rules

The ability to correctly infer constraints relies on the extraction of support for constraints from the input data. This support is generated by evaluating the set of inference rules using a given bucket's data points. This support, expressed as a *support level*, is used to determine the applicability of the constraints in the library. In this section, I describe

how inference rules are evaluated and the techniques used to produce the highest possible levels of support. Section 3.1.3 defines the term 'support level' and describes the process by which constraints are deemed applicable.

As described in Section 2.4, the role of an inference rule is to provide support for a particular constraint in the library. The rule is a logical expression over the features of the data points. Because the features describing the data points do not map directly to variables in the constraint model, the inference rules act as a bridge between the relationships present in the *feature* space and the constraints in the *constraint* space. Each rule represents a condition that must be asserted for a particular characteristic (constraint) to hold in a given problem instance.

The process by which supports are generated is as follows. Once buckets have been created and filled, the data points within each bucket are used to evaluate the inference rules (EvaluateRules in Figure 3.1). This algorithm is outlined in Figure 3.5 and works as follows. First, all of the relevant inference rules are selected for a given bucket. A relevant inference rule is one that has an equality requirement involving the feature defining the given bucket. In the BID problem for El Segundo, this means that the bucket defined by the feature *streetType* and the value *E/W* (i.e., the bucket containing all data points that lie on an East/West running street) is used to evaluate the rules that provide support for the *Odd on North* constraint in the constraint library. These rules all require that two data points being compared be on the same *streetType*. The *Odd on North* constraint limits odd numbers to the North side of E/W running streets and, as such, a precondition for all inference rules that map to this constraint requires data points to be

on an East/West running street. Therefore, it makes sense to generate support for this constraint by only comparing data points from the E/W running streets bucket.

```
EvaluateRules(B)
 1  supportedConstraints ← {}
 2  for each rule r ∈ inferenceRules
 3       do
 4          if r contains an equality condition for the feature defining B
 5            then for all pairs of points pp ∈ B
 6                    if pp satisfies r
 7                       then increase positive support for consMappedTo(r, c)
 8                       else  increase negative support for consMappedTo(r, c)
 9                    supportConstraints ← c
10  return supportedConstraints
```

Figure 3.5: The algorithm used to evaluate inference rules for a given bucket.

Given a particular bucket and a set of relevant inference rules, the framework generates support for a subset of the constraints in the library. As previously stated, each inference rule maps a logic expression x to a constraint c in the constraint library, and multiple rules, i.e. different logic expressions, can map to a single constraint. When an expression x is asserted, a positive support is registered for the corresponding constraint c, else c receives negative support. Because only relevant data points are used to evaluate the rules, non-assertion of a rule implies negative support for the corresponding constraint.

For example, if two buildings on an East/West running street are on the North side and both have an odd address, the *Odd on North* rules are asserted and these two points provide positive support for the *Odd on North* constraint. On the other hand, if the two data points have even addresses the 'else' clause of the rules is executed and negative support is provided for the *Odd on North* constraint. The algorithm finishes by returning

a set of constraints along with their supports (positive, negative, or a mix of the two) as generated by the points in the given bucket.

To generate a comprehensive set of supports, the framework must maintain a large set of inference rules. In allowing any number of inference rules to support a given constraint, rules ranging from highly specific to general can be used by the framework. By increasing the number of rules, we are also increasing the number of supports generated per constraint. To maximize the number of supports provided, I generated an exhaustive set of rules for each constraint in the library by finding all possible scenarios where data points could support a given constraint. The full rule sets for both the BID problem and Sudoku puzzles are presented in Appendix A.

IF B_1 and B_2 are on E/W-running street $\wedge$
(addr(B_1) and addr(B_2) are odd $\wedge$ B_1, B_2 are on N side of street)
THEN increment positive support of constraint 'Odd on North'
ELSE increment negative support of constraint 'Odd on North'

IF B_1 and B_2 are on E/W-running street $\wedge$
(B_1 and B_2 are on the same street $\wedge$ addr(B_1) and addr(B_2) are even
$\wedge$ B_1, B_2 are on S side of street)
THEN increment positive support of constraint 'Odd on North'
ELSE increment negative support of constraint 'Odd on North'

IF B_1 and B_2 are on E/W-running street $\wedge$
(addr(B_1) is odd $\wedge$ addr(B_2) is even $\wedge$ B_1, is on N side of street)
THEN increment positive support of constraint 'Odd on North'
ELSE increment negative support of constraint 'Odd on North'

IF B_1 and B_2 are on E/W-running street $\wedge$
(addr(B_1) is even $\wedge$ addr(B_2) is odd $\wedge$ B_1, is on S side of street)
THEN increment positive support of constraint 'Odd on North'
ELSE increment negative support of constraint 'Odd on North'

Figure 3.6: Sample BID problem inference rules providing support for *Odd on North*.

Figure 3.6 shows four sample rules in the BID problem domain. All of these rules provide support for the *Odd on North* constraint and we can see that the specificity of these rules varies. For example, the first rule is more general then the second in that it compares two points on a E/W running street while the second rule only compares these points if they are also on the same street. The additional support generated by the larger set of inference rules increases the confidence in the results of the inference mechanism. However, as I discuss in the following section, the generality of some rules can lead to incorrect positive support for certain constraints. Therefore when determining the applicability of constraints, the framework needs to contend with false positive supports.

3.1.3 Determining Constraint Applicability

After comparing all data points within each bucket, the inference engine has a set of supports, both positive and negative, for constraints in the library. Constraints with no supports may exist, and are discussed later in this section. Before the applicable set of constraints can be determined, we need to establish a metric that quantifies the applicability of these constraints. The metric used by the framework is *support level*.

A constraint's support level is a single number corresponding to a function of its positive and negative supports. A support level of 1 implies that all of the supports for the given constraint are positive, -1 implies they are all negative and 0 means there is an equal distribution of both. This metric allows the framework to describe the ratio of the positive to negative constraints as a single number that can be used to establish the applicability of the constraint. Note that the range of support levels is not restricted to {-1,0,1} but may take any rational number in the range $[-1,1]$. Because multiple rules

can provide a positive or negative support to a given constraint, we can use any ratio of positive to negative support to determine the applicability of the constraint.

After evaluating the rules, each constraint is classified as *applicable*, *non-applicable* or *unknown* based on their support level. The set of inferred constraints ($finalSet$ in Figure 3.1) is composed of all constraints classified as applicable (even those with negative support) and constraints with no support are classified as unknown. My initial work with support levels required a constraint to have a support level of 1 to be classified as applicable. While this stringent requirement helped establish the feasibility of the framework, it is not practical due to noise encountered in the problem.

The noise that the inference framework must contend with manifests itself in two forms: (1) noise generated by the data points themselves, and (2) noisy supports generated by the evaluation of the rules. In cases where the framework uses publicly available online sources to gather data points, there is no guarantee that all the points are accurate. In the BID problem, there are many online sources that provide point data. However, not all sources are maintained by institutions such as the government and most sources do not adhere to a 100% accuracy standard. For example, buildings with incorrect addresses or latitude and longitude coordinates are not uncommon in data sources that cover areas with new construction. Specific to El Segundo, Recreation Park, which spans two city blocks, is identified with a single address in the gazetteer. The data points with incorrect feature values, when included in the input information, may lead to noisy supports for certain constraints.

Furthermore, as I have shown in Section 3.1.2 and as seen in Appendix A, within a set of inference rules that provide support for a particular constraint, some rules are

more general than others. This generality can lead to false positive supports for a given constraint. Take for example two rules that provide support for the *Block Numbering* constraint in the BID problem. This constraint specifies that the addresses across city blocks must increment by some fixed factor (in El Segundo, the 100-block numbering constraint is enforced). If one rule requires the data points being compared must lie on the same street and the other more general rule does not, the more general rule can provide positive support for this constraint when in fact it does not apply to the area. This scenario is depicted in Figure 3.7 where the comparison of *Building 2* and *Building 4* leads to a false positive support for the *Block Numbering* constraint.

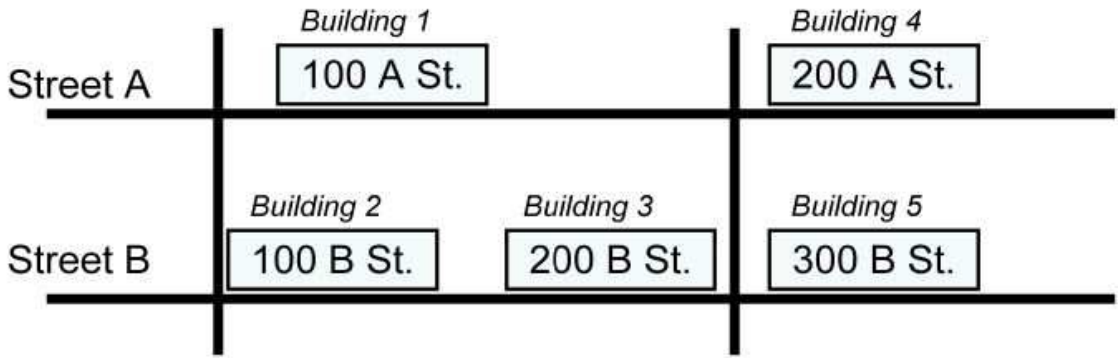

Figure 3.7: Example scenario where false support is provided for *Block-numbering*.

Relaxing the requirement that an applicable constraint have a support level of 1 (all support must be positive) is the approach used in the framework. Specifically, a support level of greater than 0, meaning there is a higher ratio of positive to negative support, classifies a constraint as applicable. Such an approach contends with both forms of noise while still maintaining the accuracy of the framework when inferring the applicable constraints for a given problem instance. This claim is validated by the experimental

results shown in Section 6.3.1. As previously mentioned, any support level can be used to classify constraints and this ratio can be adjusted as required by the problem domain.

To avoid classifying constraints as unknown, the framework generates as many supports as possible. This is done by defining as complete of a set of inference rules as possible for each constraint (see Section 3.1.2). However, if the input information is very sparse, it may be impossible to generate supports for all constraints. In these cases, the framework does not add the constraints with no support to the inferred constraint model. The contention being made is that it is better to return an under-constrained model that leads to a solution(s) rather than inferring an overly-constrained model that is not solvable and requires backtracking.

After determining the applicability of the constraints, we are left with a set of constraints that apply *somewhere* within the problem instance. We refer to the variables over which constraints apply as a constraint's scope. Determining the scope of the applicable constraints is an important step in model generation because without it we would be unable to solve BID problem instances such as those in El Segundo where address numbers do not increase monotonically (see Section 2.8). The need to find a constraint's scope is not specific to the BID problem and it also arises in additional domains such as Sudoku. Chapter 4 describes the process by which the scope of all applicable constraints is learned.

3.2 Augmenting Data Points Using Constraint Propagation

Data-point distribution within a problem instance affects the inference capabilities of the framework. No matter how exhaustive the set of inference rules and how encompassing the set of constraints in the library, without sufficient information in the input data an accurate model of a problem instance is very difficult to infer. One possible solution to the lack of input information is to augment the set of data points. This can be achieved by propagating the constraints in the generic model to induce new data points. Because the constraints in the generic model are monotonic and apply to all problem instances, applying them to the initial set of data points can derive new information not explicit in the problem definition.

The procedure described in this section is an optional component in the constraint-selection process. When a problem instance contains a small number of variables, such as a 9×9 Sudoku puzzle, the number of newly inferred data points can be significant. However, when assigning addresses to buildings in an area such as El Segundo that contains over 1650 buildings, the benefits of this procedure are greatly reduced. The procedure works well for instances where the initial data points represent a substantial percentage of the variables in the constraint model or when constraints are tight, as seen in the Sudoku puzzle domain. Therefore, the evaluation of the constraint propagation approach is only carried out on Sudoku puzzles to show its effectiveness for such domains (see Section 6.5).

The intuition behind this approach lies in leveraging the implicit relationships between variables in the constraint model to reduce their domains to a single value. A data point,

```
INFERENCE-PROPAGATION(C, F)
 1  c ← C, f ←data points in F
 2  propStrategies ← propagation methods with varying strength
 3  while propStrategies ≠ {}
 4        do
 5           s ← POP(propStrategies)
 6           newF ← PROPAGATE-CONSTRAINTS(c, f, s)
 7           if newF ≠ {}
 8              then f ← f ⋃ newF
 9              else  return f
10  return f
```

Figure 3.8: Iterative algorithm to find new data points.

as defined in Section 2.2, corresponds to a variable-value pair where the variable has a single value in its domain. To implement the idea of data point augmentation, I use the iterative propagation algorithm shown in Figure 3.8. The algorithm takes as input the problem model for the given instance, made up of the generic constraints C and the initial features F (data points) of the problem. It then propagates the generic constraints over the current set of data points (PROPAGATE-CONSTRAINTS in Figure 3.8) using different constraint propagation methods that vary in their strength of propagation, to infer new data points. If a variable's domain is reduced to a single value, this variable represents a newly inferred data point and is added to the set of returned data points. The algorithm ends when the process reaches quiescence.

It should be noted that this algorithm may require backtracking during the iterative process. If a new data point inference leads to the annihilation of another variable's domain (the domain is reduced to the empty set), then backtracking is required to eliminate the erroneously inferred data point. This phenomenon occurs when there is noise in the

initial problem definition, sometimes occurring when inaccurate data points are obtained from an online source. Therefore, this algorithm must keep track of which data points were added at each time step. An alternate solution to avoid backtracking is to validate the input information before beginning the propagation procedure.

To keep this procedure tractable, the propagation strategies need to be quick and relatively inexpensive. Therefore, I use the following sequence of propagation methods: Arc-Consistency (AC) [46], followed by Generalized Arc-Consistency (GAC) [49], ending with Singleton Arc-Consistency (SAC) [18]. Formally, a variable x_i is arc-consistent (AC) with another variable x_j if, for every value a in the domain of x_i there exists a value b in the domain of x_j such that (a,b) satisfies the binary constraint between x_i and x_j. A problem is arc consistent if every variable is arc consistent with any other one. GAC takes propagation a step further and supports non-binary constraints while SAC is strongest of the three by fixing a single value a to a particular variable x_i and determining if the problem is arc-consistent given this assignment.

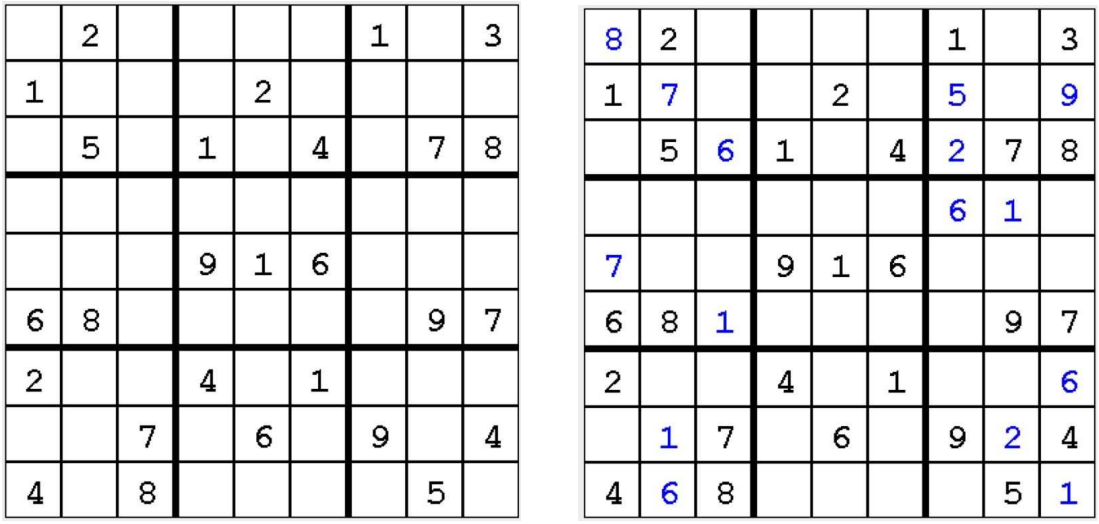

Figure 3.9: Before and after constraint propagation using SAC for a Sudoku puzzle.

These algorithms are widely used in the CP community and provide a quick yet effective method to propagate the generic constraints. Stronger propagation schemes could be used but the goal of the constraint propagation algorithm is to be a quick preprocessing step before the inference of constraints begins. As previously mentioned, as the number of variables grows and the data points no longer represent a significant percentage of these variables, the benefit of this procedure diminishes. Stronger propagation schemes may help in these cases but the overhead imposed by these schemes is substantial.

As an example, consider the Sudoku puzzle shown on the left in Figure 3.9. This is a basic Sudoku puzzle whose characteristics do not include an all-different constraint on the diagonals. However, looking at the problem definition, we would infer the applicability of the all-different constraint on the diagonals since the filled-in cells provide only positive support for this constraint. The puzzle shown on the right in Figure 3.9 is the original puzzle after applying the propagation algorithm described in this section using SAC. New data points (shown in blue) are inferred and interestingly there are three cells filled-in with the number *1* on the diagonals. These points provide negative support for the all-different diagonal constraint and this example shows that a richer set of data points improves the inference ability of the framework. In fact, in the Sudoku puzzle domain any negative support for a constraints classifies the constraint as *non-applicable*.

Chapter 4

Using Support Vector Machines to Learn the Scope of Constraints

The heterogeneity of constraints within a given problem instance requires both the inference of applicable constraints and the determination of an inferred constraint's scope. Take the BID problem in El Segundo as an example. In the constraint model for this area, there are two constraints present that enforce the direction in which addresses increase for East/West running streets. As illustrated in Figure 4.1, addresses to the west of Main Street increase to the West and addresses to the East of Main Street increase to the East. These are contradictory constraints and when applied to all buildings in the area, result in an unsolvable constraint model.

This problem is not unique to El Segundo but occurs in other areas for the BID problem. Consider the part of Belgrade Serbia shown in Figure 4.2, where the odd addresses are on the west side of the street and elsewhere in the area they are on the east side of the street. In Figure 4.2, one scope covers buildings within the circle and the second all other buildings. Again, these are conflicting constraints and when applied universally in the area, lead to an inconsistent and unsolvable constraint model. This

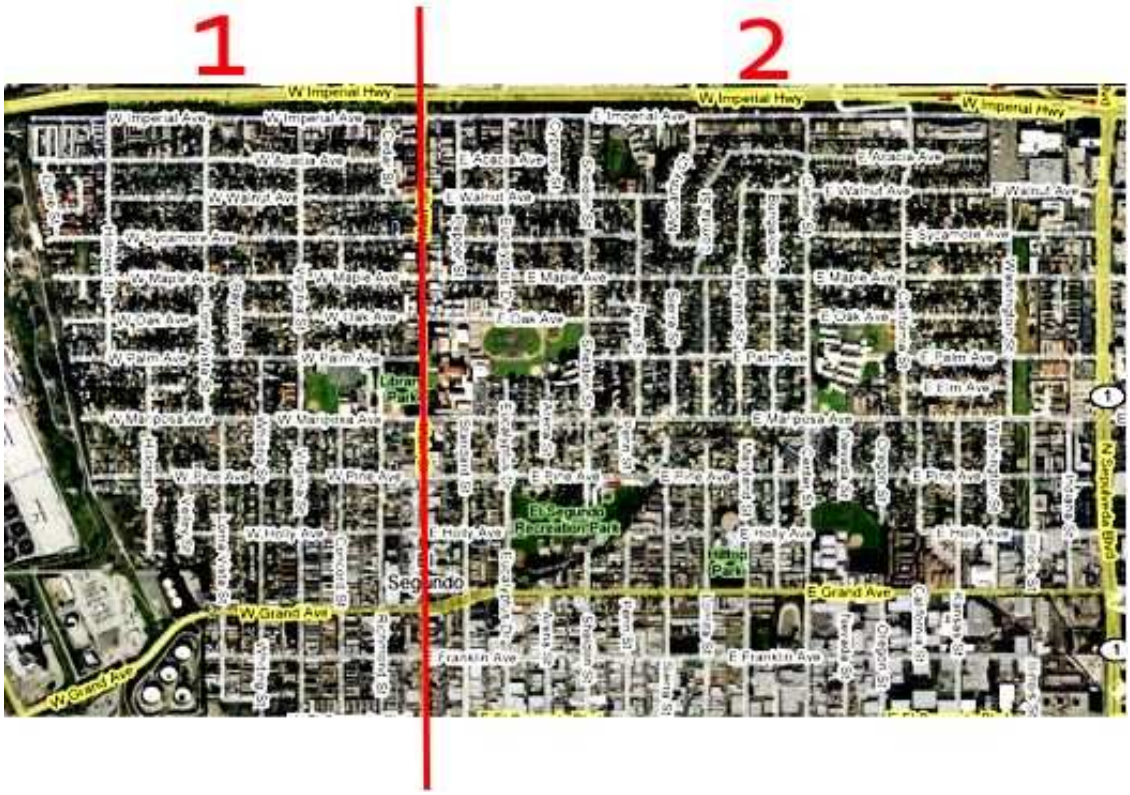

Figure 4.1: The scopes of the *Increasing East* and *West* constraints in El Segundo.

example reiterates the claim that the set of variables over which each constraint applies (defined as its *scope*) must be inferred along with the set of applicable constraints.

Determining if any set of two constraints apply over distinct sets of variables in a problem instance is a nontrivial task. One approach to this problem is to use semantic information specific to a problem domain. In El Segundo, addresses along East/West running streets change the direction in which they are increasing when the directional of a street changes (e.g., West Imperial Avenue becoming East Imperial Avenue). This information can be exploited when determining the scope of constraints but, obviously, it is domain-specific and requires interaction with a domain expert. Additionally, this information is instance-specific as it cannot be used for the area of Belgrade presented in

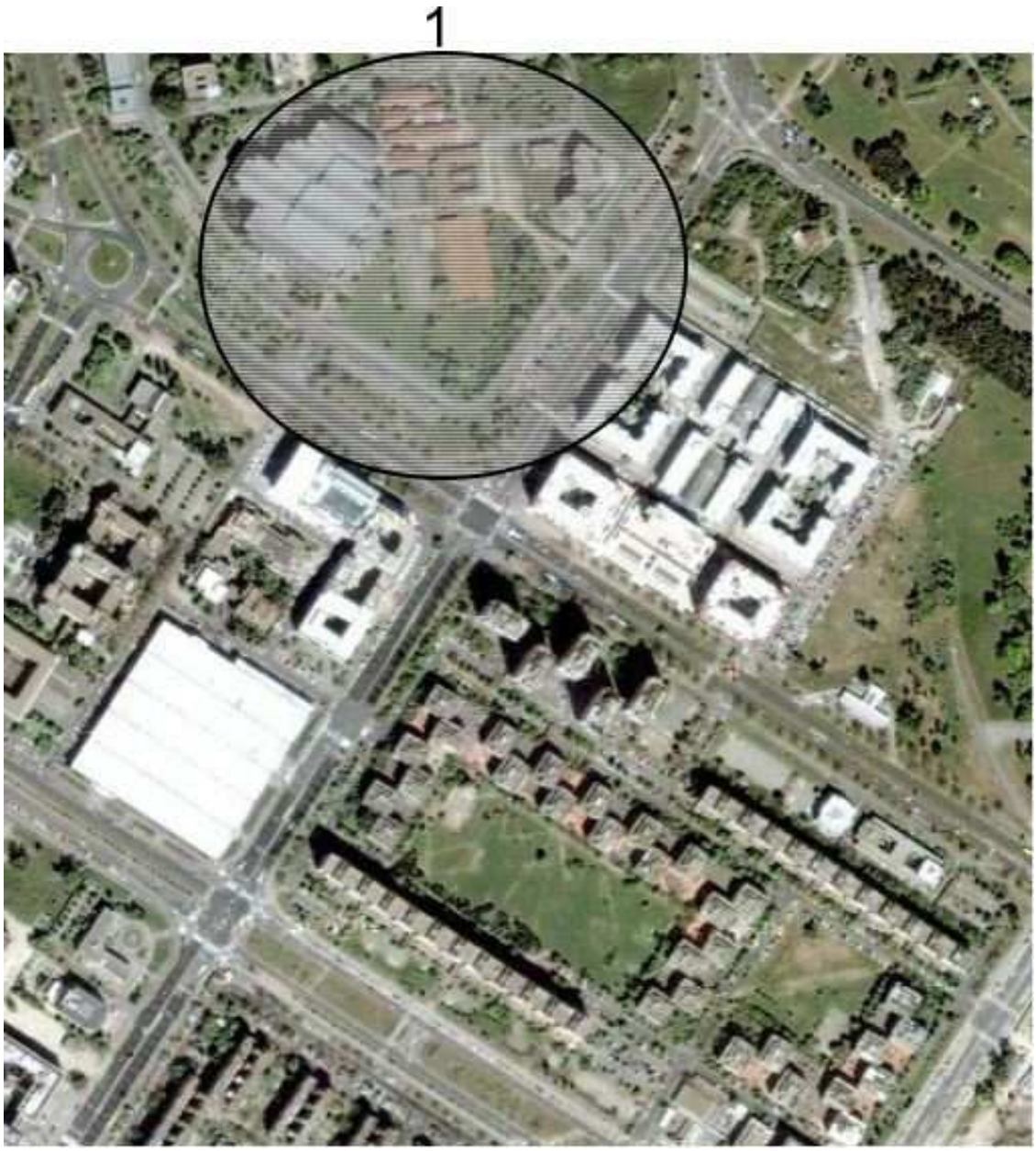

Figure 4.2: The two scopes of the *Parity* constraints in Belgrade.

Figure 4.2. The generality of the constraint-inference framework depends on an effective and domain-independent method for finding constraint scope, one that maintains a high level of automation and can be applied to any problem instance.

In this chapter, I present a new domain-independent approach that uses Support Vector Machines (SVMs) [64] to automatically learn the scopes of conflicting constraints such as those seen in El Segundo. In the end-to-end constraint inference process seen in Figure 2.7, this approach corresponds to the *Finding Scopes* component. I begin this chapter by providing an overview of SVMs and describing the algorithm used by the inference framework in Section 4.1. Section 4.2 outlines the method used to train the SVM models for a given problem instance. Finally, Section 4.3 explains how scopes are determined given the learned SVM models. To maintain continuity in this chapter, I will illustrate the concepts using the BID problem example for El Segundo but as previously demonstrated (see Figure 4.2) and as introduced in the evaluation section (see Chapter 6), conflicting constraints exist in domains such as Sudoku puzzles and others.

4.1 Support Vector Machines

SVMs are a set of methods for supervised machine learning used for classification. They map input vectors to a higher dimensional space where a maximal separating hyperplane is constructed and where the attributes of each input vector are part of the feature space. Given a set of training examples (input vectors), a set of support vectors (the SVM model) is learned and used to classify new vectors into one of two classes. This process defines the linear version of binary SVMs. The main advantage of using SVMs over other classifiers is that they minimize the empirical classification error and maximize the margin. The theory behind this method of classification is that a larger margin leads to

a better (lower) generalization error of the classifier. Figure 4.3 graphically represents a learned separating hyperplane with two parallel hyperplanes that maximize the margin.

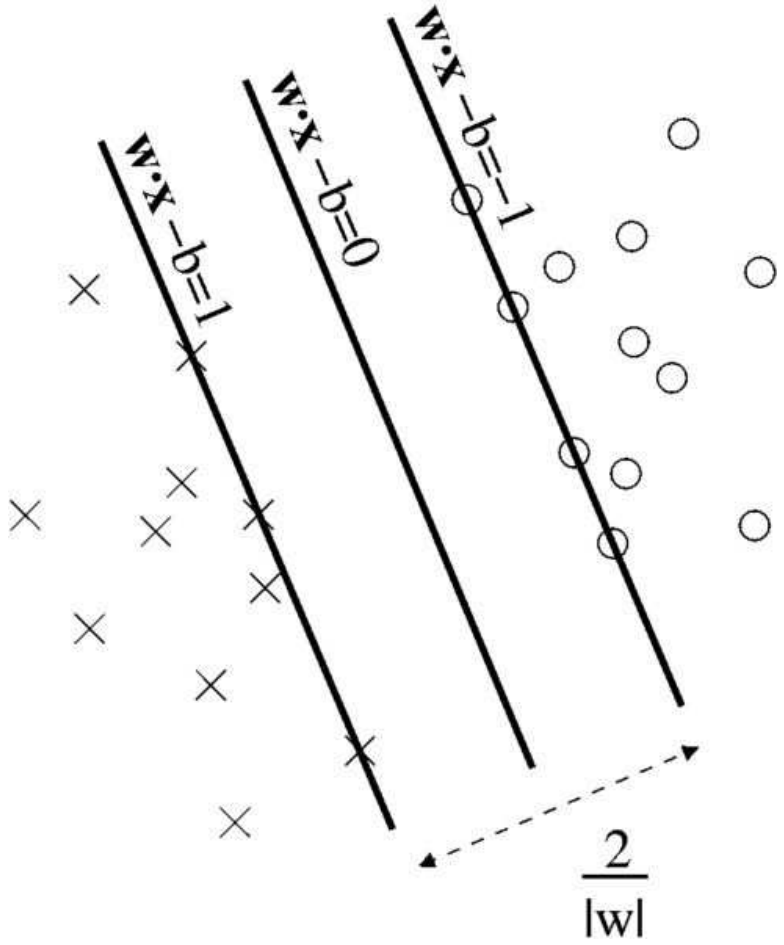

Figure 4.3: Separating hyperplanes learned by SVMs.

Non-linear versions of SVMs can be created by replacing the distance function used to create the hyperplanes with a non-linear kernel function. The non-linear kernel performs a non-linear transformation on the input space to a higher dimensional feature space. Subsequently, the hyperplanes learned in this higher dimensional space are non-linear in the original input space. Commonly used kernels are: polynomial, radial, gaussian,

and sigmoid. In this dissertation, I apply the linear and radial kernels to the process of learning the scope of constraints. Section 6.4 discusses the advantages of each kernel.

As with any classification task, the choice of which learning method to use is dictated by the nature of problems being solved. For the BID problem, I chose SVMs over other machine learning methods (e.g., neural networks and vector spaces) because they best handle the geospatial features of the problem. My intention is to represent the spatial characteristics of the problem (i.e., the location of buildings) in the variable space. SVM's ability to learn class boundaries with separating hyperplanes accomplishes this goal. This ability is also required when finding the scope of constraints for Samurai Sudoku puzzles (see Figure 1.2) and is present in the class of CSPs I study in my research.

I use support vectors to approximate the boundaries of the scopes I am trying to learn and I use predefined sets of conflicting constraints as class labels. The algorithm works as follows: Given a set of inferred constraints, the framework must determine if a conflict exists between them. To help the framework find conflicts, a domain expert provides a set of conflicting constraint types when defining the constraints in the library. I require the user to define this set of conflicting constraints because automatically identifying conflicting constraints is exponential in the number of constraints in the constraint library. Additionally, automatically determining if pairs of constraints are conflicting is a non-trivial task and considering non-binary pairs of constraints (discussed as future work in Section 8.4.2) makes the problem even more complex. Therefore, I reduce the initial complexity of the framework by using a predefined set of binary pairs of conflicting constraints.

After the applicable constraints for a given problem instance are inferred, the framework uses the set of conflicting constraints to check if any inferred constraints conflict. If a conflict exists, scopes for the conflicting constraints must be learned. An SVM model is learned for each pair of conflicting constraints and these constraints act as class labels when training the model. After learning the models, variables in the constraint model are partitioned into the scope of one of the two constraints for all conflicting pairs. Constraints with no conflicts are part of the "global" scope and apply to all the instance's variables.

4.2 Training a SVM Model

As mentioned previously, a set of conflicting constraints defined by a domain expert is used to determine if the variable space of a problem instance must be divided for two or more constraints. For the BID problem and specifically in El Segundo, addresses increasing to the east and addresses increasing to the west are examples of conflicting constraints. If after the inference process, the set of applicable constraints contains two or more conflicting constraints, we must find the scope for these conflicting constraints. Finding these scopes begins by training a SVM model for all sets of conflicting constraints.

The feature space of the SVM model is defined by the features of the data points. Therefore, an input vector represents one data point where the attributes of the vector are the features that define the data points and the values of these attributes are the corresponding feature values. Figure 4.4 shows four example feature-vectors used to train the SVM model for the conflicting constraints in El Segundo. It should be noted that

Sample BID problem data points

$\langle addressNumber, streetType, streetName, sideOfStreet, latitude, longitude \rangle$

$\langle$ 1007, EW, E Grand Ave, N, 33.9200581, -118.4054531 $\rangle$

$\langle$ 1104, EW, E Mariposa Ave, S, 34.9678123, -118.3971454 $\rangle$

$\langle$ 223, EW, W Franklin Ave, N, 33.9180688, -118.4181319 $\rangle$

$\langle$ 311, EW, W Oak Ave, N, 34.9859451, -118.4387412 $\rangle$

Input vector representation of the above data points

$\langle addressNumber, streetType, streetName, sideOfStreet, latitude, longitude \rangle$

Label 1: $\langle$ 1007, 1, 1, 2, 33.9200581, -118.4054531 $\rangle$

Label 1: $\langle$ 1104, 1, 2, 1, 34.9678123, -118.3971454 $\rangle$

Label 2: $\langle$ 223, 1, 3, 2, 33.9180688, -118.4181319 $\rangle$

Label 2: $\langle$ 311, 1, 4, 2, 34.9859451, -118.4387412 $\rangle$

Figure 4.4: Sample input vectors for conflicting constraints used to train the SVM model.

an input vector can only contain numbers as attribute values. Therefore, a number is assigned to every unique string and all strings are replaced by the corresponding number.

Each input vector used to train the SVM model must be associated with one of two class labels. In this work I only focus on pairs of conflicting constraints. However, the use of multi-class SVM [16] would enable support of non-binary sets of conflicting constraints and I discuss such an approach as possible future work in Section 8.4. In learning scopes per pair of conflicting constraints, multiple SVM models are learned and as such multiple sets of training examples are used for learning. The method by which these training examples are generated is described below.

Consider an inference rule r_i that provides support for a constraint c. Rule r_i is evaluated using two data points d_x and d_y. If r_i is asserted this signifies that the points d_x and d_y provide positive support for the constraint c. When evaluating inference rules, the framework maintains the set of data points D_{c_i} that provided positive support for each constraint. After all inference rules have been evaluated, a set of applicable constraints

AC is inferred and the framework determines if any (c_i,c_j) pair of conflicting constraints exist in AC. If (c_i,c_j) $\in$ AC then D_{c_i} is used as training examples for one class label and D_{c_j} provides examples for the other class label. Learning of the SVM models is done through the use of the SVM^{light} package[1] [34; 36; 50], a general purpose tool for learning SVM models.

The separating hyperplanes are learned by applying the linear basis kernel function to the training data. The results reported in Section 6.4 show that the linear kernel outperforms the radial one, which produces a higher-dimensional space that non-linearizes the original input space. I use the learned SVM model to classify data points with no supports for either constraint, assigning data points to the scope of either constraint label. This process is detailed in the Section 4.3.

Scopes are learned per pair p in the set PC of all pairs of conflicting constraints, where the set of conflicting constraints $ConflictCons = \{c|\forall p \in PC \;\; c \in p\}$. As such, for two conflicting constraints c_i and c_j, the scopes S_{c_i} and S_{c_j} are distinct and non-overlapping such that variable $v \in S_{c_i} \vee S_{c_j} \wedge \neg \in (S_{c_i} \wedge S_{c_i})$. As such, the scope of any constraint c where $\forall p \in PC \;\; c \in p$ will be limited to a subset of the problem variables, while the scope of any constraint $globalC$ where $\nexists p \in PC$ such that $globalC \in p$ will be defined over all problem variables. This approach also allows for overlapping scopes for two constraints c_i and c_k that do not conflict, specifically if $\nexists p \in PC$ such that $c_i \wedge c_k \in p$. Once the scope has been learned, the corresponding constraint is instantiated over the subset of variables in its respective scope, further specializing the inferred constraint model. Chapter 5 provides additional details about the instantiation process.

[1] svmlight.joachims.org

Specific to the BID problem, an SVM model is learned to determine the scope of the *Increasing East* and *Increasing West* constraints in El Segundo, and the scope of the *Odd on East* and *Odd on West* constraints in Belgrade. To generate the training examples for the SVM in El Segundo, the framework uses all of the buildings (data points) that provide positive support for the *Increasing East* constraint as examples for the class label a, and conversely all buildings that provide positive support for the conflicting constraint *Increasing West* as examples of class $\neg a$. This set of labeled examples is shown in Figure 4.5 and the full set of training examples $a \cup \neg a$ is used to learn a SVM model. A similar process is carried out to generate the training examples in Belgrade, using buildings that provide support for either parity constraint as class labels.

Given this learned model, the framework can now take a building for which it does not know which of the two constraints applies to it and classify it as belonging to the scope of one of the two conflicting constraints. The intuition is that the model will faithfully separate the buildings into the areas East and West of Main Street and each building will be added to the scope of the correct addressing constraint (*Increasing East* for buildings East of Main Street and *Increasing West* for buildings West of Main Street). As previously shown in Figure 4.1, the area labeled as *1* in this figure would correspond to the learned scope for the *Increasing West* constraint and the area labeled as *2* to the scope for the *Increasing East* constraint.

The accuracy of the learned SVM model relies heavily on the training examples. Therefore, the framework tries to augment the set of examples and this process is best explained with an example. The general technique used to create new training examples utilizes the bucketing of data points concept as described in Section 3.1.1. If all data

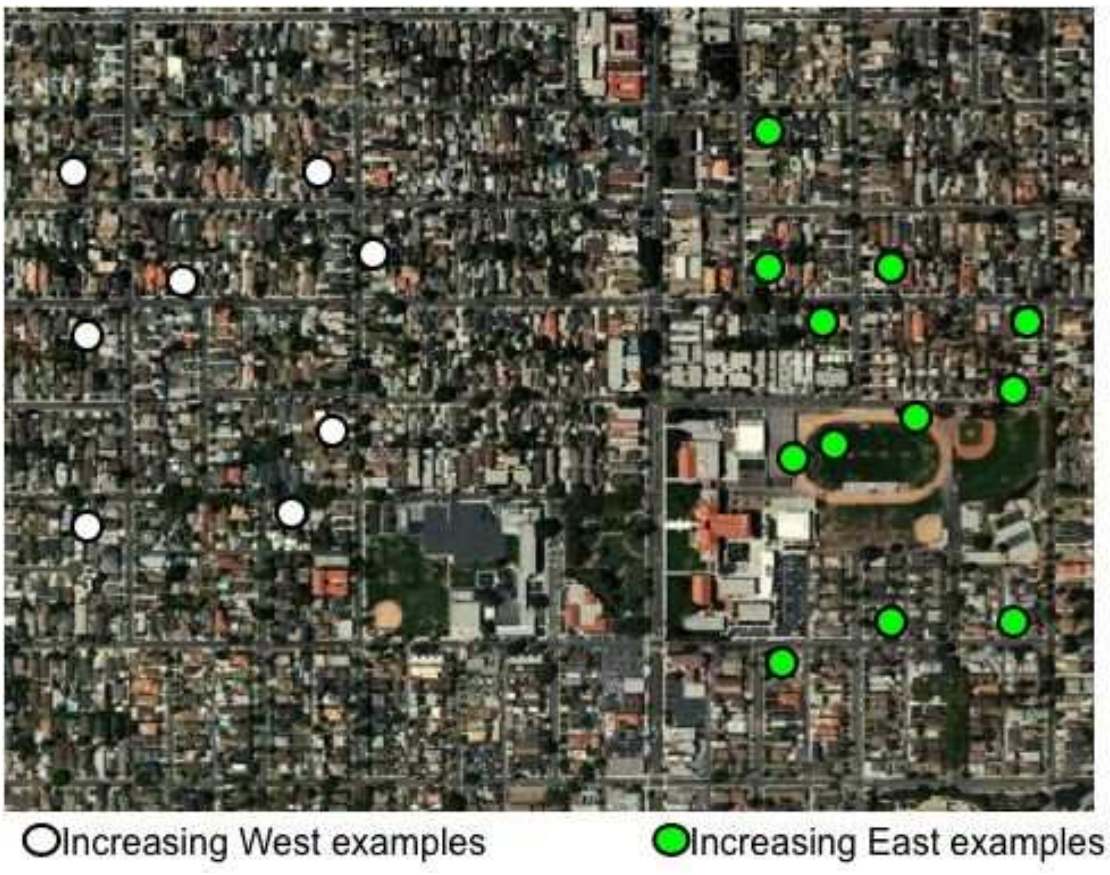

Figure 4.5: Labels used to learn the SVM model in El Segundo.

points within a bucket b provide positive support for *only* constraint c_i in the pair (c_i,c_j) of conflicting constraints, b can be used to generate additional examples for the class corresponding to c_i. These new examples are generated by creating input vectors where the feature value defining the bucket is fixed and the other attribute values are set using the known values of each attribute.

For example, consider a bucket containing two data points $\{d_1 = (1,2), d_2 = (1,3)\}$ where data points are defined by attributes $attr_1$ and $attr_2$ such that $d_x = (attr_1,attr_2)$ and the bucket is defined by the proposition $attr_1 = 1$. If both d_1 and d_2 provide positive support for only constraint c_i and $\{2,\ldots,50\}$ is the set of known values for attribute

$attr_2$, the framework generates two new examples d_{n1} = (1,40) and d_{n2} = (1,50) and these examples are used to augment the set of examples for the class label corresponding to c_i. This technique, when applied to the BID problem and Sudoku puzzles, has yet to produce noisy training data.

More specifically, consider the BID problem where the data points along the street W Imperial Avenue all provide positive support for the *Increasing West* constraint. Because the framework assumes that addressing characteristics hold for a street segment, it can augment the set of training examples by generating additional "temporary" data points along W Imperial Avenue. The new data points used for training the SVM are generated by using the bucket containing only data points on W Imperial Ave and the range of longitude values known for that street segment. This process provides additional examples for the *Increasing West* class label by using a larger sample of points along W Imperial Ave and it can be repeated for all streets that provide positive support for the *Increasing West* constraint. This technique is illustrated in Figure 4.6.

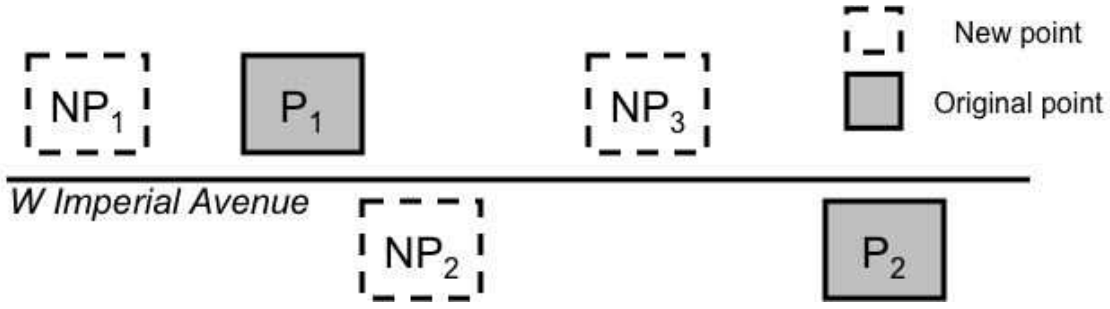

Figure 4.6: New training examples generated in the BID problem.

In the Sudoku domain, if a bucket containing points in a given row provides positive support for one of two conflicting constraints c_i, the set of training examples for c_i can be augmented by creating additional data points that are on that row. Even though

these new points do not have a filled-in number, they are defined by a location (row and column) and region and this information is useful when learning the SVM model. Hence, they help provide a better distribution of training examples along the row and lead to a more accurate learned model.

4.3 Assigning Data Points to a Scope

Given a learned SVM model for each pair of conflicting constraints, the framework is ready to classify data points into one of two classes corresponding to the scopes of the conflicting constraints. The data points that need to be classified are those that do not provide any positive support for either constraint in a pair of conflicting constraints. Using the same definition as that of the input vectors, the set of unclassified data points is converted to vectors and classified by the SVM model. The entire process is illustrated in Figure 4.7.

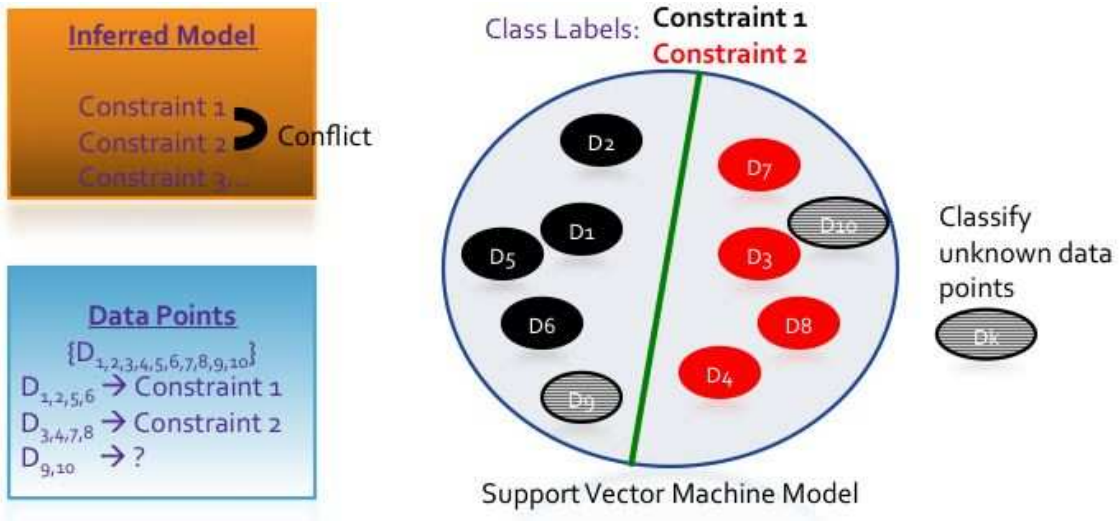

Figure 4.7: Assigning data points to a scope.

Intuitively, the SVM model represents the entire feature space of the problem instance, and the separating hyperplanes correspond to the boundaries of the scopes for the two conflicting constraints. Therefore, when the model is applied to unclassified points, these points are positioned somewhere in this feature space. Their location in the feature space falls into one of two scopes and the points are classified as belonging to one of these scopes. This classification is used when a constraint model is instantiated for the given problem instance (see Chapter 5).

The process of assigning data points to a scope is performed for all sets of conflicting constraints. Because multiple conflicts may exist within a problem instance, multiple SVM models are learned and used to classify the points. This classification is accomplished using an iterative process that terminates once all data points have been classified for all inferred constraints. As I previously described, constraints with no associated conflict apply to the "global" scope which covers all of the variables in the problem space. Furthermore, constraints apply to all variables within a scope, allowing the framework to propagate constraints to variables for which it cannot find any support (see Chapter 5).

Take El Segundo as an example and specifically the conflicting constraints *Increasing East* and *Increasing West*. Section 4.2 described the process by which the SVM model for these conflicting constraints is learned. Now consider the example set of data points shown in Figure 4.8. These data points are all on East/West running streets and they provide no support for either conflicting constraints. As such, they need to be classified as part of either the *Increasing East* or *Increasing West's* scope. Intuitively this means the framework will use the learned SVM model to determine which points are East and which points are West of Main Street. The correct classification of these points would

⟨*Name* | *AddressNum* | *StreetType* | *StreetName* | *StreetSide* | *Latitude* | *Longitude*⟩

Imperial School | 540 | EW | E Imperial Ave | S | 33.9302909 | -118.4106319
First Baptist Church | 591 | EW | E Palm Ave | N | 33.925291 | -118.4100763
El Segundo Library | 111 | EW | W Mariposa Ave | N | 33.9180688 | -118.4204327

Figure 4.8: El Segundo data points with no support for either conflicting constraint.

be *Imperial School* and *First Baptist Church* as part of the *Increasing East* class and *El Segundo Library* as part of the *Increasing West* class.

I present a method to learning scopes that is domain independent and applicable to application domains where conflicting constraints apply and 'govern' different portions of a given model. The only assumption I make is that the user specifies beforehand

what constraints are conflicting and thus cannot have the same scopes. The experimental results shown in Section 6.4 demonstrate the effect data points have on the framework's ability to learn scopes and they also show the accuracy with which it can learn scopes for a collection of BID problem instances and for the Samurai Sudoku puzzle. In this puzzle, the boundaries of a scope are defined by each one of the five puzzles that make up a Samurai Sudoku. The SVM learning method learns these boundaries automatically using the method described in this chapter.

After determining the scope for all inferred constraints for a problem instance, the framework has a set of inferred constraints along with their scopes, and input information from the problem instance definition and external data sources. It completes the model generation process by instantiating the constraint model using instance-specific information and passes the instantiated model to a custom constraint solver [6; 7]. The instantiation of the constraint model is described in Chapter 5.

Chapter 5

Instantiating a Constraint Model

In this chapter I discuss how to instantiate a constraint model given a set of inferred constraints and their scopes. In the end-to-end constraint inference process seen in Figure 2.7, this approach corresponds to the *Model Creation* component. This process is the last step in the generation of a specialized constraint model for a given problem instance. Instantiating the model allows the framework to pass it onto a constraint solver that finds a solution(s) for the problem instance. Section 5.1 describes the difference between a set of inferred constraints and an instantiated model, and outlines the transition that needs to take place from one to the other. Section 5.2 explains how the applicability of constraints is propagated through the problem instance. Section 5.3 presents the format in which instantiated constraints are represented. Finally, Section 5.4 provides the algorithm used to instantiate the final constraint model.

5.1 Transitioning to an Instantiated Model

The approach to inferring applicable constraints for a given problem instance, as described in this dissertation, uses information specific to the instance by way of data points.

These data points represent constraint variables whose values are known and these points correspond to a partial solution to the instance. However, the end goal of the framework is to generate as *complete* of a model of the problem instance as possible, which in-turn leads to a better solution than one generated when solving a sparse or generic model. In a constraint model, the completeness is defined by how constrained the variables in this model are and how likely it is that a single solution is produced when solving the model.

To generate a complete constraint model, one needs to consider what makes up a Constraint Satisfaction Problem (CSP). A CSP is represented by a tuple $P = (X, D, C)$ where $X = \{X_1,\ldots,X_n\}$ is a finite set of variables, each associated with a domain of discrete values $D = \{D_1,\ldots,D_n\}$, and a set of constraints $C = \{C_1,\ldots,C_n\}$. Each constraint C_i is expressed by a relation R_i on some subset of variable values $R_i \subseteq D_{i1} \times \ldots \times D_{ik}$ and denotes the tuples that satisfy C_i [19]. A solution to a CSP is an assignment of domain values to variables in such a way that no constraints are violated.

In the context of the constraint-inference framework presented in this dissertation, the *inferred* model of a problem instance is represented by the tuple $P_{inf} = (X_{inf}, D_{inf}, C)$ where $X_{inf} \subset X$ is a subset of the variables in the problem instance (the set of data points), each variable is associated with a domain where $D_{inf} \subset D$ and finally the relations $R_{inf} \subset R_i$ for a constraint C_i. The instantiation of a constraint specifies the subset of variables over which R_i is defined. Because the inference framework only uses the data points to make inferences, when the inferred constraints are instantiated by the framework this is done over a subset of the variables in the problem instance. The problem then lies in transforming P_{inf} such that it as closely as possible approximates the best model P of the given problem instance.

Consider the BID problem for El Segundo where the data points used by the framework are obtained from an online gazetteer. On average, a gazetteer provides between 30 and 40 data points for an area the size of El Segundo. However, this city contains roughly 1700 buildings and a solution to the BID problem is an assignment of addresses to *all* buildings. These buildings are represented as variables in the CSP model and the data points (buildings with known addresses) represent a small fraction of the entire area. As this example shows, a model instantiated over only the buildings representing data points greatly under-represents the actual instance we are trying to solve and needs to be augmented to become more complete.

The transition from an inferred model to an instantiated one is a two-step process. First, the set of variables needs to be expanded to include a set representative of the entire problem instance. In the BID problem this means representing all buildings in the area of interest with a variable, adding to the subset of variables X_{inf} (data points) used to infer the applicable constraints. Second, once the framework has a full set of variables, the relation R_i needs to be defined over the larger set of variables for each inferred constraint C_i. An example for the BID problem is instantiating the *Odd on North* constraint over the variables representing all buildings in the area and not just over the subset of buildings represented as data points. Optionally, if online sources provide additional information about a problem instance, they can be used to reduce some of the variable domains for the instance.

The full set of variables is obtained from the problem description. This description specifies what needs to be assigned a value. For example, in the BID problem the set of variables is generated by extracting the buildings in the area of interest from a satellite

image. Building identification is a separate research topic and I am assuming that I have a tool at my disposal which can be used to identify the buildings in an image [44]. This set of buildings represents all buildings in the area and it subsumes the set of variables represented by data points and used to infer the applicable constraints. Similarly in Sudoku puzzles the full set of variables represents all 81 cells of a 9x9 puzzle, augmenting the subset of variables (filled-in cells) used to infer the applicable constraints. Section 5.2 outlines the process that instantiates the constraints in more detail.

5.2 Constraint Propagation

In this section I describe how inferred constraints are instantiated over the largest possible set of variables using a propagation method. This process leads to an instantiated model of the problem. The representation of this model is described in Sections 5.3 and 5.4. The goal in generating the instantiated model is to make it highly-constrained by having the framework instantiate each constraint over a maximal set of variables. In the case of globally applicable constraints, this set is equal to the total number of variables in the final solution.

One possible instantiation policy is to instantiate constraints using only the variables that correspond to the data points in the input data. For example, an instantiated model for a BID problem instance would instantiate an inferred constraint c_i over only the variables (buildings) that provide positive support for c_i. This scenario corresponds to the scenario described in Section 5.1 where the instantiated problem model would correspond to P_{inf}. Such a model would likely be very under-constrained unless all of

the variables are present as data points. Solving an under-constrained model tends to generate imprecise solutions and the goal is to provide accurate solutions.

The approach used by the framework involves propagating the inferred constraints to variables not represented as data points. When inferring the applicability of constraints, the framework only considers known variable-value pairs (data points). As such, a set K of variables with no known values exists. The values for these variables cannot be determined from the problem description nor from any available online data-source and in turn these variables provide no insight into the defining characteristics of the given problem instance. Specifically, the set K in the BID problem represents the buildings that no gazetteer or other online source could provide an address and geographical coordinates for, or K represents the set of cells in a Sudoku puzzle that are not initially filled-in.

Because they provide no valuable information to the inference process, the variables in K are not considered when determining the applicability of constraints for the inferred problem model P_{inf} and the applicable constraints in P_{inf} are never instantiated over any variable $v \in K$. The problem lies in how to extend the applicable constraints to the variables in K and hence the goal of the framework is to include the variables in K when instantiating an applicable constraint c_i. The prevailing assumption being made is that a constraint, if applicable to some subset of data points, also applies to variables of the same *type*. Variables of the same type have similar feature values as the data points that support the given constraint.

Specifically, in the BID problem this thinking leads to the instantiation of constraints over all buildings (variables) that are on the same street, on the same side of the street, or some other common feature. Take the *Odd on North* constraint as an example. This

constraint, when applicable, ensures that the addresses of all buildings along the North side of East/West running streets are odd. If the inference framework infers the applicability of the *Odd on North* constraint for the given problem instance, the framework should instantiate this constraint over all variables representing buildings along the North side of East/West-running streets.

The algorithm used to propagate the inferred constraints is presented in Figure 5.1. It begins by using the scope of a constraint to find the relevant set of variables over which the constraint is instantiated. Every inferred constraint has a corresponding scope that defines the set of data points over which the constraint applies. In the case of conflicting constraints in the inferred set, the set of data points (variables) to which each conflicting constraint applies is a subset of all the data points. On the other hand, constraints within the global scope (having no conflicting opposite) apply to all instance variables.

```
SCOPE-AUGMENTATION(C, DP)
 1  vars ← full set of variables for the instance
 2  dataPoints ← DP
 3  for v ∈ vars ∧ ∋ dataPoints
 4        do
 5           determine all potential feature values of v
 6  for each applicable c_i ∈ C
 7        do
 8           SR_i ← data points in scope of c_i
 9           for all dp_i ∈ SR_i
10                 do
11                    comChars ← common characteristics of all dp_i
12           for v ∈ vars ∧ ∋ dataPoints
13                 do
14                    if comChars ⊂ FEATURESVALUES(v)
15                       then scope of c_i ← v
```

Figure 5.1: Algorithm used to propagate the set of applicable constraints.

The set of relevant data points SR_i for each inferred constraint c_i is composed of all data points that are within the scope of c_i. The framework finds variables that are not represented by any data point and are similar to the applicable data points based on the similarity in their feature values. First, the framework uses the input information to determine the *possible* feature values of the variables that are not represented as data points in the problem definition. The reason these aren't definite values is we do not have the exact variable-value pairs for all variables in the system (this would correspond to a full solution). For example in the BID problem, a corner building that was not represented by any data point may lie along one of two streets, which one we cannot be sure of. Therefore, the street-feature value for this variable would be one of two possible values.

With possible feature values for all variables, the framework looks at the set of data points SR_i for each constraint c_i and finds common characteristics between them. This is similar in nature to the bucketing approach described in Section 3.1.1. Once it has determined the commonalities of the points, it can select the remainder of the variables that are most similar to the data points in SR_i. Consider the *Odd on North* constraint in the BID problem. The commonality among data points within the scope of this constraint is that they are all on the North side of East/West running streets. Therefore, the framework finds all instance variables (buildings) that are on the North side of East/West running streets and stipulates that the *Odd on North* constraint must apply to these variables as well.

Figure 5.2 shows the set of streets over which the *Odd on North* constraint is instantiated before the propagation algorithm is run. We can see that it only covers a small subset

Figure 5.2: E/W street coverage of the *Odd on North* constraint **before** propagation.

of the East/West running streets in El Segundo and leads to a rather under-constrained model. Figure 5.3 shows the new set of streets covered by the *Odd on North* constraint for the same area. As we can see, the coverage is more complete and the model better represents the addressing characteristic in this area.

After propagating the constraints, the framework can instantiate a more specific constraint model. There is however the possibility that the model becomes over-constrained by incorrectly adding a variable v to the set of variables over which constraint c_i is instantiated. Incorrectly adding a building along the South side of East/West running street to the set of variables over which the *Odd on North* constraint is instantiated is an example in the BID problem. This scenario generally leads to an unsolvable constraint problem

Figure 5.3: E/W street coverage of the *Odd on North* constraint **after** propagation.

model. As such, the framework needs to have a recourse by which it can relax the model. The inference framework supports varying levels of relaxation.

First, the generated problem model can be relaxed by completely eliminating the propagation of constraints. The resulting model is the most general given the framework's inferences and corresponds to P_{inf} in Section 5.1. Second, the framework can selectively roll back the propagation on a per constraint basis. This iterative process relaxes the model until a solvable instantiation is found. In general, this later approach leads to a more constrained problem model when compared to the former however both facilities exist as options in the framework. By focusing on CSPs that are guaranteed solvable, the generation of no solution implies the inference of an incorrect problem model.

Finally, a less complete model can result from an inability to infer all applicable constraints, the worst case being the inability to infer any information specific to the given instance. This worst case has yet to be encountered in my studies of multiple instances across both the BID problem and Sudoku puzzle domains. As the results in Section 6.6 show, any solution produced by solving an inferred model still represents a significant improvement over ones obtained when solving a generic model.

5.3 Constraint Schema

Given the richer constraint problem model generated using the process described in Section 5.2, the framework is ready to represent this model. This representation uses an XML scheme in the XCSP format[1] which I developed for all constraint classes in the BID problem domain. As future work, I will develop a similar schema for Sudoku puzzles (see Section 8.4). The purpose of defining the schema using the XCSP format is two-fold.

First, this is the standardized format used to represent CSP problem instances. The steering committee of the international CSP solver competitions has decided to adopt the XCSP format as the de facto format that all competitive solvers must support. Therefore by generating problem instances using this standard, I will make a large set of problem instances available to the research community. The availability of these instances will push researchers towards non-traditional CSP problems and help drive the research in new directions. As was the case with my initial introduction of the BID problem [47],

[1] www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf

a different class of problems helps foster a different perspective with which to approach research problems.

Second, the use of a standard representation allows the framework to be more easily integrated into a large-scale application. Inevitably new advances in the fields of search and constraint programming will be discovered and incorporated into modern solvers. The current iteration of the BID problem-solving application I have developed uses our customized solver [6; 7] to assign addresses to buildings. By representing instantiated constraint problem models using a standardized format, I can plug new solvers into the application to test their scalability and efficiency in solving the BID problem. This ability to "plug and play" solvers increases the sustainability of the BID problem application. It also enhances the ability to create an online application that can be used by people throughout the world. Such an application is specifically discussed as future work in Section 8.4.

```
<inference districtid="54">
  <parity>
    <street streetname="W Grand Ave" value="N" />
    <street streetname="Binder Pl" value="false" />
    <street streetname="W Imperial Ave" value="N" />
  </parity>
</inference>
```

Figure 5.4: Instantiated example of the *Odd on North* constraint.

One XML file is used per instantiation of an inferred constraint and Figure 5.4 shows an example XML file representing the instantiated version of the *Odd on North* constraint. Two items of interest should be noted in this representation. First, the separation of constraints into individual files makes the relaxation of models easier. For example, if

the *Odd on North* is incorrectly inferred as applicable, to relax the model the XML file shown in Figure 5.4 would simply need to be removed. Additionally, when adding a new constraint to the library the only step required of a domain expert is the definition of an XCSP schema. Given this new schema, constraint models can be augmented with the addition of one XML file representative of the new constraint rather than altering all existing XML files for a given problem instance.

The second advantage of such a representation is the ability to easily backtrack over the propagation of a constraint across instance variables as described in Section 5.2. In Figure 5.4, the street *Binder Pl* is assigned the value *false*. This stipulates that the *Odd on North* constraint should not be instantiated over any buildings (variables) along this street. If the framework generates an unsolvable problem model due to the incorrect propagation of the *Odd on North* constraint across some street, a more subtle relaxation than the one described above can be accomplished. Assigning the value *false* to the street where an incorrect inference was propagated allows the framework to backtrack over the incorrect propagation rather than removing the entire *Odd on North* constraint from the problem model.

The entire set of XCSP schemas for the BID problem, with instantiated examples of each, can be found in Appendix B.

5.4 Instantiating a BID Problem Instance

An instantiated constraint model best represents the problem instance at hand given its input information. The definition of the variables and their domains, and the instantiation

of the constraints provides a solver with the information required to generate a solution(s). This solution consists of an assignment of values to the variables such that no constraints are violated. As mentioned in Section 5.2, the specificity of a constraint model greatly influences the precision of the solution. Below I outline how all of the gathered and inferred information is used to generate the XML files described in Section 5.3 for the BID problem.

The first step uses the input information to determine the layout of the problem. The satellite imagery is used to find the buildings (variables) in the instance. The vector data, which represents the street network, is used to determine the domain of the street variable for each building. The domain is determined by locating all of the *possible* streets every building can be on. In general the domain size is one, however for corner buildings the size can reach up to four.

Phone entries from online phonebooks are used to populate the domains of variables potentially on a particular street. For example, when instantiating the problem model of an area with the set {1,2,3,4,5,6} of known addresses for a given street, layout constraints are instantiated assuring that these addresses are part of the final solution. The details on how domain ranges are represented are beyond the scope of this dissertation and I point the reader to our work in Bayer et al. [7] for more details.. Finally, data points represent *landmark* points for which we know the address. Therefore, a landmark constraint is instantiated for each data point, forcing the solver to assign the specific address to the corresponding variable.

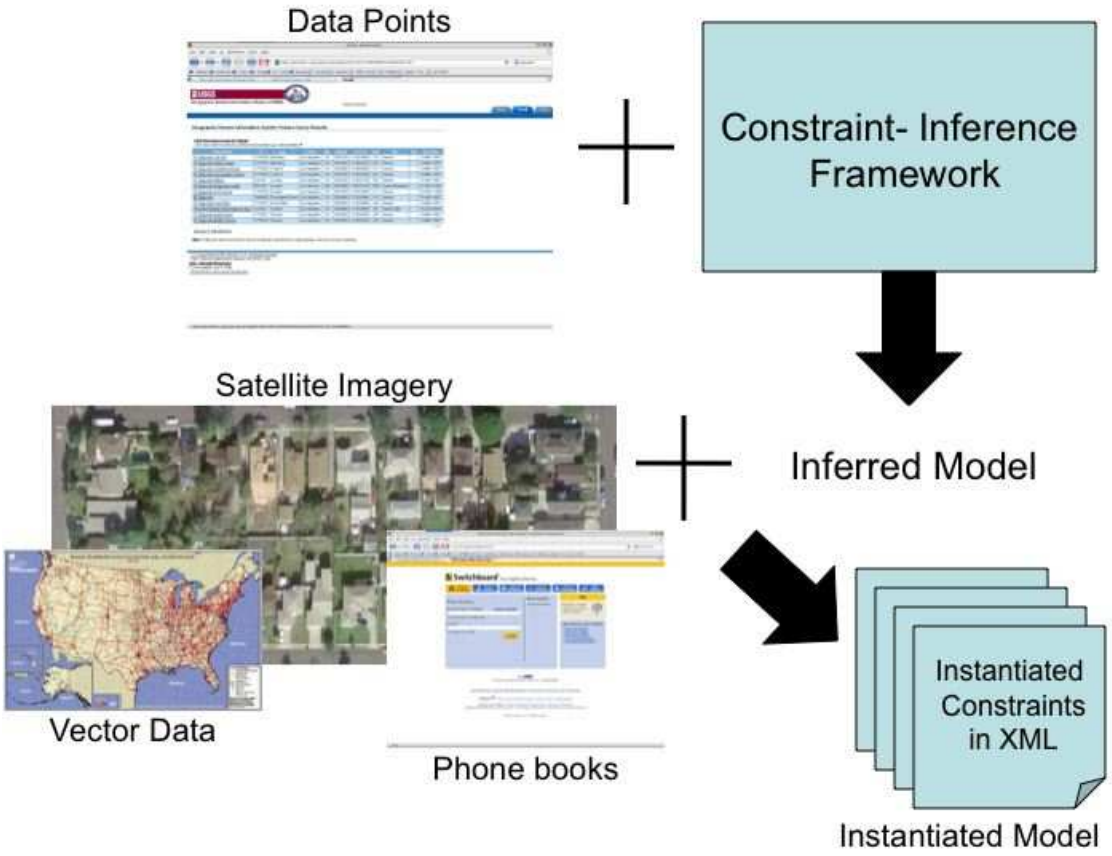

Figure 5.5: The process by which a constraint model is instantiated for the BID problem.

Having defined the layout of the problem, the framework instantiates the inferred constraints. This process is illustrated in Figure 5.5 and involves propagating the constraints as described in Section 5.2. Once the inferred constraints have been propagated to all relevant instance variables, the framework generates an XML file for each inferred constraint according to the XML schema described in Section 5.3. Because constraints are inferred based on the input information, each constraint model is unique and represents the given problem instance. It is possible that a single area of interest may be modeled with differing constraint models. If the data sources used to provide the data

points are different, or if different vector-data sources provide the road information, the inferred constraints and the instantiated model may vary.

After constructing a constraint problem model representing the given problem instance, the generated XML files provide a vehicle by which the framework connects the model-generation process to the model-solving component. Our current solver [7] used in the BID problem domain is a specialized solver implemented as a backtrack-search solver in Java. This solver uses backtrack search (BT) with nFC3, a look-ahead strategy for non-binary CSPs [9], and conflict-directed back-jumping [54]. It takes the XML files as input, generates the corresponding internal representation of the problem model, and solves this model. The solver and its model-solving methodologies are beyond the scope of this dissertation and I point the reader to our previous work [6; 7] for additional details.

Chapter 6

Experimental Evaluation

This chapter provides an experimental evaluation of the concepts introduced throughout this dissertation. Section 6.1 introduces the data sets used in the BID problem and Sudoku puzzle domains to perform the evaluation of inference-framework. Section 6.2 outlines the experimental setup for all experiments. Section 6.3 evaluates a basic version of the inference framework for the BID problem and Sudoku puzzle domains. These results demonstrate the framework's ability to infer basic models where no conflicting constraints exist. Section 6.4 evaluates the use of Support Vector Machines (SVMs) for learning the scope of constraints. Section 6.5 shows the added benefit of augmenting the initial set of data points using constraint propagation for Sudoku puzzles. Finally, Section 6.6 presents the best models inferred by the framework, and shows the performance improvement of a solver when using an inferred model over a generic one.

6.1 Data Sets

In this section, I introduce the data sets used to evaluate the constraint-inference framework. All of the problem instances are grounded in the real world and each instance has

one unique solution. Each set of instances has a unique feature(s) that distinguishes it from the others and helps highlight some ability within the framework.

6.1.1 BID Problem

For the BID problem, problem instances span different areas throughout the world. I varied the set of landmark data-points (defined in Section 2.2) for areas by using different public data sources. Each source provided a different set of data points varying in number and distribution within the area of interest. Choosing different data sources helps demonstrate the effect that data-point distribution has on the framework's inference capabilities. Other data sources could have been used but the chosen ones were easily accessible and provided varying types of buildings.

Table 6.1: Homogeneous BID problem instances.

Area	**Data points**	**~Total Buildings**	**Known Addresses**
1. (a) El Segundo	38	820	4.6%
1. (b) El Segundo	660	820	80.5%
1. (c) El Segundo	12	860	1.4%
2. Downtown LA	7	45	15.6%
3. San Francisco	16	90	17.8%
4. Boulder	7	160	4.4%
5. New Orleans	21	110	19.1%
6. Belgrade, Serbia	85	98	86.7%

The first set of instances, shown in Table 6.1, report the *area* of interest, the number of *data points* in the area, the approximate *total number of buildings*, and the percentage of *known addresses* for the area. These instances cover different areas that have no conflicting constraints in the set of defining addressing characteristics. Therefore, there is no need to learn the scope of any defining constraint as they all apply to every building in

the instance. However, each area has a unique layout of buildings and varying addressing characteristics. The first two *El Segundo* instances cover the area west of Main Street while the third covers the area east of Main Street. For instance 1(a), the 38 points are manually chosen from the LA assessor website. In instance 1(b), the roughly 660 data points are obtained from an online geocoder service. Lastly, the 12 data points for instance 1(c) are obtained from the USGS gazetteer.

The *Downtown LA* instance covers a 10 block radius in downtown Los Angeles CA and the data points are obtained from a hotels data-source found by querying Google for csv files with addresses and latitude and longitude coordinates. The *San Francisco* area covers a 21 block radius of San Francisco CA and the data points are churches and schools obtained from the publicly available USGS gazetteer. The *Boulder* instance covers a rural part of Boulder CO and again the data points are obtained from the USGS gazetteer and are composed of schools in the area. The *New Orleans* instance covers a central area of downtown New Orleans LA and uses 21 USGS gazetteer points. Finally, the *Belgrade* instances covers a 16 block area in Belgrade Serbia and uses data points from an online government planning website.

The second set of instances, shown in Table 6.2, extends some of those in Table 6.1 creating new instances that are more complex in nature, namely requiring the framework to determine the scope of conflicting constraints. This new set expands instances 1(b) (instance 7(b)) and (c) (instance 7(c)) to include both the areas East and West of Main Street. Instance 7(b) now uses roughly 1650 data points obtained from a geocoder service while 7(c) uses 20 points from the USGS gazetteer. A constraint model of each instance

now contains the conflicting constraints *Increasing East* and *Increasing West* for which scopes must be learned.

Table 6.2: Non-homogenous BID problem instances.

Area	**Data points**	**∼Total Buildings**	**Known Addresses**
7. (a) El Segundo	38	1680	2.3%
7. (b) El Segundo	1650	1680	98.2%
7. (c) El Segundo	20	1680	1.2%
8. Downtown LA	7	45	15.6%
9. San Francisco	16	90	17.8%
10. Boulder	7	160	4.4%
11. New Orleans	66	230	28.7%
12. Belgrade, Serbia	88	101	87.1%
13. Jakarta Indonesia	20	145	13.8%

Instances 8, 9, and 10 remain the same (correspond to instances 2, 3, and 4 in Table 6.1), while the *New Orleans* instance (11) covers a larger area of downtown New Orleans, LA and the 66 data points are again obtained from the USGS gazetteer. A fan-like layout of buildings is seen in this area and this leads to a set of conflicting constraints for which scopes need to be learned. The *Belgrade* instance (12) covers an area around the former Chinese embassy building in Belgrade, Serbia and the 88 data points are obtained from an online government planning website. In this area, the side of the street on which odd numbers lie is inconsistent and the scope for the *Odd on East* and *Odd on West* constraints needs to be learned. Finally, the *Jakarta* instance (13) is included which covers a large portion of downtown Jakarta, Indonesia and uses 20 hotel data-points culled from an online Indonesian hotel website. This area has two sets of conflicting constraints, both the increasing and parity constraints, and represents the area with the most complex addressing characteristics.

6.1.2 Sudoku Puzzles

The Sudoku puzzle instances used to evaluate the framework included the basic puzzle along with four variations: geometry puzzles where the nine regions of the puzzle are of irregular shapes, diagonal puzzles where additional AllDiff constraints apply to both diagonals, odd/even puzzles where numbers in colored cells have same parity, and magic puzzles where each number inside a polyomino must be no larger than the number of blue cells in the region, along with the two diagonal constraints. Each puzzle was chosen for its distinct set of defining constraints and for its unique cell characteristics such as the coloring exhibited by some of the puzzle types.

The constraint library for this domain consists of the "defining" Sudoku constraints along with all the additional constraints introduced by each puzzle variation. Data points correspond to filled-in cells as defined in Section 2.2 and are obtained from the initial problem description. All puzzles are 9x9 in size and each puzzle contains nine regions. It should be noted that there was no need to learn scopes for these puzzles. The nature of Sudoku puzzles stipulates that no conflicting constraints can apply within a single puzzle. However, the need to find scopes arises when considering Samurai Sudoku puzzles. These puzzles are discussed in further detail in Section 6.4.

I conducted three sets of experiments for each puzzle type, testing the framework on 100 puzzle instances for each puzzle difficulty level.[1] The difficulty of a puzzle generally defines the number and distribution of the cells initially filled-in and as such affects the

[1]Randomly sampled from www.menneske.no/sudoku/eng and www.printsudoku.com/index-en.html

set of data points used by the framework to infer the constraint model of the puzzle instance.

6.2 Experimental Setup

All of the experiments were run on a MacBook Pro running a Core Duo 2 processor at 2.4GHz with 2GB of RAM. The framework is written in Java and run through the Eclipse IDE. Data points for each instance are stored in individual files and adhere to a format created by me. The time to solve each instance was within 45 seconds and on average less than three seconds. The Tiger Lines vector data used in the BID problem was stored in a SQL database and covered the entire United States. For areas outside of the US, I manually constructed the vector data based on the formating used to represent the Tiger Lines data.

6.3 Inferring Constraint Models

In this section, I evaluate the framework's ability to infer constraint models for a variety of problem instances within the BID problem and Sudoku puzzle domains. The results show that when the basic set of inference components are applied to instances that do not exhibit any irregular properties, the framework can effectively infer models for these instances. Properties such as the non-monotonicty of addressing in El Segundo for the BID problem and puzzles with a sparse set of filled-in cells in Sudoku puzzles are considered irregular. These results serve as a baseline and are used to motivate and show the

improvements introduced by the learning of scopes and the augmentation of the initial set of data points.

To validate the claims made above, I test a version of the framework that implements the general concepts introduced in Chapter 2 and uses the constraint selection algorithm presented in Chapter 3. This version of the framework does not use SVMs to learn the scopes of constraints nor does it propagate basic constraints to augment the input information. Instead, it only uses the initial set of data points to make its inferences. Motivated by the results obtained when applying this basic framework to the two problem domains, Sections 6.4 and 6.5 validate the framework's ability in using SVMs to learn the scope of constraints and in applying constraint propagation to augment the initial set of data points. Section 6.6 shows the improvements in the inferred models obtained from the inclusion of these techniques in the basic framework and the ability to solve more complex BID problem instances.

The evaluation of the basic framework is divided into sections corresponding to the two problem domains discussed throughout the dissertation. The BID problem instances are selected based on two main criteria: the differing addressing characteristics that define each area and the varying online sources that provide data points specific to the areas. The Sudoku puzzles represent a wide range of puzzles, varying in the sets of constraints that define each puzzle type and in the distribution of filled-in cells for each puzzle. The results in both domains validate the framework's ability to infer models and motivate the additional techniques presented in this dissertation.

6.3.1 BID Problem

For the BID problem domain, I evaluated the framework on the problem instances described in Table 6.1 of Section 6.1.1. The hypothesis for these experiments is that given an area in which no conflicting constraints exist, the framework can infer the most representative model given the data points available for the area. As the obtained results show, the framework is able to infer a correct model of the area and is only limited by the distribution of the data points within an area.

The results summarized in Tables 6.3 and 6.4 show the accuracy with which the framework could infer the constraint model for a given area. In these tables, *Area* specifies the area from Table 6.1, *Odd on North/East* corresponds to the parity constraints for E/W and N/S running streets, *Block* corresponds to the block numbering constraint discussed previously, *Increasing North and East* are the constraints that dictate the direction in which addresses increase for N/S and E/W running streets, and *Acc.* and *Comp.* correspond to the accuracy and completeness of the inferred model calculated over the variables in the problem instance. For example, if a problem has four building variables (two on the North/South and two on the East/West running streets) and all five constraints are inferred correctly but one variable is incorrectly placed in the scope of the *Increasing North* constraint, the completeness would be 100% and the accuracy would be $11/12 = 91.67\%$. Accuracy correlates with the correctness of the model over the entire set of variables while completeness represents the tightness of this model.

The results for the three sets of experiments in El Segundo are reported in Table 6.3. The first experiment (1. (a) in Table 6.3) served as a sanity check and the results conform

Table 6.3: BID problem instance results.

Area	**Odd On North/East**	**Block** $k = 100$	**Increasing North**	**Increasing East**	**Acc.**	**Comp.**
1. (a)	✓	✓	✓	✓	100.00%	100.00%
1. (b)	✓	✓	✓	✓	100.00%	100.00%
1. (c)	✓	✓	×	✓	100.00%	89.90%

✓ correctly inferred × not inferred N/A not applicable

to the ground truth. In this area, the accuracy and completeness of the inferred model were both 100%, meaning all of the applicable constraints were correctly inferred. The second trial (1. (b) in Table 6.3) used roughly 660 points generated by a geocoder web-service. This set of data points corresponds to roughly 80% of all buildings West of Main Street. This experiment tested the scalability of the framework and required no more than 10 seconds to run (but well over an hour without the bucketing mentioned in Section 3.1.1). Additionally, all of the applicable constraints where correctly inferred.

Finally, I utilized 12 gazetteer points from the USGS gazetteer that lay within El Segundo and had an address associated with them (area 1. (c) in Table 6.3). Using these data points, the framework was able to correctly infer all but one of the other applicable constraints, failing to infer the direction in which addresses increased along North/South running streets East of Main Street. This inability lead to the lower level of completeness for the model (89.9%) and was caused by a lack of diversity in the data points. Specifically, only one of the data points East of Main Street lay on a North/South running street. This experiment helps illustrate the effect data point distribution has on the accuracy of the results. To further explore the effect of landmark point distributions

for different areas, I ran the inference engine on other parts of the world. The results of these trials are shown in Table 6.4.

Table 6.4: Other cities: inferred constraints.

Area	**Odd On North/East**	**Block** $k = 100$	**Increasing North**	**Increasing East**	**Acc.**	**Comp.**
2.	✓	✓	✓	×	100.00%	87.50%
3.	✓	✓	✓	✓	100.00%	100.00%
4.	✓	N/A	×	✓	100.00%	76.45%
5.	✓	×	✓	×	100.00%	64.92%
6.	✓	N/A	✓	✓	100.00%	100.00%

✓ correctly inferred × not inferred N/A not applicable

For downtown LA (area 2 in Table 6.4), seven landmark points where derived from an online source of hotels, each with an address and latitude and longitude coordinates. All of the constraints in the library are applicable to this area and no conflicting constraints are present. The framework was able to correctly determine all but one of the applicable constraints, resulting in a lower level of completeness. Indeed, there was not enough information in the data points to determine in which direction addresses increased along East/West running streets. However, the remainder of the applicable constraints were correctly inferred, leading to the perfect measure of accuracy.

Next I considered subareas of San Francisco CA and Boulder CO (areas 3 and 4 in Table 6.4). Using 16 and seven points respectively from the USGS gazetteer, the framework was able to correctly identify all applicable constraints in San Francisco but it was unable to infer one of the applicable constraints in Boulder. There wasn't enough information in the data to determine in which direction addresses increased along North/South running streets in Boulder, leading to a lower level of completeness. Again this was caused by

an insufficient number of data points on North/South running streets. Yet, even with relatively small sets of data points the framework was able to infer accurate and almost fully complete constraint models.

In New Orleans LA (area 5 in Table 6.4), the framework was able to infer the parity constraints and the direction in which addresses get bigger for North/South running streets. However, the distribution of the data points was such that it was unable to infer the applicability of the block numbering constraint and the direction in which addresses increased for East/West running streets, causing a drop in completeness. This drop can also be attributed to the lack of data points in this instance. Creating an area with no conflicting constraints greatly reduced the number of data points available to the framework. For Belgrade (area 6 in Table 6.4), 85 points from a government planning website were used to infer the applicable constraints. The framework was able to correctly infer all applicable constraints.

The results obtained when applying the basic framework to the areas presented in this section lead to the following observation. Given a set of data points that provide enough information about an area, the framework is able to correctly infer the set of applicable constraints. This claim is validated by the results obtained in El Segundo (instances 1. (a) and (b)), in San Francisco (instance 3) and in Belgrade (instance 5) and provides evidence for the hypothesis stated at the start of this section.

These experiments also highlight the importance of data point distribution over the problem instance. I estimate that, with a perfect distribution of data points, the minimum number needed to correctly identify all of the constraints currently considered is $4 \times n$, where n is the number of scopes defining the problem. In the case of no conflicting

constraints $n = 1$ and when the number of conflicting constraint pairs $p > 0$, $n = 2p$. Essentially, the framework requires two points for each street type (North/South or East/West running). While this minimal set of points allows the framework to infer the set of applicable constraints, it remains vulnerable to noise.

To handle the disparity in the distribution of data points, additional online sources can be used to boost the data points used to infer the applicability of constraints. Specifically, if the set of gazetteers points used in El Segundo (instance 1. (c) in Table 6.3) was augmented with new data points along East/West running streets from the LA county assessor website, the complete model of the area would be inferred. Similarly restaurants from an online restaurant source for downtown LA and additional homes from a real-estate website in Boulder would allow the framework to infer the correct model for these areas, leading to a 100% level of accuracy and completeness.

These results help establish *the feasibility of the basic components* used in the framework. The next step is to solve more complex areas, namely those that contain conflicting constraints. The experiments carried out in Section 6.6.1 show how the framework is able to handle more complex cases.

6.3.2 Sudoku

The goal of applying the basic framework to the Sudoku puzzle domain is to demonstrate its ability to infer correct models for puzzles of varying type with a well-distributed set of filled-in cells. The hypothesis being made is that if the initial filled-in cells of a puzzle are placed in locations where the relationship between these cells leads to support for all constraints in the library, then a representative model of a puzzle can be inferred. These

results serve as a validation of the generality of the framework as the same basic methodologies applied to the BID problem in Section 6.3.1 are applied to these Sudoku puzzles. Additionally, these results serve as a baseline used to demonstrate the improvement in model quality after augmenting the initial set of data points with new points through the propagation of the basic constraint model.

The results summarized in Table 6.5 solve the instances presented in Section 6.1.2 and represent the completeness and accuracy values for each inferred model as an average over the 100 puzzle instances for each difficulty level. Recall is defined as the ratio of the number of correct constraints inferred to the total number of constraints representing the puzzle type, and accuracy is defined as the ratio of the number of correct constraints to the total number of constraints inferred. $|C_{new}|$ corresponds to the number of constraint types that define each puzzle variation.

Table 6.5: Sudoku: accuracy and completeness of inferred constraints.

		Easy		**Medium**		**Hard**	
	$\|C_{new}\|$	Comp.	Acc.	Comp.	Acc.	Comp.	Acc.
Basic	3	1.0	0.88	1.0	0.87	1.0	0.87
Geometry	3	1.0	0.86	1.0	0.88	1.0	0.88
Diagonal	4	0.86	1.0	0.86	1.0	0.85	1.0
Even/Odd	4	1.0	0.93	1.0	0.94	1.0	0.95
Magic	5	(not categorized): Completeness: 0.81, Accuracy: 1.0					

In general, the framework was able to correctly infer all of the constraints. The *basic* and *geometry* puzzles are nearly the same puzzle except that geometry puzzles have irregular regions, hence the results are very similar for both puzzle types. For both of these puzzles, the framework was able to infer the three core constraints (as reflected by

the perfect completeness), but it incorrectly inferred the existence of a diagonal constraint in some puzzle instances, causing a drop in overall accuracy. This incorrect inference of the diagonal constraint is caused by a lack of information in the data points. Specifically, the data points in these puzzles were laid out in such a way that roughly 86% of the puzzles did not have two filled-in cells with the same number along either diagonal.

This same incorrect inference caused the drop in accuracy for the *even/odd* puzzle. For this puzzle variety, the framework was able to correctly infer the applicability of all four of the defining constraints, leading to the perfect level of completeness. However, as seen in the *basic* and *geometry* puzzles, the data point distribution was such that incorrect inferences of the diagonal constraint were also made. However, the accuracy values for the *even/odd* puzzle were higher than those for the *basic* and *geometry* puzzles by about 7%. This discrepancy can be attributed to a better overall distribution of the data points leading to more negative support provided for the diagonal constraint.

The *diagonal* and *magic* puzzles were cases where all inferred constraints were correct (perfect accuracy), but the filled-in cell distribution was such that the framework was unable to infer a diagonal constraint (an applicable constraint) for all the puzzle instances which resulted in a drop in completeness. In fact, only 10% of all *magic* puzzle instances contained enough data points to infer this constraint. In general, the models inferred by the framework for these two puzzles types were correct but not complete.

These results show that given a well-distributed set of data points, the basic framework can infer a representative model of the puzzle. For puzzles such as *diagonal* and *magic* that are more constrained, the framework infers a correct puzzle that is incomplete (missing inferences) resulting in lower levels of completeness but perfect accuracy. Conversely,

for puzzles such as *basic* and *geometry* which are less constrained, the framework infers complete but less precise models, resulting in perfect completeness but lower levels of accuracy. Both issues can be addressed by augmenting the set of data points to improve the coverage of the points used to infer the models.

Augmenting the set of data points motivates the constraint-propagation technique described in Section 3.2. While the basic framework can infer precise models given the set of initial filled-in cells, its inference ability is enhanced when presented with additional data points. This claim is verified by the results presented and analyzed in Section 6.5. I also show that given the right circumstances, constraint propagation leads to the generation of 5 or more data points for each constraint type, satisfying the required number of cells leading to a strong inference as stipulated by the calculated Bayes factors of n presented below.

Table 6.6: Bayes factor: strength of evidence.

K	**Strength of Evidence**
< 1.0	Negative (supports M_2)
$1 < K < 3$	Barely worth mentioning
$3 < K < 10$	Substantial
$10 < K < 30$	Strong
$30 < K < 100$	Very Strong
> 100	Decisive

An additional approach to handling incorrect inferences of constraints is to adjust the support level required for constraints to be classified as applicable. To approximate the number of data points that need to provide support per constraint to confidently infer the applicability of any constraint, I calculate the Bayes factor[2] K for the set of

[2]en.wikipedia.org/wiki/Bayes_factor

cells $c = \{1, \ldots, 9\}$. Bays factor specifies which model M_i is more strongly supported by the data given two models M_1 and M_2 and is defined as $K = \frac{p(x|M_1)}{p(x|M_2)}$. For Sudoku puzzles, M_1 corresponds to the likelihood of seeing two numbers that are the same given n filled-in cells and is defined as $M_1 = \frac{1}{\binom{9}{n}}$. $p(x|M_1)$ represents the probability that seeing n filled-in cells leads to positive support for a given constraint. M_2 corresponds to the likelihood of seeing no numbers that are the same given n filled-in cells and is defined as $M_2 = \frac{1}{9^n - \binom{9}{n}}$. $p(x|M_2)$ represents the probability that seeing n filled-in cells leads to negative support for a given constraint. Table 6.6 relates the Bayes factor to the strength of evidence for a model M_i.

Table 6.7: Bayes factor for varying numbers of filled-in cells.

n	**Bayes factor**
1	0
2	0.125
3	0.446
4	1.169
5	**2.905**
6	**7.787**
7	**25.361**
8	**117.625**
9	**1066.627**

Table 6.7 presents the Bayes factor for the set of n (the number of filled-in cells for a constraint). We can summarize the results as saying that if $n \in \{5, 6\}$ we have *strong* evidence that we have seen two numbers that are the same and if $n > 6$ we have *very strong* evidence of seeing two numbers that are the same. Therefore, we can conclude that using 5 or more filled-in cells per constraint in the library to test the applicability of a given

constraint allows us to make a strong inference of the given constraint's applicability. Obviously the more filled-in cells available the stronger the inference.

Table 6.8: Sudoku: accuracy and completeness of model with a support level of 5.

		Easy		**Medium**		**Hard**	
	$\|C_{new}\|$	Comp.	Acc.	Comp.	Acc.	Comp.	Acc.
Basic	3	1.0	1.0	0.99	1.0	1.0	1.0
Geometry	3	1.0	1.0	0.99	1.0	1.0	1.0
Diagonal	4	0.69	1.0	0.64	1.0	0.69	1.0
Even/Odd	4	0.19	1.0	0.19	1.0	0.17	1.0
Magic	5	(not categorized): Completeness: 0.41, Accuracy: 1.0					

Given the Bayes factor analysis performed above, I changed the inference framework to require a support level of 5 or above for applicable constraints and I re-ran the experiments presented in Table 6.5 and the results of these experiments are shown in Table 6.8. As these results show, the *basic* and *geometry* puzzles are much improved where all models are correct and only 2 out of 600 models are not complete. Additionally, the *even/odd* puzzle is modeled correctly (perfect accuracy) but the models are much more incomplete. For the *diagonal* and *magic* puzzles, the inferred models remain correct but become more incomplete. We can conclude that the higher support levels leads to more precise models at the expense of completeness. However, depending on the application which is driven by our framework this a preferred result and I discuss this further in Section 6.6.2.

6.4 Learning the Scope of Constraints using SVMs

The results reported in Section 6.3.1 showed that the framework can infer models for instances that are homogeneous and contain no conflicting constraints. However, we

would like to solve more complex instances and the accuracy of an inferred model for such instances will rely on the framework's ability to determine the scope of conflicting constraints. Given enough data points and pertinent semantic information, the framework can effectively determine these scopes. However, semantic information is both problem instance specific and not always easy to define. Therefore the approach used to learn a constraint's scope needs to be both domain-independent and applicable to all variations of instances within the domain.

In this section, I evaluate the framework's ability to learn the scope of constraints using Support Vector Machines (SVMs), as described in Chapter 4. My hypothesis is that given a set of relatively well-distributed data points, an accurate SVM model for the scope of two conflicting constraints can be learned. Such an approach doesn't use domain-specific information and can be applied across problem domains. Being domain independent, the automation of constraint model generation is enhanced with the use of SVMs. I show how the distribution of data points in the problem instance affects the SVM's ability to correctly classify points into one of two scopes for a pair of conflicting constraints and I show that a linear kernel outperforms a radial one for both problem domains.

Figure 6.1 presents the results obtained when using SVMs to find scopes for the *Increasing East* and *Increasing West* constraints for the El Segundo BID problem instance 2(b) in Table 6.2, chosen for its largest set of data points (∼1650 geocoded data points). The x-axis represents the percentage of the points used to train the model (randomly sampled) and the y-axis represents the percentage for the different evaluation metrics when classifying the remaining points (test cases). Please note the different scales used

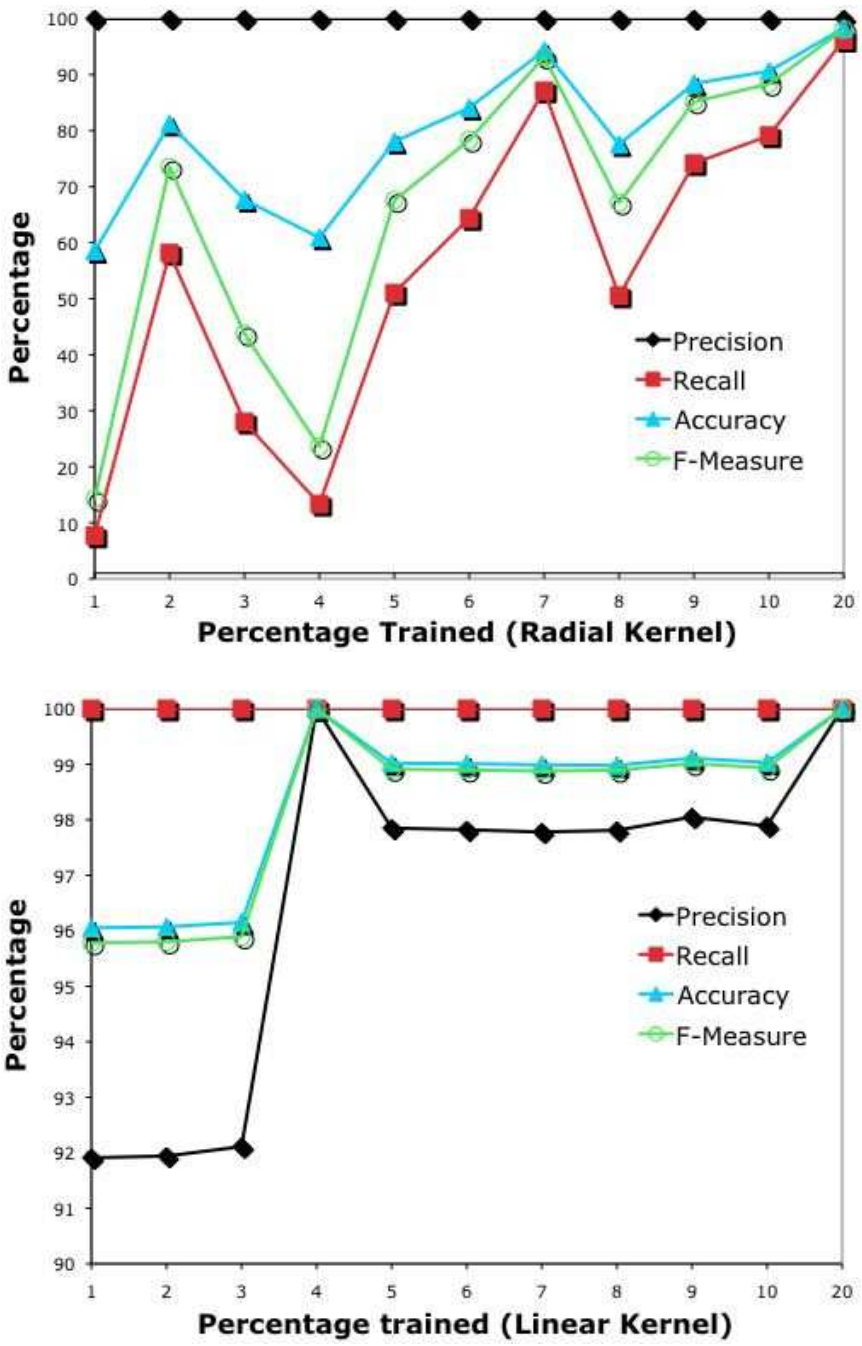

Figure 6.1: Results for finding contexts in El Segundo using SVMs.

for each y-axis. The graphs report measures of precision (number correct out of all labels), recall (number of correct labels out of all points), f-measure (the harmonic mean of the

precision and recall), and accuracy (the internal metric used by SVM^{light}). I report results obtained when using the radial and linear kernel functions.

As Figure 6.1 shows, using a linear kernel function and the procedure outlined in Chapter 4, the framework obtains 100% accuracy in labeling unknown data points when using only 4% or roughly 66 points to train the model. Obviously the results depend on the distribution of the points used to learn the SVM model. However, as additional experimental results show (see Section 6.6.1), this approach results in a high level of accuracy even when using a smaller set of data points. The linear kernel greatly outperforms the radial one because streets tend to be linear. Therefore the boundaries (hyperplanes) learned by the linear kernel more closely represent the street boundaries seen in the real-world. In contrast, the radial kernel uses non-linear approximations to learn the boundaries and as such leads to less accurate results. I do not report the times needed to train the SVM model as they are minimal (less then two seconds) for the small training sets.

An additional benefit of this approach is that the framework can use the learned scopes to classify data points that provide no support for a constraint. For example, in the BID problem the SVM model learns scope using the street attribute, allowing the framework to instantiate the correct constraints for streets with no supports for a given constraint. As seen by the results reported in Section 6.6.1, this refinement allows the framework to learn accurate constraint models for complex areas not supported by the basic framework.

This SVM-based approach is also applicable to Sudoku puzzles. Samurai Sudoku puzzles, as seen in Figure 6.2, have five Sudoku puzzles joined in a quincunx arrangement.

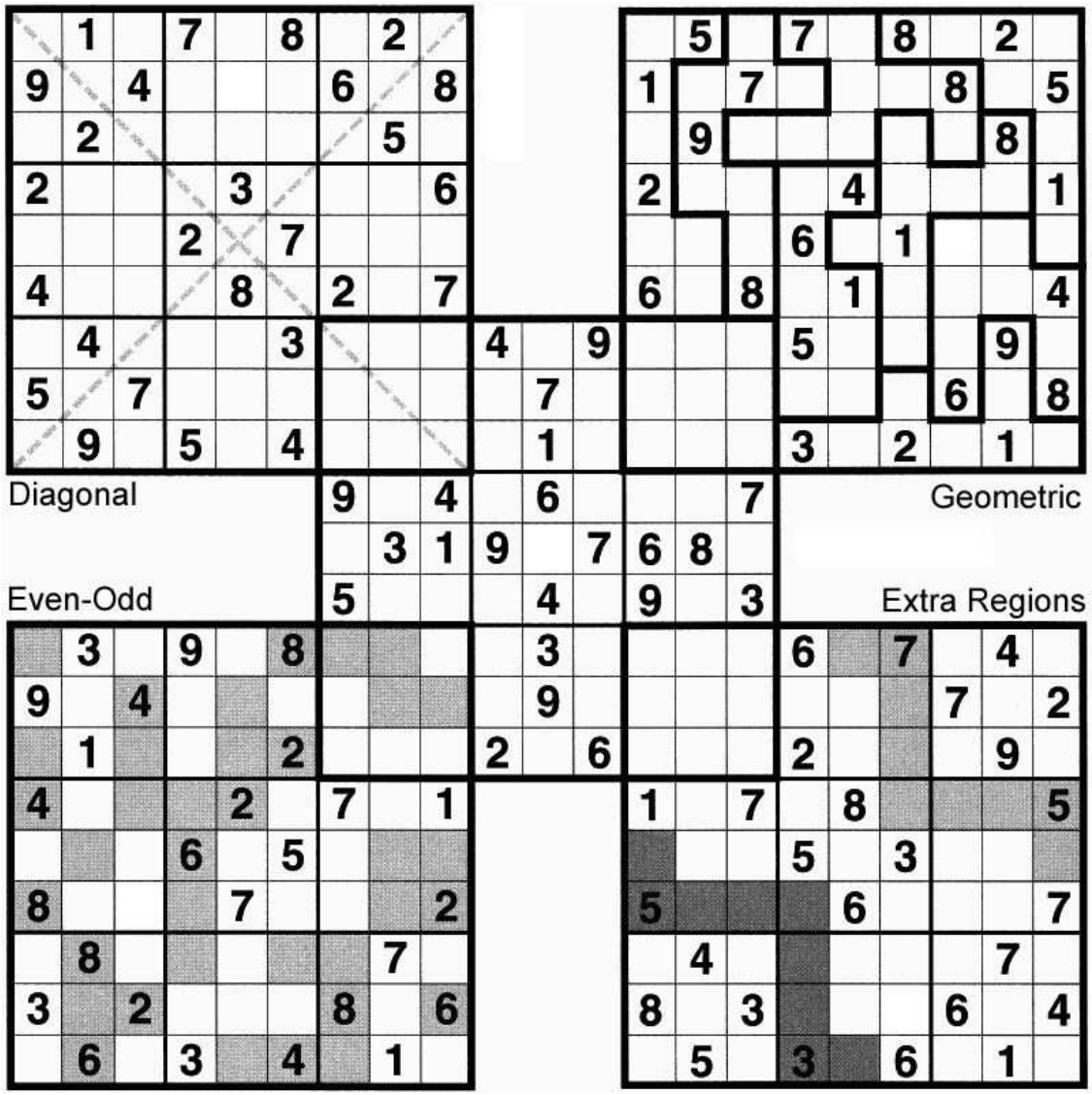

Figure 6.2: Example Samurai Sudoku puzzle.

For this puzzle variation, constraints apply within the scope of a single puzzle and these scopes overlap for a subset of the cells contained within the center puzzle. I manually created 10 Samurai-shaped puzzles to test the SVM learning algorithm in the Sudoku puzzle domain. Each puzzle was created using the template puzzle shown in Figure 6.2. In each instance, the distribution of points used for training and testing varied. Because

the goal of these experiments was to test the ability to determine the scope and not the overall ability to infer an accurate model, cells used for training and testing were filled-in with random numbers (i.e., the generated puzzle is not well-defined).

To generate results for these puzzles, the framework uses the cell locations in one of the five puzzles over which some constraint c_i applies (such as some of the cells along the two diagonals in the top left-hand puzzle in Figure 6.2) as training examples for class labels a. Different cells over which a different constraint $\neg c_i$ applies, such as some of the shaded cells in the bottom left-hand puzzle in Figure 6.2, provide training examples for class label $\neg a$. On average, 13 data points were used to train each class label. The classification task is defined as taking those cells that are part of the scope of either c_i or $\neg c_i$ but were not used to train the SVM model and classifying them as either part of class a (the scope of c_i) or class $\neg a$ (part of the scope of $\neg c_i$).

Table 6.9: Accuracy measures when applying SVMs to Sudoku puzzles.

	Accuracy	**Precision**	**Recall**	**F-Measure**
Linear Kernel	97.64%	100.00%	94.52%	97.18%
Radial Kernel	82.73%	100.00%	67.56%	80.64%

The results presented in Table 6.9 report the average accuracy, precision, recall, and f-measure of the classifications (across the 10 puzzles) as made by the SVM^{light} package. They show that an SVM-based approach can accurately learn the scope of conflicting constraints for the 10 Sudoku puzzles. These results align with those presented for the BID problem in Figure 6.1 and demonstrate the feasibility of using SVMs in the Sudoku puzzle domain. Furthermore, the positive results in both domains help establish the generality of the approach. The reduced accuracy encountered when using the radial

kernel is again attributed to the SVMs representation of the problem in the feature space. To identify the location of cells in their respective puzzles, I use a grid to generate the cell location. For example the cell in the top most corner of the top left-hand puzzle in Figure 6.2 is at the location (1,1) while the bottom most corner of the bottom right-hand puzzle is at the location (21,21). The separations between puzzles are defined as the linear boundaries of the puzzles themselves and a linear kernel better approximates these than the radial one.

6.5 Augmenting Input Data

As the results presented in Section 6.3 show, the input information, and specifically the data points, have a significant impact on the accuracy of the inferred constraint model. In Section 3.2, I presented an iterative algorithm that propagates the generic constraints of a problem class to augment the initial set of data points. The aim of the algorithm is to provide the framework with additional information that it can use to infer constraints applicable in a given problem instance.

As previously stated (see Section 3.2 for a thorough explanation), the benefits of this approach are more readily observed in Sudoku puzzles and as such I evaluate this algorithm on the Sudoku puzzle instances described in Section 6.1.2. My aim is to show that the newly inferred data points increase the number of cells that support each constraint towards a number providing evidence of *strong* inference (see Bayes factor analysis in Section 6.3.2) and they lead to the inference of more representative constraint models.

Table 6.10: Constraint Propagation: average number of new data points.

	Easy				
	Initial	AC	GAC	SAC	All
Basic	27	30	64	78	81
Geometry	28	32	51	78	81
Diagonal	22	22	25	23	25
Even/Odd	15	16	16	16	16

	Medium				
	Initial	AC	GAC	SAC	All
Basic	27	30	74	76	81
Geometry	27	30	71	76	80
Diagonal	22	22	25	23	25
Even/Odd	15	15	15	15	15

	Hard				
	Initial	AC	GAC	SAC	All
Basic	28	32	47	79	80
Geometry	27	31	45	79	80
Diagonal	22	22	26	23	26
Even/Odd	15	15	15	15	15

	Uncategorized				
	Initial	AC	GAC	SAC	All
Magic	9	9	9	9	9

The results showing the number of newly inferred data points using constraint propagation are summarized in Table 6.10. In these tables, the *Initial* column corresponds to the initial number of data points in the problem model, and the remaining numbers correspond to the number of data points after propagation using Arc-Consistency (AC), Generalized Arc-Consistency (GAC), Singleton Arc-Consistency (SAC), and all three techniques combined (All). Importantly, the constraints propagated are the generic Sudoku constraints (All-diff row, column). All reported numbers correspond to averages across the instances for a given puzzle type and difficulty level.

As these results show, GAC and SAC provide more new data points than AC and as a whole the combination of all three methods provides the most newly inferred data points. Furthermore, the *basic* and *geometry* puzzle types greatly benefit from the propagation of the constraints yet the others do not. This is caused by the initial distribution of data points within the puzzles and is most evident in the *magic* puzzle type where only one filled-in cell initially occupies a region. To further evaluate the constraint propagation method, the new sets of data points need to be used to infer the constraint model for all puzzle instances.

Table 6.11: Iterative Propagation: accuracy and completeness of inferred constraints.

			Easy		**Medium**		**Hard**	
	$\|C_G\|$	$\|C_{new}\|$	Comp.	Acc.	Comp.	Acc.	Comp.	Acc.
Basic	2	3	1.0	0.99	1.0	1.0	1.0	0.99
Geometry	2	3	1.0	1.0	1.0	1.0	1.0	0.99
Diagonal	2	4	0.89	1.0	0.89	1.0	0.88	1.0
Even/Odd	2	4	1.0	0.93	1.0	0.94	1.0	0.94
Magic	2	5	(not categorized): Completeness: 0.81, Accuracy: 1.0					

The results shown in Table 6.11 are obtained by augmenting the basic framework used in Section 6.3.2 with the richer set of data points reported in Table 6.10. When compared to the original results presented in Table 6.5, the accuracy of the constraint model was improved for puzzle variations where a significant number of new data points were inferred. Specifically, the *basic* and *geometry* puzzles saw a significant jump in accuracy values because the new data points eliminated most of the erroneously inferred diagonal constraints while still maintaining their prefect level of completeness.

Interestingly, a small number of new data points in the *diagonal* puzzle helped improve the completeness by providing enough information to enable the inference of the diagonal constraint in additional instances. Not surprisingly, the *even/odd* and *magic* puzzle variations saw no significant change in their respective models because the propagation of constraints led to very few new data points. A drop in accuracy for the hard *even/odd* puzzles is caused by the incorrect inference of the diagonal constraint for some problem instances. This incorrect inference is due to the newly inferred data points providing weak support for this constraint.

To quantify the improvement seen when using a richer set of data points, consider the average number of cells that provide support for the applicable constraints in each puzzle type. The *basic* and *geometry* puzzles saw an average increase from 3.11 to 8.87 in the number of cells that provide support. This improvement supports the increase in the number of precise models from an average of 87% to 99% (the average accuracy value of these models). The number of cells providing support correlates with a *very strong* support of evidence in the Bayes factor analysis (see Section 6.3.2) for the likelihood of seeing two numbers that are the same which helps produce more negative support for instances where the diagonal constraint does not apply. The *diagonal* puzzles saw an average increase from 2.44 to 2.89 which contributed to the small increase in the completeness (completeness value) of the inferred models. This is due to the likeliness of *not* seeing two numbers that are the same for the diagonal constraint and thus inferring its applicability. Finally, the *even/odd* and *magic* puzzles saw no increase in average number of cells that provide support and as such saw no improvement in the quality of the inferred model.

As the results in Table 6.11 show, when the number of new data points generated by constraint propagation is significant enough to increase the number of cells that provide support for a given constraint, the framework can generate very accurate models of the given instances. However, puzzle variations that are highly constrained (i.e., even/odd, magic) require propagation of more than just generic constraints. A possible solution is to propagate the set of currently inferred constraints at each iteration of the algorithm. With such an approach, the set of propagated constraints is larger and the reduction of domain values potentially greater. Theoretically, this would lead to more data points than the method used in the framework but it would still need to contend with cases where incorrect inferences are made.

6.6 Automatic Model Inference

In this section, I augment the basic framework with the SVM based approach to learning the scope of constraints and with the constraint propagation approach to generating new data points. I present results that show the framework's overall ability to infer representative models of the problem instance at hand and I show the improvement in solving performance introduced by the inferred models versus generic ones. The experiments in this section are conducted in both the BID problem and Sudoku puzzle domains and the constraint models are automatically generated. These results represent the best inferred models given the input information.

6.6.1 BID Problem

The goal of the experiments in the BID problem domain was to show that the inclusion of additional inference rules and the use of the SVM based approach to learning constraints results in precise models learned for various areas throughout the world. The inclusion of these components allows the framework to solve the more complex set of instances presented in Table 6.2 in Section 6.1.1. I also select an interesting part of El Segundo and demonstrate the effect solving an inferred model has on the performance of the solver and on the quality of the solution produced when compared to solving a generic model of the area.

To evaluate the overall performance of the model generation process, the framework solved the previously introduced instances defined in Table 6.2 and produced the results reported in Table 6.12. In this table, *Area* specifies the area from Table 6.2, *Odd on North/East* corresponds to the parity constraints for E/W and N/S running streets, *Block* corresponds to the block numbering constraint, *Increasing North and East* are the constraints that dictate the direction in which addresses increase for N/S and E/W running streets, and *Acc.* and *Comp.* correspond to the accuracy and completeness of the inferred model calculated over the variables in the problem instance.

The framework's ability to learn scopes allows it to infer models for non-homogenous areas, specifically those found in El Segundo (instances 7(b) and (c)), New Orleans (instance 11), Belgrade (instance 12) and Jakarta (instance 13). In El Segundo, the framework is able to infer accurate models for the entire city where the conflicting constraints

Table 6.12: Automated BID problem instance results.

Area	**Odd On North/East**	**Block** $k = 100$	**Increasing North**	**Increasing East**	**Acc.**	**Comp.**
7. (a)	✓	✓	✓	✓	100.00%	100.00%
7. (b)	✓	✓	Scope 1: ✓ Scope 2: ✓	Scope 1: ✓ Scope 2: ✓	98.99%	100.00%
7. (c)	✓	✓	Scope 1: ✓ Scope 2: ✓	Scope 1: ✓ Scope 2: ×	98.73%	89.90%
8.	✓	✓	✓	×	100.00%	87.50%
9.	✓	✓	✓	✓	100.00%	100.00%
10.	✓	N/A	×	✓	100.00%	76.45%
11.	✓	✓	Scope 1: ✓ Scope 2: ✓	Scope 1: ✓ Scope 2: ✓	97.67%	100.00%
12.	Scope 1: ✓ Scope 2: ✓	N/A	✓	✓	100.00%	100.00%
13.	✓	N/A	✓	✓	100.00%	100.00%

✓ correctly inferred × not inferred N/A not applicable

Increasing East and *Increasing West* co-exist. In New Orleans, N/S running streets become E/W running due to the fan-like configuration of the city. These streets maintain the addressing characteristics of a N/S even though they become E/W running and the framework's ability to learn scopes for the *Increasing East/North* constraints means it can generate a consistent model for the area. This model represents a much larger and more complex area when compared to the one inferred by the basic framework.

In Belgrade, the odd addresses are on the east and north sides of North/South and East/West running streets respectively. However, one street in the area has odd addresses on the west side of the street. Learning scopes for the constraints *Odd on East* and its opposite *Odd on West* means the framework can generate a complete constraint model for Belgrade. This is witnessed by the framework's ability to infer the applicability of both constraints resulting in a completeness measure of 100%. Additionally, the framework is

able to precisely identify the scope of each respective constraint maintaining a 100% level of accuracy. The previous model learned for this area excluded the street to which the conflicting *Odd on West* applied.

For Jakarta, the framework is able to infer a complete constraint model for the area, a task that is impossible using the basic version of the framework. The inconsistencies of the *Odd on North/East* and the *Increasing North/East* constraints throughout the city make the constraint model a difficult one to infer. However, the techniques used in the framework presented in this section allowed it to infer a model with 100% accuracy and completeness while the basic framework would do no better than returning the generic constraint model for the area

The inferred models are slightly off for the three instances with imperfect levels of accuracy. For problem instance 7(b), all constraints are inferred correctly except for the *Increasing East* constraint for a single street on the boundary of two scopes. This is caused by an incorrect classification by the SVM model and the same error occurs for instances 7(c) and 11. To contend with incorrect SVM classification, the framework implements a relaxation of the constraint model when no solution can be found. This relaxation uses backtracking where the classification of constraints for streets with no labels is reverted back to *unknown* and it is available as an optional component in the framework. Because I am dealing with CSPs that are guaranteed solvable, the generation of no solution signifies the inference of an incorrect model. A relaxation of the model may yield a less accurate final solution than one obtained from the tightest model possible. However, as the results presented below show, a solution still represents a significant improvement over ones obtained when solving a generic model.

Additionally, instances 7(c), 8, and 10 are represented by models with imperfect levels of completeness. As was the case with the models generated by the basic framework in Section 6.3.1, this loss in completeness is caused by an inability to correctly infer all applicable constraints due to the lack of information in the data points. Although these models are not as tight as they could be, they are still very precise and much tighter than a generic model for the area. As stated previously, the lack of information can be alleviated by the use of additional online sources which would provide more data points. This set of new points would provide support for the misclassified constraints and a complete model of the area would be generated.

To evaluate the performance gains of using an inferred constraint model over a generic one, I choose the area of El Segundo represented as area 1. (a) throughout the experiments in this chapter. The inferred model represents the complete model for this area and it was generated by the framework using 38 data points found in an online assessor site. Both the inferred and generic models are solved using our customized solver [7]. I validate the claims of improved solution quality and solving efficiency by applying the model to several different regions of this area which vary in size and in phone book completeness.

Table 6.13 describes the properties of the regions on which I ran these experiments. The completeness of the phone book indicates what percentage of the buildings on the map have a corresponding address in the phone book. I created the complete phone books using property-tax data, and the incomplete phone books using real-world phone books. The number of building-address combinations is the number of possible combinations of buildings and phone-book addresses. Note that this number is smaller when the phone book is incomplete than when it is complete.

Table 6.13: BID problem case studies used to evaluate performance.

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

The experimental results are summarized in Tables 6.14 and 6.15, and divided into two categories: (1) the problem model *without* the block-numbering constraint, and (2) the problem model *with* the block-numbering constraint. As our previous work [7] showed, without the block-numbering constraint (called the grid constraint) the BID problem can be solved in polynomial time using a matching algorithm. The existence of the block-numbering constraint forces the BID problem solver to use the backtrack search CSP algorithm. Therefore, to carry out a fair comparison with previous results, I demonstrate the improvement introduced by the inference framework for both the matching and search algorithms.

To further demonstrate the effect inferred constraints have on the solving process, for each algorithm I present results obtained when the Odd on North/East and Increasing North/East constraints (denoted by *orient. constraints* in Tables 6.14 and 6.15) are included and when they are unknown. *Runtime* reports the runtime, in seconds, required to solve the problem, *Domain size* reports the geometric mean of the domain size for a

building, *Runtime reduction* and *Domain reduction* report the factor by which the average domain size and runtime were reduced when using the inferred model.

Table 6.14: BID problem results for case studies without block-numbering constraints.

Matching-Based Solver

	W/o orient. constraints		W/ orienr. constraints			
	Runtime (sec)	Domain size	Runtime (sec)	Domain size	Runtime reduction	Domain reduction
NSeg125-c	90.87	1.95	5.13	1.0	17.71x	1.95x
NSeg125-i	41.25	6.84	2.42	4.68	17.05x	1.46x
NSeg206-c	393.04	2.70	22.28	1.39	17.64x	1.94x
NSeg206-i	192.98	8.75	11.08	5.83	17.42x	1.50x
SSeg131-c	152.29	3.52	9.78	1.90	15.57x	1.85x
SSeg131-i	46.62	13.05	3.04	4.06	15.33x	3.21x
SSeg178-c	379.96	3.59	19.25	1.93	19.74x	1.86x
SSeg178-i	79.24	8.68	5.05	3.41	15.69x	2.55x
				Average	**17.02x**	**2.04x**

Table 6.15: BID problem results for case studies with block-numbering constraints.

CSP Search Solver

	W/o orient. constraints		W/ orient. constraints			
	Runtime (sec)	Domain size	Runtime (sec)	Domain size	Runtime reduction	Domain reduction
NSeg125-c	22397.08	1.22	1962.53	1.0	11.41x	1.22x
NSeg125-i	22929.49	6.11	3987.73	4.18	5.75x	1.46x
NSeg206-c	198169.43	1.21	10786.33	1.0	18.37x	1.21x
NSeg206-i	232035.89	7.91	12900.36	4.99	17.99x	1.59x
SSeg131-c	173565.78	1.56	125011.65	1.41	1.39x	1.11x
SSeg131-i	75332.35	12.56	17169.84	3.92	4.39x	3.20x
SSeg178-c	523100.80	1.41	284342.89	1.31	1.84x	1.08x
SSeg178-i	334240.61	8.24	62646.91	3.23	5.34x	2.55x
				Average	**8.31x**	**1.68x**

As the results show, the use of a customized constraint model greatly improves the performance of the solver. The results for the matching algorithm presented in Table 6.14 show on average a factor of 17 improvement in runtime and a factor of two improvement in domain reduction. As we previously noted [7], every building has the correct label in its domain resulting in a perfect recall measure for the solution. Therefore a factor of two domain reduction results in a large increase in the solution's precision.

For the CSP search algorithm results presented in Table 6.15, the domain reduction is less than that of the matching algorithm results, although there is still a reduction of an average factor of 1.68. This is due to the fact that the search algorithm includes the block-numbering constraint which further constrains the problem and produces small final domain sizes. However, there still exists a significant improvement in the runtime when the inferred constraints are included in the model. On average, there is a factor of 8.31 improvement in runtime, with some scenarios seeing a reduction by a factor as large as 18.37. The performance results for both algorithms show that an inferred model greatly improves the performance of the solver and the quality of the solution.

As the results in this section demonstrate, the framework's ability to automatically infer a representative model of an area along with the improvements introduced when solving the inferred model make the framework a preferred alternative to the process of manually writing models for each problem instance. Even though a domain expert must define the constraints in the library, the inference rules that map to these constraints and the set of conflicting constraints, this is a one-off process whose information can be leveraged across instances. Once the framework has been "set up" to handle the BID problem domain, it can generate hundreds or thousands of constraint models representing

instances across varying areas of the world. The effort required of a domain expert to manually create equivalent models would be overwhelming.

Although the framework doesn't always generate complete models of the areas, the techniques in place to handle errors are robust enough to produce useful solutions even though better ones may exist. Any generated solution is better than one produced when solving a generic model of the problem and the time saved by automatically generating the models makes the inference framework a viable alternative to manual modeling of the BID problem. It is also possible that the inference framework generates an inaccurate constraint model (represented by lower levels of accuracy in my experimental results). In such cases, the model needs to be relaxed so a valid solution(s) will be found. This inaccuracy and subsequent relaxation affect the precision of the produced solution. An accurate but less complete model will result in a smaller domain reduction but as the results in Tables 6.14 and 6.15 show, the precision of the solution is still much greater when compared to the solution generated by solving a generic model.

6.6.2 Sudoku

The performance evaluation of the framework in the Sudoku puzzle domains serves the purpose of enforcing the generality and the domain-independence of the framework. The evaluation demonstrates a Sudoku solver's ability in solving problem instances represented using the inferred constraint model generated by the framework. These results validate the use of the framework for automatically modeling certain puzzle types and they show that this framework can be used as a foundation upon which a generalized Sudoku puzzle solver can be built.

To evaluate the performance of solving the inferred model for Sudoku puzzles, I applied the framework to the set of puzzle instances presented in Section 6.3.2. Additionally, the set of data points for each puzzle was augmented by applying the iterative propagation algorithm discussed in Section 3.2 and evaluated in Section 6.5. The experimental results are presented in Table 6.16. These results report the percentage of problem instances for each puzzle type and difficulty level that could be solved using the inferred constraint model along with the percentage of the solved puzzle instances that returned a single solution. This second metric is important in that I assume all of the puzzle instances are well formed, meaning they have a single solution. This assumption falls out of the class of solvable CSPs that I consider in my research.

Table 6.16: Sudoku problem results with an inferred constraint model.

	Easy		**Medium**		**Hard**	
	% solved	% one sol.	% solved	% one sol.	% solved	% one sol.
Basic	99%	100%	100%	100%	99%	100%
Geometry	100%	100%	100%	100%	99%	100%
Diagonal	100%	57%	100%	56%	100%	53%
Even/Odd	69%	100%	74%	100%	76%	100%
Magic	(¬ categorized): % solved: 100% % one sol.: 10%					

As these results show, the *basic* and *geometry* puzzles are almost all solved, except for two puzzle instances that were represented with an over-constrained model. These two puzzles can be dealt with using a backtracking approach or by increasing the support level required to classify a constraint as applicable (see Section 6.3.2). Specifically, the inferred constraints could be relaxed based on their levels of support until a solution is produced. However, as the results show for these two puzzle types, this approach would

rarely need to be taken. A similar approach could be taken with *even/odd* puzzles where a lower percentage of solved instances is caused by the incorrect inference of the diagonal constraint. Even though the relaxed models may not lead to a well-formed solution, depending on the use of the application this solution could still be useful.

For the *diagonal* and *magic* puzzles, I was able to solve all puzzle instances but a significant number of inferred models were under-constrained, leading to a lower percentage of instances with only one solution. This results from a lack of information in the data points used to infer the model. In general, a precise model is preferred over a complete one, especially when creating a tutor application such as the one described below.

The results presented in Table 6.16 show that the framework performs well enough in this domain to make it a viable backbone for a Sudoku-based application. For example if the framework and solver are applied to a Sudoku tutor application where the application teaches a user how to play Sudoku, filling cells with both unique values and sets of reduced possible values would serve the purpose of helping a user master the art of solving Sudoku puzzles of varying difficulty and type. Additionally, the ability to generate representative models of Sudoku puzzles, as demonstrated in this chapter, enforces the general applicability of the techniques used in the inference framework.

A possible solution to the problem of under-constrained models is to develop an application that uses an interactive process by which the generation and solving of a model is done with help from a user. Given the under-constrained model of the puzzle, the solver would reduce the domains of the cells as best it could and produce the possible values for all cells where singleton domains represent variable-value assignments. The application could then ask the user to assign values to cells containing multiple possible values based

on the partially generated solution. Given these new data points, the application would refine its inferred model and the process would continue until the solver produces a single solution.

Chapter 7

Related Work

My inference framework is related to work in several areas. It builds upon research on constraint programming and problem modeling. Within the modeling research area, there is relevant work in learning constraint networks, qualitative reasoning, and compositional modeling. Furthermore, the problem of building identification can be related to some existing geospatial-specific research. The related work analysis in this chapter is divided into these topics.

7.1 Constraint Programming

Puzzles have historically been an interesting domain for constraint programming (CP). Lauriere puzzles [40] were used to find new constraint solving paradigms. These paradigms have been applied to puzzles such as n-queens [31; 51], the five houses puzzle [4], the progressive party problem [60], and the social golfer problem [32]. This previous work validated the use of CP as an effective paradigm for modeling and solving combinatorial problems.

The PROVERB system [45] served as motivation for the work presented in this dissertation. PROVERB uses a probabilistic CSP approach to solving crossword puzzles. Clue-value pairs infer themes in crossword puzzles by passing them onto sets of expert modules. These clue-value pairs are analogous to the data points, and the expert modules are similar to the inference rules in my framework. Additionally, work on refining floor plans with small changes to eliminate violated design constraints [48] uses a constraint-refinement approach to solving a combinatorial problem. This work, along with PROVERB and the work on the BID problem and Sudoku presented in this dissertation further validate the modeling of combinatorial problems as CSPs.

I identified the BID problem as an interesting new domain for CSPs [47]. My initial findings showed how the BID problem can be solved as a CSP and I built upon this work by developing the full constraint-inference framework. As I have shown, this framework improves the robustness of the building-identification system and makes an online application that identifies buildings throughout the world a more realistic endeavor. Furthermore, the role of a domain expert is significantly reduced when compared to the initial approach.

Sudoku puzzles have also been modeled as a CSP [21; 59]. The work done by Simonis [59] shows how different known and ad-hoc propagation techniques affect the ability to solve basic Sudoku puzzles of varying difficulty. Combining my inference framework with the propagation schemes defined in their work would allow Simonis [59] to expand the set of Sudoku puzzles solved from only the 3x3 basic version to variations such as the ones discussed in this dissertation. Additionally, non-repetitive paths and cycles in graphs have

been used to generate algorithms that create and solve Sudoku puzzles more efficiently [21].

Filtering techniques are very useful in reducing variable domains and making search more tractable. Filtering techniques check for local consistency within the constraint network using constraint-propagation methods. Some of the most widely used filtering techniques include arc-consistency (AC) [46], generalized arc-consistency (GAC) [49], and singleton arc-consistency (SAC) [18]. AC is a method that works on binary constraints and one variable's value is said to be arc-consistent if there exists a support for this value in the second variable. GAC is an extension of AC that operates on non-binary constraints. SAC is a more thorough consistency check where a value is assigned to one variable and this value is propagated throughout the constraint network checking for inconsistencies. As I describe in Section 3.2, I use these filtering techniques in my iterative propagation algorithm to infer more information (such as the instantiation of additional variables) about a given problem instance.

Finally, in concurrent CP [57] some constraints are implications whose right-hand side is added to the constraint store only when the left-hand side is entailed. Languages such as Mozart/Oz [56; 58; 61] and Java[1] support concurrent CP. The methodology behind concurrent CP is similar in nature to my inference framework where inference rules provide support for constraints in the constraint library. However, using support levels to determine the applicability of constraints is more robust than the binary decision used in concurrent CP.

[1] java.sun.com

7.2 Constraint Modeling

There has also been work on using contextualized constraints [13; 24; 29; 30; 39]. Modeling contextualized constraints in grid workflows [24; 30] provides graph-based methods for expressing constraints and shows how multiple graphs (each corresponding to a context) can be combined to create one constraint workflow. This work is different from learning the scope of constraints in my framework in that previous work provides a means to model and merge known contexts (and the constraints within each context) while my work tries to *identify* different scopes (analogous to context in previous work) within a problem instance for all inferred constraints.

Context constraints [13; 29; 39] are used to deal with the state explosion problem in compositional reachability analysis (CRA). This work focuses in the software engineering domain and combines both derived and user-defined context constraints to build reachability graphs. Again it is more focused on deriving constraints and applying them to known contexts. The main goal of the CRA research is to create interface processes for software verification, rather than solving constraint satisfaction problem. However, the definition and application of contexts in a different research area provide insights into concepts that need to be considered when creating a generalized approach to learning scopes in my framework.

Work on learning soft constraints [37] tries to induce local preference functions for temporal constraints using global preference as defined by an expert given a set of full solutions. A key difference between our work and theirs is that we learn the scope of constraints from data points that represent variable-value assignments for only a subset

of the variables in the model (a partial solution). However, the idea of learning local preferences given global ones is interesting and provides us with helpful insight when considering our learning problem. Additionally, by learning the scope of constraints our framework "specializes" constraints and in a sense, converts them to soft constraints with a preference to local applicability (only within the learned scope).

Work on qualitative reasoning (QR) [62] and compositional modeling (CM) [22; 42; 52] focuses on using a behavioral model of a system to predict what should be observed from this system. Subias et al. [62] propose an approach to generate a qualitative model from the data clusters corresponding to classified data. Their approach uses historical data samples as landmarks for qualitative model generation, similar to the use of data points in my framework. The key idea behind compositional modeling is to store model fragments in a library and when building a model given a scenario, which includes a set of assumptions and a set of initial conditions, selecting the appropriate fragments and composing them using constraints that link the output of a fragment to the input of another.

Conceptually, these two approaches are similar. Data points in my framework are similar to the initial conditions specified in the scenario in Compositional Modeling and the library of model fragments is akin to my library of constraints. The difference lies in the approach to model generation. CM creates a brand new model for the given situation while my inference framework specializes a basic model to represent the given problem instance.

Research studying uncertain CSPs [23] is also relevant to my work. My approach to inferring models also takes a previsional approach to finding an accurate model. However,

their assumption that the applicability of constraints is independent does not hold for conflicting constraints. Also, as with all probabilistic CSPs the elicitation of probabilities is a very hard problem. Additionally, these probabilities are assumed to be uniform across all instances, which does not hold in my domains. For example, it is impractical to say the *Parity* constraint applies with 70% probability throughout the world when it can be shown that this constraint is more likely to apply in North America than in Europe.

Finally, research on using model refinement in CP also exists. The CONJURE system [26] reduces a specification of a problem into a set of alternative constraint models of a type supported by current constraint solvers. Much like my framework, this system takes a refinement approach to generating the constraint models, however, it functions at the problem class level while I attempt to refine the model of a particular instance. The CONJURE system uses ESSENCE [25] as its modeling language. This language provides a high level of abstraction by mixing natural language and discrete mathematics in the specification of combinatorial problems.

Other languages have been used to model combinatorial problems. NP-SPEC [11] can represent the specifications of all NP-complete problems and research has shown that the Z language specification is useful for the construction of constraint models [55]. OPL [33] is also a complex language for combinatorial optimization It supports a decoupling of the model (e.g., BID application) from the data (of particular instance). Both formalisms can be used to specify the constraint model inferred by my inference framework.

7.3 Learning Constraints

Recent work in CP modeling aims at automatically learning constraint networks from data. Coletta et al. [14] automatically learns constraint networks from full solutions (both consistent and inconsistent) while Bessière et al. [10] use historical data (solutions previously seen) to learn constraint networks. Machine learning approaches employing SAT-based version-space algorithms [8] and neural nets [38] have also been used to acquire constraint network models. These approaches are ways to model a problem class without having to explicitly define the constraint model.

The key differences between these approaches and our work is that they use full problem solutions to learn the constraint networks while our work requires a small number of known values (a small partial-solution) to infer the constraint model. Additionally, SVMs are best-suited to our class of problems as they handle noise better than version spaces and are less prone to over-fitting when compared to neural networks. However, some of these techniques could be extended and used to learn the constraints that make up our constraint library. Extending the techniques in this branch of research to support the types of constraints I am working with is something I propose as future work (see Section 8.4).

Additional work in using machine learning to learn constraints such as the work by Lallouet et al. [38] employs machine learning techniques to learn open constraints as well as their propagators. Furthermore, Colton et al. [15] finds redundant constraints for quasigroups and reformulate basic constraint models of these groups to improve search. Finally, Cheng et al. [12] show how ad-hoc global CASE constraints can be customized to

construct various constraint models in the STILL-LIFE game. These three papers provide insight into optimizing the inferred model by incorporating different types of constraints (i.e., redundant, CASE). Some of these techniques can be applied to the model generated by my framework and could lead to a more efficient problem-solving process.

7.4 Geospatial Integration and Reasoning

The following work is specifically related to the building identification problem. The work done by Bakshi et al. [5] presents methods to accurately geocode addresses using publicly available data sources. The authors present two different approaches that can be used to improve traditional geocoders. The end result of this work is accurate latitude and longitude coordinates for buildings in a given area. This work also uses online sources to improve the accuracy of building labels. The authors' goal is to precisely identify the location of buildings in a satellite image, which is different from my goal of providing a set of labels to buildings in an image. Furthermore, this work assumes that the sources used to identify all buildings in an image are complete (contain all of the buildings for a given area). This may be a valid assumption to make when considering property tax websites, however such sources are not available for most areas of the world. Therefore, this approach may not be universally applicable.

There has also been work done in identifying buildings in satellite imagery and merging geospatial databases using computer vision approaches, as seen in [2; 1; 20]. While some of the goals in this work are similar (identifying objects in images), the work is more focused on the actual detection of buildings in the images. This varies from my goal of

labeling and reasoning over specific buildings in images. This work could serve as a source of information for my inference framework.

Chapter 8

Discussion and Future Work

In this chapter I first summarize the contributions of my dissertation. Next, I discuss some application areas that would benefit from using my research and highlight some limitations of my approach. Finally, I list possible directions for future work.

8.1 Contributions

The key contributions of this work are as follows:

- A general framework for automatic model generation that supports varying problem instances within a problem class.
- The inference of a constraint model based on the problem instance at hand.
- The ability to deal with noisy data and incorrect inferences using support levels.
- The use of Support Vector Machines to automatically learn the scope of applicable constraints.

Each contribution is described in more detail below.

The main contribution is a generalized constraint-inference framework for constraint satisfaction problems. This framework reduces the burden placed on a domain expert by using information found in a problem instance to infer the most representative constraint model given the input information. This approach limits the involvement of the expert to the definition of the constraint library and inference rules, a task that is more manageable than having to define constraint models for all possible problem instances. Although a domain expert must "setup" a domain before it can be supported by the inference framework, the knowledge introduced by the expert is leveraged over time resulting in accurate constraint models.

In addition to the generalized framework, I identified solvable CSP problems as a class of problems amenable to modeling by an iterative process of model refinement and model solving. This class of problems supports a set of assumptions, such as the existence of a solution for all instances, that allows for the use of new techniques such as the combination of constraint propagation and model relaxation for model refinement. Further, the use of Support Vector Machines (SVMs) to automatically learn the scope of a constraint is a novel approach to handling non-homogenous problem instances. This approach is domain independent and only assumes that an instance's input information can be represented in the feature space. It can be applied to any problem where an instance must be separated to handle contentions between applicable constraints without creating multiple disjunctive constraint models.

Specific to the BID problem and Sudoku puzzles domain, I have created a set of inference rules and a library of constraints for each domain. As such, applications specific to

one of these two domains can use my framework to efficiently solve instances with minimal domain expert involvement. The flexibility of the framework makes it easy to plug-in different inference approaches and to manage different methods used to enhance the inference process, such as augmenting the input information and propagating constraints when instantiating a model. Finally, representing instances using the XCSP schema provides the research community with problem sets that can be used to further the research in various areas of Constraint Programming.

8.2 Application Areas

Although only the BID problem and Sudoku puzzles are presented throughout this dissertation to illustrate the concepts and mechanisms used, I believe this work can be applied to a general class of problems where a solution can be guaranteed. Because the techniques used in the framework are general, they can also be applied to other domains. The framework can be applied to any problem domain where commonalities existing across all problem instances can be leveraged and where variations in the instances makes it infeasible to generate and store all possible constraint models beforehand. Two such domains are machine translation and genealogical trees and I briefly discuss how my framework could be applied to these domains.

8.2.1 Machine translation

Syntactic machine translation [27; 53] is a domain where syntactic transfer rules are derived from bilingual corpa and used to translate documents from a base to a target language. To apply my inference framework in this domain, the text in the document

being translated could be used as input information specific to a problem instance. This text could determine a scope (what type of document it is) and what constraints apply. Because the subject matter contained in a document varies, the set of applicable translation rules (constraints) may also vary across documents.

By generating a constraint model specific to a particular document, a translation engine could be optimized at run-time based on the the deduced rule set (and the rules' scope) contained in the inferred model. This model would increase the efficiency of a generalized translation engine for multiple bilingual translations. Such a translation engine would be implemented using a general framework but the specificity of the inferred models and the ability to support new translations by simply adding new constraints and rules could increase its adoption in various settings. A constraint would represent grammatical conventions such as the modification of pronouns by an adjective and inference rules would specify situations that support a given convention, for example an adjective comes after a pronoun when the above mentioned convention is supported.

8.2.2 Genealogical Trees

Another example CSP problem to which I can apply my framework is the problem of constructing genealogical trees which is guaranteed to have a solution as dictated by the real world. The task that would need to be performed is, given a set of people, generate a representative model of the relationships between the people such that a genealogical tree representing these relationships can be constructed. Even though a set of generic constraints will apply to all problem instances (e.g., biological children must be younger than parents), cultural influences introduce constraints which only apply to certain instances.

For example the accepted practice of polygamy in certain cultures or adoption can introduce instance-specific characteristics and cross-cultural family trees may be represented by a non-homogenous set of characteristics.

The known relationships between family members could be used to infer the set of applicable constraints and their scopes for a given problem instance. Imagine the scenario where we would like to build an application that, when provided with a set of people (data points) between which we know some of the relationships, constructs the corresponding genealogical tree. Using my framework as the foundation for this application allows it be more robust to handling the cultural differences seen throughout the world and negates the need to determine all possible familial combinations a priori.

8.3 Limitations

In this section I describe some of the assumptions and limitations specific to the inference framework. As discussed in Section 1.1.1 the framework models only problems for which a solution is guaranteed to exist. The framework exploits this property when relaxing the model due to incorrect inferences or when it uses constraint propagation to infer new input information. Additionally, when determining if a problem instance contains multiple scopes, the framework assumes it a set of conflicting constraints has been predetermined by a domain expert. The problem of finding all pairs of conflicting constraints in the constraint library is exponential, hence the framework assumes these pairs are provided as a means to increase its efficiency and lower its complexity.

In the BID problem, two key assumptions are made. First the framework assumes it has access to vector data for the given area of interest. While the data points may come from varying data sources, road network information must be available as it is used to extract key pieces of information such as the set of potential streets for all buildings. The second assumption being made is that a building extraction tool is available to the framework. Such a tool is able to extract the buildings from satellite imagery and provide the framework with their location within the area. While I recognize that this is a difficult computer vision problem, solving it is beyond the scope of this thesis. Accordingly, the reported experiments for the BID problem use a manual process to extract the buildings in each satellite image.

8.4 Directions for Future Work

I believe there are some future directions for this work that will allow the techniques I developed to be applied more broadly. These directions include learning constraints to automatically populate the constraint library and using multi-class SVMs to determine the scope of constraints. Additional work can be done to improve the performance of the framework in the BID problem and Sudoku puzzle domains. I discuss these possible future directions in the following sections.

8.4.1 Learning Inference Rules

To further reduce the role played by a domain expert, the framework could learn the inference rules and map them to the constraints in the constraint library. One possible learning approach involves the use of agglomerative clustering [3]. This process would

work as follows. First, begin with n^r sets of n data points where r corresponds to the arity of the constraints we are trying to learn. Since all data points have a common set of features, we can generate training data for the clustering algorithm by combining the data points in each set to generate a single feature vector representing the set. When comparing data points, we will limit ourselves to the set of predicate expressions $preds = \{<, >, ==, <=, >=\}$.

Each feature vector is constructed by taking a single attribute a and the n values for that feature in the set and evaluating all predicate expressions in $preds$ to generate $|preds|$ attribute-value pairs. The size of this feature vector would be $A \times |preds|$ where A is the number of attributes that define the data points. Once we have constructed all feature vectors, we perform agglomerative clustering on these vectors using cosine similarity as the distance metric. Even though cosine similarity works only if all attribute values are binary (0/1), this is a natural result when using the set of predicate expressions in $preds$. The algorithm ends by considering clusters greater than some size c, where c must be manually set or the optimal size is determined through experimentation. The cluster center of all relevant clusters is a logical expression corresponding to an inference rule.

Once the set of inference rules have been determined, they must be mapped to the constraints in the library. This process can be done in one of two ways: (1) the domain expert can define mappings that can be used to automatically map rules to constraints, or (2) the domain expert can manually map the learned rules to constraints. The mapping of rules to constraints is not a straightforward process because the rules are defined in the *feature* space, while constraints exist in the *variable* space. Very preliminary results

validate this process of learning inference rules but more intelligent ways of generating feature vectors and handling noisy comparisons need to be determined.

8.4.2 Enhancing the Learning of a Constraint's Scope

The current approach to learning the scopes requires a domain expert to define sets of conflicting constraints. These conflicts are assumed to be binary. However, situations may arise in domains where conflicts may be ternary and beyond. To support such cases, the machine learning piece of the scope learning component in the framework needs to be enhanced to support multi-class classification. By supporting multiple classes, the framework could support more complex interactions between constraints and in turn could handle increasingly complex problem instances.

One possible solution is to replace the binary implementation of Support Vector Machines (SVMs) currently used in the framework with an implementation of multi-class SVMs [17; 35; 63]. Using multi-class SVMs would maintain the current algorithmic process and it would allow the definition of non-binary sets of conflicting constraints. As such, more complex constraint models could be learned, where the interaction between constraints is more sophisticated than what is currently supported.

8.4.3 BID Problem

Future work with respect to the BID problem can be divided into two main categories: performance improvement and application building. Specifically, to improve the performance of the framework for the BID problem, the generic constraint-propagation algorithm could be applied to augment the set of input information. The current implementation of the filtering techniques used to propagate the generic constraint model does

not support some of the constraints in the BID problem domain. Hence the propagation methods need to be updated to propagate these different constraint types.

Additionally, to fully test the performance of the framework for the BID problem, an online application for assigning addresses to buildings in satellite imagery can be built. This application would take as input the coordinates of an area of interest and return the corresponding satellite imagery with the buildings labeled with potential addresses. To make this application a reality, all of the required information would need to be automatically retrieved. Specifically, tools such as a computer vision method for identifying buildings in satellite imagery and wrappers to extract information from public sources around the world would need to be incorporated into the application along with the inference framework. This is an application building task but one that could be useful to many people.

8.4.4 Sudoku Puzzles

Future work in the Sudoku puzzle domain involves building an XML schema using the XCSP standard for Sudoku that would allow the framework to automatically instantiate the inferred constraint model. It would also provide the CP research community with additional problem instances for solver competitions. Providing a standard format in which to represent Sudoku puzzle instances could also lead to the creation of a test set of instances that could be used in submitted papers when evaluating new approaches to solving combinatorial problems. Using a base test set would make it easier to compare techniques and evaluate their respective performance when compared to the state of the art.

Bibliography

[1] Peggy Agouris, Kate Beard, Georgios Mountrakis, and Anthony Stefanidis. Capturing and modeling geographic object change: A spatio-temporal gazeteer framework. *Photogrammetric Engineering and Remote Sensing*, 66(10):1224–1250, 2000.

[2] Peggy Agouris and Anthony Stefanidis. Integration of photogrammetric and geographic databases. *International Archives of Photogrammetry and Remote Sensing, ISPRS XVIIIth Congress*, 31:24–29, 1996.

[3] Michael R. Anderberg. *Cluster Analysis for Applications*. Academic Press, Inc., 1973.

[4] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[5] Rahul Bakshi, Craig A. Knoblock, and Snehal Thakkar. Exploiting online sources to accurately geocode addresses. In *Proceedings of the 12th ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS'04)*, pages 194–203, 2004.

[6] Kenneth M. Bayer, Martin Michalowski, Berthe Y. Choueiry, and Craig A. Knoblock. Reformulating constraint satisfaction problems to improve scalability. In *Proceedings of the 7th Symposium on Abstraction, Reformulation and Approximation (SARA-07)*, pages 64–79, 2007.

[7] Kenneth M. Bayer, Martin Michalowski, Berthe Y. Choueiry, and Craig A. Knoblock. Reformulating CSPs for Scalability with Application to Geospatial Reasoning. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP-07)*, pages 164–179, 2007.

[8] Christian Bessière, Remi Coletta, Frederic Koriche, and Barry O'Sullivan. A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems. In *Proceedings of ECML'05*, pages 23–34., Porto, Portugal, 2005.

[9] Christian Bessière, Pedro Meseguer, Eugene C. Freuder, and Javier Larrosa. On Forward Checking for Non-binary Constraint Satisfaction. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP-99)*, pages 88–102, 1999.

[10] Christian Bessière, Joel Quinqueton, and Gilles Raymond. Mining Historical Data to Build Constraint Viewpoints. In *Proceedings of CP-06 Workshop on Modelling and Reformulation*, pages 1–16, 2006.

[11] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. Np-spec: An executable specification language for solving all problems in np. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL98)*, pages 16–30, 1998.

[12] Kenil C. K. Cheng and Roland H. C. Yap. Applying ad-hoc global constraints with the case constraint to still-life. *Constraints*, 11(2-3):91–114, 2006.

[13] Shing Chi Cheung and Jeff Kramer. Context Constraints for Compositional Reachability Analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, 1996.

[14] Remi Coletta, Christian Bessière, Barry O'Sullivan, Eugene C. Freuder, Sarah O'Connell, and Joel Quinqueton. Semi-automatic Modeling by Constraint Acquisition. In *Proceedings of CP-03*, pages 111–124, 2003.

[15] Simon Colton and Ian Miguel. Constraint generation via automated theory formation. *Lecture Notes in Computer Science*, 2239:575–579, 2001.

[16] Koby Crammer and Yoram Singer. The algorithmic implementation of multiclass kernel-based vector machines. Technical report, School of Computer Science and Engineering, Hebrew University, 2001.

[17] Koby Crammer and Yoram Singer. On the algorithmic implementation of multi-class svms. *Journal of Machine Learning Research*, 2:265–292, 2001.

[18] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997*, volume 1, pages 412–417, 1997.

[19] Rena Dechter. *Constraint Processing*. The MIT Press, 1989.

[20] Peter Doucette, Peggy Agouris, Maohamad Musavi, and Anthony Stefanidis. Automated extraction of linear features from aerial imagery using kohonen learning and gis data. *Lecture Notes in Computer Science*, 1737:20–33, 1999.

[21] David Eppstein. Nonrepetitive paths and cycles in graphs with application to sudoku. ACM Computing Research Repository, 2005.

[22] Brian Falkenhainer and Kenneth D. Forbus. Compositional modeling: Finding the right model for the job. *Artificial Intelligence*, 51(1-3):95–143, 1991.

[23] Hélène Fargier and Jerome Lang. Uncertainty in Constraint Satisfaction Problems: a Probabilistic Approach. In *Proceedings of the Second European Conferences on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU-93)*, pages 97–104, 1993.

[24] Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao. Chimera: a virtual data system for representing, querying, and automating data derivation. In *Proceedings of 14th International Conference on Scientific and Statistical Database Management*, pages 37–46, 2002.

[25] Alan M. Frisch, Matthew Grum, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. The design of essence: A constraint language for specifying combinatorial problems. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 80–87, 2007.

[26] Alan M. Frisch, Chris Jefferson, Bernadette Martinez Hernandez, and Ian Miguel. The Rules of Constraint Modelling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 109–116, 2005.

[27] Michel Galley, Jonathan Graehl, Kevin Knight, Daniel Marcu, Steve DeNeefe, Wei Wang, and Ignacio Thayer. Scalable inference and training of context-rich syntactic translation models. In *Proceedings of COLING/ACL2006*, pages 961–968, 2006.

[28] Michael R. Genesereth, Arthur M. Keller, and Oliver M. Duschka. Infomaster: An information integration system. In *Proceedings of ACM SIGMOD-97*, 1997.

[29] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In *Proceedings of the 2nd International Workshop on Computer Aided Verification (CAV-90*, pages 186–196. Springer-Verlag, 1991.

[30] Greg Graham, Anzar Afaq, David Evans, Gerald Guglielmo, Eric Wicklund, and Peter Love. Contextual Constraint Modeling in Grid Application Workflows. *Concurr. Comput. : Pract. Exper.*, 18(10):1277–1292, 2006.

[31] Robert Haralick and Gordon Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[32] Warwick Harvey. The Fully Social Golfer Problem. In *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'03)*, pages 75–85, 2003.

[33] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, 1999.

[34] Thorsten Joachims. Making large-scale support vector machine learning practical. In *Advances in Kernel Methods: Support Vector Machines*. MIT Press, Cambridge, MA, 1999.

[35] Thorsten Joachims. Making large-scale support vector machine learning practical. pages 169–184, 1999.

[36] Thorsten Joachims. *Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms*. Kluwer Academic Publishers, 2002.

[37] Lina Khatib, Paul H. Morris, Robert Morris, Francesca Rossi, Alessandro Sperduti, and Kristen Brent Venable. Solving and learning a tractable class of soft temporal constraints: Theoretical and experimental results. *AI Communications*, 20(3):181–209, 2007.

[38] Arnaud Lallouet and Andrei Legtchenko. Consistency for Partially Defined Constraints. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '05)*, pages 118–125, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Kim G. Larsen and Robin Milner. Verifying a protocol using relativized bisimulation. In *14th International Colloquium on Automata, languages and programming*, pages 126–135. Springer-Verlag, 1987.

[40] Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[41] Alon Y. Levy. Logic-based techniques in data integration. In *Logic Based Artificial Intelligence*, pages 575–595. Kluwer Publishers, 2000.

[42] Alon Y. Levy, Yumi Iwasaki, and Richard Fikes. Automated model selection for simulation based on relevance reasoning. *Artificial Intelligence*, 96(2):351–394, 1997.

[43] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the Twenty-second International Conference on Very Large Data Bases (VLDB-96)*, pages 251–262, Bombay, India, 1996.

[44] Chungan Lin and Ramakant Nevatia. Building detection and description from a single intensity images. *Computer Vision and Image Understanding Journal*, 72(2):101–121, November 1998.

[45] Michael L. Littman, Greg A. Keim, and Noam Shazeer. A probabilistic approach to solving crossword puzzles. *Artificial Intelligence*, 134(1-2):23–55, 2002.

[46] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, pages 99–118, 1977.

[47] Martin Michalowski and Craig A. Knoblock. A Constraint Satisfaction Approach to Geospatial Reasoning. In *Proceedings of AAAI-05*, pages 423–429, 2005.

[48] Michael D. Moffit, Aaron Ng, Igor Markov, and Martha E. Pollack. Constraint-driven floorplan repair. In *Proceedings of the Design Automation Conference (DAC-06)*, pages 1103–1108, 2006.

[49] Roger Mohr and Gérald Masini. Good Old Discrete Relaxation. In *European Conference on Artificial Intelligence (ECAI-88)*, pages 651–656, Munich, W. Germany, 1988.

[50] Katharina Morik, Peter Brockhausen, and Thorsten Joachims. Combining statistical learning with a knowledge-based approach - a case study in intensive care monitoring. In *Proceedings of the 16th International Conference on Machine Learning (ICML-99)*, pages 268–277, 1999.

[51] Bernard A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert: Intelligent Systems and Their Applications*, 5(3):16–23, 1990.

[52] Pandurang Nayak. Causal approximations. *Artificial Intelligence*, 70:277–334, 1994.

[53] Franz Josef Och and Hermann Ney. The alignment template approach to statistical machine translation. *Computational Linguistics*, 30(4):417–449, 2004.

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

[55] Gerrit Renker and Hatem Ahriz. Building models through formal specification. In *Proceedings of the First International conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization (CPAIOR04)*, pages 395–401, 2004.

[56] Peter Van Roy, editor. *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.

[57] Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[58] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 2002.

[59] Helmut Simonis. Sudoku as a constraint problem. In *Proceedings of CP 2005*, pages 13–27, 2005.

[60] Barbara M. Smith, Sally C. Brailsford, Peter M. Hubbard, and H. Paul Williams. The progressive party problem: Integer linear programming and constraint programming compared. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 36–52, 1995.

[61] Gert Smolka. Concurrent constraint programming based on functional programming. In Chris Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 1381, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.

[62] Audine Subias and Louise Travé-Massuyès. Discriminating qualitative model generation from classified data. In *20th International Workshop on Qualitative Reasoning (QR-06)*, pages 129–136, 2006.

[63] Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *Proceedings of the Twenty-First International Conference on Machine Learning (ICML04)*, pages 104–112, 2004.

[64] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, 1995.

[65] Takayuki Yato. Complexity and Completeness of Finding Another Solutions and its Application to Puzzles. Master's thesis, The University of Tokyo, January 2003.

Appendix A

Constraint Inference Rules

This appendix describes the constraint inference rules for the BID problem and Sudoku. I show all of the rules that map to each constraint in the constraint library for a given problem domain.

A.1 BID Problem Inference Rules

These rules map to the constraints in the constraint library specific to the BID problem.

A.1.1 *Odd on North* rules

The *Odd on North* constraint ensures that all buildings on the North side along East/West running streets are assigned an odd address number. The rules that map to the *Odd on North* (a more specific *Parity* constraint) constraint are as follows:

```
1. (sType(B1) = sType(B2) = EW) & (mod2(B1) = mod2 (B2) = 1)
    & (sSide(B1) = sSide(B2) = N)
2. (sType(B1) = sType(B2) = EW) & (mod2(B1) = mod2 (B2) = 0)
    & (sSide(B1) = sSide(B2) = S)
3. (sType(B1) = sType(B2) = EW) & (mod2(B1) = 1) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B1) = N)
4. (sType(B1) = sType(B2) = EW) & (mod2(B2) = 1) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B2) = N)
5. (sType(B1) = sType(B2) = EW) & (mod2(B1) = 0) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B1) = S)
6. (sType(B1) = sType(B2) = EW) & (mod2(B2) = 0) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B2) = S)
```

A.1.2 ¬*Odd on North* rules

The ¬*Odd on North* constraint ensures that all buildings on the South side along East/West running streets are assigned an odd address number. The rules that map to the ¬*Odd on North* (a more specific *Parity* constraint) constraint are as follows:

```
1. (sType(B1) = sType(B2) = EW) & (mod2(B1) = mod2 (B2) = 0)
```

```
    & (sSide(B1) = sSide(B2) = N)
2. (sType(B1) = sType(B2) = EW) & (mod2(B1) = mod2 (B2) = 1)
    & (sSide(B1) = sSide(B2) = S)
3. (sType(B1) = sType(B2) = EW) & (mod2(B1) = 0) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B1) = N)
4. (sType(B1) = sType(B2) = EW) & (mod2(B2) = 0) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B2) = N)
5. (sType(B1) = sType(B2) = EW) & (mod2(B1) = 1) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B1) = S)
6. (sType(B1) = sType(B2) = EW) & (mod2(B2) = 1) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B2) = S)
```

A.1.3 *Odd on East* rules

The *Odd on East* constraint ensures that all buildings on the East side along North/South running streets are assigned an odd address number. The rules that map to the *Odd on East* (a more specific *Parity* constraint) constraint are as follows:

```
1. (sType(B1) = sType(B2) = NS) & (mod2(B1) = mod2 (B2) = 1)
    & (sSide(B1) = sSide(B2) = E)
2. (sType(B1) = sType(B2) = NS) & (mod2(B1) = mod2 (B2) = 0)
    & (sSide(B1) = sSide(B2) = W)
3. (sType(B1) = sType(B2) = NS) & (mod2(B1) = 1) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B1) = E)
4. (sType(B1) = sType(B2) = NS) & (mod2(B2) = 1) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B2) = E)
5. (sType(B1) = sType(B2) = NS) & (mod2(B1) = 0) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B1) = W)
6. (sType(B1) = sType(B2) = NS) & (mod2(B2) = 0) & (mod2(B1) != mod2 (B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B2) = W)
```

A.1.4 ¬*Odd on East* rules

The ¬*Odd on East* constraint ensures that all buildings on the West side along North/South running streets are assigned an odd address number. The rules that map to the ¬*Odd on East* (a more specific *Parity* constraint) constraint are as follows:

```
1. (sType(B1) = sType(B2) = NS) & (mod2(B1) = mod2(B2) = 0)
    & (sSide(B1) = sSide(B2) = E)
2. (sType(B1) = sType(B2) = NS) & (mod2(B1) = mod2(B2) = 1)
    & (sSide(B1) = sSide(B2) = W)
3. (sType(B1) = sType(B2) = NS) & (mod2(B1) = 0) & (mod2(B1) != mod2(B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B1) = E)
4. (sType(B1) = sType(B2) = NS) & (mod2(B2) = 0) & (mod2(B1) != mod2(B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B2) = E)
5. (sType(B1) = sType(B2) = NS) & (mod2(B1) = 1) & (mod2(B1) != mod2(B2))
```

```
    & (sSide(B1) != sSide(B2)) & (sSide(B1) = W)
6. (sType(B1) = sType(B2) = NS) & (mod2(B2) = 1) & (mod2(B1) != mod2(B2))
    & (sSide(B1) != sSide(B2)) & (sSide(B2) = W)
```

A.1.5 *Increasing North* rules

The *Increasing North* constraint ensures that the address of all buildings along a North/South running streets increase when heading in the North direction. The rules that map to the *Increasing North* constraint are as follows:

```
1. (sType(B1) = sType(B2) = NS) & (addr(B1) > addr(B2))
    & (sSide(B1) = sSide(B2)) & (lat(B1) > lat(B2))
2. (sType(B1) = sType(B2) = NS) & (addr(B1) < addr(B2))
    & (sSide(B1) = sSide(B2)) & (lat(B1) < lat(B2))
More general (not limited to same side of the street)
3. (sType(B1) = sType(B2) = NS) & (addr(B1) > addr(B2))
    & (lat(B1) > lat(B2))
4. (sType(B1) = sType(B2) = NS) & (addr(B1) < addr(B2))
    & (lat(B1) < lat(B2))
Across blocks
5. (sType(B1) = sType(B2) = NS) & (addr(B1) > addr(B2))
    & (lat(B1) > lat(B2)) & ((lon(B1) > lon(B2)) || (lon(B1) < lon(B2)))
6. (sType(B1) = sType(B2) = NS) & (addr(B1) < addr(B2))
    & (lat(B1) < lat(B2)) & ((lon(B1) > lon(B2)) || (lon(B1) < lon(B2)))
```

A.1.6 *Increasing South* rules

The *Increasing South* constraint ensures that the address of all buildings along a North/South running streets increase when heading in the South direction. The rules that map to the *Increasing South* constraint are as follows:

```
1. (sType(B1) = sType(B2) = NS) & (addr(B1) > addr(B2))
    & (sSide(B1) = sSide(B2)) & (lat(B1) < lat(B2))
2. (sType(B1) = sType(B2) = NS) & (addr(B1) < addr(B2))
    & (sSide(B1) = sSide(B2)) & (lat(B1) > lat(B2))
More general (not limited to same side of the street)
3. (sType(B1) = sType(B2) = NS) & (addr(B1) > addr(B2))
    & (lat(B1) < lat(B2))
4. (sType(B1) = sType(B2) = NS) & (addr(B1) < addr(B2))
    & (lat(B1) > lat(B2))
Across blocks
5. (sType(B1) = sType(B2) = NS) & (addr(B1) > addr(B2))
    & (lat(B1) < lat(B2)) & ((lon(B1) > lon(B2)) || (lon(B1) < lon(B2)))
6. (sType(B1) = sType(B2) = NS) & (addr(B1) < addr(B2))
    & (lat(B1) > lat(B2)) & ((lon(B1) > lon(B2)) || (lon(B1) < lon(B2)))
```

A.1.7 *Increasing East* rules

The *Increasing East* constraint ensures that the address of all buildings along a East/West running streets increase when heading in the East direction. The rules that map to the *Increasing East* constraint are as follows:

```
1. (sType(B1) = sType(B2) = EW) & (addr(B1) > addr(B2))
    & (sSide(B1) = sSide(B2)) & (lon(B1) > lon(B2))
2. (sType(B1) = sType(B2) = EW) & (addr(B1) < addr(B2))
    & (sSide(B1) = sSide(B2)) & (lon(B1) < lon(B2))
More general (not limited to same side of the street)
3. (sType(B1) = sType(B2) = EW) & (addr(B1) > addr(B2))
    & (lon(B1) > lon(B2))
4. (sType(B1) = sType(B2) = EW) & (addr(B1) < addr(B2))
    & (lon(B1) < lon(B2))
Across blocks
5. (sType(B1) = sType(B2) = EW) & (addr(B1) > addr(B2))
    & (lon(B1) > lon(B2)) & ((lat(B1) > lat(B2)) || (lat(B1) < lat(B2)))
6. (sType(B1) = sType(B2) = EW) & (addr(B1) < addr(B2))
    & (lon(B1) < lon(B2)) & ((lat(B1) > lat(B2)) || (lat(B1) < lat(B2)))
```

A.1.8 *Increasing West* rules

The *Increasing West* constraint ensures that the address of all buildings along a East/West running streets increase when heading in the West direction. The rules that map to the *Increasing West* constraint are as follows:

```
1. (sType(B1) = sType(B2) = EW) & (addr(B1) > addr(B2))
    & (sSide(B1) = sSide(B2)) & (lon(B1) < lon(B2))
2. (sType(B1) = sType(B2) = EW) & (addr(B1) < addr(B2))
    & (sSide(B1) = sSide(B2)) & (lon(B1) > lon(B2))
More general (not limited to same side of the street)
3. (sType(B1) = sType(B2) = EW) & (addr(B1) > addr(B2))
    & (lon(B1) < lon(B2))
4. (sType(B1) = sType(B2) = EW) & (addr(B1) < addr(B2))
    & (lon(B1) > lon(B2))
Across blocks
5. (sType(B1) = sType(B2) = EW) & (addr(B1) > addr(B2))
    & (lon(B1) < lon(B2)) & ((lat(B1) > lat(B2)) || (lat(B1) < lat(B2)))
6. (sType(B1) = sType(B2) = EW) & (addr(B1) < addr(B2))
    & (lon(B1) > lon(B2)) & ((lat(B1) > lat(B2)) || (lat(B1) < lat(B2)))
```

A.1.9 *K-Block Numbering* rules

The *K-Block Numbering* constraint ensures that the addresses of buildings across city blocks increase by an increment of K. The rules that map to the *K-Block Numbering* constraint are as follows where the *block(x)* value is generated from a "virtual" grid:

```
Specific to a given street
1. (sName(B1) = sName(B2)) & (modK(B1) - modK(B2) = block(B1) - block(B2))
Across streets
2. (sType(B1) = sType(B2)) & (modK(B1) - modK(B2) = block(B1) - block(B2))
```

A.1.10 *San Francisco Block Numbering* rules

The *San Francisco Block Numbering* constraint ensures that the addresses of buildings across city blocks increase by an increment of K. This differs from the *K-Block Numbering* constraint in that the blocks of parallel streets do not need to align. The rules that map to the *San Francisco Block Numbering* constraint are as follows:

```
1. ((sName(B1) = sName(B2)) & (modK(B1) - modK(B2) = block(B1) - block(B2)))
    & ((sType(B1) = sType(B3)) & (modK(B1) - modK(B3) != block(B1) - block(B3)))
2. ((sName(B1) = sName(B3)) & (modK(B1) - modK(B3) = block(B1) - block(B3)))
    & ((sType(B1) = sType(B2)) & (modK(B1) - modK(B2) != block(B1) - block(B2)))
3. ((sName(B2) = sName(B3)) & (modK(B2) - modK(B3) = block(B2) - block(B3)))
    & ((sType(B2) = sType(B1)) & (modK(B2) - modK(B1) != block(B2) - block(B1)))
```

A.1.11 *C-Continuous Numbering* rule

The *C-Continuous Numbering* constraint ensures that the addresses adjacent to each other on a given street increase by a constant C. The rule that maps to the *C-Continous Numbering* constraint is as follows where the *Order(x)* comes from the order of a building along the given street for the given side:

```
1. (sName(B1) = sName(B2)) & (sSide(B1) = sSide(B2))
    & (order(B1) - order(B2) * C = addr(B1) - addr(B2))
```

A.1.12 *Marker Distance* rule

The *Marker Distance* constraint ensures that the addresses of buildings are related to the buildings' distance from a fixed marker. The rule that maps to the *Marker Distance* constraint is as follows where the *Distance(X,Y)* is the distance between two buildings calculated using the lat/lon coordinates and f is a factor in the set {.001,.01,.1,1,10,100,1000,10000}:

```
1. (sName(B1) = sName(B2)) & (distance(B1,B2) = (addr(B1) - addr(B2)) * f)
```

A.2 Sudoku Inference Rules

These rules map to the constraints in the constraint library specific to the Sudoku puzzle domain. Each constraint type is indicated in **bold**.

AllDiff Row: Ensures that all cells in the same row are assigned a different number.

```
(Row_1 == Row_2) & (Number_1 != Number_2)
```

AllDiff Column: Ensures that all cells in the same column are assigned a different number.

```
(Column_1 == Column_2) & (Number_1 != Number_2)
```

AllDiff Region: Ensures that all cells in the same region are assigned a different number.

```
(Region_1 == Region_2) & (Number_1 != Number_2)
```

AllDiff Diagonol_1: Ensures that all cells along the first diagonal are assigned a different number.

```
(Row_1 == Column_1) & (Row_2 == Column_2) & (Number_1 != Number_2)
```

AllDiff Diagonol_2: Ensures that all cells along the second diagonal are assigned a different number.

```
(Row_1 + Column_1 == 10) & (Row_2 + Column_2 == 10) & (Number_1 != Number_2)
```

Color Even: Ensures that all cells of the same color are assigned an even number.

```
(Number_1 % 2 == 0) & (Number_2 % 2 == 0) & (Color_1 == Color_2)
```

Color Odd: Ensures that all cells of the same color are assigned an odd number.

```
(Number_1 % 2 == 1) & (Number_2 % 2 == 1) & (Color_1 == Color_2)
```

Color Small: Ensures that all cells of the same color are assigned a number in the range $\{1 \ldots 4\}$.

```
(Number_1 < 5) & (Number_2 < 5) & (Color_1 == Color_2)
```

Color Big: Ensures that all cells of the same color are assigned a number in the range $\{5 \ldots 9\}$.

```
(Number_1 > 4) & (Number_2 > 4) & (Color_1 == Color_2)
```

Appendix B

Constraint Model: XML Schema

This appendix describes the XML schema for all XML files that compose a problem instance. I show the XSD schema definition, followed by an example XML file.

B.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.

An example XML file 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>
```

B.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.

An example XML file is the following.

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

B.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.

An example XML file 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>
```

B.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.

An example XML file 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>
```

B.5 Inferred Ranges XML file

The schema for the inferred ranges 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="ranges">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="street" minOccurs="0" maxOccurs=
              "unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="segment" minOccurs="0" maxOccurs=
                    "unbounded">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="startpt">
                            <xs:complexType>
                              <xs:attribute name="buildingid" type="xs:int"
                              use="required" />
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="endpt">
                            <xs:complexType>
                              <xs:attribute name="buildingid" type="xs:int"
                              use="required" />
                            </xs:complexType>
                          </xs:element>
                          <xs:element name="startAddress" type="xs:int" />
                          <xs:element name="endAddress" type="xs:int" />
                        </xs:sequence>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="streetname" type="xs:string" 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 inferred ranges on addresses XML file, the *ranges* node contains multiple *street* children. Each *street* has a *streetname* attribute, which indicates the common name for the street. The *street* node contains multiple *segment* children. The *segment* node has a *startpt* and *endpt* child, which indicate the start and end buildings of the segment. The *segment* node also has a *startAddress* and *endAddress* child, which indicate the start and end address number of the segment.

An example XML file is the following.

```
<constraint districtid="1">
  <ranges>
    <street streetname="W Grand Ave">
      <segment>
        <startpt buildingid="6532" />
        <endpt buildingid="6539" />
        <startAddress>100</startAddress>
        <endAddress>200</endAddress>
      </segment>
      <segment>
        <startpt buildingid="4321" />
        <endpt buildingid="5412" />
        <startAddress>201</startAddress>
        <endAddress>300</endAddress>
      </segment>
    </street>
    <street streetname="Binder Pl">
      <segment>
        <startpt buildingid="1043" />
        <endpt buildingid="231" />
        <startAddress>100</startAddress>
        <endAddress>200</endAddress>
      </segment>
    </street>
  </ranges>
</constraint>
```

B.6 Ascending/Descending Value XML file

The schema for the Ascending/Descending XML file is the following.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="inference">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ascending">
          <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="value" 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:enumeration value="false" />
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:attribute>
                </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 an ascending/descending value XML file, the *ascending* node contains multiple *street* children. Each *street* has a *streetname* and a *value* attribute, which indicate the common name for the street and in which direction addresses increase in value. False indicates that a constraint could not be inferred for that street.

An example XML file is the following.

```
<inference districtid="54">
  <ascending>
    <street streetname="W Grand Ave" value="N" />
    <street streetname="Binder Pl" value="false" />
  </ascending>
</inference>
```

B.7 Parity XML file

The schema for the Parity XML file is the following.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="inference">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="parity">
          <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="value" 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:enumeration value="false" />
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:attribute>
                </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 parity XML file, the *parity* node contains multiple *street* children. Each *street* has a *streetname* and a *value* attribute, which indicate the common name for the street and which side of the street is odd. False indicates that a constraint could not be inferred for that street.

An example XML file is the following.

```
<inference districtid="54">
  <parity>
    <street streetname="W Grand Ave" value="N" />
```

```
    <street streetname="Binder Pl" value="false" />
  </parity>
</inference>
```

B.8 Continuous Numbering XML file

The schema for the Continuous Numbering XML file is the following.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="inference">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="continuous">
          <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="value" use="required">
                    <xs:simpleType>
                      <xs:restriction base="xs:string">
                        <xs:pattern value="false | [1-9][0-9]*" />
                      </xs:restriction>
                    </xs:simpleType>
                  </xs:attribute>
                </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 continuous numbering XML file, the *continuous* node contains multiple *street* children. Each *street* has a *streetname* and a *value* attribute, which indicate the common name for the street and the fixed increment amount between addresses. False indicates that a constraint could not be inferred for that street.

An example XML file is the following.

```
<inference districtid="54">
  <continuous>
```

```
    <street streetname="W Grand Ave" value="2" />
    <street streetname="Binder Pl" value="false" />
  </continuous>
</inference>
```

B.9 District Boundaries XML file

The schema for the District Boundaries XML file is the following.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="inference">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="district" minOccurs="1" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="area">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="point" minOccurs="3" maxOccurs=
                    "unbounded">
                      <xs:complexType>
                        <xs:attribute name="lat" type="xs:double" use=
                        "required" />
                        <xs:attribute name="lon" type="xs:double" 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:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

In a district boundaries XML file, the *boundary* node contains a single *area* children. The *area* node has multiple *point* children (a minimum of three). Each *point* has a *lat* and a *lon* attribute, which indicates the latitude and longitude of the boundary point.

An example XML file is the following.

```
<inference>
```

```
  <boundary districtid="1">
    <area>
      <point lat="54.1" lon="46.1" />
      <point lat="54.2" lon="46.1" />
      <point lat="54.2" lon="46.2" />
      <point lat="54.1" lon="46.2" />
    </area>
  </boundary>
  <boundary districtid="2">
    <area>
      <point lat="54.1" lon="46.1" />
      <point lat="54.2" lon="46.1" />
      <point lat="54.2" lon="46.0" />
    </area>
  </boundary>
</inference>
```