USING CONSTRAINT PROCESSING TO MODEL, SOLVE, AND SUPPORT INTERACTIVE SOLVING OF SUDOKU PUZZLES

by

Christopher G. Reeson

AN UNDERGRADUATE THESIS

Presented to the Faculty of
The College of Arts and Sciences at the University of Nebraska
In Partial Fulfillment of Requirements
For the Degree of Bachelor of Science with Distinction

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry and Dr. Charles Riedesel

Lincoln, Nebraska

May, 2007

USING CONSTRAINT PROCESSING TO MODEL, SOLVE, AND SUPPORT INTERACTIVE SOLVING OF SUDOKU PUZZLES

Christopher G. Reeson,

University of Nebraska, 2007

Advisor: Berthe Y. Choueiry and Dr. Charles Riedesel

In this thesis, we model the Sudoku puzzle, a known **NP**-complete problem [Yato, 2003], as a Constraint Satisfaction Problem (CSP) and investigate solving it using Constraint Processing (CP) techniques. We study and compare the effectiveness of several constraint propagation algorithms. We investigate the use of CP techniques to support interaction with the human players to guide and train them in solving Sudoku puzzles. We explore the use of the appealing and familiar setting of Sudoku puzzles as a vehicle to teach CP techniques to students in Computer Science and to explain the power of these techniques to the general public.

We found empirically that algorithms that achieve relatively low levels of consistency are able to solve most (SAC [Debruyne and Bessière, 1997]) or all (SGAC [Debruyne and Bessière, 1997; Régin, 1994]) common Sudoku puzzles that have one solution. We designed and implemented a Java applet, Solver, that allows a user to interactively solve a Sudoku using CP techniques. Solver is built to maximize the interactions between the human users and CP techniques. It allows the users to apply different consistency algorithms, work specifically on certain constraints, and make assignments and domain reductions on their own. We also designed a 'hint' functionality that uses increasingly complex propagation algorithms, in a controlled manner, to guide the users and train them playing the game.

ACKNOWLEDGEMENTS

I would like to convey my deepest gratitude to my advisor, Professor Berthe Y. Choueiry who has given me her full support in completing this thesis. This thesis would not have been possible without her hard work testing the software, inputting new instances, explaining concepts, and revising and reviewing my writing just to name a few things. I would like to thank Dr. Charles Riedesel for serving as my co-advisor. He carefully read this manuscript and provided invaluable feedback and advice to help me improve it. I would also like to thank my committee members: Professor Steve Dunbar, who inspired this research and Professor Steve Reichenbach.

The CONSTRUCTOR was built in collaboration with Angelo Kai-Chen Huang, a master's student at the Department of Computer Science of the University of Southern California. Angelo built all the mechanisms for the CONSTRUCTOR around my interface, search, and constraint propagation algorithms.

Finally, I would like to thank my family for their support.

This research is supported by a UCARE (Undergraduate Creative Activities and Research Experiences) grant of the University of Nebraska-Lincoln, and CAREER Award #0133568 from the National Science Foundation.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In this thesis, we study the design and application of Constraint Processing (CP) techniques to solve the popular combinatorial game known as the Sudoku puzzle. In particular:

1. We study the modeling of the Sudoku puzzle as a Constraint Satisfaction Problem (CSP).

2. We explore the design and use of CP techniques, both search and constraint propagation, for systematically solving the problem.

3. We develop strategies grounded in CP to dynamically assist a human player solving a Sudoku puzzle.

4. We investigate techniques developed by human players to solve Sudoku puzzles, and characterize them in terms of the theoretical concepts of consistency levels of a CSP.

In addition to the above theoretical investigations, we have also developed and implemented a fully interactive and web-based tool that demonstrates each aspect of our

theoretical investigations.

1.1 Background of the Sudoku Puzzle

The Sudoku Puzzle was created by Howard Garns in 1979 and originally appeared in the Dell Pencil Puzzles and Word Games magazine. It is sometimes called Number Palace, which was its original name when first published. In Japan, Nikoli began publishing Sudoku Puzzles in 1986 and introduced the Sudoku name, which it trademarked in Japan. More recently, newspapers across the United States have begun publishing puzzles daily [Sudopedia, 2007].

Figure 1.1 shows a typical Sudoku puzzle. This puzzle has nine lines, nine columns,

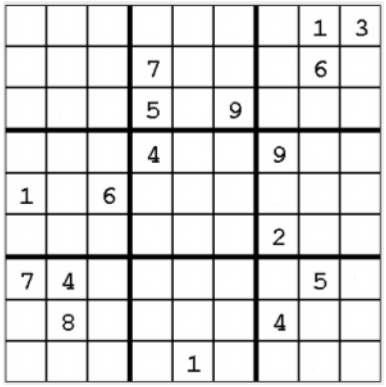

Figure 1.1: Instance number 15 on Royle's web site.

and 9 square blocks delimited by darker lines in Figure 1.1. The goal of this puzzle is to place the numbers 1...9 exactly once in each line, column, and block. Some numbers are already given in the cells. We will call these `preassigned values`. This Sudoku variation is the most commonly published one in the United States, and we will refer to it as the `basic` type. There are many other types of Sudoku puzzles. Two common ways to modify the puzzle are as follows:

1. *Changing the constraints:* One way of doing this is to add additional diagonal constraints. Another is to make the blocks non-uniform, with a jigsaw puzzle appearance.

2. *Enlarging the size of the puzzle.* The size of the grid can be increased, possibly becoming rectangular, or some cells can be split into two, requiring the placement of two numbers instead of one.

We call a *cell* an atomic square of a puzzle, and a *block* the collection of cells where every value (or color) has to appear exactly once. According to the SudoCue Solving Guide [2007], 'blocks' are are also called 'boxes,' 'nonets,' 'regions,' and 'minigrids.' In order to avoid introducing new terminology, we will refer to these units as blocks.

A number of properties of Sudoku puzzles are worth mentioning.

Definition 1 Well posed Sudoku (Simonis [2005]): *A Sudoku is well posed if and only if it has exactly one solution.*

Definition 2 Minimal Sudoku (Royle [2005]): *A Sudoku is minimal if and only if it is well posed and and no preassigned value can be removed without the puzzle becoming not well posed.*

Figure 1.1 shows an example of a minimal Sudoku taken from Royle's web site [2005].

1.2 Thesis outline

This report is structured as follows.

Chapter 2 introduces CSPs and our modeling of the Sudoku puzzle as a CSP. In Section 2.3 we describe the consistency algorithms we implemented to solve the Sudoku. In particular, in Section 2.5, we propose two conjectures:

1. The algorithm for enforcing SGAC (where SGAC denotes Singleton Generalized Arc Consistency [Debruyne and Bessière, 1997; Régin, 1994]) solves any well posed 3×3 Sudoku.

2. The consistency properties SGAC and R(1,2)C (where R(1,2)C denotes relational (1,2)consistency [Dechter, 2003]) are equivalent on all-different constraints, such as the ones used to model a Sudoku.

Chapter 3 describes the functionalities we implemented to support and guide a human playing Sudoku.

Chapter 4 discusses related work in the field of Computer Science and within popular human solution methods. It ends this report with the conclusions we have drawn from our investigations.

Finally, Appendix A explains our implementation, including the XML format we designed for storing Sudoku instances, and the data structures we used for storing and processing a CSP instance.

Chapter 2

The Sudoku Puzzle as a CSP

In this chapter, we introduce Constraint Satisfaction Problems (CSPs) and our model of the Sudoku Puzzle as a CSP. We also review the various constraint propagation algorithms that we have implemented, and introduce two conjectures relating the consistency properties and solvability of a 3×3, well posed Sudoku.

2.1 Constraint Satisfaction Problem (CSP)

A CSP is defined a $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, where

- $\mathcal{V} = \{V_1, V_2, \ldots, V_n\}$ is a set of n variables.
- $\mathcal{D}$ is a set of domains, one domain per variable. The domain D_{V_i} of variable V_i is a set of values that V_i can take. And,
- $\mathcal{C}$ is a set of constraints that apply to the variables. A constraint C_i that applies to a set of variables $\{V_a, V_b, \ldots, V_k\}$ (called the *scope* of the constraint) is a relation over the domains of these variables, and specifies the combination of values that the variables can take at the same time.

Solving a CSP corresponds to assigning a value to each variable such that all the constraints are simultaneously satisfied.

2.2 Sudoku as a CSP

We represent a $n \times m$ Sudoku as a CSP as follows:

- Each cell is a CSP variable. There are $n^2 \times m^2$ variables.
- The domain of the cell is the set of values in the interval [1, $(n \times m)$].
- We consider two models for constraints: a binary model and a non-binary one.

Figure 2.1 shows a 3×3 Sudoku in a grid of 9 lines and 9 columns. Note that the

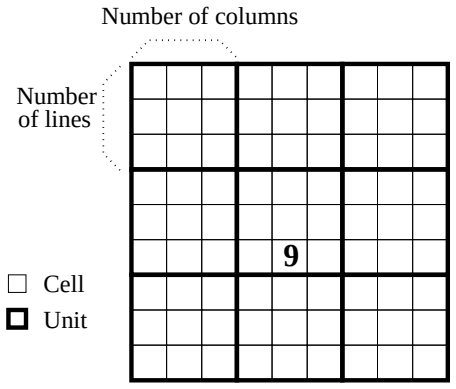

Figure 2.1: Grid of a 3×3 Sudoku.

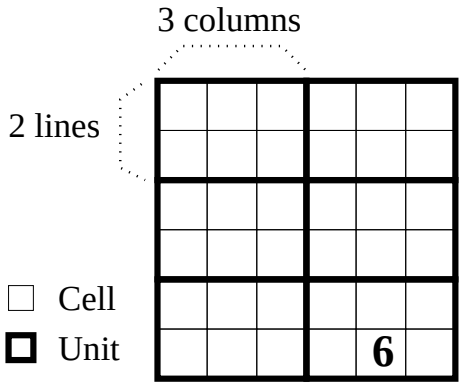

Figure 2.2: Grid of a 2×3 Sudoku.

number of lines and columns need not be equal in general. If there are n lines and m columns, then the puzzle will have $n \times m$ blocks, and there will be n blocks in a line and m lines of blocks. For example, $n = 2$ and $m = 3$ is a Sudoku where the variables can take the values [1,6] as shown in Figure 2.2.

Note the Latin Square is a special case of Sudoku where either n or m is equal to 1. All three representations of the Latin Square shown in Figure 2.3 are equivalent. The

block constraints in the leftmost and center representations duplicate the function of the line and line constraints respectively. Thus, block constraints can be safely ignored resulting in a Sudoku equivalent to a Latin Square.

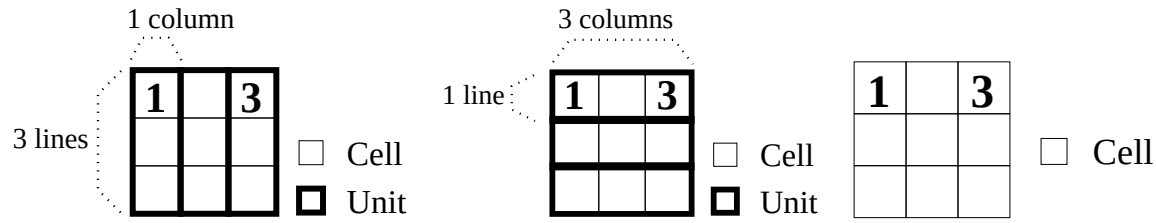

Figure 2.3: Three equivalent representations. *Left:* 3x1 Sudoku. *Center:* 1x3 Sudoku. *Right:* 3 Latin Square.

2.2.1 Binary constraint model

In the binary model, we consider only one type of constraint, which is the binary mutex (or difference) constraint. We generate one such constraint between every two cells that appear in the same line, column, or block. The constraint enforces that the two cells cannot take the same value.

We generate just enough binary constraints to completely model the problem, meaning that we allow exactly one constraint between every pair of related variables. There are $(n \times m)^2 \times \ [\frac{3(n \times m-1)-(m-1)-(n-1)}{2}]$ binary constraints in this model. Figure 2.4 shows the constraints in one block, line, and column.

2.2.2 Non-binary constraint model

In the non-binary model, we consider three types of the non-binary mutex (or all-different) constraints, where the type is defined by the scope of the non-binary constraint. These are:

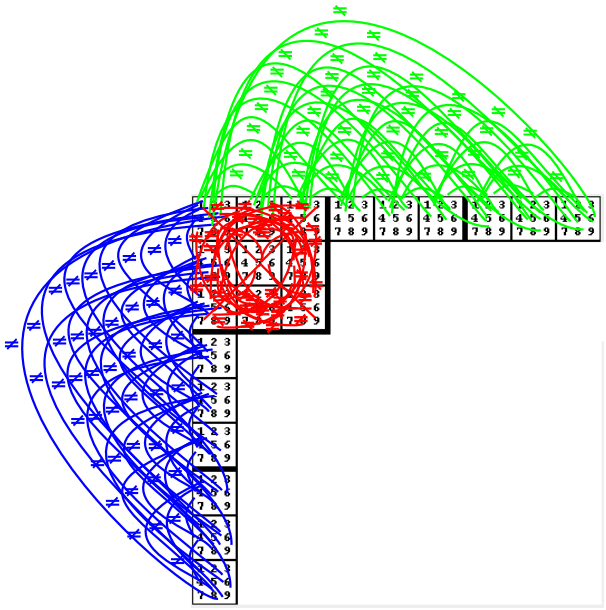

Figure 2.4: The binary constraints in one block, line, and column.

1. The `line-constraint`, whose scope is all the cells in a given line.

2. The `column-constraint`, whose scope is all the cells in a given column). And

3. The `block-constraint`, whose scope is all the cells in a given block.

There is a total of $3 \times (n \times m)$ non-binary constraints in this model. Figure 2.5 shows the constraints in one block, line, and column.

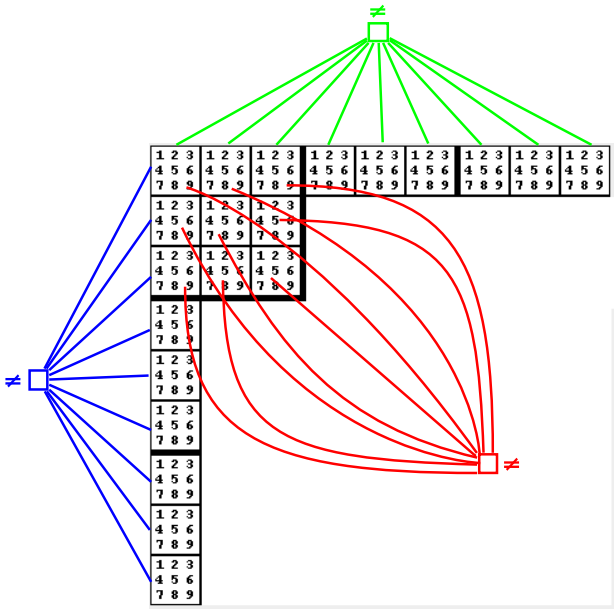

Figure 2.5: The non-binary constraints in one block, line, and column.

2.3 Constraint propagation algorithms

Among the most important contributions of the CP community is the definition of the consistency properties of a CSP and the design and evaluation of constraint propagation algorithms that aim at enforcing these consistency properties on a problem.

In this thesis, we restrict ourselves to algorithms that modify the domains of the variables in a CSP, by removing values, in order to guarantee a given level of consistency. These algorithms should be contrasted to techniques that add new constraints to the problem to explicitly prevent some inconsistent combinations from being explored.

We implemented two algorithms that achieve arc consistency and also extensions of these algorithms using the shaving technique. Shaving is somehow similar to the human method of 'guessing.' In shaving, formally called Singleton Consistency [Debruyne and Bessière, 1997], a variable is assigned a value from its domain, and then consistency algorithms are run to check the validity of such an assignment. If a consistency algorithm returns a failure, then the value is eliminated from the variable. If no broken constraints are found, then the value is retained. In the rest of this section, we review the various constraint propagation algorithms we implemented:

- Forward checking (FC) [Haralick and Elliott, 1980].
- Arc Consistency (AC) [Mackworth and Freuder, 1984].
- Generalized Arc Consistency (GAC) for all-different constraints [Régin, 1994]
- Singleton Arc Consistency (SAC) [Debruyne and Bessière, 1997], also known as shaving. And,

- Singleton GAC (SGAC) [Debruyne and Bessière, 1997; Régin, 1994], applying shaving similarly to SAC, but in combination with GAC instead of AC.

2.3.1 Arc-consistency (AC)

The basic operation in arc-consistency is to update the domain of a variable using the REVISE operation. This operation revises the domain of one variable given the constraint that links the variable to another variable. Algorithm 1 shows the simple REVISE operation we use for binary mutex constraints.

Input: V_i, V_j
Output: true if D_{V_i} is updated; false otherwise.
1 **if** $(|D_{V_j}| = 1) \wedge (D_{V_j} \subset D_{V_i})$ **then**
2 | $D_{V_i} \leftarrow D_{V_i} \setminus D_{V_j}$
3 | **return** true
4 **else**
5 | **return** false
6 **end**

Algorithm 1: REVISE(V_i,V_j) for a binary mutex constraint.

When a number of variables in a CSP have been instantiated, we can revise the domains of the un-instantiated variables, given the current instantiations. This operation is known as *forward checking* (FC), and usually used at each variable instantiation during backtrack search (see Section 2.4). The algorithm is shown in Algorithm 2.

For arc-consistency, we implement AC-3 [Mackworth and Freuder, 1984], which revises the domains of all variables until quiescence, revisiting variables that are connected to at least one variable whose domain has been modified. Algorithm 3 is the pseudo code we implemented.

Input: $\mathcal{P}$: a binary CSP
Output: true if the assigned variables are arc-constent; false otherwise.

if *No domain is empty* **then**
  $\mathcal{V}_i = \{ V_i \in \mathcal{V} \mid V_i$ is an instantiated variable$\}$
  $\mathcal{V}_u = \{ V_u \in \mathcal{V} \mid V_u$ is an un-instantiated variable$\}$
  for *every* V_u *in* $\mathcal{V}_u$ **do**
    for *every* V_i *in* $\mathcal{V}_i$ **do**
      REVISE(V_u, V_i)
      if $D_{V_u} = \emptyset$ **then**
        return false
      end
    end
  end
  return true
else
  return false
end

Algorithm 2: FC($\mathcal{P}$)

Input: $\mathcal{P}$: a binary CSP
Output: true if the set of constraints is arc-consistent; false otherwise.

$Q \leftarrow \cup_{C_{V_i,V_j} \in \mathcal{C}} \{e(V_i, V_j), e(V_j, V_i)\}$
while Q *is not empty* **do**
  $e_{V_i,V_j} \leftarrow$ POP(Q)
  if REVISE(V_i, V_j) **then**
    $Q \leftarrow Q \cup \{e_{V_k,V_i} | V_i, V_k \in \mathcal{V}$ and $k \neq j\}$
  end
  if $D_{V_i} = \emptyset$ **then**
    return false;
  end
end
return true

Algorithm 3: AC-3($\mathcal{P}$).

2.3.2 Shaving Arc Consistency (SAC)

SAC iterates over each value of every variable. The algorithm assigns a value to a variable and then runs AC on the entire problem. It will remove the value from the domain of the variable if a constraint is broken while performing AC. It continues to iterate over all variables while any value is removed. Algorithm 4 is the pseudo code we implemented.

Input: $\mathcal{P}$: a binary CSP
Output: true if all variables have at least one value; false otherwise

```
1  Q_0 ← 𝒱 \ {V_i | V_i is instantiated}, Q ← Q_0
2  while Q is not empty do
3  |  V_i ← Pop(Q)
4  |  t ← D_{V_i}
5  |  for each x ∈ t do
6  |  |  V_i ← x
7  |  |  if not AC-3(𝒫) then
8  |  |  |  Q ← Q ∪ {V_j|(V_j ∈ 𝒱) ∧ (C_{V_i,V_j} ∈ 𝒞) ∧ (V_j is not instantiated)}
9  |  |  |  t ← t \ {x}
10 |  |  |  if t = ∅ then
11 |  |  |  |  return false
12 |  |  |  end
13 |  |  end
14 |  |  Unassign(V_i)
15 |  end
16 |  D_{V_i} ← t
17 end
18 return true
```

Algorithm 4: SAC($\mathcal{P}$).

2.3.3 Generalized Arc Consistency (GAC)

We have implemented Régin's [1994] algorithm to achieve GAC in polynomial time on non-binary all-diff constraints. GAC dominates AC, which, in turn, dominates FC in terms of filtering power. Figure 2.6 shows, on a single block within a puzzle, the effects of applying these algorithms by showing how the the domains of the unassigned variables are reduced. We run FC on the entire puzzle yielding the reductions indicated by the arrow marked FC. We then run AC on the individual block shown in Figure 2.6 yielding the results indicated by the arrow marked AC. Finally, when GAC is run on the same block, more values are removed. Note that the filtering achieved with a given algorithm is independent of the sequence shown in the figure.

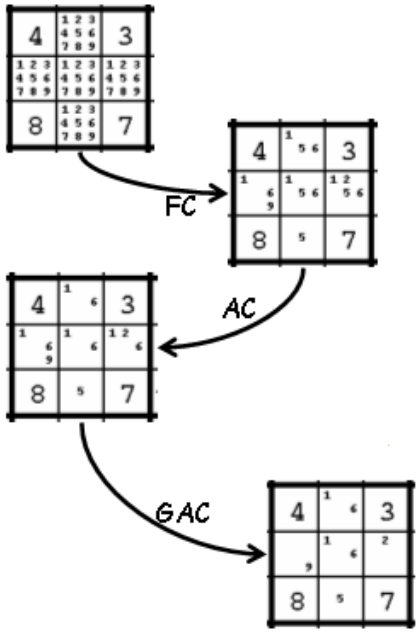

Figure 2.6: An illustration of GAC's power.

2.3.4 Shaving Generalized Arc Consistency (SGAC)

SGAC works the same way as SAC substituting GAC for AC. SGAC is the most powerful algorithm we implemented.

2.3.5 Consistency algorithm summary

Figure 2.7 shows the relative strengths of the algorithms. Forward Checking (FC)

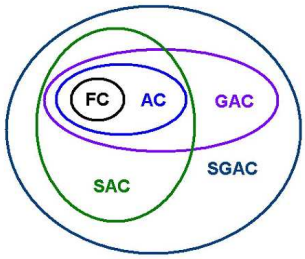

Figure 2.7: Sets denote the values removed by propagation.

is the least powerful form of arc consistency that we implemented. It iterates over only the constraints between the instantiated variables and the un-instantiated ones. Arc Consistency (AC) is more agressive than FC because it considers all the constraints in the CSP (i.e., include the constraints among un-instantiated variables), and continues revising domains until no additional reductions are possible. Generalized Arc Consistency (GAC) is more powerful than AC [Régin, 1994]. For example, in instances like the one shown in Figure 2.6 where a value only appears in exactly one variable, or n values appear as the only choices in n variables, GAC yields more domain reductions. Shaving or Singleton Arc Consistency (SAC) is more powerful than AC because it enforces AC for each combination of variable-value pair of the un-istantiated variables, removing those values that do not yield an arc-consistent network. Similarly, Singleton Generalized Arc Consistency (SGAC), or shaving with

GAC, is stronger than GAC. SGAC is stronger than SAC because as we have already shown GAC is stronger than AC. However, GAC and SAC are not comparable.

2.4 Search algorithms

Search algorithms proceed in two main ways:

1. Constructive search: We assign a value to one variable at a time (i.e., instantiate the variable), gradually building a solution.

2. Iterative-repair search: We consider an assignment of values to all variables (i.e., a complete but potentially inconsistent assignment), and gradually remove inconsistencies by locally modifying assignments to some variables.

Search algorithms may explore part or all of a search space. Some search algorithms implement constraint propagation techniques to prune the search space.

We have implemented a depth first backtrack search algorithm for finding all the solutions to a Sudoku. Backtrack search (BT) is a constructive search algorithm that is both sound and complete. It keeps building a partial assignment until it finds a solution or an error. If it encounters an error, it 'backtracks' by unassigning the last instantiated variable and trying another value for it taken from its domain.

To check the consistency of a solution that is being built interactively, we implement the following mechanisms:

1. Back-checking (BC): Back-checking ensures that any binary constraint between the variable being assigned and any previously assigned variable is not broken.

2. Forward checking (FC): Forward checking revises the domain of every unassigned variable connected with a binary constraint to the variable being instan-

tiated (i.e., current variable). It removes, from the domain being revised, the values that are not consistent with the instantiation of the current variable.

3. Maintaining arc-consistency (MAC): Maintaining arc-consistency runs AC-3 (see algorithm 3) after every variable instantiation to make the entire problem arc consistent.

2.5 Conjectures about the consistency properties of a Sudoku

Below, we introduce 2 conjectures about the consistency properties of a Sudoku puzzle. We verified the first conjecture empirically, and draw the beginning of the second conjecture. Both remain to be established.

2.5.1 Consistency level for solving a Sudoku puzzle

We conjecture that SGAC is sufficient to solve any well posed 3×3 Sudoku puzzle (which has a 9×9 grid). The proof remains open, but the results listed by Simonis [2005] and our experiments support this conjecture.

2.5.2 Comparing SGAC and R(1,2)C

We conjecture that SGAC and R(1,2)C (where R(1,2)C denotes relational (1,2)consistency [Dechter, 2003]) are equivalent on all-different constraints, such as the ones used to model a Sudoku.

Our goal is to prove that SGAC($\mathcal{P}$) $\equiv$ R(1,2)C($\mathcal{P}$), where $\mathcal{P}$ is a non-binary CSP. We need to prove that R(1,2)C($\mathcal{P}$) $\Rightarrow$ SGAC($\mathcal{P}$), and R(1,2)C($\mathcal{P}$) $\Leftarrow$ SGAC($\mathcal{P}$).

Without loss of generality, we restrict the proof on how R(1,2)C and SGAC operate on a CSP composed of only two constraints C_1 and C_2, where S_1 and S_2 are the scopes of C_1 and C_2 respectively, and $S_1 \cap S_1 \neq \emptyset$. for this purpose, we consider the CSP $\mathcal{P}=(\mathcal{V}=S_1 \cup S_2, \mathcal{D}, \mathcal{C}=\{C_1,C_2\})$, where $\mathcal{D}$ is the set of domains of the variables in $\mathcal{V}$.

R(1,2)C $\Rightarrow$ SGAC

We need to prove that the set of variable-value pairs filtered out by SGAC is a subset of the set of variable-value pairs filtered out by R(1,2)C.

Any variable-value pair eliminated by SGAC is guaranteed not to appear in any solution, because SGAC is a (safe) consistency propagation algorithm. Any such variable-value pair is necessarily filtered out by R(1,2)C, which eliminates every variable-value pair that does not appear in a solution to $\mathcal{P}$. Consequentely, any variable-value pair eliminated by SGAC is also eliminated by R(1,2)C.

(Intuitively speaking, whereas R(1,2)C preserves only the variable-value pairs that participate in a solution (i.e., the minimal CSP), SGAC is a propapation mechanism that filters the domains as a step 'towards' finding the minimal network.)

R(1,2)C $\Leftarrow$ SGAC

We need to prove that the set of variable-value pairs filtered out by R(1,2)C is a subset of the set of variable-value pairs filtered out by SGAC. We distinguish two situations for this proof:

1. $S_1 \cap S_2 = \{V_i\}$, and

2. $|S_1 \cap S_2| > 1$.

Our proof for the first situation holds for general constraints. The proof of the second situation remains open. If a proof cannot be found for arbitrary constraints, an alternative would be to prove the second situation for the restricted case of all-different constraints that occur in Sudoku puzzles.

1. Consider first the case $S_1 \cap S_2 = \{V_i\}$.

   - Assume that a value x for a variable V_i is filtered out by R(1,2)C but not by SGAC. Given that $V_i \leftarrow x$ is not filtered out by SGAC on C_1, it must appear in a solution to S_1. Similarly, it must appear in a solution to S_2. Because S_1 and S_2 overlap only on V_i and have both a solution with $V_i \leftarrow x$, $V_i \leftarrow x$ appears in a solution to $S_1 \cup S_2$. This solution requires that R(1,2)C not remove $V_i \leftarrow x$, which is in contradiction with our assumption. Consequently $V_i \leftarrow x$ must be filtered out by SGAC.
   - Assume that a value x for a variable $V_j \in S_1$ and $V_j \notin S_2$ is filtered out by R(1,2)C but not by SGAC. Because this variable-value pair is not filtered out by SGAC, then there is a solution s_1 to S_1 where it appears. Let y be the value of V_i in this solution. Because $V_i \leftarrow y$ is preserved by SGAC, there must be a solution s_2 to S_2 where $V_i \leftarrow y$ appears. These two solutions agree on the value of there unique common variable V_i, and thus form together a solution to the CSP. Consequently, we have a solution to the CSP with $V_j \leftarrow x$, and x should be preserved by R(1,2)C, which is in contradiction with our assumption. Consequently $V_j \leftarrow x$ must be filtered out by SGAC.

2. Now, consider the case where $|S_1 \cap S_2| > 1$, and let $S = S_1 \cap S_2$. The proof is complete only when this case is solved.

Chapter 3

Interactive Functionalities

We developed a number of interactive functionalities in order to support and guide the user in solving a Sudoku puzzle. This chapter describes the functionality without going into the implementation specifics. Figure 3.1 gives an illustrated introduction to the interface.

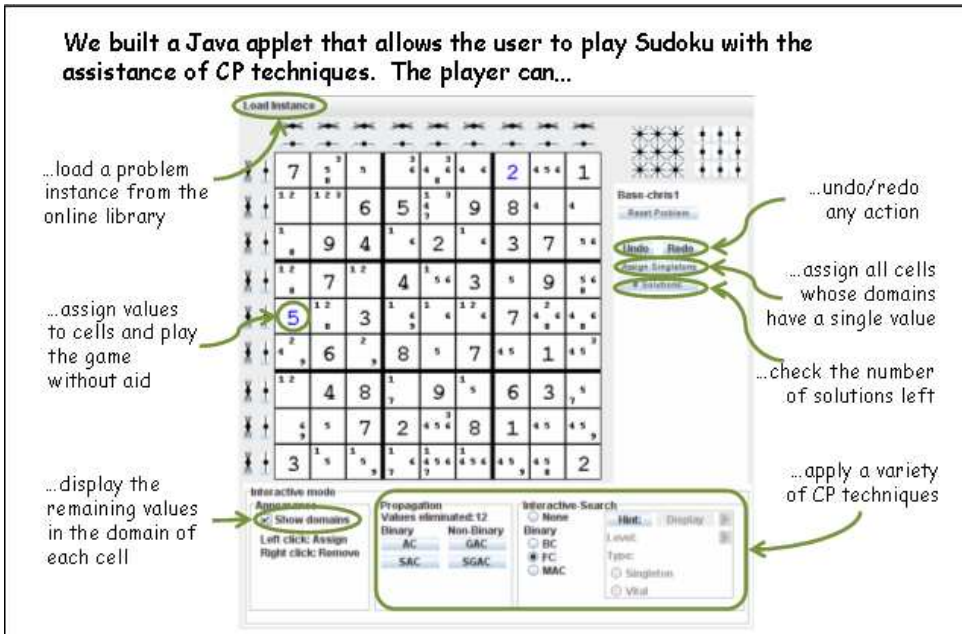

Figure 3.1: An introduction to the solver interface.

3.1 Assigning singletons

Assign Singletons takes each variable with one value in its domain and assigns that value to that variable. Assigning singletons before executing a shaving algorithm prevents the algorithm from uselessly iterating over singleton variables.

3.2 Hint: A tool for aiding human players

This functionality iterates through increasing levels of propagation looking for two domain conditions:

1. *Singleton:* A singleton hint occurs when there is a variable with only one value. The rightmost variable in Figure 3.2 has only one value thus illustrating a singleton, in this case 4.

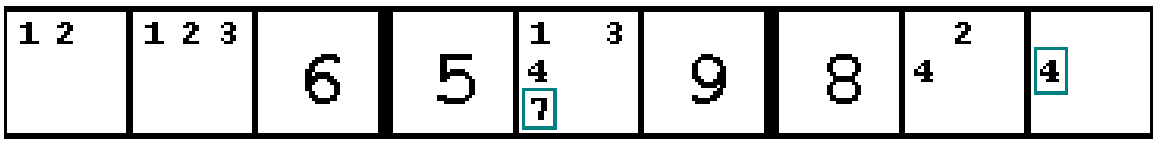

Figure 3.2: A line containing both a vital and singleton hint.

2. *Vital:* A vital hint occurs when a value appears only once within a constraint. Figure 3.2 also contains a vital value which is the boxed 7 in the middle variable. The value 7 does not occur in the domain of any other variable in the line.

The hints are organized depending on (1) the level of consistency required for guessing the hint, and (2) the condition of the domain of a cell. Given the current state of the grid, the 'Hint' functionality checks for these two conditions as enforced by the following levels of consistency in increasing complexity order: FC, AC, GAC on

each individual constraint, GAC on all constraints, SAC on each individual variable, SAC on all constraints, SGAC on each individual constraint, SGAC on all constraints. Starting from the simplest consistency level (FC), 'Hint' records the total number of cells that exhibit a given condition at the current consistency level, and displays this number to the users. Users can switch between the two conditions (Singleton and Vital), and visit each of the hints as they are highlighted on the grid. Users can then move to the next level of consistency, thus gradually sharpening their abilities to solve the puzzle.

Figure 3.3 shows the hint button in action.

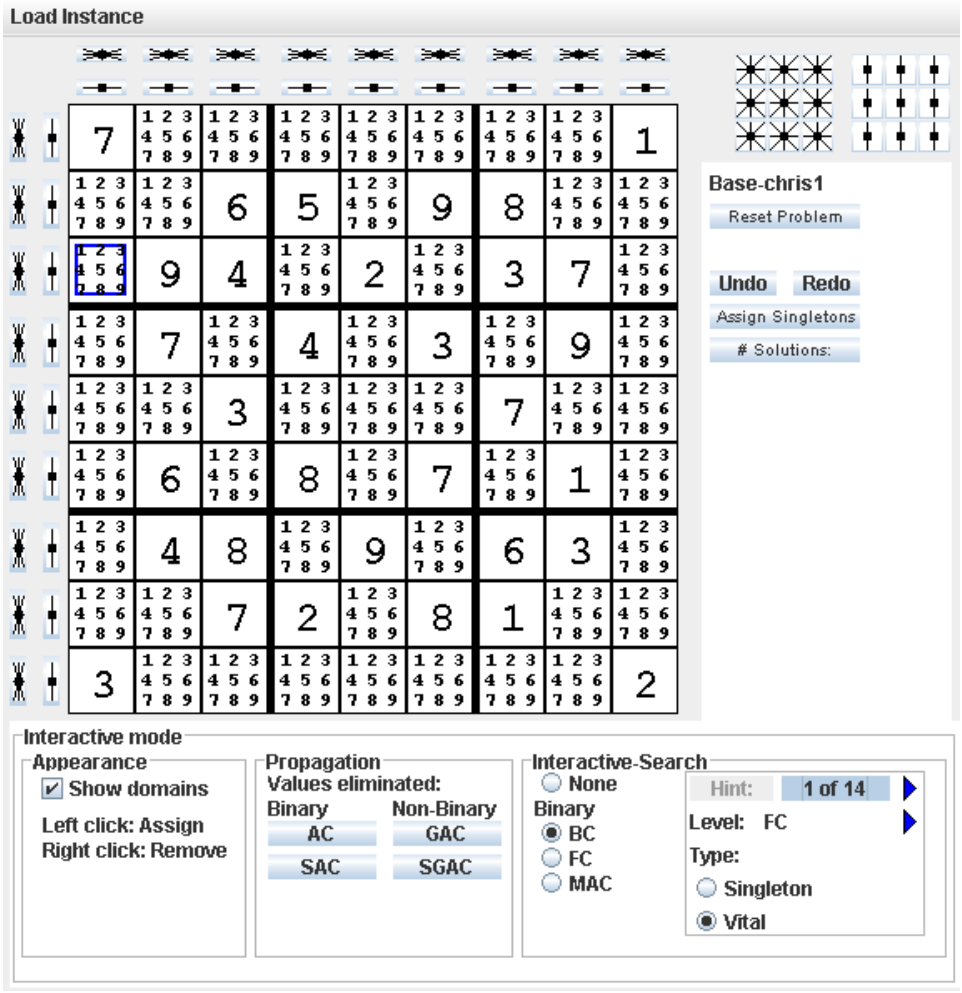

Figure 3.3: The hint panel displaying a hint.

3.3 Enforcing consistency levels

This functionality allows the user to execute GAC on any individual non-binary constraint in the puzzle or AC on the binary constraints within any of the non-binary constraints within the puzzle. These two sets of buttons are shown in Figure 3.4. the buttons on the left and on the top are the nonbinary buttons, and the buttons with one line and a centered square are the binary buttons.

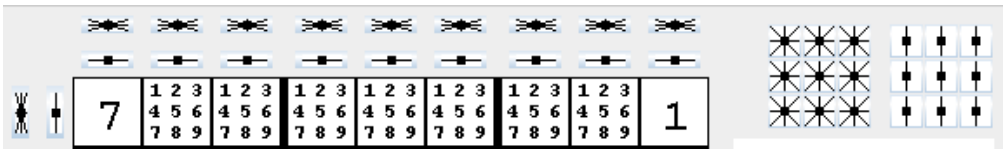

Figure 3.4: A view of all of the column and unit consistency buttons and one line's consistency buttons.

This allows the user to experiment on partial solutions they have uncovered. Users can test what-if scenarios where they assign one or more variables and see the outcome of the consistency algorithms in this context. In this way a human can do shaving by assigning a variable and investigating how the propagation algorithms then strip values from variables across the board.

This functionality is also useful for teaching consistency algorithms, allowing a student to see in which cases GAC achieves more reductions than AC and investigate why that happens. Since the interface also displays the number of values eliminated by the propagation, the user gets feedback in terms of number of values eliminated.

3.4 Showing errors

This functionality displays a red square in any variable that is in a broken constraint after running any of the consistency algorithms. Figure 3.5 shows the result of vi-

olating a binary constraint. Figure 3.6 shows the result of violating a non-binary constraint.

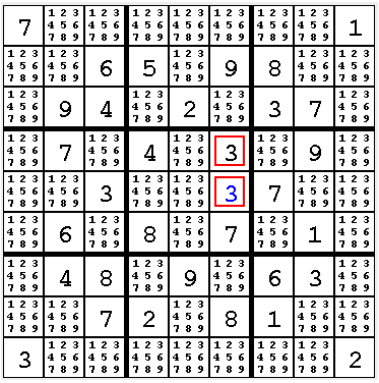

Figure 3.5: Display of a binary constraint in error.

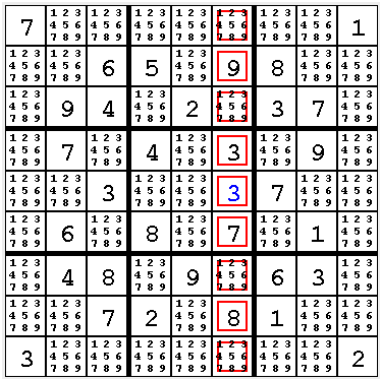

Figure 3.6: Display of a non-binary constraint in error.

3.5 Showing number of solutions

There is a button to display the number of solutions of the Sudoku currently on display in the grid. For this purpose, we create a copy of the current state of the puzzle, apply constraint propagation in an increasing complexity order as a preprocessing to search, then run an exhaustive backtrack search with forward checking. This process finds and stores all solutions, then prints out their total number.

3.6 Showing/hiding domains

There is a button to display or hide the possible values a variable can take. If the user wants to turn this off they will still have the correct options in the menu when making assignments or reductions, but will not see them on the board. These marks are equivalent to human pencil marks.

3.7 Assigning/removing values

This functionality allows the human to make assignments of values to variables. Also, by left clicking the user is allowed to remove values from the domains of variables. In this way the user can do any sort of filtering by hand including algorithms that are implemented.

3.8 Undo/Redo Implementation

After any propagation, assignment, or reduction is made an object recording the action is placed on the stack that allows the user to undo and subsequently redo any such action.

3.9 Input new instances

Angelo Kai-Chen Huang at the University of Southern California has used our code to implement a program that allows the user to input new Sudoku instances. CONSTRUCTOR's interface stores the new instances in the same library SOLVER retrieves them from. The interface is shown in Figure 3.7.

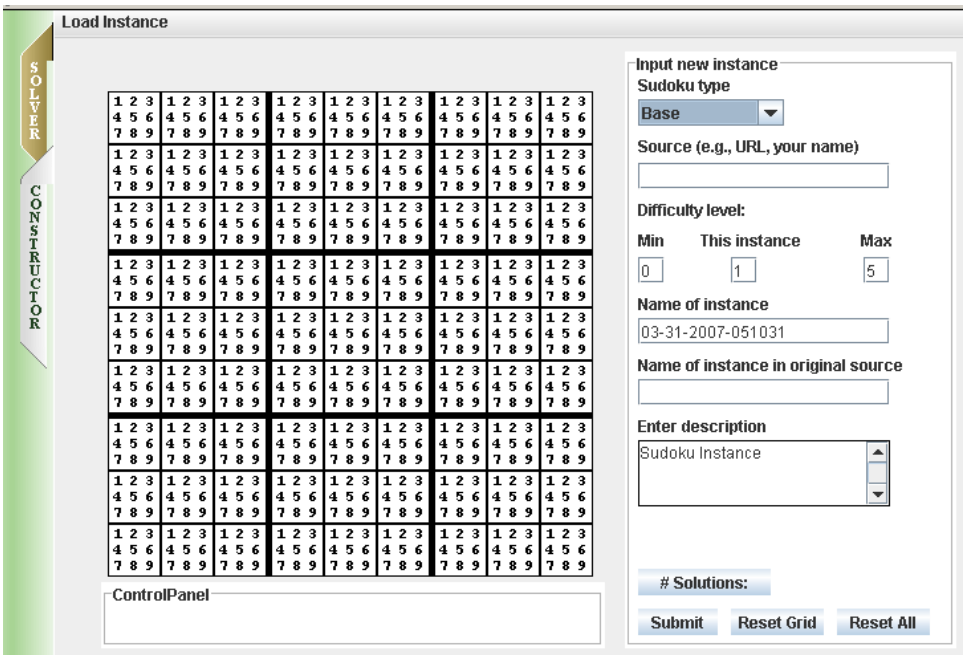

Figure 3.7: The constructor interface.

Chapter 4

Related Work and Conclusions

In this chapter we discuss related work that we have found and explore some of the methods that humans have developed to solve Sudoku Puzzles. Finally, we comment on the impact our work has already had, give directions for further research, and state our conclusions.

4.1 Related work in Computer Science

Although countless web pages on the Internet are devoted to Sudoku, our investigations have not found many published papers on the subject.

Yato [2003] showed that Sudoku is a NP-Complete problem by expanding on the Latin Square using the Another Solution Problem (ASP) as a framework.

Although a Sudoku puzzle was used in the logo of the 2006 International Conference on the Theory and Practice of Constraint Processing (CP 2006), the only publication on the topic within the framework of Constraint Processing that we are aware was published by Simonis [2005].

He used several different constraint propagation algorithms including SAC and SGAC to explore solving Sudoku. He focuses primarily on adding constraints to speed up propagation. He tries to use Constraint Programming with high level constraints based on flow problems to speed up solving a puzzle. However, we have found that R(1,2) and SGAC are sufficient to solve any 3×3 Sudoku. Our approach differs from Simonis's in that we restrict ourselves to the most basic model of the constraints, and use only the most common constraint propagation algorithms (a subset of those used by Simonis).

Unlike Simonis, we focus on the interactive aspects of the game, and have designed a 'hint' functionality that uses constraint propagation mechanisms of increasing complexity to guide the human players and train them in playing the game.

4.2 Related work by the general public

There are many internet sites dedicated to Sudoku and finding ways to solve it. First, we introduce some common terminology that is used, and then we list all of "rules" or patterns for solving Sudokus that we have found documented on the internet.

4.2.1 Terminology

- `house` A house is the term used for the line, column, or block that a variable is in. It is a non-binary constraint that the variable takes part in.

- `peers` The peers of a variable are all of the variables that share a constraint with that variable.

4.2.2 The rules

There are many rules that have been developed by humans to guide puzzle solvers. Ruud Van Werf has collected many of them on Sudopedia [2007] and his guide on SudoCue[2007]. We have compiled an exhaustive list here. `Full House`, `Last Digit`, `Hidden Single` (a.k.a., `Pinned Digit`), `Naked Single` (a.k.a., `Forced Digit`, `Sole Candidate`), `Locked Candidates` (a.k.a., `Line-Box Interaction`, `Claming`), `Locked Pair`, `Locked Triple`, `Naked Subset`, `Naked Quad`, `Naked Triple`, `Naked Pair`, `Almost Locked Candidates`, `Hidden Subset`, `Hidden Pair`, `Hidden Triple`, `Hidden Quad`, `X-Wing`, `Swordfish`, `Jellyfish`, `Squirmbag`, `Finned Fish`, `Sashimi Fish`, `Franken Fish`, `Mutant Fish`, `Kraken Fish`, `Skyscraper`, `2-String Kite`, `Empty Rectangle`, `Simple Colors`, `Multi-Colors` (a.k.a., `Supercoloring`), `Weak Colors`, `X-Colors`, `Color Trap`, `Color Wrap`, `Color Wing`, `3D Medusa Coloring` (a.k.a., `Advanced Coloring`, `Ultracoloring`), `Uniqueness Test`, `Bivalue Universal Grave`, `Bivalue Universal Grave Lite`, `Forcing Chain`, `X-Chain`, `XY-Chain`, `Remote Pairs`, `Fishy Cycle` (a.k.a., `X-Cycle`), `Broken Wing`, `Nice Loops`, `Double Implication Chain`, `Alternating Inference Chain`, `XY-Wing`, `XYZ-Wing`, `ALS-XZ`, `WXYZ-Wing`, `ALS-XZ`, `ALS-XY-Wing`, `Death Blossom`, `Aligned Pair Exclusion`, `Aligned Triple Exclusion`, `Subset Exclusion`, `Sue de Coq`, `Subset Counting` (a.k.a., `Extended Subset Principle`), `Traveling Pairs` (a.k.a. `Braiding`, `Braid Analysis`), `Constraint Subsets`, `Equivalence Marks`, `Forcing Net`, `Tabling` (a.k.a., `Trebor's Tables`), `Graded Equivalence Marks` (a.k.a., `GEM`), `Bowman Bingo`, `Trial and Error` (a.k.a., `Bifurcation`, `Ariadne's Tread`), `Nishio`, `Templating` (a.k.a., `Pattern Overlay Method`, `POM`), and `Guessing`.

Some of these can be accomplished by the same consistency algorithm. For ex-

ample, `Naked Singles` and `Full House` and can both be accomplished with Forward Checking. These are both equivalent to Method A on the Sudoku Solver by Logic website [2007]. These methods identify the same singletons that our hint functionality will find on the hint level FC type Singleton.

The `Hidden Singles`, `Squeezing`, and `Cross Hatching` (eqivalent to Method B on Sudoku Solver by Logic [2007]) rules are a bit more tricky. Our interface will identify the cell containing the '`Hidden Single`' as a Vital hint after running FC. However, the variable will not have any of the other incorrect values removed by the consistency algorithm until GAC is run. Simonis [2005] got around this limitation of FC by using channeling.

Many techniques are subsumed by GAC. `Naked Pair`, `Naked Triple`, `Naked Quad`, `Hidden Pair`, `Hidden Triple`, and `Hidden Quad` in addition to the above are all achieved by GAC.

Some human techniques do go beyond what GAC is capable of resolving. `Locked Candidates` and `X-Wing` will be achieved by SGAC.

4.3 Conclusions and future work

Our work has already inspired the research of Mr. Martin Michalowski, a doctoral student at the Information Sciences Institute of the University of Southern California. He chose to use Sudoku puzzles as an application domain to demonstrate his idea for an iterative approach to building CSP models from data. He is using our implementation of the propagation algorithms described in this thesis as a reformulation step in the modeling process.

In addition, Mr. Angelo Kai-Chen Huang, a graduate student in the masters

program of the University of Southern California, has expanded our code to create CONSTRUCTOR, which allows users to input Sudoku instances. Also, using the basic blocks of our code, he is expanding his constructor to include additional types of Sudokus beyond basic, such as diagonal Sudokus.

We hope to expand our solver to work with these different types of problems, and to further investigate the relationships between the constraint propagation used and the ability to solve the Sudoku Puzzle.

We have found that propagation is very useful in guiding a human playing Sudoku in an interactive environment. The process helps them sharpen their skills and become better Sudoku players.

We have found that adding additional constraints, as Simonis did, is unnecessary for solving Sudokus as SAC will solve most puzzles and SGAC will solve all puzzles we have encountered. In this way Sudoku is a good illustration of the power of SAC and SGAC which are otherwise thought to be more expensive than effective.

Appendix A

Implementation

A.1 The XML file

We have developed an XML schema to facilitate storing Sudoku puzzles. The file format is based on that of the Constraint Solver Competition [CPAI, 2005]. The specification can be found at http://cpai.ucc.ie/05/xml.html. We have kept the opening and closing `<instance>` tags as well as the `<presentation>` tag. We also retained the `description`, `nbSolutions`, and `format` attributes within the `<presentation>` tag. The `format` is Sudokuv2.1 to distinguish our files from the standard format.

A Sudokuv3.0 format exists, and code to interpret it is included in Solver and Constructor. However, this format and the code to interpret it was created as an extension of this thesis by Angelo Kai-Chen Huang.

Several new tags are present in our schema to capture information specific to Sudoku puzzles. These are the `<source>`, `<difficulty>`, `<difficultyScale>`, `<dimensions>`, and `<preassigns>` tags.

A.1.1 The <source> tag

The <source> tag has one attribute: src. Source can either be a website, a name of a publication, or any other source of Sudoku puzzles.

A.1.2 The <difficulty> tag

The <difficulty> tag has one attribute: level. The level is expected to correspond to one of the value attributes of a <difficultyLevel> tag described in the <difficultyScale> section. This is the difficulty as given by the source of the problem.

A.1.3 The <difficultyScale> tag

There are many different difficulty scales. The SudoCue Solving Gude[2007] lists at least 10 different scales currently being in publications. There has been no effort made to standardize these ratings. In order to allow input from many different sources and retain the relative difficulty as rated by the source of the problem. The <difficultlyScale> tag allows a complete description of all the levels in the rating system where the problem comes from.

Within the <difficultyScale> tag there is the <difficultyLevel> tag. The <difficultyLevel> tag has two attributes: value and rank. The value attribute is the name of the difficulty level as given in the source. The rank attribute begins at 0 for the easiest and continues by incrementing one for each new difficulty. The difficulty always increases as the rank increases.

A.1.4 The <dimensions> tag

The <dimensions> tag specifies the dimensions of a Sudoku instance in terms of the number of lines and columns in a block.

The <dimensions> tag has two attributes: lines and columns, where lines and columns specify, respectively, the number of lines and columns in a block. Note that the number of lines and columns need not be equal in general. If there are n lines and m columns, then the puzzle will have $n \times m$ blocks, and there will be n blocks in a line and m lines of blocks. For example, <dimensions lines="2" columns="3"> is a Sudoku where the variables can take the values [1,6] as is shown in Figure 2.2.

A.1.5 The <preassigns> tag

The final additional tag is the <preassigns> tag. This tag takes one optional attribute, nbPreassigns, which gives the number of pre-assigned cells in the Sudoku instance. For example, <preassigns nbPreassigns="26"> means that 26 cells are pre-assigned.

We specify the pre-assigned cells within the <preassigns> tag using the <preassign> tag. The <preassign> tag has three attributes: value, line, and column. line and column denote the cell position in a $n \times m$ matrix, and value gives the value assigned to the cell, which should be in the interval [1, $(n \times m)$].

For example, <preassign value="9" line="6" column="5"> states that the cell in line 6 and column 5 is pre-assigned the value 9 as shown in Figure 2.1. Additionally, <preassign value="6" line="6" column="5"> states that the cell in line 6 and column 5 is pre-assigned the value 6 as shown in Figure 2.2.

A.1.6 Sample XML file

Below, we show an example file for the leftmost Sudoku shown in Figure 2.3:

```
<instance>
  <presentation
     name="2-13-06LS"
     description= "This is the 3 by 3 latin square"
     nbSolutions="at least one"
     format="sudoku2.1"/>
  <source src="original" />
  <difficulty level="easy"/>
  <difficultyScale>
     <difficultyLevel value="easy" rank="0"/>
     <difficultyLevel value="med" rank="1"/>
     <difficultyLevel value="hard" rank="2"/>
  </difficultyScale>
  <dimensions lines="3" columns="1"/>
  <preassigns nbPreassigns="4">
     <preassign line="1" column ="1" value="1"/>
     <preassign line="1" column ="3" value="3"/>
  </preassigns>
</instance>
```

A.2 Data structures for the Sudoku as a CSP

In this section, we introduce the data structures for implementing in Java the Sudoku as a CSP.

A.2.1 Variables

Every cell in the grid is represented by a CSP variable and implemented as a class with the following attributes (called private variables in Java):

- `problem` holds the problem that the variable is located in.
- `index` holds the position (line, column) of the cell in a $[1, n \times m]$ grid.
- `assigned` stores the value from the interval $[1, n \times m]$ assigned to the cell.
- `initial-domain` stores the set of all possible values that the variable can take, which is by default $\{1,2, \ldots, n \times m\}$. When the cell is pre-assigned, this attribute stores only the pre-assigned value as a singleton.
- `current-domain` stores the set of all possible values that the variable can take after some decisions (e.g., constraint propagation or arbitrary domain reductions) have been made. The value of this attribute is initialized to that of `initial-domain`.
- `b-constraints` stores the list of binary constraints that apply to the cell. There are $3(n \times m - 1) - (m - 1) - (n - 1)$ mutex binary constraints for each variable.
- `nb-constraints` stores a list of non-binary constraints that apply to the cell. There are 3 non-binary constraints that apply to each cell: one line constraint, one column constraint, and one block constraint.

A.2.2 Constraints

There are two types of constraints: `binary` and `non-binary`. We have chosen to implement both types. They are both stored separately and represented as different classes. Additionally, the constraints may be stored in either `extension` or `intension` depending on the needs of the search being used and constraints on memory usage. In SOLVER we never generated constraints in `intension` because doing so would take a lot of space and was never necessary. The constraints are not read in from the XML file, but are generated from the dimensions of the problem[1]. Every Sudoku of the same dimensions will have the same constraints.

1. *Constraint in the binary model:* In the binary model we generate binary mutexes between all cells of the same line, column, and block. We are careful not to duplicate the constraint between two variables in the same column and block, or line and block. Only one binary mutex is necessary in overlapping situations. The constraints are only stored in intension. In intension there is simply a function that checks to see if the two variables in the scope of the constraint are not assigned the same value, or one assigned a value and another with that value as its only remaining value in the domain, or both with a singleton domain of the same value.

   If the constraints were to be written out in extension there would a total number of tuples equal to the number of constraints $((n \times m)^2 \times [\frac{3(n \times m-1)-(m-1)-(n-1)}{2}])$ multiplied by $(n \times m) \times (n \times m-1)$ because there are $(n \times m)^2$ total tuples but $(n \times m)$ of them are not valid. For example, (1,1), (2,2), and (3,3) would not be in the list of all valid pairings for variables in the same line, column, or

[1]Format v3.0 designed by Angelo Kai-Chen Huang differs in this respect and includes all the constraints of the CSP model.

block.

While the XML file is being read in, the binary constraints are created in intension by simply specifying the scope of the constraint. A built-in function is used to ensure the mutex property of the constraint.

2. *Constraint in the non-binary model:* The non-binary model has three separate types of constraints. This is the most natural way of thinking of a Sudoku, and is indeed how it is often explained. The values in every `line`, `column`, and `block` must all be different. We will implement and test the algorithm developed by Régin [1994] for efficiently filtering and propagating an `all-different` constraint defined in extension. Since there are $(n \times m)!$ tuples in the definition of such a constraint, we will refrain from defining it in extension until enough cells in its scope have been assigned.

   - `Line constraints`: There is an all-different constraint over the variables appearing in a row of a Sudoku puzzle.
   - `Column constraints` There is an all-different constraint over the variables appearing in a column of a Sudoku puzzle.
   - `Block constraints` There is an all-different constraint over the variables appearing in a block in a Sudoku puzzle.

Bibliography

[CPAI, 2005] Workshop CPAI. Constraint Solver Competition, 2005. cpai.ucc.ie/05/xml.html.

[Debruyne and Bessière, 1997] Romuald Debruyne and Christian Bessière. Some Practical Filtering Techniques for the Constraint Satisfaction Problem. In *Proc. of IJCAI*, pages 418–423, 1997.

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

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

[Mackworth and Freuder, 1984] Alan K. Mackworth and Eugene C. Freuder. The Complexity of Some Polynomial Network Consistency algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, (25) 1:65–74, 1984.

[Régin, 1994] Jean-Charles Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the National Conference on Artificial Intelligence (AAAI 94)*, volume 1, pages 362–367, 1994.

[Royle, 2005] Gordon Royle. Minimum Sudoku, 2005. people.csse.uwa.edu.au/gordon/sudokumin.php.

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

[SudokuSolver, 2007] SudokuSolver. Sudoku Solver by Logic, 2007. www.sudokusolver.co.uk.

[Sudopedia, 2007] Sudopedia, 2007. Founder: Ruud van der Werf. www.sudopedia.org.

[van der Werf, 2007] Ruud van der Werf. SudoCue–SudokuSolving Guide, 2007. www.sudocue.net/guide.php.

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