HOW COMPUTERS SOLVE SUDOKU

by

Ian S. Howell

A THESIS

Presented to the Faculty of

The College of Art and Sciences at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Bachelor of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry (adviser) and Professor Mohammad Rashedul Hasan (co-adviser)

Lincoln, Nebraska

December, 2017
The Constraint Systems Laboratory of UNL offers an online Sudoku solver. This tool, developed in 2007, is outdated. It is built in Java 4, which is deprecated and blocked by modern web browsers for security concerns. The goal of this thesis is to design, implement, and test a new online Sudoku solver that uses current web technologies and software development tools and implements advanced mechanisms and algorithms proposed in the area of Constraint Processing (CP), a sub-field of AI.

The artifact resulting from this thesis offers a new modern, intelligent, and interactive graphical interface that guides the user in the process of solving Sudoku puzzles by inference (i.e., without search), search, and by interactively interleaving the two at any point in the solving process. To this end, we implement a variety of constraint propagation algorithms (i.e., AC, GAC, SAC, and SGAC) and various types of search mechanisms (i.e., back-checking, forward checking, and realfull lookahead on binary and nonbinary constraints). An important aspect of our work is the design and implementation of ‘user-friendly’ and interactive functionalities that graphically illustrate to the user the operations of the various propagation mechanisms at various levels of detail as well as the ability to interactively switch between various processing modalities and to undo/redo complex decisions. Finally, our platform is flexible enough to accommodate extensions to other types of Sudoku-based puzzles in the future.

Our tool is useful in a variety of settings for research, education, outreach. In research, it supports the investigation of the level of consistency needed to solve $9 \times 9$
Sudoku puzzles without search and can be useful to explore methods for generating new Sudoku puzzles. In teaching, it is useful to introduce students to modeling problems with constraints, explain consistency properties, and illustrate the operations of constraint propagation and lookahead. Finally, this tool is invaluable in the outreach activities of the Department of Computer Science and Engineering in order to explain to children and to the general public ‘how computers think.’
ACKNOWLEDGMENTS

I would like to acknowledge Dr. Berthe Choueiry for all of her hard work in guiding me through this project and for the delightful conversations and debates we had about user experience during our weekly meetings. They were truly a pleasure and I am grateful for all of her efforts.

I am grateful to Dr. Mohammad Rashedul Hasan for being my co-adviser, reading through this document, and challenging me with insightful questions and comments.

I would also like to acknowledge Robert Woodward, Daniel Geschwender, and Anthony Schneider for their aid in debugging the constraint algorithms and running through ideas with me about this project. They were a pleasure to have in the lab.

This research was supported by NSF Grant No. NSF RI-1619344 and by a UCARE award from the University of Nebraska-Lincoln.
# Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Motivation ............................................. 1

1.2 Contributions .......................................... 2

1.3 Thesis Organization .................................... 2

2 Constraint Processing 3

2.1 Design Principles ..................................... 3

2.2 Background ........................................... 4

2.3 Modeling ................................................ 5

2.4 Variables ............................................... 5

2.5 Domains ................................................ 6

2.6 Propagation Algorithms ................................. 7

2.6.1 Arc Consistency ..................................... 7

2.6.2 The diff constraint .................................. 8
2.6.3 Generalized Arc Consistency ........................................ 8
2.6.4 The \texttt{alldifferent} constraint ................................. 8
2.6.5 Singleton Arc Consistency ........................................... 11
2.6.6 Singleton Generalized Arc Consistency ......................... 12
2.7 Search ............................................................................ 12
  2.7.1 BCSSP ................................................................. 12
  2.7.2 \texttt{LABEL} .......................................................... 13
  2.7.3 \texttt{UNLABEL} ......................................................... 14
  2.7.4 Backchecking and Lookahead ..................................... 14
  2.7.5 Variable Ordering Heuristics: \texttt{dom/wdeg} ................. 15
  2.7.6 Domino Effect ......................................................... 15

3 Application Interface ....................................................... 17
  3.1 Design Principles ....................................................... 17
  3.2 Board .......................................................................... 18
  3.3 Board Features .......................................................... 19
  3.4 Control Panel ............................................................. 20
    3.4.1 Load Section ......................................................... 20
    3.4.2 Solve Section ......................................................... 22
    3.4.3 Submit Panel ......................................................... 27
    3.4.4 Settings Panel ....................................................... 29

4 Database ........................................................................... 32
  4.1 Server ................................................................. 32
  4.2 Schema ................................................................. 32
  4.3 Security ................................................................. 34
## 5 System Architecture

5.1 Overview .................................................. 36

5.2 React ...................................................... 37

5.3 Redux ....................................................... 38

5.4 Constraint Solver ......................................... 39

5.5 File Structure ............................................... 39

## 6 Conclusions ................................................ 41

A Domain Interface ........................................... 42

Bibliography ................................................... 44
List of Figures

2.1 Binary scope .................................................. 5
2.2 Nonbinary scope ............................................. 5
2.3 Sparse Set Pre-Removal ................................... 6
2.4 Sparse Set Post-Removal ................................... 6
2.5 alldifferent .................................................. 9
2.6 Value graph .................................................. 9
2.7 Max matching .................................................. 9
2.8 Directed graph from matching ............................. 10
2.9 Edge classification ........................................... 10
2.10 Post-GAC ..................................................... 10
2.11 BCSSP Algorithm .......................................... 13

3.1 Board with column hover ................................. 18
3.2 Domain removal modal ..................................... 19
3.3 Board features ............................................... 19
3.4 Singleton Assignment ...................................... 20
3.5 Load Panel .................................................... 21
3.6 Solve Section ............................................... 22
3.7 Solve Enforce Sub-Panel ................................... 23
3.8 Solve Propagate Panel ........................................ 23
3.9 Solve Filter Panel ............................................. 24
3.10 Solve Find Panel ............................................. 25
3.11 Solve History Sub-Panel ...................................... 26
3.12 Propagation History Navigation ............................ 28
3.13 Solutions Found History .................................... 28
3.14 Submit Panel ................................................... 29
3.15 Settings Panel ................................................. 30

5.1 Data flow in a React-Redux Application ...................... 39
List of Tables

3.1 User actions and their associated History Record types 27

4.1 Server queries 33

4.2 Structure of table activities 33

4.2 Structure of table activities (continued) 34
Chapter 1

Introduction

In this chapter, we describe the motivation behind reimplementing the Sudoku solver of the Constraint Systems Laboratory.\footnote{http://sudoku.unl.edu} We overview the technology choices and contributions of this work. Finally, we review the organization of this thesis.

1.1 Motivation

The online Sudoku solver developed in the Constraint Systems Laboratory as an undergraduate thesis [Reeson, 2007; Reeson et al., 2007] went through many iterations over the past decade.\footnote{http://sudoku.unl.edu} The first and second versions are feature complete and written as a Java applet. Version 3 was an attempt at redesigning the user interface by a team enrolled in the Human-Computer Interaction course, but unfortunately never properly functioned. Lastly, Version 4 expanded the functionality of Version 2 by implementing a functionality to enforce domain minimality to aid in the construction of new Sudoku puzzles.

Technology has advanced since the project began and most modern web browsers
no longer support Java, especially Java 4; hence, the need to design and implement a new solver that anyone can easily access online. This project constitutes Version 5 of the Sudoku solver. It is written using native web languages: HTML, CSS, and JavaScript. Due to JavaScript’s adoption by the web industry as the sole natively supported scripting language, coupled with its ever increasing performance across platforms, we expect this web application to last 15 to 30 years, making the result worth the invested effort.

1.2 Contributions

This project is a complete rewrite of the old Sudoku solver. However, the visualization tools are original research contributions. They are conceived and implemented to allow increased flexibility and total interoperability between the various algorithmic components. Our goal is to help the general public and youth understand how computers ‘think’ by observing ‘algorithms at work’ and interacting with them. Our tool has already been used in the classroom to explain to students the notions of consistency, consistency properties, and constraint propagation algorithms.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 introduces the constraint processing algorithms and data structures used by our solver. Chapter 3 describes the interface that we designed and its functionality. Chapter 4 reviews database and server configuration. Chapter 5 discusses the architecture of our system. Finally, Chapter 6 concludes the thesis.
Chapter 2

Constraint Processing

In this chapter, we review some basic knowledge about Constraint Satisfaction Problems, present our constraint solver, and discuss the implemented consistency algorithms.

2.1 Design Principles

We intentionally designed our solver as a stand alone engine. It should be easy to integrate in any Constraint Processing project that is written in JavaScript or another language that transpiles to JavaScript. To keep this solver agnostic to the submitted problem instance, we access the data structures through interfaces, which should make it easy to use the solver in different contexts. Also, we take a special care to not modify the input inside the solver. Instead we make copies of the input and modify only the copies, which avoids causing undesirable side affects.
2.2 Background

A Constraint Satisfaction Problem (CSP) is given as \( \langle V, D, C \rangle \) where \( V = \{V_0, \ldots, V_n\} \) is a set of variables, \( D = \{D_0, \ldots, D_n\} \) is a set of domains where \( D_i \) is a set of values that variable \( V_i \) can be assigned, and \( C = \{C_0, \ldots, C_m\} \) is a set of constraints, where each constraint is a relation over a subset of the variables and restrict the combinations of values that the variables can be assigned. The query is to determine whether or not there is an assignment to the variables such that all constraints are satisfied.

The scope of a constraint is the set of variables that the constraint applies to. The arity of a constraint is the cardinality of its scope. A binary constraint is defined over two variables. A constraint is defined by a relation that is a subset of the Cartesian product of the domains of the variables in its scope. The relation lists the combinations that are allowed by the constraint. Each of which is called a support or a solution to the constraint. For a given binary constraint \( C_i \) with scope \( \{V_0, V_1\} \), a support for a value \( v_0 \in D_0 \), where \( D_i \) is the domain of \( V_i \), is \( v_1 \in D_1 \) where the assignment of \( (v_0, v_1) \) to \( (V_0, V_1) \) is consistent. The diff constraint is an example of a binary constraint that allows the two variables in its scope to be assigned different values. If the two variables are assigned the same value, the the diff constraint is said to be violated and the assignment inconsistent. A global constraint is a ‘template’ that captures a relation between a non-fixed number of variables. An example is the constraint \( \text{alldifferent}(V_1, \ldots, V_k) \), which specifies that the values assigned to the variables \( V_1, \ldots, V_k \) must be pairwise distinct.
2.3 Modeling

Our model of the 9×9 Sudoku puzzle represents each cell as a variable whose domain is {1,...,9}.

Similar to previous work [Simonis, 2005; Reeson, 2007], we use two constraint models: one with binary constraints and the other with nonbinary constraints:

- The binary model has one binary \texttt{diff} constraint between every two variables that appear on the same row, column, or block of the Sudoku board, see Figure 2.1. This model results in 810 \texttt{diff} constraints for the 9×9 Sudoku puzzle.

- The nonbinary model has one nonbinary \texttt{alldifferent} constraint defined over all the variables of a row, column, or block of the board, see Section 2.6.4 and Figure 2.2. This model results in 27 \texttt{alldifferent} constraints of arity 9 for the 9×9 Sudoku puzzle.

2.4 Variables

Variables are modeled after each cell in the 9×9 Sudoku problem. Each model keeps track of its domain, neighbors, binary constraints that have the variable in its scope,
and nonbinary constraints that have the variable in its scope. This redundant information allows for quickly testing neighboring variables in AC and other various optimizations.

## 2.5 Domains

Because this component of the project could be used in the future for other applications, the `Domain` interface was designed to work for any domain implementation, see Appendix A. We describe below our implementation designed to improve search performance.

During search, the most common operations to perform on a domain are removal and insertion. We implement domains as *sparse sets* [Schaus et al., 2013] in order to execute these operations in constant time. Sparse sets map each element in the domain to its position in a list with a ‘cutoff’ point, which allows us to remove or restore the element by swapping its position. See Figure 2.3 and Figure 2.4 for an example of removing $c$ from a sparse set.\(^1\) In addition, sparse sets can restore sections of its elements at once, by moving the cutoff point arbitrary distances.

\[
\begin{array}{cccc}
\text{size}_D & 1 & 4 & 3 & 2 \\
\text{map}_D & 1 & 4 & 3 & 2 \\
\text{dom}_D & a & d & c & b \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{size}_D & 3 & 4 & 1 & 2 \\
\text{map}_D & 3 & 4 & 1 & 2 \\
\text{dom}_D & c & d & a & b \\
\end{array}
\]

Figure 2.3: Sparse Set Pre-Removal  \hspace{1cm}  Figure 2.4: Sparse Set Post-Removal

\(^1\)The values to the left of $size_D$ are considered to be in in the domain and those to the right are considered to be removed.
2.6 Propagation Algorithms

We implement the following algorithms for enforcing consistency:

- A version of algorithm AC3 for arc consistency [Mackworth, 1977] that is tailored for the `diff` constraint.
- The algorithm `AllDiff` for generalized arc consistency [Régisn, 1994].
- We implement SAC on top of our specialized AC3.
- We implement SGAC on top of `AllDiff`.

2.6.1 Arc Consistency

The property of Arc Consistency (AC) ensures that every value has a support in the domain of every other variable. Differently stated, “any value in the domain of a variable can be extended consistently by any other variable” [Dechter, 2003, p. 54]. Any algorithm of arc consistency enforces this property on a CSP.

One such algorithm, AC3 maintains a dynamic (set) queue of tuples of variables. For each binary constraint defined over variables $V_i, V_j$, the tuples $(V_i, V_j), (V_j, V_i)$ are added to the queue. The algorithm pops a tuple $(V_i, V_j)$ from the queue and removes from the domain of $V_i$ values that lack a support in the domain of $V_j$. The function that implements this operation is called `Revise3(V_i, V_j)`. If the domain of $V_i$ is updated, all tuples $(V_x, V_i)$ with $V_x \neq V_j$ are added to the queue in case they are not already there because values in the domain of $V_x$ may have lost their support in the domain of $V_i$. The algorithm repeats the above until the propagation queue is empty. AC3 runs in time $O(n^2d^3)$, where $n$ is the number of variables in the problem and $d$ the size of the largest domain.
2.6.2 The diff constraint

In the binary model of the Sudoku, the only binary constraint is the different constraint, which restricts the variables in its scope to be assigned different values. Thus, for two variables $V_i, V_j$ in the scope of a different binary constraint, a value for variable $V_i$ may lack a support in the domain of $V_j$ if and only if the domain of $V_j$ is a singleton. For the sake of practical efficiency, we implement a special version of $\text{REVISE3}(V_i, V_j)$ for the different constraint that updates the domain of $V_i$ only when the domain of $V_j$ is a singleton. When this is the case, we simply remove the unique value in the domain of $V_j$ from that of $V_i$.

2.6.3 Generalized Arc Consistency

Generalized Arc Consistency (GAC) is the generalization of Arc Consistency to non-binary constraints. The GAC property ensures that every value in the domain of every variable in the scope of a given nonbinary constraint appears in a solution to the constraint, that is, it can be consistently extended to all the other variables in the scope of the constraint.

2.6.4 The alldifferent constraint

The alldifferent constraint has the same definition as its binary version. However, enforcing the GAC property is not a straightforward operation because we would like to avoid enumerating all permutations of values to the variables. Régis [1994] proposed an efficient algorithm $\text{ALLDIFF}$ for enforcing GAC on a nonbinary constraint in $O(s^2d^2)$ where $s$ is the arity of the constraint and $d$ the number of distinct values in the domains of the variables.
For a given constraint such as the one shown in Figure 2.5, ALLDIFF first builds a value graph. The value graph is a bipartite graph with, on the left, vertices representing the variables in the scope of the constraint and, on the right, vertices representing all the values in the domains of those variables. An edge links a variable with a value in its domain. Figure 2.6 shows the value graph of the alldifferent constraint in Figure 2.5.

The algorithm then finds a maximal matching in the value graph using any maximal matching algorithm such as the one of Hopcroft and Karp [1973]. Figure 2.7 shows a maximal matching for the example of Figure 2.5. If the maximal matching does not cover all the variables, then the constraint is inconsistent. Otherwise, we use the obtained maximal matching to identify:

\[\text{Such a situation never arises in a consistent well-formed puzzle.}\]
• All edges that do not appear in any maximal matching. Those edges must be removed.

• All edges that appear in every maximal matching. Those edges are called vital and correspond to assignments that the variables must take.

• Edges that appear in some but not all maximal matching. Those are values that must be kept in the domain of the variables and are GAC.

To this end, we orient the edges of the bipartite graph (i.e., value graph) as follows: the edges that appear in the matching are directed from the variables to the values and all other edges are directed from the values to the variables. Figure 2.8 shows how the bipartite graph of Figure 2.6 is directed based on the matching in Figure 2.7.

Now, we identify all strongly connected components. A strongly connected component is a collection of vertices within a graph where there is a path from every
vertex to every other vertex. Any edge appearing in a strongly connected component is guaranteed to appear in some but not all matchings. For each such edge, the value must remain in the domain of the corresponding variable. In Figure 2.9, the strongly connected components are highlighted in blue.

In the case of a well-formed Sudoku, we are guaranteed not have any free vertices in the matching.\(^3\)

All edges that appear in the matching but not in a strongly connected component are necessarily vital edges. Those edges are shown black in Figure 2.9. All remaining ones are edges representative to inconsistent variable-value pairs: values that must be removed from the domains of their corresponding variables. Those edges are shown red in Figure 2.9.

Figure 2.10 shows the alldifferent constraint of Figure 2.5 after enforcing GAC.

When propagating GAC over a CSP, we exploit the fact that we already know a maximal matching after the first run to quickly find new matchings. By removing the edges removed by other constraints from the matchings, we build maximal matchings from non-maximal ones using Hopcroft and Karp [1973].

### 2.6.5 Singleton Arc Consistency

The Singleton Arc Consistency (SAC) property ensures that for each value in every variable’s domain, the value can be assigned to the variable and the resulting state is arc consistent [Debruyne and Bessière, 1997].

Like in AC3, the propagation of SAC maintains a dynamic queue of variables to be checked. Initially, this queue is set to all the variables in the CSP. The algorithm then pops a variable \(V_i\) and instantiates it to a value \(v_i\) from \(D_i\). The algorithm then

\(^3\)Because the number of vertices on the left of the value graph is equal to the number of vertices on its right.
runs AC3. If this filtering causes inconsistency then \( v_i \) is removed from \( D_i \). Then SAC undoes the filtering. The process is repeated for all the values in \( D_i \). The process is repeated until not values can be removed from any variables.

2.6.6 Singleton Generalized Arc Consistency

The Singleton General Arc Consistency (SGAC) property ensures that for each value in every variable’s domain, the value can be assigned to the variable and the resulting state is generalized arc consistent [Debruyne and Bessière, 1997].

The propagation of Singleton Generalized Arc Consistency (SGAC) works in the same way that the propagation of SAC does, except instead of enforcing AC, it enforces GAC.

2.7 Search

This section describes the different components of the search algorithm used to find solutions of the Sudoku state.

2.7.1 BCSSP

Our search algorithm is a modified version of the binary constraint satisfaction search problem, or BCSSP, procedure of Prosser [1993], see Figure 2.11.

The BCSSP algorithm provides a framework for using generic \texttt{LABEL} and \texttt{UNLABEL} functions that assign and unassign variables during search. BCSSP instantiates \( i \) to 1, \textit{consistency} to \texttt{True}, and \textit{status} to \texttt{unknown}. While the status of the problem is \texttt{unknown}, if \textit{consistent} is true \texttt{LABEL} is executed to instantiate the current variable, otherwise (i.e., \textit{consistent} is \texttt{False}) \texttt{UNLABEL} is executed to backtrack to
Data: The size of the graph: \( n \), \textsc{Label} and \textsc{Unlabel} methods

Result: If the problem has a solution or is impossible

\begin{algorithm}
\begin{algorithmic}
\STATE \textit{consistent} \leftarrow \text{True};
\STATE \textit{status} \leftarrow \text{unknown};
\STATE \textit{i} \leftarrow 1;
\WHILE {\textit{status} = \text{unknown}}
\IF {\textit{consistent}}
\STATE \textit{i} \leftarrow \textsc{Label}(\textit{i}, \textit{consistent});
\ELSE
\STATE \textit{i} \leftarrow \textsc{Unlabel}(\textit{i}, \textit{consistent});
\ENDIF
\IF {\textit{i} > \textit{n}}
\STATE \textit{status} \leftarrow \text{solution};
\ELSEIF {\textit{i} = 0}
\STATE \textit{status} \leftarrow \text{impossible};
\ENDIF
\ENDWHILE
\end{algorithmic}
\end{algorithm}

Figure 2.11: BCSSP Algorithm [Prosser, 1993]

the previous level. \textsc{Label} and \textsc{Unlabel} set \textit{consistent} and return the level \( i \) on which to operate. If \( i \) reaches 0, the problem has no solution. If \( i \) reaches a number greater than the number of variables in the CSP, a solution has been found. Our implementation of BCSSP keeps running after a solution is found and records all consistent solutions.

2.7.2 Label

The \textsc{Label} function represents a forward movement in the search tree, i.e., the instantiation of a variable, or the assignment of a domain element to a variable. In our implementation, when we use \textsc{Label}, we attempt to assign a value \( v_i \) in \( D_i \) to \( V_i \) and then run GAC to check for inconsistency and filter the domains of the future variables, i.e., variables to be instantiated in the future. We keep track of the reductions that GAC makes to the domains of the variables for backtracking. \textsc{Label} returns \( i \leftarrow i + 1 \) and sets \textit{consistency} to \textbf{True} if an assignment was successful. If all
assignments were unsuccessful, it returns $i$ and sets $consistency$ to $False$.

### 2.7.3 Unlabel

The Unlabel function executes a backtracking in the search, i.e., it moves to the last instantiated variable and makes it the current variable. It removes the assigned value from the domain. Importantly, it restores all the values removed as the effect of the instantiation that was just undone by the lookahead mechanism from the domains of the future variables. If no values are left in the domain of the current variable domain, then it sets $consistency$ to $False$, otherwise it sets $consistency$ to $True$.

### 2.7.4 Backchecking and Lookahead

It is important during search to expand only paths that are consistent with the constraints. To this end, we implement a backchecking procedure and two lookahead procedures.

At each instantiation, backchecking (BC) ensures that the value with the current variable is consistent with each of the past assignments. Backchecking need only check the binary constraints.

Lookahead removes from the domains of the future variables values in their domains that are not consistent with the current instantiation. We implement two types of lookahead: forward checking (FC) and realfull lookahead (RFL) for each of the binary and nonbinary models. FC revises the domains of only the variables connected by a constraint to the current variable; RFL enforces AC/GAC on the entire future subproblem. Naturally, when using any kind of lookahead, backchecking becomes obsolete because the values remaining in the domains of the future variables are necessarily consistent with all past instantiations.
2.7.5 Variable Ordering Heuristics: \textit{dom/wdeg}

The order in which the variables are instantiated during search affects the size of the explored search tree. The common wisdom is to assign \textit{the most constrained variable first}. Various heuristics were proposed in the literature to implement this principle. We use the dynamic \textit{dom/wdeg} heuristic [Boussemart \textit{et al.}, 2004].

At each call to \texttt{Label}, the \textit{dom/wdeg} heuristic [Boussemart \textit{et al.}, 2004] chooses the variable that has the smallest value of \textit{dom/wdeg} to instantiate, where:

- \textit{dom} is the size of the current domain of the variable
- \textit{deg} is the degree of the variable in the future subproblem
- \textit{w} is the weight of the variable computed as the summation of the weights of the constraints that apply to the variable.

At the beginning of search, the weights of all constraints are initialized to 1. During lookahead, if revision to enforce AC/GAC causes a domain wipe out, the weight of the culprit constraint is incremented.

2.7.6 Domino Effect

After lookahead, if the domain of any future variable becomes a singleton, the variable is immediately instantiated, thus saving one application of the \textit{dom/wdeg} heuristic. Further, the instantiation of this singleton and the application of lookahead can cause the domain of another future variable to become a singleton. The resulting chain of singletons is called the ‘Domino Effect’ and can result in much quicker search times. Therefore, if any singletons exist while applying a variable ordering heuristic, including \textit{dom/wdeg}, they are immediately moved to the front of the ordering.
Summary

In this chapter, we reviewed the various constraint-processing algorithms implemented in our tool.
Chapter 3

Application Interface

In this chapter, we describe the various components of the Single-Page Application (SPA) that allows users to interact with our Sudoku solver. The interface’s functionalities are organized in the three main panels of the SPA. We describe the Board and Board Features panels, while going into further details of the extensive Control Panel.

3.1 Design Principles

Version 2 is the version of the Sudoku solver that is currently available online. It has three pages: one page for the solver, one page for loading puzzles, and one page for entering and submitting new puzzles. These three pages form an incohesive application, with duplicate functionalities split across multiple pages.

In our new design, we choose to maintain most of the functionalities of the old Sudoku solver while adding a few new advanced features and creating a more cohesive experience. To create a more cohesive User Experience (UX), we choose to make our interface an SPA and design UI components to look and feel distinct, based on the functionalities that they provide.
3.2 Board

The board’s primary purpose is to convey the current state of the Sudoku problem. Its secondary purpose is to highlight certain details within the application, e.g., to aid users in visualizing filtering and constraint scopes, see Figure 3.1.

![Board with column hover](image)

**Figure 3.1: Board with column hover**

The board provides many functionalities to users, including:

- *Assign a value to a variable.* This is done by either clicking within the cell and entering a value, or simply hovering over the cell and pressing a number key.

- *Remove a value from a variable’s domain.* This is done by right clicking on the cell that you want to remove the value from and then clicking on the values you want to remove\(^1\), see Figure 3.2.

\(^1\)To reassign a cell, a user would first remove the value from the cell and then assign a new value
• *Perform local AC or GAC.* This is done on a row, column or group by clicking on the buttons on the border of the board and in the Solver Panel, see Section 3.4.2.1.

![Figure 3.2: Domain removal modal](image)

In addition, the columns and rows are numbered and lettered, respectively, to properly supplement messages in the Solver History, see Section 3.4.2.5.

### 3.3 Board Features

The board features panel, located in the bottom, left hand corner of the interface, allows the user to interact with the board in a few basic ways. It allows the user to toggle the viewing of cell domains, reset the board, and interact with singleton assignment.

![Figure 3.3: Board features](image)

Singletons are domains that have a single element in them, which means that the cell cannot possibly have any other value. Therefore, we assign the value to the cell,
see Figure 3.4. The buttons here either search for and assign singletons immediately, or automatically assign them as they show up through consistency propagation and domain removals.

![Figure 3.4: Singleton Assignment](image)

### 3.4 Control Panel

The Control Panel is located on the right hand side of the interface and provides the four pages of functionality: Load, Solve, Submit, and Settings.\(^2\)

#### 3.4.1 Load Section

The Load Section allows users to load various submitted puzzles into the solver, see Figure 3.5. The first column contains the puzzle name and can be sorted alphabetically. The filter box underneath the sorting button filters by Sudoku type. Currently the only Sudoku type is ‘base’, but this solver can be extended to other Sudoku types in the future. The second column is level and is sortable and filterable by the fractional hardness of a puzzle, which is calculated by dividing the level by the max level. For instance a puzzle with difficulty level 3 with a max of 5 will rank higher than a puzzle with level 4 and max of 9.

The third column is the minimum level of consistency needed to solve a problem and is sortable and filterable. In the ranking, AC is strictly weaker than GAC, with

\(^2\)A cog wheel represents the Settings section.
the overall ranking being: AC ≺ GAC, SAC+GAC (because GAC and SAC are not comparable), SAC ≺ SGAC ≺, Un solved (i.e., by the implemented consistency algorithms). The fourth column is the number of solutions each problem has and is sortable and filterable on that number. The last column is the source, or where each puzzle comes from. This is sortable alphabetically. Each time a filter changes, the number of matching puzzles, indicated under the name-column title, updates and is presented to the user immediately.
3.4.2 Solve Section

The Solve Section helps users to solve Sudoku puzzles, see Figure 3.6.

![Figure 3.6: Solve Section](image)

There are many different tools in this section that allow users to interactively solve puzzles, test for solutions to puzzles, and visualize how the computer synthesizes constraints to use propagation techniques.
3.4.2.1 Solve Sub-Panel: Enforce

The sub-panel shown in Figure 3.7 allows users to enforce AC or GAC on a block of cells. This functionality is supplementary to the row and column controls on the board itself, see Section 3.2.

![Figure 3.7: Solve Enforce Sub-Panel](image)

3.4.2.2 Solve Sub-Panel: Propagate

This sub-panel, see Figure 3.8, allows users to run constraint propagation algorithms on the current puzzle. These propagations are then tracked in the Solve History Sub-Panel, see Section 3.4.2.5. The possible propagations include: AC, GAC, SAC, and SGAC, see Section 2.6.

![Figure 3.8: Solve Propagate Panel](image)
3.4.2.3 Solve Sub-Panel: Filter

This sub-panel, see Figure 3.9, allows users to perform consistencies as they assign values on the board. These filters include:

- **None**: No filtering is applied on assignment.

- **Back Checking (BT)**: No filtering is applied, but errors are marked on assignment (i.e., backchecking).

- **Binary Forward Checking (FC)**: Forward checking using binary constraints is applied after assignment.

- **Binary Real Full Lookahead (RFL)**: AC is enforced after every assignment.

- **Nonbinary Forward Checking**: Forward checking using nonbinary constraints is applied after assignment.

- **Nonbinary Real Full Lookahead (GAC)**: GAC is enforced after every assignment

![Figure 3.9: Solve Filter Panel](image-url)
3.4.2.4 Solve Sub-Panel: Find

The Find Sub-Panel, see Figure 3.10, contains two buttons. The first, labeled ‘Solvable?’, checks if solutions exist, given the current state of the solver. A solutions history is added to the Solve History Sub-Panel, containing all solutions found, and kept for reference as the user continues to assign values on the board. The second button, labeled Print PDF, will generate a PDF of the Sudoku puzzle with its assignments and current, enumerated solutions.\(^3\)

![Find Sub-Panel](image)

Figure 3.10: Solve Find Panel

3.4.2.5 Solve Sub-Panel: History

The Solve History Sub-Panel, see Figure 3.11, keeps track of all user actions on the board and aids users in visualizing how a computer synthesizes constraints to solve Sudoku problems. For each action a user takes, a history element is added to this sub-panel, see Table 3.1.

When a user takes an action, it appears in the form of a history within the Solve History Sub-Panel. The user can navigate back and forth through their actions to see the resulting effects of each action on the board.

For histories that include filtering, a child history drop-down is present. If filtering took only one step, two buttons will be present, a Before button and an After button.

\(^3\)This functionality is to be implemented
Clicking on either will show the state before or after the filtering takes place, respectively. If more than one filtering step took place, a slider is shown instead, which allows the user see the state of the problem at arbitrary points during the filtering process. Setting the slider to an arbitrary position causes the board to render domain values removed in that step in red and domain values to be removed in later steps in bolded black, see Figure 3.12. If the propagation done is AC or GAC, the scope of the constraint used to do the filtering is highlighted. If the propagation is not AC or GAC, the variables affected by the filtering are highlighted.

For solution histories, all solutions can be shown be clicking the expansion button, see Figure 3.13. This will display a list of numbered solutions, with cells that differ from solution to solution, or between all solutions; according to the setting, highlighted green.
Table 3.1: User actions and their associated History Record types

<table>
<thead>
<tr>
<th>Action</th>
<th>History Record Type</th>
<th>Description of contained data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign Value</td>
<td>AssignHistoryRecord</td>
<td>All assignment actions and immediate filtering actions.</td>
</tr>
<tr>
<td>Load Puzzle</td>
<td>LoadHistoryRecord</td>
<td>Loaded puzzle assignments.</td>
</tr>
<tr>
<td>Filtering Method Change</td>
<td>MethodHistoryRecord</td>
<td>Changes to the interactive filtering method and immediate domain removal actions after said method is applied to all assignments.</td>
</tr>
<tr>
<td>Propagation Runnings</td>
<td>RemoveHistoryRecord</td>
<td>All removal actions along with method used, remaining domains after removal, and variables affected.</td>
</tr>
<tr>
<td>Singleton Assignments</td>
<td>SingletonAssignHistoryRecord</td>
<td>Singleton assignments.</td>
</tr>
<tr>
<td>Solve</td>
<td>SolutionsHistoryRecord</td>
<td>All solutions found, along with the status of possible solutions, i.e., if the problem was inconsistent to start with, solutions found, or impossible puzzle.</td>
</tr>
</tbody>
</table>

3.4.3 Submit Panel

The Submit Panel, see Figure 3.14, enables users to submit new puzzles to the database, so that others can view and solve them. This panel lets users enter the following metadata attributes about the puzzle: Sudoku type, source, difficulty level, name of the instance, name of the instance in the original source, and description. These parameters, along with the number of solutions and minimal solvable consistency, are stored in the database, see Chapter 4. Once the metadata has been entered, users can test the number of solutions their assignments generate and the minimum consistency needed to solve their puzzle. They can then submit their puzzle to the database if the number of solutions is known and less than 100. If the puzzle has already been submitted to the database, the user is warned and not allowed to resubmit. The same process prevents two puzzles in the database from having the same
name, as submitting a different puzzle with the same name would lead to confusion.

Finding solutions to a user’s puzzle works differently than finding a normal set of solutions; only the current assignments are considered, not user removed domain values. To find these solutions, a blank copy of the puzzle is made and assignment actions are taken. Then solutions are found, separate from the actual Sudoku state represented on the board. This keeps the search algorithm’s changes separate from
the current state and ensures that the puzzle with just the assignments given has solutions.

To check the minimum consistency needed to solve the puzzle, users can click the ‘Test Consistencies’ button. This button results in either the minimum necessary consistency, which can be AC, GAC, SAC, GAC + SAC, or SGAC, an error pertaining to the consistency of the problem and at what level the inconsistency is discovered, or state that the problem is unsolved by any of the consistency algorithms.

### 3.4.4 Settings Panel

The Settings Panel, see Figure 3.15, houses a few global user settings. Currently there are three settings made available:
• *Hide tooltip explanations*: For those that already know what consistencies do and how they work, this option stops the explanations from displaying when hovering over filtering buttons. This value is defaulted to `False`, as this application is designed to explain everything and aid in the learning of others.

• *Stop search after ‘so many’ solutions*: This stops search after however many solutions users deem to be ‘too many’. This setting affects both the board state search and submit search algorithms. The default is 50 and max is 100.

• *Solution cell diff style*: When viewing solutions, there are two different styles to calculating differences. The pairwise style, designated with ‘2’, calculates changes and displays them to the user on a pair by pair basis. For example, if there are three solutions, the differences between solutions one and two will be highlighted in green on solution two, while the differences between solution two and solution three will be highlighted in green on solution three. The second style, designated with ‘ALL’ calculates the cells with differences between at least one pair of solutions and highlights those cells on all solutions.
Summary

In this chapter, we reviewed the design of the graphical interface and the main functionalities currently implemented.
Chapter 4

Database

In this chapter, we review the Sudoku server interface, database, and security.

4.1 Server

The basic PHP interface has only one endpoint for clients to access. This endpoint, however, is a query style endpoint, meaning that different data objects can be passed to it to achieve different results, see Table 4.1.

4.2 Schema

The Sudoku database is not changed from previous iterations, which allows the previous versions to still interact with it correctly. The database contains a single table, puzzles, that houses all of the puzzles that we have in storage that others can access through the Sudoku application, see Table 4.2.
### Table 4.1: Server queries

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>Query</th>
<th>Response Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td></td>
<td>All puzzles currently in database as a JSON object.</td>
</tr>
<tr>
<td>POST</td>
<td>INSERT_NEW PUZZLE</td>
<td>Checks the given puzzle against the database and if it does not exist, adds it to the collection.</td>
</tr>
<tr>
<td>POST</td>
<td>CHECK_NAME</td>
<td>Checks to see if there exists a puzzle with a given name. If there does exist one, it returns the puzzle’s ID, otherwise it returns an empty OK response.</td>
</tr>
<tr>
<td>POST</td>
<td>CHECK_VALUES</td>
<td>Checks to see if there exists a puzzle with the same assignments. If there does exist one, it returns the puzzle’s ID, otherwise it returns an empty OK response.</td>
</tr>
</tbody>
</table>

### Table 4.2: Structure of table activities

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>PuzzleID</td>
<td>int(11)</td>
<td>No</td>
<td>NULL</td>
</tr>
<tr>
<td>SudokuType</td>
<td>varchar(25)</td>
<td>No</td>
<td>Base</td>
</tr>
<tr>
<td>Source</td>
<td>varchar(50)</td>
<td>Yes</td>
<td>NULL</td>
</tr>
<tr>
<td>MinDiffLvl</td>
<td>int(2)</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>DiffLvl</td>
<td>int(2)</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>MaxDiffLvl</td>
<td>int(2)</td>
<td>No</td>
<td>5</td>
</tr>
<tr>
<td>PuzzleName</td>
<td>varchar(50)</td>
<td>No</td>
<td>Sudoku</td>
</tr>
<tr>
<td>OrigName</td>
<td>varchar(50)</td>
<td>No</td>
<td>NULL</td>
</tr>
<tr>
<td>Description</td>
<td>varchar(150)</td>
<td>Yes</td>
<td>NULL</td>
</tr>
<tr>
<td>PuzzleValues</td>
<td>varchar(100)</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>Solutions</td>
<td>int(2)</td>
<td>Yes</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 4.2: Structure of table activities (continued)

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Null</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency</td>
<td>varchar(10)</td>
<td>No</td>
<td>Unsolved</td>
</tr>
</tbody>
</table>

4.3 Security

While this application is run primarily on the client side with little networking, it does pull data from the Sudoku database for known puzzles. This interaction with the database requires scrutiny, as malicious code could come from the database and run arbitrary scripts on the client’s machine.

To be able to protect our users from these sorts of attacks and others, various security concerns need to be taken into account. The largest security concerns, by far, are SQL Injection and Cross Site Scripting (XSS), the kind of attack described previously. SQL Injection is a form of security breach that allows attackers to execute arbitrary SQL commands on the MySQL database. The kinds of vulnerabilities that can make this happen are ones that do not clean or manage arbitrary text well. XSS works in nearly the same way; when JavaScript is entered into the database, but is parsed as code instead of text, running arbitrary scripts on a client’s machine when they pull that data.

To prevent both of these vulnerabilities, we escape our text fields with special characters, using PHP functions built for security. Escaping text fields encodes all text as actual text and not as possible commands. For example if a `<script>` tag was embedded in one of the text fields, it would be escaped to `\<script\>`, rendering the script tag invalid.
Summary

This chapter reviewed the Sudoku server interface, database, and security.
Chapter 5

System Architecture

In this chapter, we review the system architecture, along with the usage of the web application packages: React and Redux.

5.1 Overview

The architecture of this solver is a React-Redux web application written in JavaScript, with the Constraint Solver written in Typescript. React and Redux are libraries written by Facebook that together create manageable and maintainable web applications. React manipulates the DOM and physical HTML elements of the web page, while Redux manages the application state and actions incurred by users.

These packages work together to create maintainable, transparent web applications that feature logical, unidirectional data flow with state-determined markup. These components are written in JavaScript, a dynamically typed language that enhances development speed and flexibility. The Constraint Solver is written in TypeScript, a statically typed version of JavaScript, which enables clean, type-enforceable

\(^1\)https://www.typescriptlang.org/
development, before transpiling into usable JavaScript code.

5.2 React

React\(^2\) is a DOM manipulation framework written by Facebook, released to the public in 2013. Facebook created React to control the DOM and render large, frequent changes efficiently. One of the major drawbacks of working with the DOM, in general, is that it can take a lot of time to do many individual tasks. Therefore, lots of small changes to the DOM can slow down page render times and lessens the amount of time that calculations are allowed to take place.

React helps with this problem by calculating the necessary changes needed to update an application to a wanted state and then updating all changes in a batch update. This is done by keeping a copy of the DOM, a virtual DOM object, that React will calculate differences against. Changing the virtual DOM takes little time, so developers can change it without major drawbacks. When there are differences between the DOM and the virtual DOM after the event loop has completed, React performs a batch update, updating the DOM to match the virtual DOM all at once, instead of in many little changes. This procedure cuts down on unneeded overhead and increases the performance of web applications.

To use React in a web app, developers create components. Components can render HTML to the screen and be used within each other. For instance, a tree component could display a tree with many apple components on its branches. These components also define event handlers that the user interacts with, for instance, when the user clicks a submit button on a form on click handler will run code to submit the form.

\(^2\)https://reactjs.org/
5.3 Redux

React can handle components and updating the DOM well, but with anything larger than a small application, application state can become unmanageable and unmanageable. Data has to be passed around components in a opaque and unintuitive way. To fix this problem, Facebook created Redux\(^3\), a package that centralizes and globalizes the application state while enforcing unidirectional data flow. This data flow is followed through a schema:

- **Actions** are called from containers and separated from the changes they make to the state.

- **Components** render elements into HTML and send messages back to their containers when events occur, e.g., a click of a button.

- **Reducers** store the application state and update their state according to actions called.

- **The Store** is a global store that the actions are dispatched to, which forwards the actions to the reducers.

- **Containers** link the state in reducers and possible actions a component can take together and pass these as properties to the components.

Figure 5.1 provides a visualization of the way that data flows in this one-way environment.

If an action will take a while to perform, or needs to call other actions, for example a callback after loading some data from another page asynchronously, thunks can be created. Thunks pass the global store and dispatch functions to a ‘thunk action’,

\(^3\)http://redux.js.org/
so that the thunk can perform necessary work and then call other actions that will actually affect the state.

### 5.4 Constraint Solver

The Constraint Solver is written agnostically, so that it can function with any other JavaScript based constraint application. Although only constraints and domain structures necessary for this project were created, the interfaces and algorithms were created with extensibility in mind so that other projects could expand and improve upon this one.

### 5.5 File Structure

Due to the nature of a React-Redux web application, much of the file structure is pre-determined. Thus, we have the following key folders:

- **actions**: Contain actions that can be done to alter the state.

- **components**: Contain components that make up the actual user interface.

- **containers**: Contain components that link functionality into the components.

- **models**: Contain local models, like history functionality.
- **records**: Contain state models, that are stored immutably.

- **reducers**: Contain the application state and are modified by actions.

- **selectors**: Calculate properties of components using memoization.

**Summary**

In this chapter, we described the various dependencies and design decisions made throughout this project. We familiarized ourselves with the basic file structure used and principles this application is built upon. This chapter is a must read for anyone wanting to add to the system functionalities.
Chapter 6

Conclusions

In this thesis, we described the various components of the interactive Sudoku solver that we built. This application is already live on the sudoku.unl.edu subdomain and will soon be the default solver. Because of the rapid development nature of this project, testing was mostly done manually, although a unit-testing framework for the CSP algorithms is set up and in use.

Two useful extensions of this project include expanding to full functionality of Version 4 (i.e., using minimality to aid in construction) and implementing user-interface testing. Other extensions of this project would include implementing the ‘hint’ functionality of Version 2 and implementing other consistency algorithms and their visualizations, which can be quickly implemented using the graphics packages in the extensive NPM repository\(^1\).

\(^1\text{https://www.npmjs.com/}\)
Appendix A

Domain Interface

The following are the methods seen as needed for usable domains in our Constraint Solver and make up the Domain interface:

- contains\((value)\) → boolean: Checks to see if the domain contains the value. Returns true if it does contain and false if not.

- current\()\) → value\[
\]: Returns an array containing all of the values currently in the domain.

- next\()\) → value: Returns the first value in the domain, if one exists. This function is useful when searching through the domain for a consistent value.

- original\()\) → value\[
\]: Returns the original elements of the domain in an array.

- remove\((value)\) → void: Removes the value from the domain.

- restore\()\) → void: Restores all values within the domain.

- restore\((value)\) → void: Restores that value within the domain

- size\()\) → number: Returns the current size of the domain.
• COPY() \rightarrow Domain: Returns a copy of the domain in its current state.
Bibliography


