Spring 2016

Nested Monte Carlo Tree Search as Applied to Samurai Sudoku

Laura Finley
Governors State University

NESTED MONTE CARLO TREE SEARCH AS APPLIED TO SAMURAI SUDOKU

By

Laura Finley

B.S., Loyola University Chicago, 2010

THESIS

Submitted in partial fulfillment of the requirements

For the Degree of Master of Science,

With a Major in Mathematics

Governors State University

University Park, IL 60466

2016

Acknowledgements

First, special thanks go out to my advisor, Dr. Tweddle, for all the advice, time, and support he has provided. I could not have done it without his guidance, and I could not have asked for a better mentor. Many thanks to my committee members, Dr. Zhang and Dr. Galante, for taking the time from their busy schedules to work with me and provide advice. It has truly been a wonderful learning experience thanks to my committee.

I would like to thank C.J. Sheehe for the thought-provoking discussions that helped me find issues in my code and make improvements to it. It would not have been as successful without his help.

I am grateful for the assistance from my sister, Rose Nee, in setting up puzzles, as well as the endless support and encouragement.

Finally, I thank my husband, Ryan Finley, for all of his help and support. From setting up puzzles with me to helping me maintain balance between my responsibilities, he has done so much to help me with this project, and I am truly grateful.

Contents

Abstract

As Sudoku has come into prominence as a favorite logic puzzle, mathematicians and computer scientists alike have analyzed the game for interesting properties. The large search space presents a challenge for both generating and solving Sudoku puzzles without relying on techniques that simply permute a valid puzzle. These permutations result in puzzles that are essentially the same since they follow the same solution path. Many Sudoku generating or solving programs rely on brute-force methods to avoid this pitfall, but this is inefficient since there is no heuristic to navigate the huge search space. A nested Monte Carlo tree search has some basis in brute-force methods, but guides the search in order to achieve better results by using random games within nested search stages. In this paper, we show that when the nested Monte Carlo search algorithm is implemented for solving Samurai Sudoku, a version of Sudoku in which a standard Sudoku puzzle is placed with four other standard Sudoku puzzles overlapping on each of the corners, it performs better than a completely random brute-force algorithm. Additionally, an improvement to the nested Monte Carlo search is made by implementing a heuristic that is used at each level of search.

Keywords: Sudoku, Samurai Sudoku, Monte Carlo search, rollouts, solver, tree search

1 – Introduction

The popular puzzle game Sudoku was introduced to the world in *Dell Pencil Puzzles and Word Games* in the May 1979 edition. The creator was Howard Garns, who unfortunately never saw the success of his puzzle. He called it “Number Place” when it was first published, but a magazine in Japan later picked up the puzzle in 1984, naming it “Sudoku.” The puzzle did not become popular until Wayne Gould wrote a computer program in 1997 that was able to create Sudoku puzzles automatically and made a deal in 2004 with the *London Times* to have the puzzles published. The following year, he struck a similar deal with the *Daily Telegraph*, and the puzzle spread like wildfire from there. However, Sudoku is actually a variation of Latin Squares, which were first created in the Middle Ages, but later named and studied by Leonhard Euler. A Latin Square is an *n x n* matrix that is filled with *n* values in such a way that each symbol appears only once in each row and column. Clearly, Sudoku puzzles are a subclass of Latin Squares, since they follow the same rules with an added condition of dividing the grid into blocks in which each value must also appear only once (Delahaye, 2006).

1.1 – Definitions

The puzzle is played on a *Sudoku grid*, which is a 9 x 9 grid that is divided into 3 x 3 blocks with the numbers 1 through 9 placed exactly once in each *row*, *column*, and *block*. To create the puzzle, numbers are removed strategically from a selection of *cells* on the grid, and the player must solve the puzzle by deducing the correct numbers that must be placed to restore the grid using the numbers left on the grid, known as *clues*. Figure 1 gives a graphical description of the basic terminology used in this paper for Sudoku. Cells are called *neighbors* if they share a column, row, or block. *Candidate sets* are the lists of numbers that may be placed in a non-clue cell without violating the rules of the puzzle. There are many *solving strategies* that have been developed by players, ranging from simple logical steps to complicated deductions.

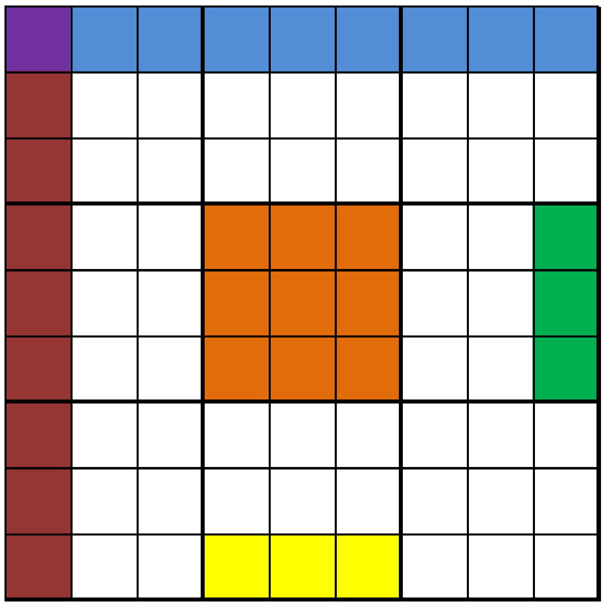

Figure 1 – A cell is a single square within the puzzle (purple). A block is one of the 3 x 3 squares within the puzzle (orange). Minor columns are formed from 3 vertically adjacent cells in a block (green). Minor rows are made up of 3 horizontally adjacent cells in a block (yellow). A row is formed from 3 horizontally adjacent minor rows (blue), and a column is made up of 3 vertically adjacent minor columns (red).

It is conventional for a puzzle to have a unique solution, and for aesthetic purposes, the clues are often arranged in a symmetric pattern. Sudoku can be generally extended from the 9 x 9 case, known as standard Sudoku, to any $n^2 \times n^2$ grid divided into $n \times n$ blocks filled with numbers 1 through n^2, and many other variations have been created by adding one or more conditions to the grid. The particular variation used for the algorithm discussed in this paper is known as

Samurai Sudoku. Samurai Sudoku is composed of five standard Sudoku grids arranged in a quincunx, such that the corners of the grids overlap. In most Samurai Sudoku puzzles, each of the standard grids cannot be solved independently. An example of a Samurai Sudoku puzzle is given in Figure 2.

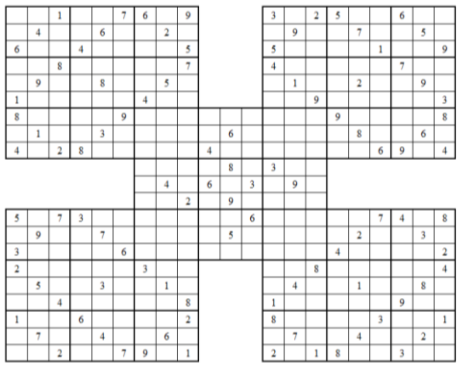

Figure 2 – From *Taking Sudoku Seriously* (Rosenhouse & Taalman, 2011).

1.2 – Algorithms for Sudoku

Sudoku puzzles have piqued the interest of both mathematicians and computer scientists alike. Solving Sudoku puzzles is an NP-complete problem, shown by Yato in 2003, immediately placing it into an intriguing class of problems (Lewis, 2007). This comes from one of the great questions of mathematics and computer science: whether P = NP. Solution times in computer science are given based on the number of variables that must be set in an algorithm, denoted as N. In other words, P is the class of problems with solution times that are proportional to some polynomial and NP is the class of problems with solutions that can be verified in polynomial time. Polynomial time means that the number of computations in the algorithm is bounded by a polynomial function of N. It is unknown whether NP and P are equivalent. A majority of NP problems are NP-complete, like Sudoku, which means that if a polynomial-time solution can be developed for one instance, it can be used to solve all other instances of that problem. This seems to go against intuition since it is theorized that many of the problems belonging to NP are solved in exponential, rather than polynomial, time. If it can be shown that P = NP, then the class of very difficult problems actually has simple solutions that can be found (Hardesty, 2009). As pointed out by Lewis, since solving Sudoku is an NP-complete problem, there is no polynomial time algorithm that can be applied to every possible Sudoku puzzle unless P = NP (Lewis, 2007).

To get some idea of how humongous this search space is, there are 6,670,903,752,021,072,936,960 (approximately 6.7×10^{21}) possible, valid Sudoku grids (Delahaye, 2006). Clearly the search space of solutions to a particular puzzle is somewhat reduced from this, since the clues that are given will rule out many grids, but on the other hand, the search space is also variable depending on those clues. There are many websites devoted to Sudoku that rely on algorithms to generate and solve puzzles so players can quickly load a puzzle to play. For example, the puzzles used to test the algorithm discussed in this paper were found on dkmGames.com and SamuraiSudoku.org, and each of these websites provide the player with puzzles to play online and can generate the solution. In order to provide an enjoyable playing experience and bring players back again, efficient and accurate algorithms are desired. There are many possible implementations that have been tried, but some have proven more effective than others.

Most solving algorithms, the focus of this paper, fall into two camps: using human solving strategies to deduce the correct solution or using search methods with backtracking to find the correct solution. Each presents its own benefits and drawbacks. Algorithms that imitate human solving patterns can be useful when also implementing a Sudoku puzzle generator, as it ensures that the solution can be obtained through logical steps rather than guessing. One such method was designed by Boothby, Svec, and Zhang. They defined operations to apply each solving strategy and then found the inverse of each operation (Boothby, Svec, & Zhang, 2008). This enabled them to attempt to generate Sudoku puzzles by using the inverse operations, with a goal in mind of being able to create a puzzle using a certain set of solution strategies. They hoped this would guarantee puzzles to be a certain difficulty by allowing only solving strategies classified for that difficulty or lower (Boothby, Svec, & Zhang). Another example is the `hsolve` algorithm created by Chang, Fan, and Sun. This approach attempts to simulate the behavior of a human solving the puzzle while simultaneously calculating a difficulty based on the solving strategies used (Chang, Fan, & Sun, 2008). It starts with the simplest level of solving strategies, applying each one to the grid to determine how many could possibly be applied to the grid at the current state. It calculates how many strategies must be tried before finding one that advances the solution. One of the possible valid solution strategies is randomly selected and applied, and it continues through each level of solving strategies until a solution is found. Again, the focus remains on using the solving algorithm in combination with a generation algorithm (Chang, Fan, & Sun).

Of course, sometimes the focus is not on generating Sudoku puzzles. There is also interest in finding efficient algorithms that can navigate the enormous search space of Sudoku puzzles. The simplest among the searching methods use brute-force techniques. These may only place numbers randomly in the puzzle, perhaps shuffling the numbers 1 through *n*^{2} and then checking for rule violations in the rows, columns, and blocks. More commonly, these programs employ backtracking algorithms. Numbers are randomly placed, but checks are performed after each placement to find a valid number before moving to a new cell. If it reaches a cell that has no valid placements available, it steps backward and erases the previous placement, testing out a new number. It may take several steps backward before it finds a valid solution (Delahaye, 2006).

An integer programming model, a binary integer program (BILP) more specifically, for solving Sudoku was applied by Bartlett, Chartier, Langville, and Rankin. This model uses decision variables that record whether each number is present in the cell (Bartlett, Chartier, Langville, & Rankin, 2008). These variables are defined as follows:

$$x_{ij\_k} = \begin{cases} 1, if\ the\ integer\ k\ is\ placed\ in\ cell\ (i,j)\ of\ the\ grid \\ 0, otherwise \end{cases}$$

Each of the rules for Sudoku puzzles is formulated as a constraint for the program. Their method poses solving the Sudoku puzzle as a constraint programming problem and uses Matlab's `bintprog` function, which finds a solution through a series of LP-relaxation problems. They showed that this can also be easily extended to variations of Sudoku, since each additional rule is simply added to the set of constraints (Bartlett, Chartier, Langville, & Rankin).

Other methods approach solving Sudoku strictly as an optimization problem. Lewis followed this path, using a representation of the grid, a neighborhood operator, and a function for evaluating the grid, and he applied a simulated annealing metaheuristic (Lewis, 2007). A Sudoku grid is considered optimal if it is complete and satisfies all the rules. This method fills the grid with random values, and the evaluation function calculates a cost (or how far from optimal the grid is) based on the number of contradictions found in the grid. While contradictions to the rules exist, the neighborhood operator is called on to choose and then swap two non-clue cells within the same block to test if it eliminates any contradictions. The application of the simulated annealing means it searches for a neighbor with a lower cost so a solution is found quickly. This was possibly the first application of a metaheuristic to a Sudoku solving algorithm, and the author noted that it was successful at solving any Sudoku puzzle (Lewis).

Genetic, evolutionary, and many other types of search algorithms have all been explored as well. One such evolutionary algorithm, Harmony Search Algorithm, was developed by Geem (Geem, 2007). This algorithm was applied to several optimization problems, including solving Sudoku. It is used to mimic the behavior of musicians, based on factors such as memory consideration and adjustments to pitch. Similar to the simulated annealing application, a cost is calculated based on the number of contradictions in the puzzle and compared to neighbor solutions in an effort to reach the solution with the lowest cost. While it was not successful in solving every puzzle, it proves to be an interesting application (Geem).

1.3 – Nested Monte Carlo Search

While using a metaheuristic to guide the search can be helpful in reducing program run time, the exploration of the search space can also be guided by making random choices while playing step-by-step through the game. A nested Monte Carlo tree search works this way, creating a tree as a random choice of the available options is made at each step towards finding a solution. At a given step, or level, the search determines the correct choice to be played by searching the lower steps. This method of guiding the search through successive nested levels of game play is known as rollouts (Rosin, 2011). In the case of Sudoku, this means placing a number in a cell and removing that number from all neighbor cells. The grid is checked for contradictions caused by the random play of the game to determine if the rollout was successful at this level. Although this could be a lengthy search, the search space is reduced since the game is played throughout the rollout, which in a sense is optimizing the game at each level of search. Many other types of algorithms do not perform this optimization during the search, but only at the first level of search (Cazenave, 2009).

There is some variation in how a Monte Carlo search algorithm may work. The basic type of nested Monte Carlo search is known as Iterative Sampling, which plays the game with simple random choices. Rollouts may be used to improve the Monte Carlo search, and Cazenave notes that this was successful for

Tesauro and Galperin (Cazenave, 2009). A heuristic may be applied to the algorithm as well. In some applications for other games, the heuristic was adaptable, and the rollouts actually improve the heuristic as the search continues through lower levels. A similar type of Monte Carlo search is a Reflexive Monte Carlo search. Instead of playing an undetermined number of levels until a solution or contradiction is reached, a static number of levels are played and evaluated (Cazenave). The algorithm discussed in this paper is based off an Iterative Sampling search, but uses nested rollouts and applies a basic heuristic to improve the search performance.

2 – Algorithm Description

Previously, a similar algorithm was developed by Cazenave and applied to Morpion Solitaire, SameGame, and 16 x 16 Sudoku (Cazenave, 2009). The algorithm developed for this paper is applied to Samurai Sudoku to try a somewhat different and more difficult application. Additionally, the nested Monte Carlo search algorithm for Samurai Sudoku uses nested, recursive calls and applies a simple heuristic at each level of the search to guide the rollouts. The set-up function `CreateGrid(n`^{2}`, total grid size, clue set)` is called to create the grid. For simplicity, the representation of the grid is laid out as a square, with cells that are not part of the grid set to 0. The function places the clues in the grid as it creates it, and it calls on the evaluation function, `ClueEliminationCheck(n`^{2}`, total grid size)`, which performs a base level optimization by removing clue values from the candidate sets of neighboring cells. This means that there are fewer branches that need to be checked since values are removed before the search is started. The search is then started with `PerformSearch(n`^{2}`, total grid size, stepCount)`, tracking the current level of the rollout with `stepCount`.

```
CreateGrid(n^2, total grid size, clue set)
    for each cell, create candidate sets
        if clue set has value > 0 for current cell
            set candidate set to size 1 with given
    value
        else
            set candidate set to size n^2 with values
    1 through n^2

    ClueEliminationCheck(n^2, total grid size)
    stepCount = 0
    PerformSearch(n^2, total grid size, stepCount)
    return grid
```

`ClueEliminationCheck(n`^{2}`, total grid size)` executes the 'game play' that optimizes the grid at each level. For any cell with only one value (either a placed number or a clue), that value is removed from the candidate sets of all neighbors. Following each elimination check, the grid is evaluated for contradictions; once a contradiction is found, the program sets a flag and immediately breaks out of the current play. It does not calculate the number of contradictions since this is not set up as an optimization problem, but simply checks if one exists to see if the current search path is unsuccessful. Because there is interdependence between the five standard Sudoku grids, the algorithm must go through each grid individually to evaluate.

```
ClueEliminationCheck($n^2$, total grid size)
    for each grid
        for each cell, if candidate set is size 1
            remove candidate set value from each
cell that is a row neighbor
            check for contradiction
            if contradiction
                break
            remove candidate set value from each
            cell that is a column  neighbor
            check for contradiction
            if contradiction
                break
            remove candidate set value from each
            cell that is a block  neighbor
            check for contradiction
            if contradiction
                break
    return contradiction
```

`PerformSearch(`n^2`, total grid size, stepCount)` is the recursive search function. The heuristic is first applied, following the example of Cazenave: the grid is checked for the smallest candidate set size that is greater than one, and a cell is randomly chosen with a minimal candidate set (Cazenave, 2009). Before a random number from the candidate set is chosen and set in that cell, the current state of the puzzle is stored. This allows the algorithm to travel back along the current path through the tree when the rollout path terminates in contradiction rather than solution. After the random play is made, `ClueEliminationCheck(n`^{2}`, total grid size)` is called. If a contradiction is found, the grid is restored to the state at the previous level of search and the contradictory value is removed from the candidate set. In terms of the tree search, this means that it terminates the lower level search on the current branch and moves to another branch that has not been explored yet. Since we are choosing minimal candidate sets, it is possible that the candidate set would be reduced to size one after removing a value, so the evaluation function is called again in such a state to check if the remaining value causes a contradiction. If it does, then the program needs to follow the path further back and remove the previously set value. This repeats until a layer of search is found that does not lead to a contradiction on the current leaf. Once that leaf is found, it proceeds to start a new rollout from there by making a recursive call to `PerformSearch(n`^{2}`, total grid size, stepCount)`.

```
PerformSearch(n^2, total grid size, stepCount)
     minimum = 10;
     for each cell
          if candidate set size < minimum
               minimum = candidate set size
     if minimum < 10
          for random cell
               if candidate set size = minimum
                    store current grid and location of
random play
                    choose random play
                    restoreGrid =
               ClueEliminationCheck(n^2, total grid
               size)
```

```
            if restoreGrid = true
                set grid to state before
                random play and remove the
                chosen value from candidate
                set
                if candidate set size = 1
                    restoreGrid =
                    ClueEliminationCheck($n^2$,
                    total grid size)
            while restoreGrid = true and
        candidate set size < 2
                restoreGrid = false
                set grid to previous state at
                previous level and remove the
                chosen value from candidate
                set
                if candidate set size = 1
                    restoreGrid =
                    ClueEliminationCheck($n^2$,
                    total grid size)
            if restoreGrid = false
                stepCount increases
                PerformSearch($n^2$, total grid
size, stepCount)
    return
```

Although cells with minimal candidate sets are chosen, this type of nested search does not work like an optimization problem, where it is continuously improving the result found. It is possible that a less optimal grid will be chosen in the next layer of search. Using the Samurai Sudoku variation increases the complexity of the algorithm since the search may follow a path that results in an optimal grid for one of the five standard Sudoku grids, but as that path travels through one of the other four, it may not lead to a solution due to the interdependence between grids. Of course the increased size of the grid that must be solved also complicates the solver, especially since it uses recursive calls, which can be taxing on performance. However, since this interdependence plays a role and since one grid can usually not be solved independently from the others, a recursive function makes sense so that the nested layers can go as deep as needed without fixing a bound.

3 – Analysis of Application to Samurai Sudoku

The program performs well most of the time when solving Samurai Sudoku puzzles. The algorithm always produced a correct solution when it was able to solve the puzzle. However, there were ten puzzles that it could not solve because the program crashed before a solution was found due to making too many recursive calls for the program to track. Although most puzzles are constructed to be logic solvable, a benefit of this method is that it does not rely on this assumption. It does apply the most basic solving strategies to the grid to reduce candidate set sizes, but these strategies are really just checking the constraints applied to the grid by Sudoku rules. This allows for a reduction of the search space by applying constraint satisfaction within the algorithm. Table 1 gives the times to solve eighty grids of varying difficulty, where the difficulties that were used are ‘easy’, ‘standard’, ‘hard’, and ‘tough’, and Figure 3 displays the run times graphically. The computer used to run these puzzles is an HP laptop with 8 GB of memory and dual 1.9 GHz processors.

	# of Puzzles	Average Time (seconds)	Range of Time (seconds)
Easy	20	0.127	0.001 – 2.423

Standard	20	43.638	0.062 – 247.042
Hard	20	438.443	0.904 – 1229.890
Tough	20	457.752	0.093 – 2440.220
All Puzzles	80	209.359	0.001 – 2440.220

Table 1: Run times for Samurai Sudoku

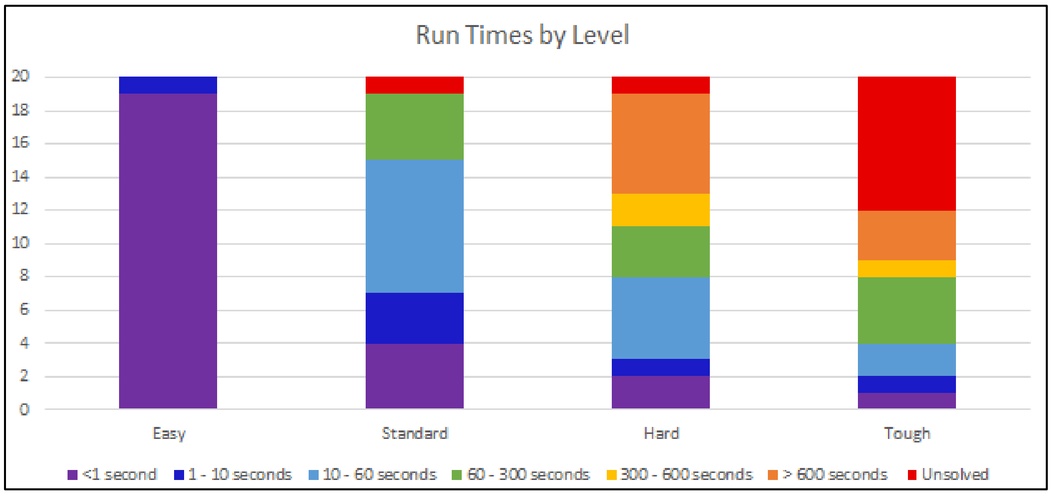

Figure 3: Distribution of run times

Run time for puzzles that were not solved are not factored into average solve time. While the run time remains low for easy and standard puzzles, a large jump in average time required to solve occurs when the difficulty increases past standard. Since the difficulty of a Sudoku problem is bounded in a sense (because it enforces a condition of logic solvability for a human player), the large jump run time is not extremely problematic, although not ideal.

Interestingly, puzzles of any given difficulty were solved with a relatively large range of run times. The difference in run time is likely due to the random nature of the nested rollouts; the algorithm sometimes made ‘lucky’ random choices to guide the rollouts or find a cell with a minimal candidate set. To examine the effect of the randomness, one tough puzzle was chosen to be run 25 times. The run time range for this puzzle was 57.252 to 92.039 seconds, with an average of 85.512 seconds and a median run time of 85.956 seconds. The variance is 37.073, indicating that there is a fair amount of dispersion in the data set. However, this is expected since the game play at each level of search is randomized, and choosing a cell to make a play from is also randomized. Since a cell can potentially be picked multiple times, including solved cells, it takes longer on average to find a cell that meets the minimal candidate set requirement as it approaches finding the solution of the puzzle.

3.1 – Comparison to other solvers

Comparing to a handful of other Sudoku solvers, the run time is acceptable considering that the puzzle being solved is much larger. As mentioned before, Cazenave implemented a similar algorithm for 16 x 16 Sudoku. For 100 Sudoku puzzles with 66% non-clue cells (compared to an average of 72% non-clue cells for Samurai Sudoku), his algorithm had an average run time of 61.83 seconds, which is only about one-third of the Samurai Sudoku algorithm run time. Of course, Samurai Sudoku puzzles are about 44% larger than those puzzles and have multiple Sudoku grids with interdependence, so it is not a perfect comparison. It is hard to say how much of the extra run time for this algorithm is accounted for by the larger size and interdependence, but Cazenave's nested Monte Carlo search also implements memorization of best sequences, or the sequences that lead to lower costs on average, which is likely a factor in his improved run time (Cazenave, 2009). The simulated annealing algorithm developed by Lewis had a more constant run time across difficulty levels. He noted that for 9 x 9 standard Sudoku, the algorithm typically solved the puzzle in about half a second, but for 16 x 16 standard Sudoku, the range of run time was 5 to 15 seconds. However, this program sometimes ended up ‘stuck’ and required one or two restarts to achieve a solution (Lewis, 2007). Similarly, the Harmony Search Algorithm took 3 to 38 seconds to solve most puzzles. Geem notes that his algorithm could only solve the puzzle 33 out of 36 runs, which is similar to the failure rate that the nested Monte Carlo algorithm for Samurai Sudoku faces. However, the failures of the Harmony Search Algorithm were due to the program becoming entrapped in a local optimum and timing out the search, whereas the algorithm examined in this paper becomes ‘lost’ in the search; this will be discussed further in Section 3.2. The median for the successfully solved puzzles by Harmony Search Algorithm was 8 seconds, while the median for Samurai Sudoku was 13.679 seconds (Geem, 2007). The median is quite good considering that Harmony Search Algorithm was only tested on 9 x 9 Sudoku, and Samurai Sudoku is around 4.5 times larger. However, the average and range of run time is considerably longer, even with size taken into account. The results for the Binary Integer program are similar in run time to the Harmony Search Algorithm, solving a puzzle in 16.08 seconds. Unfortunately, only one puzzle was tested with this method, so it is hard to say whether it is truly better or not (Bartlett, Chartier, Langville, & Rankin, 2008).

Cazenave noted in his analysis that the most difficult problems to solve in the 16 x 16 standard Sudoku case were those with 66% non-clue cells. Puzzles outside this range were often over- or under-constrained, both leading to easily successful searches. However, as seen in Figure 4, there does not seem to be much correlation between the percentage of non-clue cells and run time for the Samurai Sudoku algorithm.

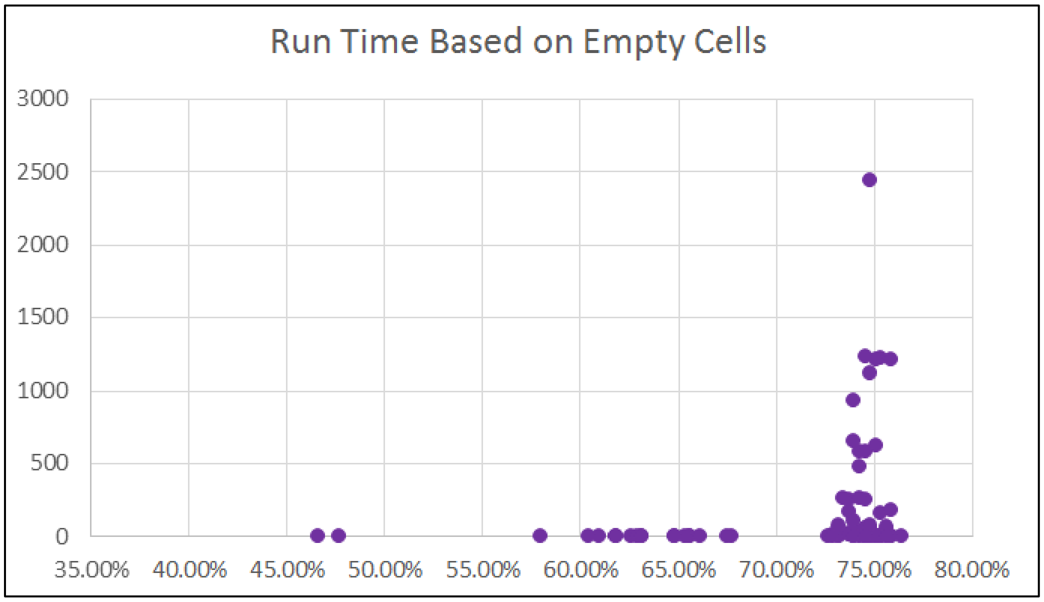

Figure 4: Run time predicted by percentage of non-clue cells

Most of the puzzles have around 75% non-clue cells with a large range of run times centered there, suggesting that this percentage of non-clue cells does not guarantee a hard or easy puzzle for the algorithm. Puzzles with less than 50% non-clue cells always ran quickly, suggesting that similar to Cazenave's findings, puzzles with too many clues were over-constrained and easy to solve (Cazenave, 2009).

It is obvious that the difficulty rating of the puzzle or the extreme ends of the range of given clues affects run time, but what drives this difficulty level if not the number of clues? Given that Samurai Sudoku generally will not be solvable without working on the overlapping areas, perhaps the number of clues in the overlapping areas of the quincunx layout is driving the run time increase. Figure 5 seems to suggest this is so; the run times are plotted against the number of clues found in the overlapping areas.

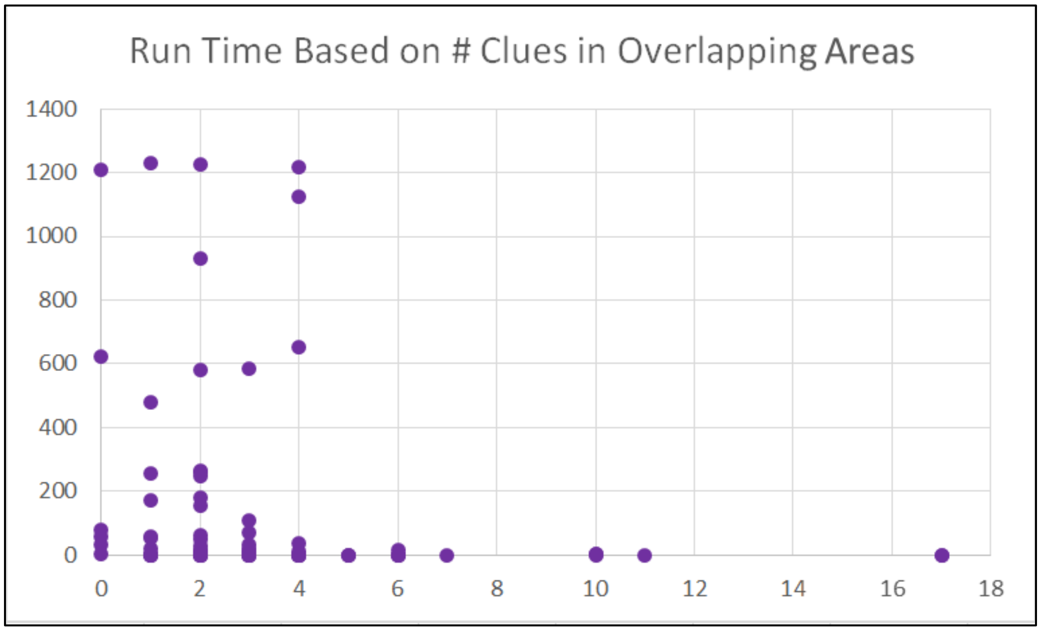

Figure 5: Run time predicted by the number of clues in the overlapping areas

All the run times are low for high values of clues, and a peak is found for puzzles in which 4 or fewer clues were given in overlapping areas. The unsolved puzzles all had 1 to 4 clues in the overlapping areas, suggesting that it is not as under-constrained as having 0 clues, but does not have enough constraints to guide the search as when there are 5 or more clues. It is possible that a better heuristic may improve the search significantly here if it can keep the search from becoming 'lost' among the large number of possible rollouts.

3.2 – Drawbacks of the algorithm

There are pitfalls to be wary of with the nested Monte Carlo search as applied to Samurai Sudoku. Since it relies on a recursive function call in the search, it becomes strenuous when many nested rollouts are necessary. This leads to an overflow of the call stack as the search depth and number of branches increase too much, causing the program to crash before the algorithm can solve the puzzle. Theoretically, if the size of the call stack was not a limitation, the program could solve any puzzle it was fed. Since it does not rely on optimization to find a solution, it has no risk of becoming stuck in a local optimum, which is what caused the Harmony Search Algorithm to sometimes fail; theoretically, this means that a ‘lucky’ run could reach the solution if there are correct random choices more often throughout the game play (Geem, 2007).

The search comes to a stop once a solution has been found, so the algorithm is also incapable of conclusively determining if multiple solutions exist to the puzzle. Logic-based solvers have the advantage here, since multiple solutions typically exist when sets of numbers can be swapped to create another solution. Solving such a puzzle often ends with a set of cells that contain the same candidate sets without any further solving strategies that can deterministically place numbers in those cells; this is known as an *unavoidable set* (Vanpoucke, 2012). One cell can have a random number placed from its candidate set that then determines the numbers that must be placed in the remaining cells. Since the nested Monte Carlo search makes random choices at each level of game play, it only needs to randomly choose one of the numbers that could lead to a correct solution, and it will not step back to evaluate un-checked numbers. Of course, adding this capability would also increase run time, and most algorithms of this type have the same drawback. Depending on the intended use of the solver, this could be considered a downside. A user intending to use it in combination with a puzzle generator would not have a way to guarantee that the generator produces uniquely solvable puzzles—only that that it produces a solvable one. Since the game plays made during the search are random, the algorithm may be run many times to attempt to find multiple solutions, but it is not guaranteed to find other solutions if they exist or to find them in a timely manner.

4 – Future Work

Remarkably, it seems that despite the Sudoku craze, very little work has been done on Samurai Sudoku or even other variations. Many types of algorithms could be attempted on Samurai Sudoku, such as genetic or evolutionary algorithms, like Geem's Harmony Search Algorithm, or any of the other ones mentioned in this paper (Geem, 2007). It certainly provides a way to stress the algorithm and reveal its limitations. Additionally, more research into equivalence classes for Samurai Sudoku could reveal helpful information for future algorithms, especially if focus is placed on examining how the overlapping areas affect the search tree or solving process of the grid.

There is plenty of work that could be done with the nested Monte Carlo search algorithm as well. As mentioned before, by applying a better heuristic, or possibly multiple heuristics, the search can be guided closer to the rollouts that will more quickly lead to a solution. If the program proceeds along the correct rollouts more often, then less backtracking and fewer recursive calls are required. This may reduce or even eliminate the stack overflow issue. Given the difference in run time after accounting for the different type of puzzle being examined, using memorization like Cazenave's algorithm shows potential (Cazenave, 2009).

However, this must be approached carefully. Cazenave's algorithm only needs to consider one grid, while this algorithm needs to look at five grids with interdependence. A better sequence on one grid may not translate well to another grid within the puzzle. It may be worth it to examine whether certain sequences are better on certain classes of Sudoku puzzles.

Sudoku grids can be considered "essentially the same," or part of the same equivalence class, if a mapping can be made from grid 1 to grid 2 using actions that do not cause contradictions to the rules of Sudoku when applied to any given grid. For example, rotating any Sudoku puzzle by multiples of 90° results in a puzzle within the same equivalence class. There are 5,472,730,538 equivalence classes of Sudoku grids, making it a daunting task to examine whether the performance of a sequence corresponds to certain equivalence classes (Chapman & Rupert, 2012). The various grids that are part of a Samurai Sudoku grid may come from distinct equivalence classes, so if sequences are better depending on which equivalence class they are applied to, memorization would need to be applied for each grid individually. Additionally, given that the overlapping areas of the grid play such a pivotal role in the solution path for these puzzles, guiding the search to start the nested rollouts from this area could potentially provide significant improvement over the more random method currently used.

The performance can likely be improved by applying additional solving strategies to the algorithm. It currently only removes placed numbers from the candidate sets of neighbor cells, whereas applying actual solving strategies could potentially place more numbers and reduce the size of the candidate sets further

(Lewis, 2007). If more cells are solved or candidate set sizes are reduced, there are fewer random choices (and thus fewer levels of nested search). Although some extra computing time will be necessary to perform the solving strategies, it is likely that the use of extra solving strategies will improve the run time overall since each constraint applied reduces the overall search space (Bartlett, Chartier, Langville, & Rankin, 2008). For example, the "Covering Set" solving strategy as outlined in Boothby, Svec, and Zhang's work, could be applied easily in a computer program. This solving strategy checks for *k* neighbor cells whose candidate sets contain the same *k* numbers; since these cells are neighbors, and there are *k* such cells with the same *k* possibilities, all other shared neighbor cells can remove those *k* numbers from their candidate sets (Boothby, Svec, & Zhang, 2008).

If there was a desire to enhance the program to check for multiple solutions as well as finding a solution, there could be some promise in examining unavoidable sets. This concept was discussed in section 3.2, but this definition from Vanpoucke (2012) states it more formally:

Definition (Vanpoucke) *– Consider an n^2 x n^2 Sudoku grid S. A subset U of S is called an unavoidable set if S\U has more than one completion to an n^2 x n^2 Sudoku grid.*

An unavoidable set of *m* cells has degree *k* if the puzzle must have at least *k* values from the unavoidable set given as clues in order to be uniquely solvable. Furthermore, Vanpoucke's work (2012) gives us the following theorem:

Theorem (Vanpoucke) – *Consider an n^2 x n^2 Sudoku grid S and suppose that $U \subseteq S$ is an (m,k)-unavoidable set. Then we will need to add at least k elements from U to S\U to obtain an n^2 x n^2 Sudoku puzzle with a unique completion. Moreover, if $V \subseteq S$ is an (m',k')-unavoidable set, such that $U \cap V = \emptyset$, then $U \cup V$ is an (m+m',k+k')-unavoidable set.*

Vanpoucke points out that "if a set of clues does not intersect every unavoidable set, then…there is more than one completion" (Vanpoucke, 2012, p. 21). Thus, if the potential unavoidable sets can be identified quickly and each of these sets can be checked for *k* clues, the algorithm could identify whether the puzzle could be solved uniquely. However, Vanpoucke (2012) notes that a program written to find unavoidable sets was not fast enough when attempted before, so improvements to this method would first be necessary (p. 22). With future research though, this could become a feasible avenue to explore.

In a more general sense, the program can likely be improved by making changes to enhance efficiency. There may be programming languages that would be better suited for this algorithm than C++. It could potentially improve run time (although would not eliminate failures) to use a faster computer to run the solver. The program currently uses vectors to store previous states of the grid and locations that were used in play, which gives the program great flexibility for managing the candidate sets, but requires the program to constantly resize these variables. There could be a more efficient method of storage, perhaps using a structure or class instead, that has not been attempted here. There are likely no changes to the methods used, such as how it is checking the grid for contradictions or choosing a minimal candidate set, that will significantly improve the run time, but it may be worth some experimentation at a later date.

In earlier versions, the program started searching for a cell with a minimal candidate set in the first cell of the representation of the grid, proceeding systematically through the rows and columns. However, this version also failed to solve a large number of more complicated puzzles; this is likely because larger numbers of clues were concentrated in other areas of the puzzle, meaning that it did not choose cells that would quickly determine values in neighbor cells and thus required more steps to compute. When the program was revised to choose cells from a randomly determined column and row, it was able to solve more puzzles, but run time was driven up due to this randomness. The random choice of cell is not guided at all, so it can choose cells multiple times or choose cells that do not have a minimal candidate set. Particularly towards the end stages of a search, where many cells have only one value, the computation time for choosing an eligible cell increases significantly. Finding some balance between these two methods would likely produce a program that is faster and able to solve more puzzles.

5 – Conclusion

This paper presents, to the author's knowledge, the first application of a nested Monte Carlo search algorithm to the Sudoku variation called Samurai Sudoku. Although it was not entirely successful, it certainly solves the puzzles much faster than a human is able to without the need for multiple iterations of the program. These solutions are always correct when the program is able to handle the number of recursive calls necessary to find it. Since there is so little work done on Samurai Sudoku, it is difficult to say how successful this algorithm was compared to previous work. Further examination of the overlapping areas and how they affect the puzzle undoubtedly will lead to an improved search method.

Section 1.2 shows that there are many possible approaches to solving Sudoku grids that each provide certain benefits and pitfalls, depending on the desired use for the solver. Most notably, it is clear that algorithms can be developed that can solve any Sudoku grid and any variation on Sudoku, although there may be some limitations in technology for the more complicated cases. There is generally a distinction made between algorithms using some type of random search and algorithms using solving strategies to mimic human solving techniques, but experimentation in combining these two approaches may be the key to overcoming the limitations exhibited by this application. Similarly, looking at advances in the mathematics behind Sudoku could lead to improvements in the algorithms for solving and generating puzzles. In particular, examining the equivalence classes and unavoidable sets shows promise in revealing more about Sudoku that will guide future work. Continued exploration of the areas of interest of both mathematicians and computer scientists alike will likely reveal much more work to be done with Sudoku and its many variations.

References

Bartlett, A., Chartier, T., Langville, A., & Rankin, T. (2008). *An Integer Programming Model for the Sudoku Problem.* Retrieved from http://langvillea.people.cofc.edu/sudoku5.pdf

Boothby, T., Svec, L., & Zhang, T. (2008). *Generating Sudoku Puzzles as an Inverse Problem.* Retrieved from http://www.math.washington.edu/~morrow/mcm/team2306.pdf.

Cazenave, T. (2009). Nested Monte-Carlo Search. *International Joint Conferences on Artificial Intelligence Organization Proceedings* (pp. 456-461). Retrieved from http://www.aaai.org/ocs/index.php/IJCAI/IJCAI-09/paper/view/343.

Chang, C., Fan, Z., & Sun, Y. (2008). *hsolve: A difficulty metric and puzzle generator for Sudoku.* Retrieved from http://web.mit.edu/yisun/www/papers/sudoku.pdf.

Chapman, H., & Rupert, M. E. (2012). A Group-theoretic Approach to Human Solving Strategies in Sudoku. *Colonial Academic Alliance Undergraduate Research Journal, 3*, Article 3.

Delahaye, J.-P. (2006). The Science behind Sudoku. *Scientific American*, 81-87.

Free Online Samurai Sudoku. (2008). Retrieved from Samurai Sudoku: http://www.samuraisudoku.org/online.php

Geem, Z. W. (2007). Harmony Search Algorithm for Solving Sudoku. In B. Apolloni, R. Howlett, & L. Jain (Eds.), *Knowledge-Based Intelligent Information and Engineering Systems* (Vol. 4692, pp. 371-378). Springer. doi:10.1007/978-3-540-74819-9_46

Hardesty, L. (2009, 10 29). *Explained: P vs. NP*. Retrieved from MIT News: On Campus and Around the World: http://news.mit.edu/2009/explainer-pnp

Lewis, R. (2007). Metaheuristics can Solve Sudoku Puzzles. *Journal of Heuristics, 13*(4), 387-401.

Rosenhouse, J., & Taalman, L. (2011). *Taking Sudoku Seriously.* New York: Oxford University Press.

Rosin, C. D. (2011, July 16-22). *Nested Rollout Policy Adaptation for Monte Carlo Tree Search.* Retrieved from International Joint Conference on Artificial Intelligence: http://ijcai-11.iiia.csic.es/program/paper/1356

Sudoku. (2015). Retrieved from dkmGames: http://dkmgames.com/SudokuGames.htm

Vanpoucke, J. (2012). Mutually Orthogonal latin squares and their generalizations (Master Thesis). Retrieved from http://homepages.vub.ac.be/~jvpouke/MasterThesisMOLS.pdf

Appendix

The following is the content of main.cpp:

```
#include "SudokuSolver.h"

using namespace std;

int main()
{
     //Set the size of the grid and if playing
standard or samurai version
     int numBands = 9;
     int numTotalBands = (numBands * 2) +
sqrt(double(numBands));

     //Need 3 pieces of information for clues--column,
row, and value, so create 2D vector
     std::vector<std::vector<int> > clueSet;
     for (int iColumn = 0; iColumn < numTotalBands;
iColumn++)
     {
          std::vector<int> newRow(numTotalBands, 0);
          clueSet.push_back(newRow);
     }

//dkmgames.com #61743
     clueSet[0][2] = 3;
     clueSet[0][3] = 4;
     clueSet[0][8] = 7;
     clueSet[0][17] = 3;

     clueSet[1][5] = 8;
     clueSet[1][6] = 9;
     clueSet[1][14] = 9;
     clueSet[1][17] = 8;

     clueSet[2][1] = 5;
     clueSet[2][3] = 3;
     clueSet[2][4] = 7;
     clueSet[2][16] = 5;
     clueSet[2][18] = 6;

     clueSet[3][0] = 1;
     clueSet[3][4] = 5;
     clueSet[3][7] = 8;
```

```
clueSet[3][13] = 4;
clueSet[3][17] = 5;
clueSet[3][20] = 7;

clueSet[4][4] = 3;
clueSet[4][14] = 3;
clueSet[4][16] = 4;
clueSet[4][19] = 2;

clueSet[5][0] = 9;
clueSet[5][1] = 2;
clueSet[5][3] = 6;
clueSet[5][15] = 2;
clueSet[5][20] = 9;

clueSet[6][15] = 7;
clueSet[6][18] = 9;
clueSet[6][19] = 3;

clueSet[7][3] = 2;
clueSet[7][5] = 9;

clueSet[8][2] = 6;
clueSet[8][3] = 7;
clueSet[8][4] = 1;
clueSet[8][12] = 3;
clueSet[8][15] = 6;
clueSet[8][18] = 5;

clueSet[9][8] = 1;

clueSet[10][11] = 9;

clueSet[11][9] = 4;
clueSet[11][11] = 2;
clueSet[11][13] = 5;

clueSet[12][1] = 5;
clueSet[12][9] = 3;
clueSet[12][17] = 1;

clueSet[13][4] = 6;
clueSet[13][9] = 8;
clueSet[13][13] = 6;
clueSet[13][14] = 2;
clueSet[13][15] = 4;
clueSet[13][19] = 5;
```

```
	clueSet[14][1] = 3;
	clueSet[14][3] = 5;
	clueSet[14][10] = 6;
	clueSet[14][14] = 5;
	clueSet[14][17] = 9;
	clueSet[14][20] = 8;

	clueSet[15][0] = 7;
	clueSet[15][2] = 2;
	clueSet[15][5] = 1;
	clueSet[15][12] = 5;

	clueSet[16][6] = 8;
	clueSet[16][7] = 1;
	clueSet[16][13] = 2;
	clueSet[16][19] = 4;

	clueSet[17][0] = 9;
	clueSet[17][5] = 7;
	clueSet[17][6] = 4;
	clueSet[17][14] = 4;
	clueSet[17][18] = 3;
	clueSet[17][19] = 8;
	clueSet[17][20] = 6;

	clueSet[18][1] = 7;
	clueSet[18][3] = 6;
	clueSet[18][4] = 3;
	clueSet[18][8] = 4;
	clueSet[18][15] = 7;
	clueSet[18][17] = 3;

	clueSet[19][0] = 4;
	clueSet[19][1] = 9;
	clueSet[19][13] = 1;
	clueSet[19][16] = 9;
	clueSet[19][18] = 6;

	clueSet[20][2] = 3;
	clueSet[20][8] = 2;
	clueSet[20][16] = 8;
	clueSet[20][18] = 5;
	clueSet[20][19] = 3;

	//Now create our grid object to be modified in
our function
```

```
     //Use a 3D array since cells can hold multiple
possible values as we search
     std::vector<std::vector<std::vector<int> > >
m_sudokuGrid;

     //Now that the grid is ready, start the timer &
then start the algorithm
     std::clock_t startTime = std::clock();
     m_sudokuGrid = CreateGrid(numBands,
numTotalBands, clueSet);

     std::clock_t endTime = std::clock();

     //Puzzle is now solved, so calculate the run time
     double runTime = (endTime - startTime) / (double)
CLOCKS_PER_SEC;

     //Print the solution and run time
     for (int iColumn = 0; iColumn < numTotalBands;
iColumn++)
     {
          ofstream outputFile;
          outputFile.open("solvedpuzzle.txt");

          outputFile << runTime << " seconds" << endl
<< endl;

          for (int iRow = 0; iRow < numTotalBands;
iRow++)
          {
               for (int iColumn = 0; iColumn <
numTotalBands; iColumn++)
               {
                    if (m_sudokuGrid[iRow][iColumn][0]
== 0)
                    {
                         outputFile << "    ";
                    }
                    else
                    {
                         outputFile <<
m_sudokuGrid[iRow][iColumn][0] << "  ";
                    }
               }
               outputFile << endl;
          }
          outputFile.close();
     }
     return 0;
}
```

The following is the content of the header file:

```
#if !defined(SUDOKUSOLVER_H)
#define SUDOKUSOLVER_H (1)

#include <ctime>
#include <time.h>
#include <math.h>
#include <cstdio>
#include <string>
#include <vector>
#include <stdio.h>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <iostream>
#include <algorithm>
#include <sys/utime.h>

//Functions
bool ClueEliminationCheck(int numBands, int
numTotalBands);
void PerformSearch(int numBands, int numTotalBands,
int currentStep);
std::vector<std::vector<std::vector<int> > >
CreateGrid(int numBands, int numTotalBands,
std::vector<std::vector<int> > clueSet);

#endif
```

The following is the content of SudokuSolver.cpp:

```
#include "SudokuSolver.h"

//Create the Sudoku grids we'll use during the search
as well as vectors to store the info at each layer of
the search
//Use a temp grid as well so we can undo changes when
a contradiction is reached
std::vector <int> valuesSet;
std::vector<std::vector <int> > cellsSet;
```

```
std::vector<std::vector<std::vector<int> > >
sudokuGrid;
std::vector<std::vector<std::vector<std::vector<int> >
> > sudokuGridTemp;

std::vector<std::vector<std::vector<int> > >
CreateGrid(int numBands, int numTotalBands,
std::vector<std::vector<int> > clueSet)
{
     for (int iColumn = 0; iColumn < numTotalBands;
iColumn++)
     {
          //Create a 2D vector to hold each cell in a
row, which holds multiple values
          std::vector<std::vector<int> > newRow;
          for (int iRow = 0; iRow < numTotalBands;
iRow++)
          {
               //Each cell is its own vector to hold
the potential values in non-clue cells
               std::vector<int> newCell;

               //Areas on screen that aren't part of a
grid (since we use a square for simplicity) are set to
0
               //This will make the search skip over
them since there is only 1 value & it is not 1-9
               int bound1 = numBands - 1;
               int bound2 = numBands +
sqrt(double(numBands));
               int bound3 = numBands -
sqrt(double(numBands));
               int bound4 = (numTotalBands - 1) - (2 *
sqrt(double(numBands)));
               if ( ((iRow > bound1) && (iRow <
bound2)) && ((iColumn < bound3) || (iColumn > bound4))
)
               {
                    newCell.push_back(0);
               }
               else if ( ((iColumn > bound1) &&
(iColumn < bound2)) && ((iRow < bound3) || (iRow >
bound4)) )
               {
                    newCell.push_back(0);
               }
               else
               {
                    //If clueSet has a 0, there was no
clue, so fill in all possible values
                    if (clueSet[iColumn][iRow] == 0)
                    {
                         for (int iClue = 0; iClue <
numBands; iClue++)
                         {
                              newCell.push_back(iClue
+ 1);
                         }
                    }
                    //Otherwise it gets 1 value--the
clue value
                    else
                    {

     newCell.push_back(clueSet[iColumn][iRow]);
                    }
               }
               //Add our newly created cell to the row
               newRow.push_back(newCell);
          }
          //Add our newly created row to the grids
          sudokuGrid.push_back(newRow);
     }

     //To make the search more efficient, we remove
possibilities ruled out by the clues
     bool setUpError = ClueEliminationCheck(numBands,
numTotalBands);
     if (setUpError == true)
     {
          std::cout << "Contradiction found in initial
set up. Please check clues give in main.cpp.";
          system("pause");
     }

     //If no errors, continue with the search
     int stepCount = 0;
     PerformSearch(numBands, numTotalBands,
stepCount);

     return sudokuGrid;
}
```

```
bool ClueEliminationCheck(int numBands, int
numTotalBands)
{
     //Grid 0 is top left grid; grid 1 is top right
grid; grid 2 is center grid
     //Grid 3 is bottom left grid; grid 4 is bottom
right grid
     bool contradiction = false;
     int valueCount[9] = {0,0,0,0,0,0,0,0,0};
     int rowStartingValues[5] = {0,0,6,12,12};
     int columnStartingValues[5] = {0,12,6,0,12};

     //Need to check each grid individually to make
sure clues are only eliminated within the correct grid
     for (int iGrid = 0; iGrid < 5; iGrid++)
     {
          for (int iColumn =
columnStartingValues[iGrid]; iColumn <
(columnStartingValues[iGrid] + numBands); iColumn++)
          {
               for (int iRow =
rowStartingValues[iGrid]; iRow <
(rowStartingValues[iGrid] + numBands); iRow++)
               {
                    //If cell only contains a given
clue, remove that value from all other cells in the
row, column, & block
                    if (
(sudokuGrid[iRow][iColumn].size() == 1) &&
(sudokuGrid[iRow][iColumn][0] != 0) )
                    {
                         int value =
sudokuGrid[iRow][iColumn][0];

                         //Move along the row removing
the value from each cell
                         for (int jColumn =
columnStartingValues[iGrid]; jColumn <
(columnStartingValues[iGrid] + numBands); jColumn++)
                         {
                              if ( (iColumn !=
jColumn) && (sudokuGrid[iRow][jColumn].size() > 1) &&
(std::binary_search(sudokuGrid[iRow][jColumn].begin(),
sudokuGrid[iRow][jColumn].end(), value) == true) )
                              {

     sudokuGrid[iRow][jColumn].erase(std::remove(sudok
uGrid[iRow][jColumn].begin(),
sudokuGrid[iRow][jColumn].end(), value));

                                   if
(sudokuGrid[iRow][jColumn].size() == 0)
                                   {
                                        contradiction
= true;
                                        break;
                                   }
                              }
                              else if
(sudokuGrid[iRow][jColumn].size() == 1)
                              {
                                   int valueCheck =
sudokuGrid[iRow][jColumn][0];

     valueCount[valueCheck - 1]++;
                              }
                         }

                         //Check for repeated values
                         for (int iValue = 0; iValue <
numBands; iValue++)
                         {
                              if (valueCount[iValue] >
1)
                              {
                                   contradiction =
true;
                              }
                              valueCount[iValue] = 0;
                         }

                         //Exit if reached a
contradiction
                         if (contradiction == true)
                         {
                              break;
                         }

                         //Move along the column
removing the value from each cell
                         for (int jRow =
rowStartingValues[iGrid]; jRow <
(rowStartingValues[iGrid] + numBands); jRow++)
                         {
                              if ( (iRow != jRow) &&
(sudokuGrid[jRow][iColumn].size() > 1) &&
(std::binary_search(sudokuGrid[jRow][iColumn].begin(),
sudokuGrid[jRow][iColumn].end(), value) == true) )
                              {

     sudokuGrid[jRow][iColumn].erase(std::remove(sudok
uGrid[jRow][iColumn].begin(),
sudokuGrid[jRow][iColumn].end(), value));

                                   if
(sudokuGrid[jRow][iColumn].size() == 0)
                                   {
                                        contradiction
= true;
                                        break;
                                   }
                              }
                              else if
(sudokuGrid[jRow][iColumn].size() == 1)
                              {
                                   int valueCheck =
sudokuGrid[jRow][iColumn][0];

     valueCount[valueCheck - 1]++;
                              }
                         }

                         //Check for repeated values
                         for (int iValue = 0; iValue <
numBands; iValue++)
                         {
                              if (valueCount[iValue] >
1)
                              {
                                   contradiction =
true;
                              }
                              valueCount[iValue] = 0;
                         }

                         //Exit if reached a
contradiction
                         if (contradiction == true)
                         {
                              break;
                         }
                         //Move through block removing
the value from each cell
                         int rowMod = iRow %
int(sqrt(double(numBands)));
                         int columnMod = iColumn %
int(sqrt(double(numBands)));

                         for (int kRow = (iRow -
rowMod); kRow < (iRow + (3 - rowMod)); kRow++)
                         {
                              for (int kColumn =
(iColumn - columnMod); kColumn < (iColumn + (3 -
columnMod)); kColumn++)
                              {
                                   if (
(sudokuGrid[kRow][kColumn].size() > 1) &&
(std::binary_search(sudokuGrid[kRow][kColumn].begin(),
sudokuGrid[kRow][kColumn].end(), value) == true) )
                                   {

     sudokuGrid[kRow][kColumn].erase(std::remove(sudok
uGrid[kRow][kColumn].begin(),
sudokuGrid[kRow][kColumn].end(), value));

                                        if
(sudokuGrid[kRow][kColumn].size() == 0)
                                        {

     contradiction = true;
                                             break;
                                        }
                                   }
                                   else if
(sudokuGrid[kRow][kColumn].size() == 1)
                                   {
                                        int valueCheck
= sudokuGrid[kRow][kColumn][0];

     valueCount[valueCheck - 1]++;
                                   }
                              }
                         }

                         //Check for repeated values
                         for (int iValue = 0; iValue <
numBands; iValue++)
```

```
                         {
                              if (valueCount[iValue] >
1)
                              {
                                   contradiction =
true;
                              }
                              valueCount[iValue] = 0;
                         }

                         //Exit if reached a
contradiction
                         if (contradiction == true)
                         {
                              break;
                         }
                    }
               }

               //Exit this loop too if reached a
contradiction
               if (contradiction == true)
               {
                    break;
               }
          }

          //Exit this loop too if reached a
contradiction
          if (contradiction == true)
          {
               break;
          }
     }

     return contradiction;
}

void PerformSearch(int numBands, int numTotalBands,
int currentStep)
{
     int minimum = 10;

     //Start minimum at 10, so if no cell has more
than 1 value, it stays at 10 which is clearly not a
possible state for a cell
```

```
     //Find cell with lowest number of clues to keep
the algorithm fast
     for (int iColumn = 0; iColumn < numTotalBands;
iColumn++)
     {
          for (int iRow = 0; iRow < numTotalBands;
iRow++)
          {
               if ( (sudokuGrid[iRow][iColumn].size()
< minimum) && (sudokuGrid[iRow][iColumn].size() > 1) )
               {
                    minimum =
sudokuGrid[iRow][iColumn].size();
               }
          }
     }

     if (minimum < 10)
     {
          int iRowRand = rand();
          int iRow = iRowRand % numTotalBands;

          int iColumnRand = rand();
          int iColumn = iColumnRand % numTotalBands;

          while (sudokuGrid[iRow][iColumn].size() !=
minimum)
          {
               iRowRand = rand();
               iRow = iRowRand % numTotalBands;

               iColumnRand = rand();
               iColumn = iColumnRand % numTotalBands;
          }

          if (sudokuGrid[iRow][iColumn].size() ==
minimum)
          {
               //Save the current grid & location
being set
               sudokuGridTemp.push_back(sudokuGrid);
               std::vector <int> location;
               location.push_back(iRow);
               location.push_back(iColumn);
               cellsSet.push_back(location);
               //While cell has multiple
possibilities, randomly select one value to fill in
               int random = rand();
               int randomChoice = random %
sudokuGrid[iRow][iColumn].size();
               int value =
sudokuGrid[iRow][iColumn][randomChoice];
               valuesSet.push_back(value);

               //Change the cell to the randomly
selected value & check for contradictions
               sudokuGrid[iRow][iColumn].assign(1,
value);
               bool restoreGrid =
ClueEliminationCheck(numBands, numTotalBands);

               //If a contradiction was found, restore
the grid & erase the value that caused it
               if (restoreGrid == true)
               {
                    sudokuGrid =
sudokuGridTemp[currentStep];

     sudokuGrid[iRow][iColumn].erase(std::remove(sudok
uGrid[iRow][iColumn].begin(),
sudokuGrid[iRow][iColumn].end(), value));
                    sudokuGridTemp[currentStep] =
sudokuGrid;

                    //If one possible value left, need
to check for contradiction
                    if
(sudokuGrid[iRow][iColumn].size() == 1)
                    {
                         valuesSet[currentStep] =
sudokuGrid[iRow][iColumn][0];
                         restoreGrid =
ClueEliminationCheck(numBands, numTotalBands);
                    }
               }

               //If we did find a contradiction in the
last possible value, we need to step back 1 layer &
try again
               while ( (restoreGrid == true) &&
(sudokuGrid[iRow][iColumn].size() < 2) )
               {
                    currentStep--;
                    restoreGrid = false;

                    iRow = cellsSet[currentStep][0];
                    iColumn =
cellsSet[currentStep][1];

                    //Restore grid to previous step's
state, erase the problematic value, & re-save the grid
                    sudokuGrid =
sudokuGridTemp[currentStep];

     sudokuGrid[iRow][iColumn].erase(std::remove(sudok
uGrid[iRow][iColumn].begin(),
sudokuGrid[iRow][iColumn].end(),
valuesSet[currentStep]));
                    sudokuGridTemp[currentStep] =
sudokuGrid;

                    //Erase the saved info for the
step that led to a contradiction
                    cellsSet.pop_back();
                    valuesSet.pop_back();
                    sudokuGridTemp.pop_back();

                    //If one possible value left, need
to check for contradiction
                    if
(sudokuGrid[iRow][iColumn].size() == 1)
                    {
                         valuesSet[currentStep] =
sudokuGrid[iRow][iColumn][0];
                         restoreGrid =
ClueEliminationCheck(numBands, numTotalBands);
                    }
                    //If all values lead to
contradiction, need to step back another layer
                    else if
(sudokuGrid[iRow][iColumn].size() == 0)
                    {
                         restoreGrid = true;
                    }
               }

               //If no contradictions, start another
layer of search
               if (restoreGrid == false)
```

```
                        {
                                currentStep++;
                                PerformSearch(numBands,
numTotalBands, currentStep);
                        }
                }
        }

        return;
}
```