Search, Constraint, and Sudoku

Wes Robbins wesley.robbins@student.montana.edu
Department of Computer Science
Montana State University
Bozeman, MT 59717, USA

Dave Miller david.miller26@student.montana.edu
Department of Computer Science
Montana State University
Bozeman, MT 59715, USA

Editor: Asad Noor

Abstract

Constraint satisfaction problems (CSPs) are a class of problems that can be defined by set of variables and a set up constraints on the variables. In order for a solution to a CSP to be correct, all variables must fit all constraints. In this paper we examine constraint satisfaction problems along with common algorithms that are used to solve them. Specifically, we view Sudoku as a CSP such that each square is a variable and the set of constraints are the standard sudoku rules. To solve sudoku puzzles, we implement a total of five algorithms from two different approaches. We implement three depth first search algorithms and two local search algorithms. We discuss the algorithms and our implementations and run experiments to compare the efficiency of each algorithm for solving sudoku puzzles. Additionally, we test the algorithms on puzzles of different difficulty levels. This allows use to examine how each algorithm's efficiency changes between difficulty levels and how the changes in efficiency compare across algorithms. It was found that simulated annealing performed the best, followed by the depth-first search methods, and that the genetic search is not well suited for this problem. Further, it was shown that more difficult puzzles do not necessarily take more time to solve.

1. Introduction

In practice, constraint satisfaction problems are solved using some form of state-space search (Russell and Norvig (2021)). Thus, one must search a graph created out of variables and constraints on those variables in order to find a goal state. Regular graph searching methods such as depth-first search, breadth-first search, etc. can be used on these problems but with heuristic methods tailored to the individual problem we can speed up the search significantly. This paper uses the constraint satisfaction problem Sudoku (9x9) as its main task and shows the advantages and downfalls of five different search algorithms to solve this problem.

Sudoku can be formulated as a CSP as follows. We are given an initialized board with some squares filled in (at least 8 unique numbers must be represented for it to have a unique solution). Then we can formalize the constraints as: each row must contain the numbers 1-9 without repeat, each column must contain the numbers 1-9 without repeat, and each sub-square must contain the numbers 1-9 without repeat. Notice that we will perform all experiments on 9x9 Sudoku boards to keep down computation time but that the results will still apply to any NxN board. Each algorithm will be tested on 20 different boards, including five easy, five medium, five hard, and five extremely hard. A count of major decisions made within each solver will be used to compare the performance across algorithms.

Hypothesis We expect forward checking and arc consistency to be more efficient than backtracking since these algorithms utilize context-specific information during the search and are able to prune large portions of the graph of paths. We expect arc consistency to have the greatest improvement because arc consistency has more constraints (and therefore prunes more paths of the search tree) than only forward checking. Additionally, we expect arc consistency and forward checking to have the greatest improvements in efficiency on the hardest puzzles, because we suspect harder problems have a greater number of possible paths and therefore pruning the search tree of paths will have a greater effect.

For the local search methods, we hypothesis that simulated annealing will be more likely to get stuck in local optima the the genetic algorithm (GA). However, we expect that simulated annealing will be more efficient if it is able to solve the problem, because the GA requires has higher overhead computation.

When comparing local search and DFS algorithms, we expect the local search algorithms to be more efficient. In a local search algorithm, you can trivially initialize the board such that one of the three constraints (i.e rows, columns, or sub-squares) are satisfied for all variables. This takes the problem from a 3-constraint problem to a 2-constrain problem. Due to this factor we believe local search algorithms will find solutions more efficiently.

2. Algorithms

We implement five different algorithms for solving the Sudoku puzzles. Three are depth first search algorithms. The other two are variation of local search. In this section, we provide a description of each algorithm and our implementation. Additionally, to help explain our implementations we provide high level pseudo-code for each algorithm, but leave out fine-grained details for brevity. More exact implementation details can be found in the comments of our code.

Depth First Search Algorithms

Simple Backtracking

Backtracking is a method of programming a simple depth first search where each branch is searched until no branch if found(Russell and Norvig (2021)). The algorithm then 'backtracks' to the most recent possible solution. The time complexity of backtracking to algorithm for Sudoku is $O(9^M)$ where M is the number of empty squares.

We do a recursive implementation of the backtracking algorithm such that each time a new variable is assigned, a recursive call is made with the updated board. If the recursive function receives a board state that is a dead end, the function exits which facilitates 'backtracking' to previous recursive calls. Once an accurate solution is found, the solution is returned from board. This procedure is further detailed by the pseudo-code in Algorithm 1.

Backtracking with Forward Checking

Forward checking is an improvement on backtracking that works by checking the implications of the current variable assignment on its adjacent unassigned variables (Russell and Norvig (2021)). In this context, adjacent variables are those it shares a row, column, or sub-square with. It does this by keeping track of the domains of all variables- the values they can hold without violating constraints. If while assigning the current variable we notice that it leaves an adjacent variable with an empty domain, we can backtrack early and thus speed up computation by pruning parts of the graph earlier than vanilla backtracking. The pseudo code is shown in Algorithm 2.

Algorithm 1 Backtracking(board, currentSpace)

```
1: Values = 1...9
2: if board is filled then return board
3: end if
4: if currentSpace ≠ '?' then                          ▷ Handle case for fixed values
5:     Backtracking(board, nextSpace)
6: end if
7: for n ∈ Values do
8:     board[currentSpace] ← n
9:     if board fulfills constraints then      ▷ If board is valid make recursive call to next state
10:        updated ← Backtracking(board, nextSpace)
11:    end if
12:    board[currentSpace] ← '?'                                  ▷ reset to previous state
13: end for
```

Algorithm 2 ForwardChecking(board, currentSpace, domains)

```
1: if board is filled then return board
2: end if
3: for n ∈ domains of currentSpace do
4:     board[currentSpace] ← n
5:     if board is valid then
6:         domains ← forwardCheck(board, currentSpace, domains)
7:         if all domains ≠ ∅ then
8:             board ← ForwardChecking(board, nextSpace, domains)
9:         end if
10:    end if
11:    reset domains
12:    board[currentSpace] ← '?'                                  ▷ backtracking
13: end for
```

Backtracking with Arc Consistency

Arc consistency is a more specific case of local consistency (Russell and Norvig (2021)). Local consistency enforces consistency between variables that are local to each other. More specifically, Arc consistency requires that all neighbouring variables can be satisfied after assignment to a variable. In other word, a move in the state space does not create an unsolvable position for a future state that is local to the current state.

We implement arc consistency by enforcing consistency after assignment of a new variable. We do so by tracking the domain of each variable throughout execution. If a new variable assignment results in an empty domain for neighbouring value, the function exits and the algorithm recursively backtracks.

Algorithm 3 ArcConsistency(board, currentSpace, domains)

```
1: if board is filled then return board
2: end if
3: for n ∈ domains of currentSpace do
4:     if board is legal then
5:         board[currentSpace] ← n
6:         domains ← deleteInconsistentDomains(domains)
7:         if check then
8:             board ← ArcConsistency(board, nextSpace, domains)
9:         end if
10:    end if
11:    reset domains
12:    board[currentSpace] ← '?'                          ▷ backtracking
13: end for
```

Local Search Algorithms

Local search is different from the above algorithms in that instead of starting from the given, unfilled board state in the graph, the board is filled completely and the search starts from that state. Thus, local search permutes the state one square at a time and searches within the locality of that random initialization. The algorithms below are both based on physical processes and give direction to the local search for a goal state.

In the implementations of both of our local search algorithm, we initialize the board such that one constraint is fulfilled for all variables. This can be done trivially. For example, in our simulated annealing implementation, all rows are solved during initialization by putting values 1..9 in each row without checking for violations in columns or sub-squares. During execution we never replace variables, but only swap variable values within the same already solve sections. This implementation detail has no affect on the core algorithmic approach, but does greatly decrease the search space.

Local Search with Simulated Annealing

Annealing is the physical process by of letting a metal cool slowly in order to toughen it (Humphreys and Hatherly (2012)). Simulated annealing takes inspiration from this process by introducing a temperature variable T, which is cooled in order to harden the robustness of an algorithm (Russell and Norvig (2021)). Specifically, the stochasticity introduced by simulated annealing can prevent an algorithm from getting stuck in a local optima. In the simulated annealing algorithm, a high temperature value corresponds to greater randomness. The temperature starts high and *cools* throughout program execution. The temperature can be decreased through one of many schedules, including

linear or geometric. This simulated annealing method can be applied to different variations of local search algorithms.

We implement a local search algorithm such that a random square $x_{i,j}$ is selected from the board (excluding pre-assigned squares) of state S. We only consider squares that have constraint violations (this implementation detail decreased clock time by an average of 25%). The value of the chosen square is swapped with the value of another chosen square in the same row $x_{k,j}$ to create a board in state S'. If the fitness of $S' > S$, then the swap is accepted. If not, the swap is chosen by sampling a probability distribution calculated from the following equation

$$P = e^{\frac{-(f(S)-f(S'))}{kT}} \tag{1}$$

where f calculates the fitness of a board, k is a constant, and t is the temperature. Next, we geometrically decrease T such that $T_{t+1} = T_t * x$ where $0 < x < 1$.

Our implementation loops through the above process until the puzzle is solved or the search algorithm becomes stuck in a local optima. If the latter is true, the temperature t is reset to the initial temperature T_0. The algorithm is determined to be stuck if the board has not changed position in r iterations.

During testing, we tune the parameters to the following values: initial temperature $T_0 = 10$, scalar value $k = 10$, scheduling value $v = .9$, and the number of iterations before reset $r = 100$.

Algorithm 4 SimAnnealing(board)

```
while not solved do
    swap to random squares in same row in board
    calulate fitness
    if fitness improved then
        keep swap
    else
        prob = e^(change in fitness / kT)
        sample from prob to decide whether to leave swap
    end if
end while
```

Local Search with Genetic Algorithm

The genetic algorithm is a process that reflects the ideas of natural selection and survival of the fittest from evolutionary biology. It is a population based method that keeps a number of individuals that 'reproduce', 'mutate', 'compete', and 'die' in a way somewhat similar to biology. An important part of the GA is a fitness function which indicates how good an individual is compared to the goal state. Since the goal state of Sudoku is zero collisions, fitness was measured by counting the total number of collisions on a board (rows and columns, since the sub-squares have no collisions by construct). Since this is not an optimal algorithm for Sudoku, it was found that most of the time multiple restarts are required to find a solution. This was done by generating a new population and breeding it with the old one. Tunable parameters included population size, the number of generations to restart after without change, the number of individuals used in tournament selection.

First, a number of individuals (boards) are filled randomly such that one of the three constraints (rows, columns, sub-squares) is not violated. Then, for each generation, half of the boards are chosen via a selection process, these boards are mated via a crossover process to produce children, the children are mutated, then replaced into the population. In this implementation, selection was done by tournament, where $k = 2$ members were chosen to compete randomly, and the one with higher fitness was taken until half the population was selected. Crossover was done by choosing a

random number of points, $k \in \{1, ..., 8\}$ and then taking a random sample of k sub-squares and switching the chosen sub-squares to create a child board. Mutation was done by choosing a random sub-square and switching two squares inside of it. Thus, by this construction, and new child would not have a conflicting sub-square. Finally, replacement was done via an elitist steady-state process where the weakest half of the population was killed and replaced by the children. (Deng and Li (2011))

Algorithm 5 Genetic(board, populationSize, maxGenerations)

```
1: while not solved do
2:     for i = 0...populationLength * 2 do
3:         population[i] ← getRandomBoard(board)
4:     end for
5:     population.sort(key = getFitness)
6:     population ← population[:populationLength]        ▷ take better half of random population
7:     for i = 0...maxGenerations do
8:         selected ← selection(population)
9:         children ← crossover(selected)
10:        population ← replace(population, children)
11:        population.sort(key = getFitness)
12:        if getFitness(population[0]) is 0 then return population[0]        ▷ solution found
13:        end if
14:    end for
15: end while
```

3. Experimental Approach

We design our experiments with the goal of comparing efficiency between algorithms. We design decision metrics (discussed below) such that reasonable comparisons can be made between algorithms. The most natural comparisons can be made between algorithms of the same type (i.e DFS or local search), and we do this first. Additionally, we loosely compare across types (DFS vs. local search), making interesting observations since we cannot compare directly. Finally, we compare how each type of search performs against each difficulty level and contrast those results.

Comparing DFS Algorithms

In order to compare the three variations of the backtracking algorithms we use a decision metric such that each time a move in the state space is *evaluated* we increment a decision counter. In the *simple backtracking* and *forward checking* algorithms, the algorithm only directly *assigns* values and never evaluates future state. On the other hand, the *arc consistency* algorithm *considers* and evaluates future possible board positions. Because *arc consistency* considers future positions, using the number of recursive calls is not a viable metric. Instead, we consider directly making a move in the state space and considering a future state space are of equal cost, because in practice both require that the entire board state is checked for correctness. Our cost metric can generally be described as the $C(x_i \leftarrow v) = 1$ where x_i is a variable on the board and v is any value.

Comparing Local Search Algorithms

Since our local search methods are fundamentally different from the DFS methods, we separate it into a different section for comparison. These methods are highly sensitive to hyper-parameter values as well as implementation tactics and randomness. However, using the following cost metric,

we can achieve a reasonably accurate idea of how each search is performing. In these algorithms, we consider a move in state space to include: a swap for simulated annealing (shown by one iteration of the main loop), and genetic operators performed on one individual for the genetic search. Now, this is still difficult as one generation of the genetic search involves numerous swaps, based on the population size and how crossover and mutation are performed. Thus, we use a single fitness check as our cost metric. This means that for every iteration / generation, simulated annealing will have one major decision and genetic search will have *populationSize* major decisions.

Comparing DFS and Local Search Approaches

To draw conclusions across all the algorithms, we will use both the number of major decisions and wall time as rough estimators. Let the reader be aware that since these are not accurate metrics for comparing these algorithms, we will not draw any hard conclusions and only postulate basic comparisons. However, we can compare performance across difficulty levels for DFS and local searches with some accuracy and will do so. A decision is counted for all algorithms when the board is checked for correctness.

4. Results

DFS Algorithms

For our first analysis we run all three DFS algorithms on all 20 puzzles and average the costs. After running this experiment, we see that forward checking algorithm performs the best according to our metric. Arc consistency is second best performing algorithm. Simple backtracking is the worst performing algorithm (5x worse that forward checking and 2x worse than arc consistency). These results can be visualized in Figure 2.

Next, we sorted the problems by difficulty according to the performance of the simple backtracking algorithm (i.e higher cost is more difficult problem [1]. Experimentally, we find that there is no statistically significant correlation between the relative performance between algorithms and the difficulty of the problem. The results from all problems sorted by difficulty can be seen in Figure 1. In Figure 1, we can see that arc consistency almost always outperforms simple backtracking and forward checking almost always outperforms arc consistency. While there are some variations in the comparisons, it does not correlate with increasing difficulty. We did the same comparison by ordering the puzzles by their denoted difficulty (i.e 'evil','easy', etc.) and also found that relative performance between the algorithms was not affected by the different problem subsets.

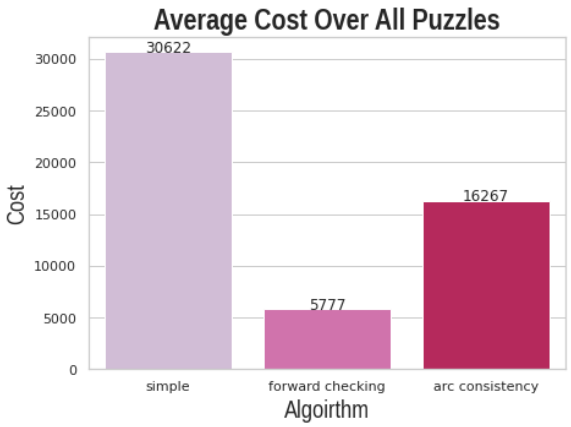

Figure 2: Results for Backtracking Algorithm

The fact that forward checking and arc consistency outperformed simple backtracking on our performance measure is consistent with out hypothesis. However, our hypothesis state that arc consistency would be the best performing algorithm. Experimentally, we find that forward checking

1. The difficulties given in the problem names (i.e 'easy', 'evil') seemed somewhat arbitrary for the performance on all five of our algorithms (e.g 'Evil-P4' was one of the easiest problems)

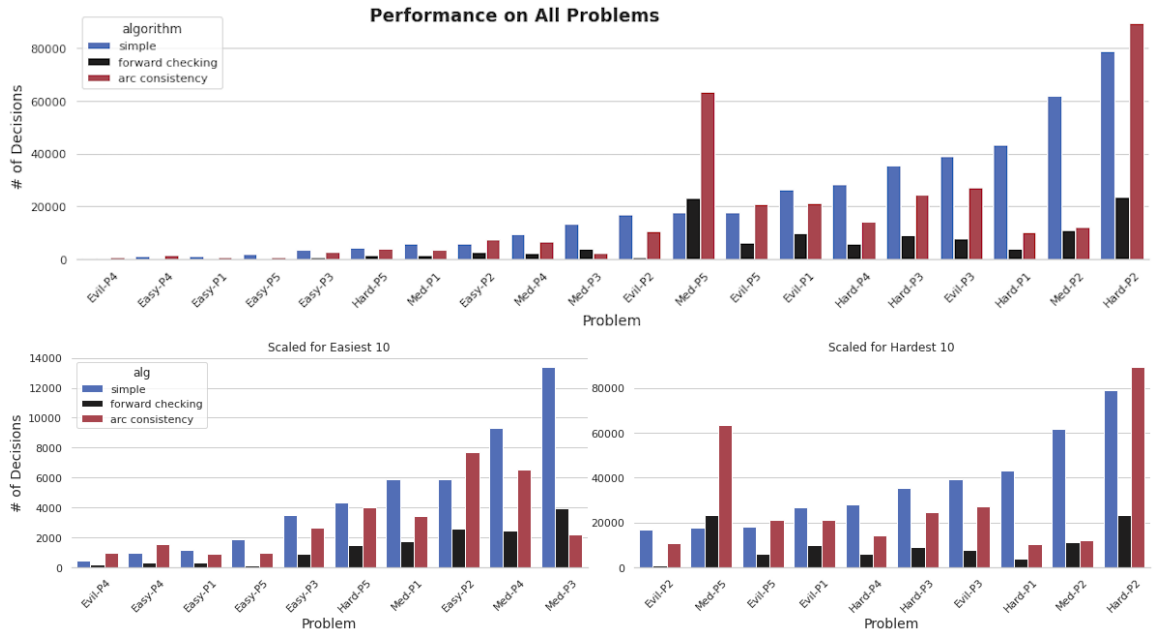

Figure 1: Performance for each algorithm across all problems. Problems ares sorted from easiest to hardest based on performance of the simple backtracking algorithm. This figure allows analysis of relative performance between algorithms as problem difficult increases. The graph on the second row is the same as the bar graph on the first except that the left half of the graph has 8x smaller increments.

is the most efficient DFS algorithm. Our hypothesis also stated that forward checking and arc consistency would offer the greatest improvements on the hardest problems. Experimentally, we see that relative improvement between algorithms is *not* correlated with the difficulty of the problem.

Local Search Algorithms

We did the same analysis on the local search methods, using a single fitness check as the main decision measure. After running the experiment, we see that simulated annealing is the more efficient algorithm by far, and although the genetic search found solutions for all the puzzles, it does not seem to be a good choice for Sudoku. It was noticed that these algorithms are highly prone to randomness, i.e. on one run a solution might be found almost instantly, and in the next run it will take the search much longer. This is mostly based on the initial random fill of the board and the randomness of the trajectory of the search.

Looking at figure 3, we can see that simulated annealing is around 100x more efficient that the genetic search. This can be explained by our metric (fitness check) and the fact that the genetic search has a population where the fitness of each individual must be checked for every generation, while simulated annealing only requires one check per iteration. On difficulty of the puzzles, the local searches performed similar to the DFS searches and we can see that there is little to no correlation between the difficulty and the number of steps to solve. This is interesting because it loosely shows that puzzle difficulty is somewhat arbitrary if you are not a human.

It is interesting that the genetic search performed so poorly compared to all the other algorithms. We noticed that almost every puzzle required multiple restarts, even though the search quickly converged to less than 10 conflicts. Thus, we postulate that Sudoku has a very large number of

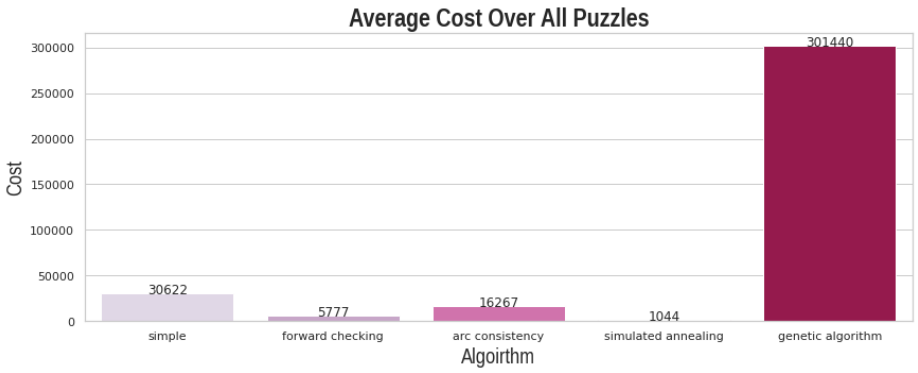

Figure 3: The average number of decisions made over all 20 puzzles for each of the five algorithms.

local optima compared to other problems and that genetic search tends to get stuck in these optima. Further, the given boards only have one solution and it is likely that most of these local optima do not lead to that solution. It is also likely that the construct of GA just does not work well with this kind of problem. The search was tuned so that one of the constraints was never violated, but both crossover and mutation were able to produce boards that had the same number of conflicts as before, only in different spots. Thus, the genetic search has a difficult time as it approaches the solution when compared to simulated annealing, which has the advantage of working by changing only one square at a time, which guarantees that it can follow the gradient. Therefore, we conclude that the genetic algorithm is better used for optimization problems.

Comparing All Algorithms

Our hypothesis state that local search algorithms would out perform the depth first search algorithms. Our experimentation showed this was not completely true. While simulated annealing was the best performing algorithm, the genetic algorithm did not perform better than any of the DFS algorithms. From these results in is hard to determine whether DFS or local search is a better general approach for Sudoku.

Wall Time

While wall time is generally not a reliable efficiency metric, we find it interesting to compare wall times in addition to the decision metrics previously discussed. To improve statistical significance of the wall times, we run the algorithms on all puzzles ten times and average the results. To improve dependability, we run them algorithms on the same laptop. We run the algorithms on all puzzles and sum the time to solve each algorithm. The results our in the table below.

Algorithm	All 20 puzzles (seconds)	Avg Time per Puzzle (seconds)
Simple Backtracking	3.41	0.17
Forward Checking	2.98	0.14
Arc Consistency	8.20	0.40
Simulated Annealing	8.20	0.40
Genetic Algorithm	3620.43	180.07

Table 1: Times in seconds to solve all 20 puzzles, averaged over 10 runs.

5. Summary

This paper compares various search algorithms used to solve the constraint satisfaction problem Sudoku. Five algorithms were considered, including three heuristic-based depth first search methods and two local search methods. The algorithms were tested on 20 different Sudoku grids varying from easy to extremely difficult and compared using both time and counts of major decisions. We find that both local search and depth first search are viable approaches for constrain satisfaction problems, with all algorithms showing good performance with the exception of genetic search. Simulated annealing was found to be the most efficient for solving Sudoku puzzles, and it was seen that given puzzle difficulty has little to no correlation with the time or steps it takes to find a solution.

References

Xui Qin Deng and Yong Da Li. A novel hybid genetic algorithm for solving sudoku puzzles. 2011.

Frederick John Humphreys and Max Hatherly. *Recrystallization and related annealing phenomena*. Elsevier, 2012.

Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach, 2021.