Cooper Strahan, Riley Slater
CSCI 446 - A.I.
September 20, 2021
Dr. Sheppard

Sudoku Project Report

Abstract

Algorithms used to solve Constraint Satisfaction Problems (CSPs) can be applied to solve sudoku puzzles. The backtracking algorithms, simple, forward checking, and arc consistency, were able to generate solutions for every puzzle that they were presented with. Note here that simple backtracking was not run on the more challenging puzzles due to time constraints. The Local Search algorithms, simulated annealing (SA) and genetic, were in general able to find solutions, but were not guaranteed to find a solution. In the case of the genetic algorithm (GA) on the "evil" puzzles the GA was not able to find a solution in any of the experimental runs. These results could be improved with tuning of the independent variables, given that variables were selected with regard to time constraints and optimization for all test cases rather than a specific test set.

Problem Statement

The primary focus of the project was to create efficient algorithms that correctly found a solution to any given sudoku puzzle. Five algorithmic approaches were to be implemented, potentially yielding different results for each algorithm. The algorithms were separated into two sets, backtracking algorithms and local search algorithms. Backtracking algorithms are "brute force" algorithms that are guaranteed to find a solution, while local search algorithms use a stochastic search methodology in order to generate a solution without a guarantee of correctness.

It is expected that the backtracking algorithms will require less backtracking to find a solution to sudoku as the optimization among algorithms increases. Optimization in this context means the increase in pruning abilities as algorithmic complexity increases from simple backtracking to arc consistency. The simple backtracking algorithm will take the most backtracks. Then, the forward checking algorithm will be more efficient and require less backtracks. Finally, the arc consistency algorithm will be the most efficient. This is because the optimization of the algorithms is increasing with each new implementation. Thus, reducing the number of times the algorithm will have to backtrack to get to a final solution, given that a solution exists. Forward checking on average will be less efficient than arc consistency because arc consistency is a more rigorous prune of potential values.

The local search algorithms are less related as they are implementations of two different methodologies. Our intuition led us to believe that the GA would outperform the SA algorithm in both speed and finding a correct answer. This thought arose because of the higher amount of complexity involved with the creation of a GA. As well as the assumption that the likelihood of finding a solution is higher when you are checking for a correct answer amongst an entire population of candidates rather than the updating of a single candidate.

Hypothesis

Backtracking algorithms will find a correct solution to any given Sudoku Puzzle and will require less steps to find that solution as the pruning abilities of the backtracking algorithm are improved. Local search algorithms more often than not will find a solution to any given Sudoku puzzle while the GA will outperform the SA algorithm.

Algorithm Descriptions

The backtracking algorithms use a graph data structure to model the sudoku board. Each cell in the board is a vertex in the graph and an adjacency list is constructed between the vertices that are related based on the constraints of sudoku. This adjacency list, and the given values, are then used to prune the potential values for each empty cell in the forward checking and arc consistency versions of the backtracking algorithm. This pruning is what reduces the number of backtracks for each algorithm. The simplest way to design these algorithms was to have functions for pruning inside the graph object for each type of algorithm that prunes; forward checking and arc consistency. Pruning works by assigning a value to a "first" empty cell, and then removing the value from the domains of the empty cells dependent on the "first" cell.

The simulated annealing algorithm takes a sudoku board represented by a two dimensional array as input, and asserts pseudo random values in all of the non given cells. The assertion is called pseudo random because the random values are inserted so as not to validate the constraint that each 3x3 subset of the puzzle contains the values 1-9. This design decision was made in order to improve the efficiency of the algorithm, while reducing the time spent evaluating the conflict statistics of a puzzle. The algorithm then enters into a loop with a starting temperature T of 1. A puzzle is then evaluated to determine the cost. The cost of the puzzle is equivalent to the number of constraints said puzzle currently violates. If a puzzle has a cost of 0, the puzzle is said to be correct, the algorithm terminates by returning the puzzle. If a puzzle is not returned, the algorithm continues. The algorithm then swaps two random values within a subset of a puzzle to create a new puzzle, a design decision made to prevent the violation of the initially satisfied constraint. The new puzzle is then accepted as the working puzzle in two instances. The first instance is if the cost of the new puzzle is lower than the cost of the working puzzle. The second is if the puzzle's cost is higher it is accepted with a probability P. P is defined with the following equations.

$$P = exp^{-\Delta/T}$$
$$\Delta = cost(new\ puzzle) - cost(working\ puzzle)$$
$$i = current\ number\ of\ iterations$$
$$T = (1 - (.001 * i))$$

This probability function allows the algorithm to accept a worse solution where the degree of how bad a solution is, is taken to effect. Solutions with many more conflicts will be taken less times than solutions with only one or two additional conflicts. The probability function, because of the decreasing number T, allows worse solution acceptance more often at the beginning of the algorithm but nearly never accepts a worse solution when reaching the end of our predetermined number of iterations. This piece of the algorithm allows our SA algorithm to jump out of local minimums in order to find the global minimum that it is searching for. If the algorithm does not find the correct solution over it's 10,001 iterations, the current best solution is returned.

The genetic algorithm accepts a sudoku puzzle represented by a two dimensional array as input and performs the same pseudo random number generation as simulated annealing. The difference being the GA performs this number generation 1000 times to create a population of sudoku boards. The algorithm then checks within the population for a sample with a fitness = 1. Fitness is determined by counting the number of unique values in each row and column, summing those values, and dividing the sum by 162 (i.e. the correct number of unique values for a solved board). If a value in the population achieves a fitness of 1, the algorithm is terminated and the solved puzzle is returned. The next step in the GA serves to create a new population, which is the next generation in the progression of the algorithm. The algorithm does this by creating another loop that randomly selects groups of individuals from the general population. The group size was selected to be 10 to keep computation costs down. A variation of Tournament Selection is then used to select the most fit members from the group. Each member of the group is evaluated for its fitness. The more fit individual wins with a probability S. The S equation is described below.

$$A = \textit{current puzzle}$$
$$h = 0.75$$
$$f_A = \textit{fitness}(A)$$
$$\textit{iter} = \textit{current iteration}$$
$$T = (1 - 0.01 * \textit{iter})$$
$$S = 1 - \exp((-h * f_A)/T)$$

This equation serves a similar purpose to the SA equation, the difference being that the SA puzzle's values are being compared against one another where the GA accepts puzzles based only on the fitness score of the individual. The h value is a tuned value that was adjusted based on run experiments. The function allows for worse scores to be accepted as winners in the tournament in early iterations of the algorithm but makes it more difficult for worse solutions to win later on. The worse solutions are said to be penalized by the equation across the entire running of tournament selection. The algorithm selects two winners using tournament selection on random subsets of the population. The two winners are then "crossed" to generate two new puzzles. This is done by swapping random values from the same 3x3 subsets of the puzzles. A fixer function is then run to also swap duplicates created by the swap function. This fixer finds the new duplicate from each puzzle and swaps across the two puzzles again. All of these swap actions are performed within the equivalent subsets in order to maintain the correctness of the initially satisfied constraint. The two new puzzles are then mutated with a probability of 0.07. This value was tuned based on experimental results. The new puzzles are added to the new generation. This action is performed 500 times in order to create a generation of 1000 new puzzles. We selected generational replacement in order to maintain lower scoring puzzles to attempt to avoid being stuck in local maximums. The T value is then reduced by 0.01, to allow the algorithm 100 iterations to find a solution. This algorithm is run fewer times because new population generation is computationally expensive and slow to run on our machines. If a solution is not found we return our best solution at the 100^{th} iteration.

SA and the GA were built as subclasses of an overarching superclass LocalSearch. The Local Search class contains random assertion and swap functions that are utilized by both the SA and GA.

Experimental Approach

The experiments were iterative. Each algorithm was tested against each set of puzzles 10 times, giving 50 observations for each algorithm at each level of difficulty; easy, medium, hard, evil. Noting that Simple Backtracking was not run on the Hard or Evil puzzles due to time constraints. Different information based on the type of algorithm were tracked and reported. Backtracking algorithms kept track of the number of backtracks. Local Search algorithms kept track of the correctness heuristic and the number of iterations to termination. The summary statistics were then gathered for each algorithm, and organized by puzzle difficulty. These statistics were written to a file placed in the project directory in order to maintain result continuity.

Results

Backtracking Results:

Algorithm	Difficulty	Obs	Min	Max	Mean	Variance *	Skew*
Simple	Easy	50	46	57,930	29044.5	339,838,245	-1.3
Fwd Chk	Easy	50	0	2368	1184	58,2290	-1.1
Arc Con	Easy	50	0	711	355.5	52,539	0.0
Simple	Medium	50	1763	63,530	32169.9	342,097,479	$4.9x10^{-4}$
Fwd Chk	Medium	50	3	2386	11981.1	582,326	-6.4
Arc Con	Medium	50	0	715	357.9	52,541	-9.8
Fwd Chk	Hard	50	19	2420	1222.1	582502	-2.1
Arc Con	Hard	50	4	720	361.5	52,545	-5.1
Fwd Chk	Evil	50	95	2630	1362.5	585890	0.0
Arc Con	Evil	50	23	790	406.5	52,939	0.0

Local Search Results:

Algorithm	Difficulty	Obs Type	Obs	Min*	Max*	Mean	Var*	Solution Found
Sim Anneal	Easy	Cost	50	0	4	1.48	1.56	18
		Iterations		3,100	10,001	8779	3,504,774	
Genetic	Easy	Fitness	50	0.975	1.0	0.991	7.5	22
		Iterations		24	100	79.72	669.59	
Sim Anneal	Medium	Cost	50	0	6	2.82	1.74	2
		Iterations		6211	10,001	9899.18	317,108	
Genetic	Medium	Fitness	50	0.975	1.0	0.984	4.94	3
		Iterations		64	100	98.16	55.04	
Sim Anneal	Hard	Cost	50	0	7	3.24	2.961	3
		Iterations		8748	10,001	9,946.96	58,550	
Genetic	Hard	Fitness	50	0.969	1.0	0.983	5.86	3
		Iterations		53	100	97.92	84.93	
Sim Anneal	Evil	Cost	50	0	6	3.32	2.99	4
		Iterations		6765	10,001	9844.46	382,417	
Genetic	Evil	Fitness	50	0.963	1.0	0.981	5.33	1
		Iterations		89	100	99.78	2.42	

* Rounded for brevity

Local Search Heuristic Counts **

		Min Conflict						
Algorithm	Difficulty	0	2	3	4	5	6	7
Sim Anneal	Easy	18	26	3	4	0	0	0
	Medium	2	27	4	12	3	2	0
	Hard	3	4	4	15	0	6	2
	Evil	4	17	4	12	6	7	0
		Fitness Score						
		0.962	0.969	0.975	0.981	0.9876	1	
Genetic	Easy	0	0	6	2	20	22	
	Medium	0	0	17	4	26	3	
	Hard	0	2	19	3	23	3	
	Evil	1	2	22	3	21	1	

**All numbers are rounded for brevity

Conclusion and Algorithm Behavior

The backtracking algorithms all performed consistently with what was expected of them. As problem difficulty increased for each algorithm, the number of backtracks increased. Backtracks also reduced as the pruning capabilities of the algorithms got better. This means that for each set of problems, the arc consistency performed the best, forward checking the second best, and simple backtracking performed the worst. The number of backtracks had high variability as shown by variance, and typically skewed negative, or had little to no skew.

SA performed differently than expected. On the easy puzzles it found a solution 36% of the time. The medium, hard, and evil puzzles found results 4%, 6%, and 8% of the time respectively. This result was interesting as typical intuition would lead one to assume that solutions would be found more often for easier puzzles. Contrary to this result, by examining the means, we show that the algorithms did come closer to finding a solution more often for puzzles by their orders of difficulty. The mean number of iterations does not follow a linear distribution, as the evil puzzles seemed to find solutions on average more quickly than the medium and hard puzzles. These results can be explained as “lucky” guesses during our 50 iterations. During previous experiment runs, the algorithms behaved in a more linear fashion. These results have been included in the project’s github repository. The results for the SA algorithm were disappointing, but could be improved by tweaking the tunable variables. It was interesting to see the fairly consistent results returned from the SA algorithm despite the algorithm difficulty. Future work for this

algorithm could include lowering iteration constraints and increasing the total number of runs; we believe these tweaks could vastly improve the performance of SA.

The GA was much slower than the rest of the algorithms. The GA was slower because population construction ended up being very computationally expensive. The results are more linearly correlated than the SA results. It is shown that the mean of the observations decreases as puzzle difficulty increases. The GA found slightly more valid solutions for easy puzzles in comparison with the SA algorithm, three solutions each for the hard and medium difficulty puzzles, and reached a valid solution a single time when run against the Evil puzzles. The algorithm also found solutions in a linear manner, where solutions were more consistently found earlier with respect to the easy puzzles and were found in later iterations when run on harder puzzles.The results show that variance in relation to the fitness heuristic is fairly stable for all of the difficulty levels. On average the GA found correct solutions a similar amount of times relative to the SA algorithm, disproving our hypothesis about its performance being better than SA.

Summary

Through this project we have shown that algorithms used to solve CSPs can be used to solve the sudoku puzzle. Implementations and simple analysis show that any of these algorithms can be used to solve a sudoku puzzle. The findings were consistent that Backtracking with Arc Consistency was the fastest and most reliable solution when writing an algorithm to solve the sudoku puzzle. All of the backtracking algorithms demonstrated the ability to solve any valid sudoku puzzle that was given to them as input. The implementations of the Local Search algorithms were slower and did not return correct solutions as often as desired. Our team assumes that this can be improved by tweaking tunable variables. In general Simulated Annealing was faster than the Genetic Algorithm, while both algorithms produced correct solutions at a similar rate.

References

"Backtracking Algorithms." *GeeksforGeeks*, www.geeksforgeeks.org/backtracking-algorithms/.

Edelkamp, Stefan, and Stefan Schrödl. "Arc Consistency." *Arc Consistency - an Overview | ScienceDirect Topics*, 2012, www.sciencedirect.com/topics/computer-science/arc-consistency.

Barták, Roman. "Constraint Propagation." *Constraint Guide - Constraint Propagation*, 1998, ktiml.mff.cuni.cz/~bartak/constraints/propagation.html

Lewis, R. Metaheuristics can solve sudoku puzzles. *J Heuristics* **13,** 387–401 (2007). https://doi.org/10.1007/s10732-007-9012-8

Schloss, J. (2017, February 22). *How algorithms evolve (genetic algorithms)*. YouTube. Retrieved September 19, 2021, from https://www.youtube.com/watch?v=qiKW1qX97qA.

Svardekar, P. (2020, November 10). *Lecture 16: Tournament SELECTION | isc*. YouTube. Retrieved September 19, 2021, from https://www.youtube.com/watch?v=9OXJapW8vqM.