Sudoku

Jonathan Meerson , Harel Farkash , Avi Korzac

Introduction to Artificial Intelligence course by Prof. Jeff Rosenschein

The Hebrew University of Jerusalem 23.8.2018

Introduction

Sudoku (数独), originally called Number Place) is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution. (Wikipedia)

Approach and Methods

Sudoku can be viewed as a search problem but since it has a big branching factor, it might be inefficient to solve it using uninformed search algorithms.
The main purpose of this paper is to find whether using CSP heuristics and techniques like Forward-Checking and Arc-Consistency are better in terms of efficiency and running time than basic uninformed search algorithms, like backtracking.

We've formulated the Sudoku Game as a Constraint-Satisfaction problem as follows:
Variables: X = All the tiles on the board.
Domains: $D = \{\{1, 2, ..., 9\}^{81}\}$
Constraints: C = Each value is assigned exactly once in each row, column and in a 3x3 block.

We also defined the **Neighbors** of a given tile as all the tiles which are in its 3x3 block, its row and its column.

In Addition, after reading some papers[1] [2] [3] about stochastic approaches to solving Sudoku, we wanted to check whether probabilistic algorithms like Simulated Annealing can solve it quickly.

[1] "stochastic optimization approaches for solving sudoku - arXiv."
[2] "Metaheuristics can Solve Sudoku Puzzles. - Rhyd Lewis."
[3] "Sudoku solving algorithms - Wikipedia."

API and GUI

The main goal of our project and paper was to evaluate and compare different algorithms for solving Sudoku boards. We chose to compare them on two aspects: time and number of actions (writing and deleting from the board). Each algorithm is implemented in a different class inheriting from a class named Solver which defines the behavior and outcome of inserting and deleting from the board.

When running a game with a specified solver type, the game creates a solver object, runs it, calculate its running time and when the solver finishes, the game is returned with a queue of actions. It then evaluates how many actions the solver made and can graphically display them.

Each Solver holds a reference to the Sudoku object which contains the board, a list of the read-only tiles and provides a set of static methods for general use like getting all values from a given row, column and block, getting all indexes of a block, all legal values that can be put to a given tile etc.

We also support a GUI that the user can watch with a simple parameter command line definition. The graphic interface we wrote uses the pygame library and is heavily referenced on brk3's github repository[4] with some changes of our own regarding the interface itself, since the user doesn't do anything with it rather than just watch some solver running on it.

Algorithms

Backtracking

A basic uninformed algorithm for solving CSPs that can be viewed as a **Depth First Search** with single variable assignment as a successor function. It traverses a tree of depth $|X|$ where each level represents a single variable X_i and each level has $|D_i|$ nodes, one for every possible value X_i can be assigned with. During its run it picks nodes that represent values which are consistent with its partial assignment.
If the current node isn't consistent with its partial assignment, it would traverse to its sibling nodes. If none of the nodes in the current level are consistent with its partial assignment, it would backtrack a level upwards and pick a node which represents a different value, and so on.

[4] "GitHub - brk3/Sudoku: Simple Sudoku game in Python."

If our traversal resulted in a complete assignment we return the assignment we've generated.
If none of our traversals resulted in a complete assigned, we return fail.
Our implementation of the backtracking algorithm iterates over the empty tiles and inserts legal values in them. When it got to a tile where no legal values could be inserted, it would go back to the previous tile, delete its value and put a different one, until the board is completely filled.

Constraint Satisfaction Problem Heuristics

Regular backtracking is based on an arbitrary way of choosing the variables, and ordering the values they can be assigned with.
We have implemented an improvement to the regular backtrack using three heuristics:
Minimum Remaining Values (MRV) - Choose the variable with the least remaining possible legal values to be assigned.
In Sudoku, it means to choose the tile that has the fewest numbers that can be assigned to it - we iterated the empty tiles in the board, and checked which are the tiles that can be assign the least number of values.
Degree Heuristic - It is a tiebreaker for MRV for a situation of 2 variables with the same number of MRV. We will choose the variable that will affect the maximum number of neighbor variables.
In Sudoku, if after calculating the MRV we got two or more tiles with minimal number of values to assign, we iterated over these tiles and chose the the tile with the highest number of unassigned neighbors.
Least Constraining value (LCV) - After a variable was chosen, we will choose a value that will affect the domain of the variables the least.
In sudoku**,** we will choose the value that will leave the chosen tile's neighbors the most values to be assigned to. In our implementation we sorted the list of legal values for the chosen tile by the amount of possible values for all its neighbors if assigning it. We also added some sort of Forward - Checking technique to it by eliminating values that gave to some neighbor zero legal values to be assigned to.

Arc Consistency

This algorithm makes sure that for every value that is set in a variable, is consistent to the other neighbor variables, i.e. It removes those values from the domain of *X* which aren't consistent with the constraints between *X* and *Y*. The algorithm keeps a collection of arcs that are yet to be checked; when the domain of a variable has any values removed, all the arcs of constraints pointing to that pruned variable (except the arc of the current constraint) are added to the collection.

In our implementation, we create a domain matrix with all the possible values that can be assigned into every variable, then we create an arcs queue which represents all the neighbors of all the variables (empty tiles). On that queue we activate the ac3 algorithm.

Forward Checking

This algorithm works very similarly to the Backtracking algorithm, with a single improvement:
Whenever the algorithm assign a variable X_i, it establishes arc consistency for it: for each unassigned variable Y_i that is connected to X_i by a constraint, we remove from Y_i's domain any value that is inconsistent with the value assigned to X_i.
If after this removal Y_i's domain size is zero, then our assignment for X_i is incorrect, and we backtrack our steps and assign a different value for X_i.
This algorithm allows us to fail faster, and prune any branch of our search tree which represents a consistent partial assignment that cannot be extended to a complete consistent assignment.

We’ve implemented this algorithm by checking after each assignment of a value to a tile whether one of its neighbors has an empty domain, meaning the tile has no legal values to be assigned to it. If such neighbor is found the assignment is canceled and we continue to the next legal value.

Simulated Annealing

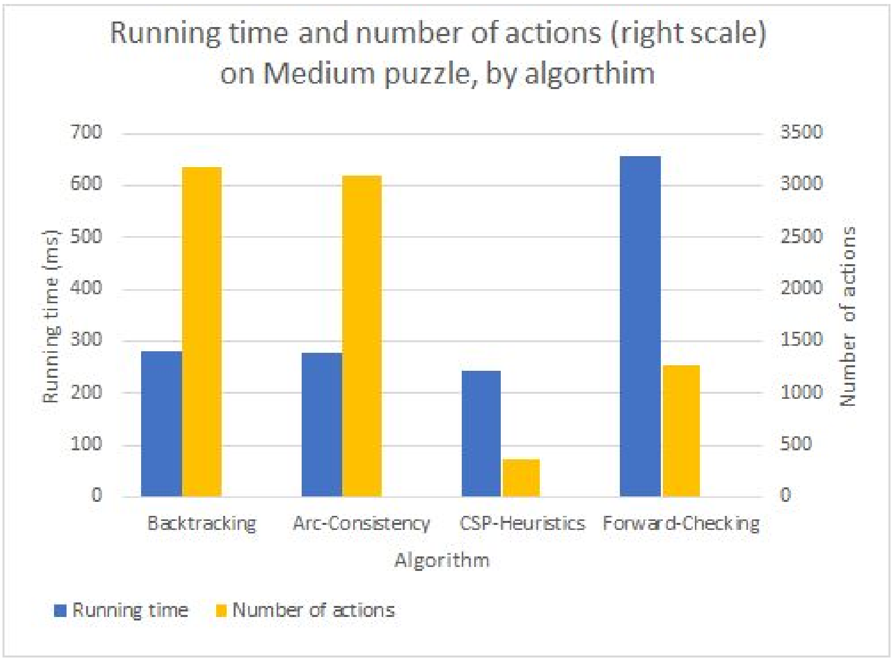
Simulated Annealing is a stochastic technique for finding global minimum or maximum of a function. Given a large search space and some scoring function, it finds a global optimum of this function and eventually returns the optimal final state. It is an iterative algorithm starting with some initial state. where at each iteration some successor state is calculated and scored using the funcion. It then decides whether to move to the new state or stay in the old one. It continues iterating until the optimal state is found. The choice between staying in the old state or moving the new state is probabilistic: it selects the state with the better score, but with some decreasing probability (over the iterations) it can “make a mistake” and go against the score difference and stick with the worse scored state. The probability to make this “mistake” depends on some predefined temperature which decreases in each iteration.
Our implementation was based on an article[5] by Rhyd Lewis from Edinburgh Napier University in Scotland with some changes.

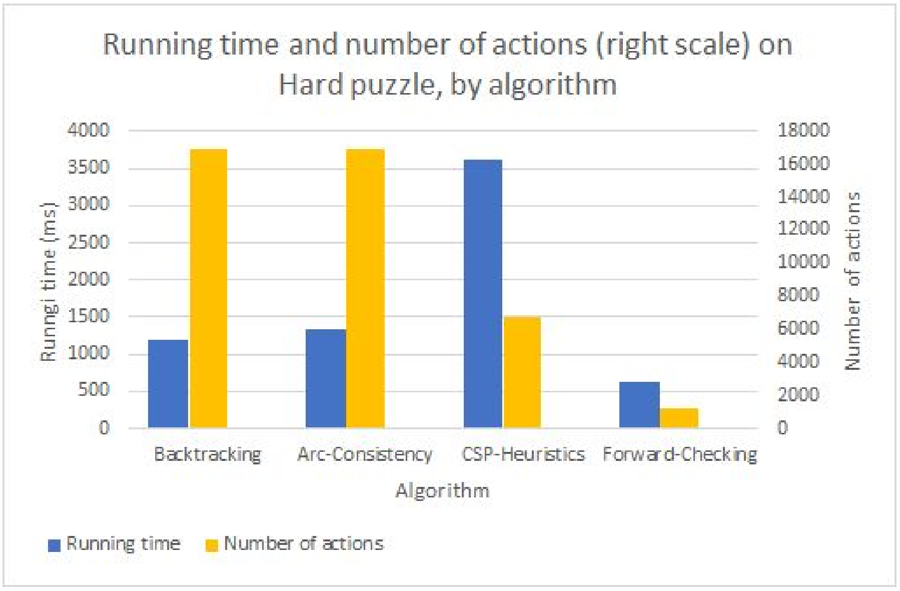
[5] "Metaheuristics can Solve Sudoku Puzzles. - Rhyd Lewis."

In the article, the board was initially randomly filled where only the 3x3 blocks had to be legal, meaning the rows and columns were in an illegal situation where a number could appear twice in the same row or column.
The scoring function was total number of violations (missing numbers) in each row and column.
In each iteration the successor state was calculated by swapping two tiles in some randomly chosen block (keeping the legality of a block) and with a chance to improve the legality of the rows and columns.
When we tried to implement the algorithm using this method we didn’t get many successful attempts and the running times were atrocious (around 10s). We decided to change it and add some new features:
We changed the first random fill to be column-wise and not block-wise. We decided to make that change because usually in Sudoku boards from the newspaper (which we used to test our results) the variance of number of initially filled tiles in the columns is lower that than the variance of number of filled tiles in the 3x3 blocks.
The successor generator was changed accordingly: pick a random column and swap two tiles in it. The scoring function was also changed accordingly: we calculated number of violations in the 3x3 blocks and in the rows since the columns were always legal.
Finally we added a “total escape” feature: we noticed that sometimes the algorithm was stuck for many iterations on an almost optimal score (with only two violations). So we decided to check at each iteration if the algorithm is stuck on the same score for a lot of iterations (depending on the score itself, with some formula that increases the higher the score), it randomly selects some columns and refills them again randomly. We thought to also reset the temperature in such case but it didn’t work well and was against the principle we saw in class where the probability of finding the global optimum approaches 1.
After many attempts the best temperature we found was 1 with a decreasing factor of 0.999.

Results

As mentioned before, we wanted to see if CSP heuristics and techniques had an improvement on the uninformed backtracking algorithm.
These are the results of each of our CSP algorithms compared to the Backtracking algorithms in terms of number of actions performed and running time, after running each one 100 times on 3 different puzzles:



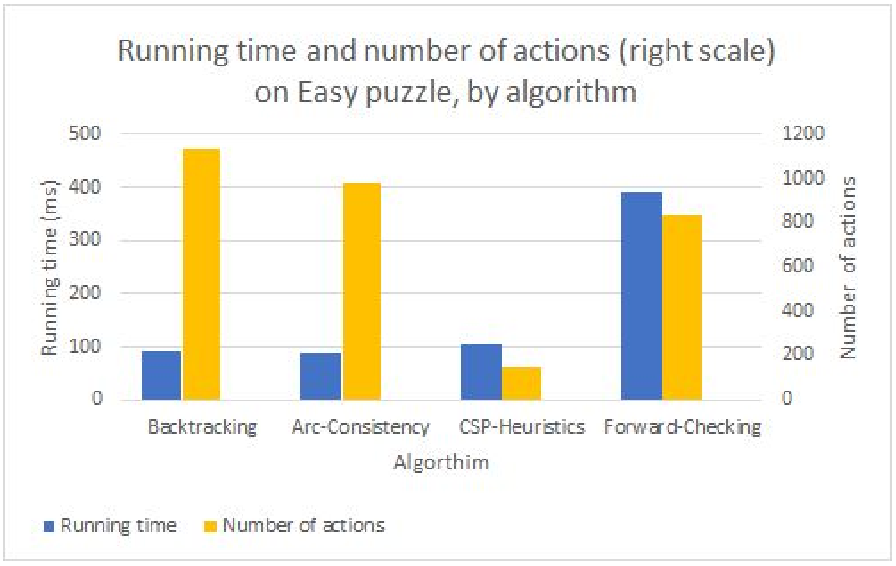


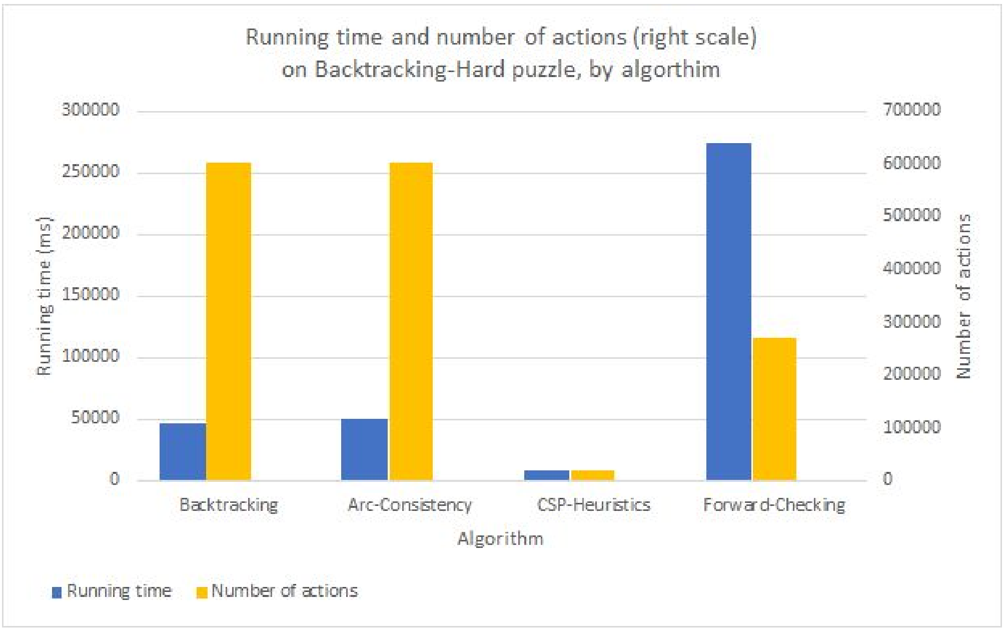
In addition, we found a “hard for backtracking” puzzle:

1	2		4			3		
3				1			5	
		6				1		
7				9				
	4		6		3			
		3			2			
5				8		7		
		7						5
							9	8

1	2	8	4	6	5	3	7	9
3	7	4	2	1	9	8	5	6
9	5	6	8	3	7	1	4	2
7	6	5	1	9	8	4	2	3
2	4	9	6	7	3	5	8	1
8	1	3	5	4	2	9	6	7
5	9	2	3	8	6	7	1	4
4	8	7	9	2	1	6	3	5
6	3	1	7	5	4	2	9	8

This puzzle is special and hard for backtracking because when iterating from left to right, top to bottom, the first cells contain very big values, so a backtracking algorithm that iterate over values from the smallest to largest (like ours), makes many mistakes before reaching the right solution.
The results for running each of the algorithms for 5 times on this board:





Besides CSP heuristics and techniques, we wanted to see if Simulated Annealing was a good algorithm for solving Sudoku. We ran it on each puzzle for 100 times for 30,000 iterations each, and calculated the percentage of successes (meaning the algorithm could in fact solve it under 30,000 iterations), and the average time of the successful attempts.
The results:

	Successes (%)	Time of success (s)
Easy	78	1.033
Medium	21	1.357
Hard	3	2.161

Discussion and Conclusions

The main goal of this paper was to compare the CSP heuristics and techniques to the simple backtracking algorithm.

On its own, the backtracking algorithm didn't perform as bad when looking at the running times. On the boards we took from newspaper (easy, medium and hard) it ran for less than a second (a bit more and the hard one) and in a perspective of a user waiting for an application to solve his sudoku, he performed well. Due to its lack of calculations (like heuristics or forward checking) it sometimes ran as fast or even faster than the algorithm that we implemented to improve him. On the other hand, the running time for the "hard for backtracking" board was almost 50 seconds. It can be explained by the fact that the true values in the starting tiles were big and the algorithm iterates over the smaller values first (when we changed the implementation to run on the bigger values first he was obviously really fast and made no mistakes). When looking at the the number of actions taken (deletion and insertion of values in tiles), he was always the worst. In the medium and hard puzzles he was tied for worst with the Arc-Consistency algorithm which will be discussed next.

The Arc-Consistency algorithm didn't perform well at all. On the medium board the number of actions was slightly smaller than the number of actions it took for the backtracking algorithm, and on the hard board and on the "hard for backtracking" it was the same. It means that the process of reducing the size of the domain of each tile did not work well and reduced the domains just slightly if at all. The reason for this is the fact that a value is removed from the domain of a tile if and only if it appears in the domain of its neighbor which is a strong condition given the board is usually only very partially filled in the beginning. This is why it did somewhat improve the number of actions in the easy puzzle which was already somewhat full in the first place.

The running time wasn’t good accordingly: since the initial process of reducing domains wasn’t very effective and consumed a lot of time, the running time of the algorithm was worse than the backtracking algorithm on the medium, hard and “backtracking hard” puzzles. On the “backtracking hard” it took almost 4 seconds more than the backtracking algorithm with the exact same number of actions.
We thought about implementing the Arc-Consistency in a different way that might be more effective: instead of declaring the tiles as the variables and the values 1-9 as the domains, do the opposite and declare the values 1-9 as the variables and the tiles as the domain. This might have had better results but the implementation of the next part of the algorithm, the backtracking itself would have been entirely different and not comparable to the regular backtracking algorithm.

The algorithm which used CSP heuristics (MRV, degree and LCV) had somewhat good results in the running time aspect and excellent results in number of actions:
On the easy puzzle it was a little worse than the backtracking algorithm but was almost 8 times better in the number of actions (144 vs 1134). The difference in time could be easily explained with the additional time the CSP algorithm had to use to calculate the heuristics.
On the medium puzzle it was the same with the number of actions taken: the CSP algorithm used almost 9 times less actions than the backtracking one (363 vs 3,173) and in this board it was also reflected in the running time where the CSP algorithm did a bit better.
On the other hand, on the hard puzzle the CSP algorithm had a somewhat bad running time compared to the other algorithms but still made a lot less actions than the backtracking algorithm.
The algorithm showed its greatness on the “hard for backtracking” puzzle. It took him less than 8 seconds to solve it while the backtracking algorithm ran for 46 seconds. The ratio between the number of actions was also amazing - the heuristics improved the number of actions by a factor of almost 33 (18,430 vs 600,984).

The Forward-Checking algorithm did poorly on most of the puzzles. Due to its heavy computation (at each assignment check all the empty neighbors for a tile with no possible legal values) it wasted a lot of time in comparison to how many actions it saved when comparing to the regular backtracking algorithm.
On the easy and medium puzzles it took the most time to finish and the difference between the number of actions between it and the backtracking algorithm wasn’t that big, in contrast to the CSP algorithm which sometimes took a bit more time but used a lot less actions.
On the hard puzzle it did surprisingly well and had the lowest running time and number of actions. When we checked how many neighbors on average it went through before finding out the inspected value was not possible, the result was 11

out of 20 which means it didn’t waste that much time, and checking those neighbors was made an improvement to both running time and the number of actions.
On the “hard for backtracking” puzzle it performed terribly. It did improve the number of actions by more than times 2 compared to the regular backtracking, but since it iterated through so many values and had to check potentially all the neighbors each time, it wasted a lot of time and ended up with the worst running time of more than 4.5 minutes.

Our conclusion from the results is that using the uninformed backtracking algorithm is a good method. It performed well on the boards from the newspaper and could have been improved with shuffling the possible values for the “hard for backtracking board” which was specifically designed to fail him with big values on the first tiles.
In addition, the Forward-Checking and the Arc-Consistency algorithms are not very reliable: Arc mostly doesn’t make significant changes in the domains and wastes time on trying to reducing their sizes. The forward checking algorithm is not effective in a small search space where checking violations is wasting more time than encountering mistakes on the run itself.
The CSP heuristic are a great way to reduce the number of actions but the computation time make the running time usually worse than the backtracking algorithm.

The second goal of this project was to figure out whether Simulated Annealing was a good method for solving Sudoku.
Our conclusion from the results is that it's definitely not: Although the running times of the successful attempts were quite low, the actual success were rare most of the times - only 3% successes for the hard puzzle.