Sudoku Solver

AI Final Project

Maria Tseytlin, Bar Mualem, Adi Brock, Tehilah Nagar

The Hebrew University of Jerusalem

Introduction

Sudoku is a logic-based number-placement puzzle. The standard sudoku grid consists of a 9x9 grid, broken into 9 squares of 3x3. Each row, column and square must contain all the numbers between 1 and 9.
Each sudoku puzzle starts with a few given values that are used as clues to fill in the rest of the numbers. The amount of numbers given determine the difficulty of the game.

There is a version of the game where the grid can be a 16x16 and also a version where the objective is to avoid collisions in the two main diagonals in addition to the usual constraints.
In this project we will attempt to solve a Sudoku by various methods and compare their performance.

Popularity of Sudoku

Modern Sudoku appeared in Japan already in 1979 but only in 2004 became known around the world, when it was published in an American newspaper.
The game has gained lots of popularity all around the world since then and appears often in newspapers and online.

Sudoku has been popular among mathematicians and has been studied frequently by them. The sudoku problem seem to be an NP-complete problem for a n^2xn^2 puzzles, though when n=3, the problem is easily solvable by the computer.

AI Solution Methods

The most naïve way to solve Sudoku is by brute force. This means trying every possible option of assignment (while checking for collisions) until the solution is reached. The problem with this approach is that it is very inefficient. In the worst case it takes over 600 million steps to solve a 9X9 puzzle. In order to solve the problem more efficiently we can incorporate AI algorithms.

There are two reasonable AI methods to solve a Sudoku puzzle by a computer:
By modeling it as a CSP problem.
By using a local search algorithm.

CSP

A Constraint Satisfaction Problem is a model of defining a specific kind of problems. Every CSP model is defined by three things:
1) a list of variables.
2) a domain for each variable e.g. possible values which a variable can receive.
3) a list of constraints between variables – usually given by a function since the number of constraints might be considerably big.

Solving CSP problems is usually done by backtracking, which can be improved by using different heuristics. For example, these heuristics help us find inevitable failures early and also decide what value to assign to which variable.

Local Search

Local search is a method which is mainly applied when trying to solve computationally hard problems. As opposed to standard search methods (such as BFS and A*), local search does not care about the path to the solution, but only cares about finding the solution itself. Local search algorithms move from state to state in the space of states by applying local changes, trying to get closer to the solution each time.
There are many different local search algorithms. The most naïve one is the hill climbing algorithm. Hill climbing is an iterative algorithm that starts with an arbitrary complete assignment, then attempts to find a better state by making an incremental change and does so until finds the solution or until it gets stuck in a local maximum (minimum).
The problem with hill climbing is that we don’t allow any bad moves and therefore we can get stuck quite quickly, depending on the states space.
Other algorithms involve some degree of probability. We will still look for better successors while allowing random moves. This way we can escape regions in the search space that don't lead to the solution.

Goals of Our Project

Sudoku is a classical problem to be modeled as a CSP. The cells of the board can be seen as variables, the numbers 1-9 are the domain and neighboring cells constrain each other, thus it makes a lot of sense to model the problem this way. A Sudoku problem is solved when a value has been assigned to each variable without violating the constraints. The search process for assigning a valid value to a variable is significantly sped up by narrowing down the domain of each variable.

In our project we have two main goals:

1. We intend to examine combinations of different heuristics for the **backtracking algorithm** and check which of them yields the best results. We are looking to solve the problem with the least backtracks possible. We will measure on a 9x9 grid in two levels of difficulty and also on a 16x16 grid.

2. Afterwards we will try our luck finding a solution using a **local search** algorithm. Solving sudoku this way is not an easy task. The states space is huge; hence we might get stuck very quickly without finding the solution. It is hard to compare the results of the CSP algorithms with the results of the local search, since we measure number of backtracks in the former while we measure number of steps (and the time) in the latter. We still want to see how far we can get by using this method, which is less intuitive for sudoku, yet still possible.

Search Algorithms and Heuristics Employed

Backtracking

Backtracking is an algorithm that finds the solution to a computational problem by determining all the possible paths to the next state, then testing them one by one for validity by continuing to determine further next states, until either reaching a goal state or failing to produce a valid next state. Upon failing, the algorithm backtracks through the tree and recursively repeats the process again. The performance of backtracking is improved by utilizing the forward checking and arc consistency strategy heuristics.

Backtracking is an algorithm which can be used for brute force, thus securing a solution, since it goes through all the possibilities. In sudoku, though, there is a huge amount of possible states.
To overcome this problem, we implemented heuristics which improve the performance of the algorithm drastically:

- Choosing the next variable:
  1. **By order**: the standard way is choosing an arbitrary order which is decided beforehand. For example, we can go through each row, from left to right.
  2. **Minimum remaining values**: the next assigned variable will be the one which has the least possible values left for assignment.

- In which order should we try the values:
  1. **Random order**: after choosing a cell, we randomly choose a value from the possible values left.
  2. **Least constraining value**: we choose the value which rules out the fewest values in the remaining variables.

- Detecting failures early:
  Sometimes a certain decision can cause us to fail in solving the problem and can only be detected after many steps. These heuristics help us detect those decisions early and avoid making them:

  1. **Forward checking**: a heuristic which is meant to detect failures early. We keep track of the possible values that remain for each variable. When we assign a value to some variable, we update all its neighbors. If one of them has no remaining possible values, we know we reached a dead end and have to reassign.

  2. **Arc consistency**: this heuristic checks if the neighbors of an assigned variable are consistent with the assignment. For any assigned value, we go through its neighbors and remove the new assigned value from their possible values. It does the same for any new variable that is left with only one possible value, until a failure is detected i.e there is a variable with no possible values, or until all the necessary variables are checked.

Local Search

To solve sudoku using local search, we first filled in each row randomly, so that it contains all the numbers 1-9. This way we ensured we have all the numbers needed for a solution.
The utility function we used is one that counts the number of collisions in the board. The puzzle is solved when the number is 0.
In order to improve the current state, we swapped between two random numbers in a random row. This way we made sure that the numbers don't change but rather change their place.
We used two different local search algorithms in our project:

1. **Random-restart hill climbing**:

   Using standard hill-climbing, we could not hope for solving the puzzle. As we mentioned before, the state space is huge and therefore even the first step can be fatal and lead us very quickly to a local minimum, without being able to escape it.
   Therefore, we tried the random-restart version of it, where we start again from the beginning upon getting stuck in a local minimum (maximum).

2. **Stochastic beam search**:

   We used a kind of stochastic beam search for solving sudoku. In local beam search, instead of producing one successor, we produce 7 different successors randomly. The algorithm tries to improve each one of them separately. The advantage is that we are more spread out over the state space. In addition, for each member in the beam, we decide in each step to take its best successor with a probability of 0.75 and otherwise choose a random successor, which doesn't necessarily improve it. The stochastic element helps us escape local minima, which are unavoidable.

Results

The following graphs demonstrate the number of backtracks that resulted in each combination of heuristics for each of the boards we used.

Abbreviations used for the different heuristics:
AC - arc consistency
FC - forward checking
BO - by order
RV - random value
MRV - minimum remaining value
LCV - least constraining value

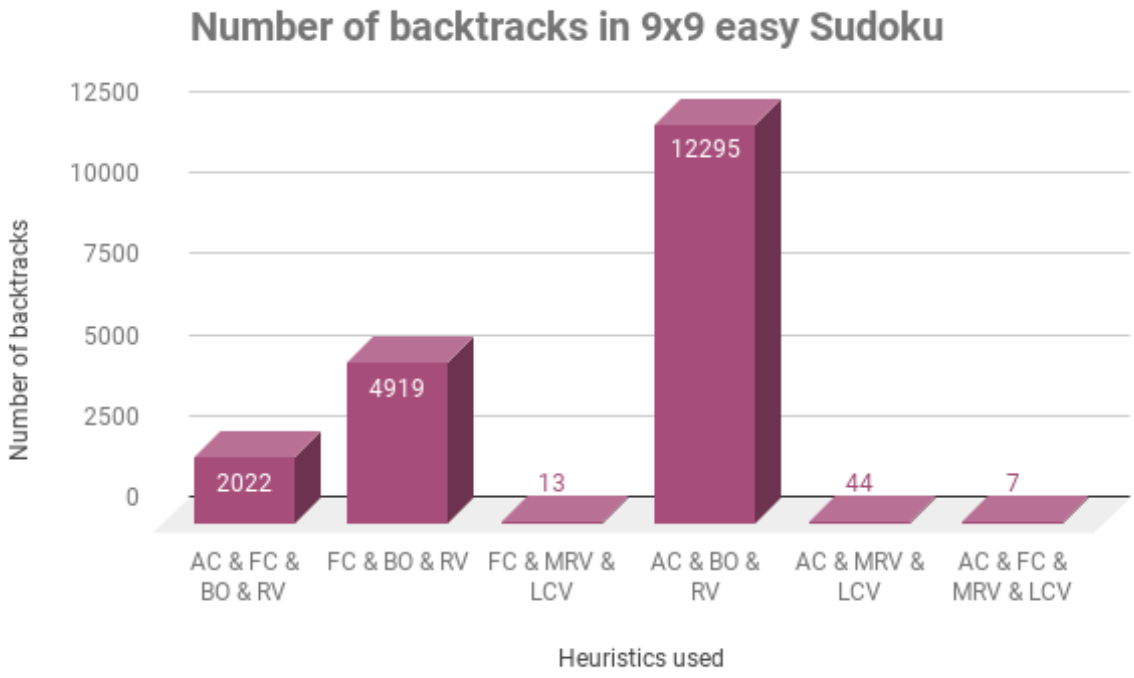

Fig.1. comparing heuristics in an easy 9x9 sudoku.

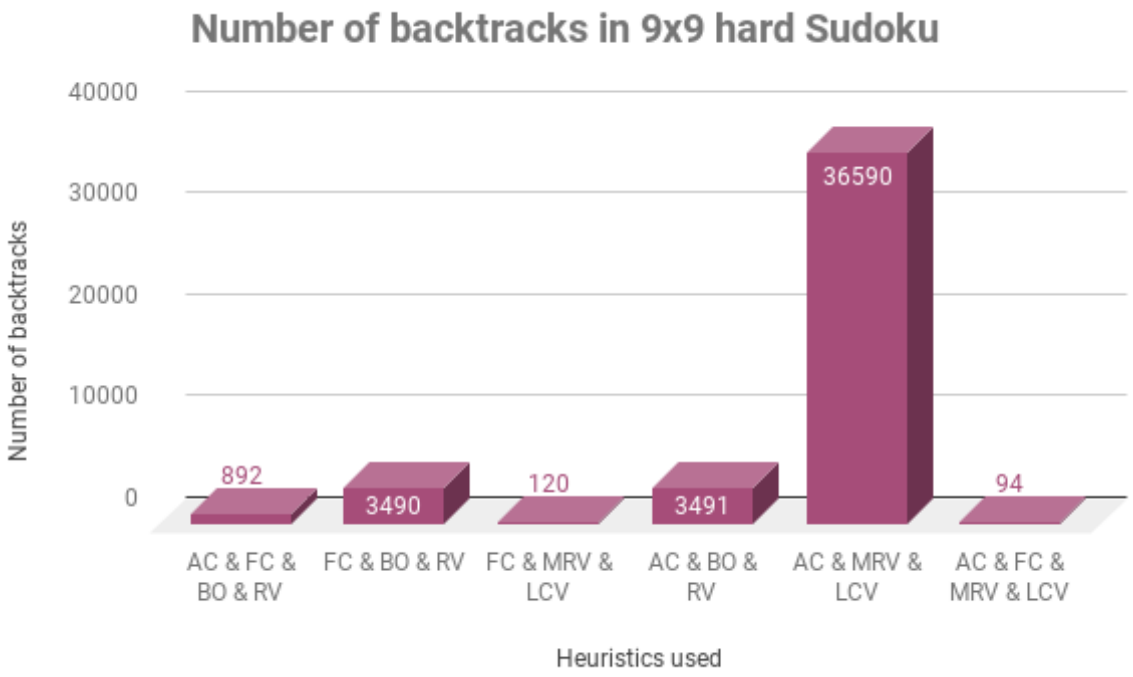

Fig.2. comparing heuristics in a hard 9x9 sudoku.

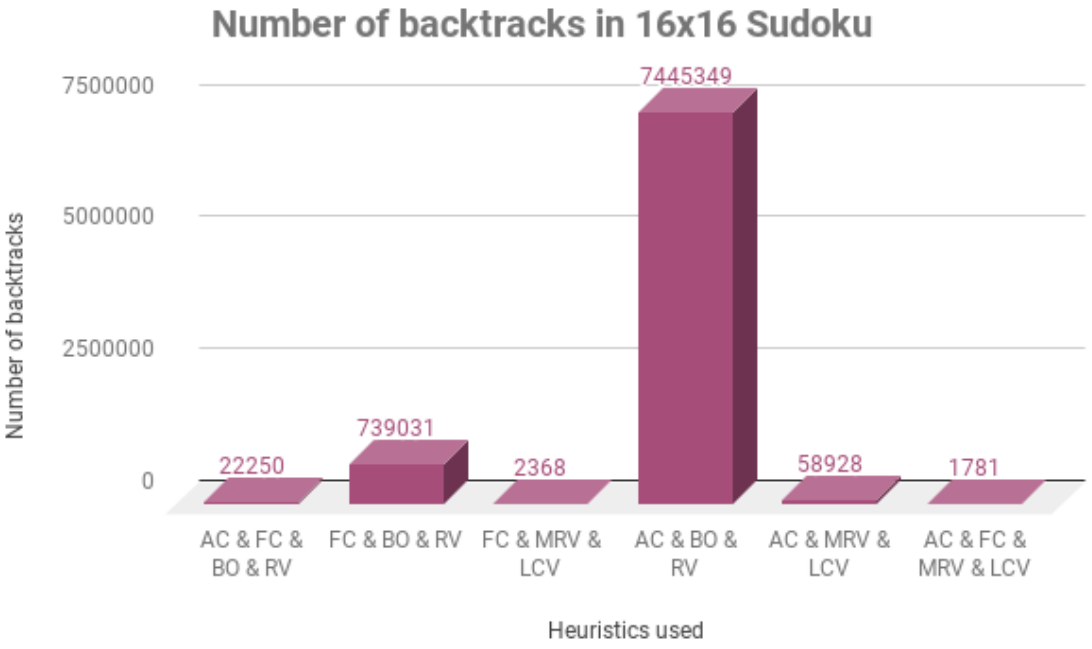

Fig.3. comparing heuristics in a 16x16 sudoku.

As we can see in all the graphs, the best combination that yields the smallest number of backtracks, is where we apply all the heuristics together: arc consistency, forward checking, minimum remaining values and least constraining value.
It is of no surprise, since we detect failures very early combining both of the failure detecting heuristics. Moreover, the other two heuristics help us choose better variables and values, which prevent failures.

We measured the time it takes to solve the puzzle using stochastic beam search. We ran it several times – and it took an average of 1 hour to finish on the easy 9x9 sudoku, which is reasonable considering the problem.

Conclusions

- After running our program with different parameters, we concluded that using all the heuristics together yields the best result. It is of no surprise, as combining forward checking and arc consistency helps us to detect failure much more quickly than using each one separately. Thus, the program has to backtrack much less.

- We were surprised to discover that forward checking always performed better than arc consistency, even though arc consistency is supposed to detect failures earlier. The reason can be related to our specific implementation or to the problem itself.

- In figures 1 and 3, the combination of heuristics that performed the worst is arc consistency, by order and random value. The second worst is forward checking, by order and random value. We can conclude that using heuristics for choosing variables and values improve the results drastically. On the contrary, on figure 2, the combination that preformed the worst is arc consistency, MRV and least constraining value. The result was 10 times worse than the second one. This is an anomaly that we cannot explain. It might be related to the amount of given numbers, but we are not sure.

- The random restart hill climbing algorithm solved the puzzle only once, but mostly got to 2-4 collisions. We gave it at most 1000 restarts before giving

up. We assume that there are many approximate solutions to the problem while there is one (or very few) full solution. For example, there are approximately 135 different boards that contain 2 collisions. This is due to the fact that we only swap numbers situated in the same row. This is over 100 times more possibilities than the solution itself. Usually when we get to 2 collisions we are very far from the solution and get stuck in a local minimum. Getting from there to the solution may require many changes, even though it seems we are close.
The first swap strongly affects our chance of finding the solution, because of the specific structure of the state space. In some other types of problems, this algorithm might yield better results.

- The stochastic beam search algorithm worked slightly better for the problem. It managed to solve the problem most of the times, though it took quite a long time. There is of course a trade-off between computation time and the ability to solve the problem. We kept track of 7 different threads in the same state space, thus spreading out over it and increasing our chance of reaching the solution. This made the algorithm slower, as we had to make all the calculations 7 times for each iteration. The stochastic element enables us to avoid getting stuck in local minima, so that we don't need to randomly restart the search. It allows to make "bad moves" which are not possible in standard hill climbing, for example.
  There might be a problem, if the 7 generated successors of the start state are concentrated in a small region of the state space, therefore lacking diversity.

Sources

Mainly course presentations, Wikipedia and our brains ;)

Running the code

In order to run the program, you will need to navigate to the file Sudoku/solve. First, run Makefile in the command line.

Then, if you would like to use CSP to solve the sudoku, run the command line -

Sudoku CSP <sudoku_name> <heuristics_combination>

where *sudoku_name* is either

- easy – the easy 9x9 sudoku
- hard – the hard 9x9 sudoku
- 16 – the 16x16 sudoku

heuristics_combinations can be

- 1 - Arc consistency, Forward checking, By order, Random value
- 2 - Forward checking, By order, Random value
- 3 - Forward checking, MRV, Least constraining value
- 4 - Arc consistency, By order, Random value
- 5 - Arc consistency, MRV, Least constraining value
- 6 - Arc consistency, Forward checking, MRV, Least constraining value

If you would like to use Local search on the easy 9x9 sudoku, run the command-

For stochastic beam search - *Sudoku LCL beam*
For random-restart hill climbing - *Sudoku LCL RR*

Examples for running the code:

- Running the problem using CSP with the heuristic combination of Forward Checking, Arc consistency, MRV, Least constraining value on the hard 9x9 sudoku:

  Sudoku CSP hard 6

- Running the problem using CSP with the heuristic combination of Forward Checking, By order, Random Value on the 16x16 sudoku:

  Sudoku CSP 16 2

- Running the problem using Local Search – stochastic beam:
  (Local search always runs on the easy 9x9 sudoku)

  Sudoku LCL beam