Project 2 – Artificial Intelligence

Arnar Bragi Bragason
Haraldur Jón Friðriksson

Introduction

For this project we decided to implement combinations of algorithms and heuristics to solve Constraint Satisfaction Problems (CSPs) in forms of a sudoku puzzle.

Sudoku is a number-based logic puzzle that is played on a square grid consisting of several sub-grids. Each cell in the grid must be filled with a number, typically from 1 to the size of the grid. The objective of the puzzle is to fill every cell in the grid with a number, while ensuring that each row, column, and sub-grid contains each number exactly once.

To put it another way, you need to fill the grid with numbers in a way that meets the following requirements:

1. Each row of the grid must contain each number exactly once.
2. Each column of the grid must contain each number exactly once.
3. Each sub-grid (also known as a "box") of the grid must contain each number exactly once.

The size of the grid and the sub-grids can vary, a 9x9 is the most common Sudoku puzzle but we test grids of size 4x4 and 16x16 as well.

To solve the boards we used backtracking as the main component in our search algorithms and implemented different versions to answer these questions:

- How does the solver's performance change as the difficulty of the sudoku puzzle increases?

- How does the choice of backtracking algorithm affect the solver's performance?

- How does the choice of variable selection heuristic (MRV, degree, and random) affect the solver's performance?

Methods

Now we will break down the methods we use to answer each question:

How does the solver's performance change as the difficulty of the sudoku puzzle increases?

To be able to test each solver's performance on different difficulties we decided it would be best to generate our own boards to do testing on, that way we can have more control of our testing and be more consistent. The boards start off as a full and valid sudoku board created by a backtracking algorithm and then we remove the numbers from the board, the amount of numbers we remove depends on the difficulty of the puzzle.
After studying top ranking sudoku websites we found that the amount of revealed numbers were usually around:

- Easy = 46%
- Medium = 39%
- Hard = 30%

For example in a medium game of 9x9 sudoku, you would get 81 * 0.39 = 31.59, which would be rounded to 32 hints for that puzzle.
Of course this is not some standard in the sudoku world but instead a solid idea of difficulty for our testing purposes.

How does the choice of backtracking algorithm affect the solver’s performance?

To answer this question we needed to implement different version‘s of the backtracking algorithm. We made:

- Backtracking (Brute)
- Backtracking
- Backtracking with forward checking

The difference in code for these algorithms is not signifcant and usually just a change of a couple of lines.
All of these algorithms go from the first box (top left) all the way to the last in the opposite corner filling the board along the way.

What makes the Backtracking (Brute) different from the others is that it tries to put every number within the size of the board into a square (if needed) and backtracks to the last one if there are none safe. It has no regard to the domains of the squares and just tries everything which eventually leads to a completed puzzle.

The other two algorithms, Backtracking and Backtracking with forward checking use a different approach when placing numbers into the squares. They both utilize the domains of the variables (squares) to try out numbers. This reduces the amount of numbers each square has to try out which is great, but to counter that upside the downside is needing to maintain the domains. Every time these algorithms choose a number from a given domain and place in the board they have to remove that number from other domains in the same row, column and box, which slows down the backtracking a lot compared to the brute version.

The Backtracking with forward checking algorithm however has one extra difference compared to the other two which is obviously the ability to forward check. We didn´t want make this version even slower by checking each square of the board for an empty domain when placing into the board. Instead when we reduce the domains we check if the domain being reduced has more than 1 (len(domain) > 1) number in it‘s domain, if not then the domain we are trying to reduce is becoming 0. If a domain becomes 0 then the last placement in the board is not valid since every square will have to be filled for a complete sudoku puzzle. In this case we backtrack right away in contrast to the Backtracking algorithm, where it would blindly keep filling the board until it stumbles upon the empty domain.

With these different versions of the backtracking algorithm in place we could test each one with different size boards and difficulties to test their performance.

How does the choice of variable selection heuristic (MRV, degree, and random) affect the solver's performance?

After implementing the algortihms we could add heuristics and further test their performance. We made separate functions for the different combinations of algorithm and heuristic.

To select the MRV (mininum remaining value) in a given state we created our own priority queue which holds nodes with the variables y-cord, x-cord and MRV value (aka length of domain). Our queue is built as a linked list with a head, tail and next pointer. When we insert into the queue we place the node where it will keep the queue sorted by MRV value. In our backtracking algorithms that use this heuristic we simply pop the front of the queue to consistently get the square with the lowest MRV value possible. To keep the queue updated we created a copy of the reduce domains function where we also update the queue of any changes.

The next variable selection heuristic we implemented added on to the MRV. To decide between squares that all had the same MRV we added a degree heuristic to choose the one that was most constrained. This addition would go through a list of the nodes and get their degree. To calculate their degree we counted all the empty squares that were in the same row, column or box. In the end we would have the square that had the MRV and highest degree.

The last variable selection heuristic we implemented chose random squares to place in. We call a set_up_search_rand function that puts tuples of all cells in a list called rand_queue. The search function then randomly selects a cell from that rand_queue and starts trying to place numbers in it. Everytime a cell is selected it is removed from the rand_queue. When the rand_queue is empty we double-check if there are any cells that the function missed by calling the set_up function again. If the rand_queue is still empty after the check, then the sudoku is finished.

Results

To answer our questions from before we had to find a way to test the algorithms and heuristics in a fair way. We decided it be to be best to generate 100 boards in each (size x difficulty) so each algorithm would be on a level playing field. To save time and ourselves from staring at a screen waiting for results we put a 30 second max run time on each test so in our code you will see something like:

```
elapsed_time = time.time() - self.start_time

if elapsed_time >= 30:
    return True, expansions
```

Our results would have been different if we had let each test run its course but in some cases it could have taken many hours. So in our results 30 second average run time will be your maximum.
To answer our questions we had to decide what combinations of algorithms and heuristics we would need to do so, here is what we implemented.

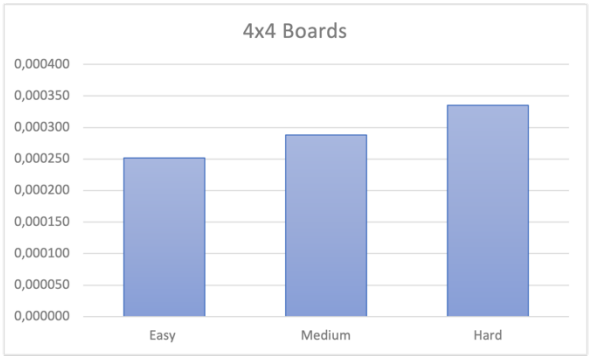
- Backtracking (Brute)
- Backtracking
- Backtracking With Forward Check
- Backtracking Using MRV Heuristic
- Backtracking With Forward Check Using MRV Heuristic
- Backtracking Using MRV & Degree Heuristics
- Backtracking With Forward Check Using MRV & Degree Heuristics
- Backtracking Using Random Heuristic

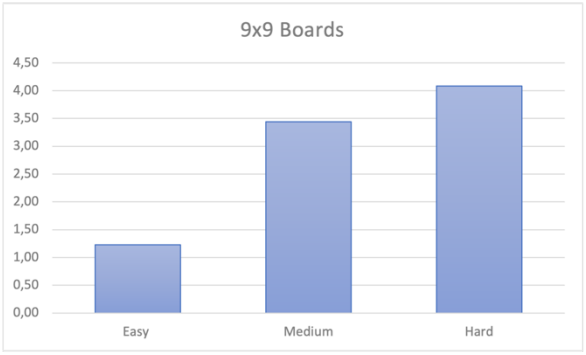
We ran the tests generated on each algorithm and documented the average of all the tests taken in an excel sheet. Every algorithm ran 100 tests for 4x4 and most for 9x9 but for 16x16 we had to cut it to the first 20 boards. If each test had ran for 30 seconds on each algorithm in 16x16 it would have taken many days.

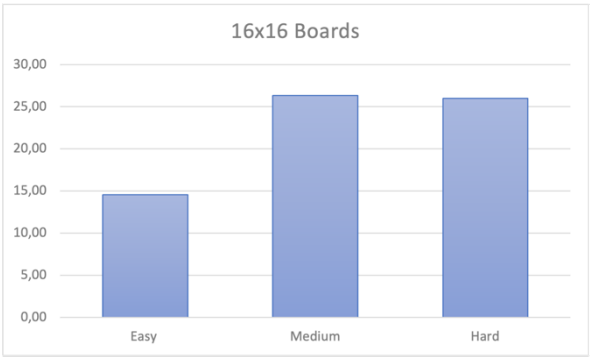
After getting the results we could go back to our original questions and answer them.

How does the solver's performance change as the difficulty of the sudoku puzzle increases?

In our excel sheet we can clearly see that in most cases as the difficulty increases the average search run-time does as well. I think it´s safe to say the reason for this is simply because there are more possibilities. When you have more blank squares then most domains are going to be larger than if there were more constraints from the start, resulting in a longer run-time.
Here are simple bar charts taking average of the average run-times so you can see this visually:







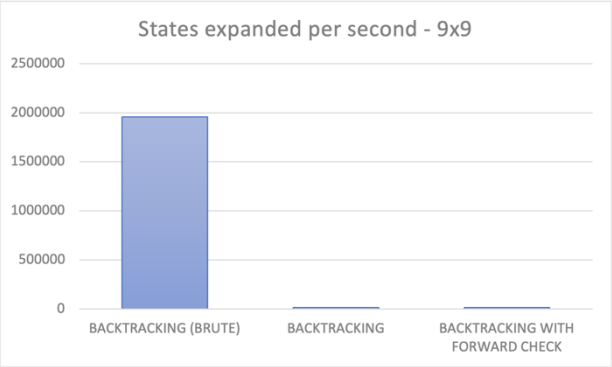
The difference in the amount of blank squares in 4x4 isn‘t a lot as our formula from before results to 7, 6 and 5 hints for those boards, which translates well in the bar chart. However, the results of the other two are not as easy to explain. We are still confused as to why hard was sometimes faster than medium in 16x16 but our guess is that there might be a certain threshold where more blank spaces in a board of that size makes it easier for some algorithms to operate. But to conclude this question, the difficulty more often than not has an effect on a solver‘s performance.

How does the choice of backtracking algorithm affect the solver's performance?

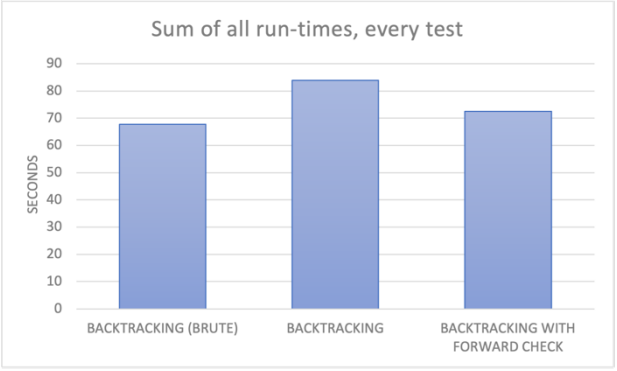
This will be decided by the algorithms without any heuristics as we will go into that in our next question.
We will be taking a look at:

- Backtracking (Brute)
- Backtracking
- Backtracking With Forward Check

The Backtracking Brute algorithm was the fastest, it has the lowest average run-time at each difficulty at every single size. To understand why it performed so well compared to the other two we can take a look at the states expanded per second stat in these tests. The algorithm is just way faster at running through the states and finding a solution, it doesn´t stop to think for a second and just tries everything. It often went into the millions while the other two maxed out at 60 thousand.
Here is a pretty funny bar graph for average states expanded per second in the 9x9 tests:



The downfall of the other algorithms is the amount of time spent taking care of the domains. This doesn´t mean the other two are horrible solutions, they just might need some heuristics to make looking after the domains worth while.
Backtracking With Forward Check was often very close to being as fast as Brute but could never beat it. However plain Backtracking was by far the worst and was slowest in every test. Here is a chart summing all average run-times which pretty much shadows what I just said:



So to conclude this question the choice of backtracking algorithm does indeed have a great effect on the performance. Even if we take Brute out of the scope and focus on the other two then we can see having forward checking is superior and a better choice of backtracking algorithm for our project.

How does the choice of variable selection heuristic (MRV, degree, and random) affect the solver's performance?

Now we will be looking more closely at how these algorithms performed:

- Backtracking Using MRV Heuristic
- Backtracking With Forward Check Using MRV Heuristic
- Backtracking Using MRV & Degree Heuristics
- Backtracking With Forward Check Using MRV & Degree Heuristics
- Backtracking Using Random Heuristic

Let‘s start off by comparing results between using MRV and using no heuristic at all (the ones discussed in the previous question).
First thing we noticed was that using MRV led to fewer states being expanded, this is likely because the square it chooses to put in usually has a small domain containing it´s solution.
But is it faster to use MRV?
Simple answer is yes, but it‘s not really faster until after the 9x9 medium tests. In the easier, smaller puzzles they perform about the same but MRV really shines in the harder tests.
Instead of making another bar chart here are some run-time averages (in seconds) from our excel sheet:

	9X9 HARD	16X16 EASY	16X16 MEDIUM	16X16 HARD
BACKTRACKING	1,792	23,937	28,059	30
BACKTRACKING W FC	0,537	19,043	25,871	27,041
BACKTRACKING MRV	0,0248	6,845	22,614	21,342
BACKTRACKING W FC MRV	0,0220	5,670	22,164	20,312

As we discussed before plain Backtracking performs worst and Backtracking with forward check somewhat better. But it is safe to say that using MRV is far superior than no heuristic at all. A brief look at these averages and you can see that using MRV is way faster, so to use it for our project should be a no-brainer.

Now that we have covered MRV we can go into the degree heuristic. Does the addition of the degree heuristic improve our run-time?

Let‘s start with looking at the backtracking using MRV & degree algorithm. With a quick look at our results in the excel sheet we see that isn´t faster than backtracking using only MRV in any tests.
If we examine the one with forward check we see that it as well doesn´t seem to be able to improve from using only MRV. In the tests these algorithms perform about the same but in the 16x16 tests we see that the ones using degree do significantly worse in most cases.

Why isn‘t the addition of degree better?
Intuitively, the degree heuristic may seem like it should perform better than the MRV heuristic alone. However, the additional computational overhead required to compute the degree of each variable for each iteration can actually slow down the algorithm‘s overall performance.
It‘s also possible that the Sudoku boards we generated have relatively simple constraints (since we removed random values) and that the additional overhead of the degree heuristic is not offset by any improvement in performance.

The last heuristic we implemented was random. We did not have high hopes for this one and the results were just as we expected. In the 4x4 boards it was somewhere in the middle of all the algorithms regarding average run-time. But the problems started as soon as we touched 9x9 boards, taking on average 9 seconds to finish the easy boards. After that it‘s average run-time is the max in each test. The explanation behind these results is the fact that random heuristics are not effective at handling complex problems with large search spaces and numerous constraints. The larger the problem, the more likely the random heuristic is to take a long time to find a solution, if it finds one at all. This is because a random heuristic simply guesses values without considering any problem-specific information.

So to wrap up the results.
How does the choice of heuristic effect the performance of a solver? It can have a huge impact on performance if the right one is chosen, for our project using only MRV is clearly best.

Conclusion

In this project we implemented combinations of algorithms and heuristics to solve Constraint Satisfaction Problems (CSPs) in forms of a sudoku puzzle. To summarize this report we will briefly answer our questions once again.

How does the solver's performance change as the difficulty of the sudoku puzzle increases?
In most cases increasing difficulty leads to higher average run-time, lowering its performance.

How does the choice of backtracking algorithm affect the solver's performance?
For our project some algorithms are clearly faster than others and perform better so choosing the right one is crucial.

How does the choice of variable selection heuristic (MRV, degree, and random) affect the solver's performance?
As with the algorithms themselves some heuristics clearly perform worse than others. Choosing the right one will make the performance better.

In the end our choice for the best algorithm is the backtracking with forward check using MRV (only).

There is still potential for future work in this project, particularly in developing more sophisticated heuristics that can handle more complex constraints. Additionally, there may be opportunities to improve the implementation of existing heuristics to make them more efficient and reduce run-times.