Háskólinn í Reykjavík

Reykjavik University

Spring 2015

Artificial Intelligence

T-622-ARTI

Teacher: Stephan Schiffel

Final Project

Nilli solves Sudoku

anagram: Nilli loves Sudokus

:: NerðirMeðSkapgerðir ::

Ægir Már Jónsson - aegir13@ru.is

Arnar Freyr Bjarnason - arnarb12@ru.is

Atli Sævar Guðmundsson - atlisg12@ru.is

Stefanía Bergljót Stefánsdóttir - stefania13@ru.is

Dags: 7. apríl. 2015

0. Introduction

Our project was to implement an agent that solves Sudoku puzzles because we were interested in implementing a solution to a constraint satisfaction problem. As with all our agents this semester, he was named Nilli. Sudoku is interesting since it is a simple and straightforward example of a constraint satisfaction problem (CSP) but is also very popular. If solving regular 9x9 Sudokus would be too simple, we wanted to take it a step further and try it on bigger Sudokus and make him as efficient as possible. That would require implementing strong heuristics and mixing search methods and CSP methods in a smart way. The purpose of the project was to learn more about CSP, work with it first hand, and get to know it on a personal level. The first goals we set for Nilli were: Solve a regular sized Sudoku (9x9) under 0.1 sec and a larger sized Sudoku (16x16) in a reasonable amount of time (preferably under a minute).

1. What did we do?

1.1. Implementation

1.1.1 Brute force

The first version we implemented was a brute force search with backtracking. The board was represented with a 2-dimensional int array. The program went linearly through the board, put a number in a square and then checked if it the assignment was valid (that number not in that row, column or section) and backtracked if it was not. That worked surprisingly well for a regular sized sudoku (9x9), but was really slow for a larger board (16x16).

1.1.2 Domains

The next thing we did was to keep track of the domain for each space. We created a class Variable to keep track of this. Each variable had a domain that started as {1,2,3...n}. Before starting the search we restricted the domain for each variable according to the starting assignments. Every time we made an assignment during the search, we removed the assigned value from the domain of the connected variables. This meant that whenever we made a new assignment all the values in the domain were valid assignments and therefore we did not have to check if the new board was valid. Modifying the domains was faster than checking the validity of each assignment, so this version was slightly faster than the brute force version. For each variable we represented the domains using an array of integers. The value 1 represented that the number vas in the domain, a lower value meant that it was not. Whenever a variable V was given an assignment N, we restricted the domains by lowering the N-th value by 1 in the array for each of V's neighbours (neighbours of V are defined as the variables in the same row, column or section as V). Then the search was conducted recursively and finally the N-th values were increased by 1. This way we did not need to make a copy of the array or the variables at each level, which meant that the amount of memory used by the recursion was fairly low.

1.1.3 Variable heuristics

We used a variable heuristic where we looked at each unassigned variable and chose the one with the smallest domain to be the next one to assign a number to. We experimented with different methods of ordering the variables, including using priority queues, but we ended up simply going through the array, examining them, and keeping track of which variable had the smallest domain so far. This feature turned out to be very useful and improved the time and state expansion a lot.

1.1.4 Value heuristics

We experimented with different ways of ordering the values when assigning a value to a variable. We found out that the order makes a big difference in the number of state expansions, sometimes changing it by a factor of 10 or larger. However we could not come up with a heuristic that significantly and consistently improved the solver. The heuristic included in the final solution simply counts the number of empty spaces where the given value would be a valid assignment.

1.1.5 Inference

Before the backtracking-search for the solution starts, the solver tries to solve the board using simple inference rules. It looks at all the spaces: for any space where only one value is possible, it assigns that value to it. It also checks if there is a number that can only be placed in one space in some line, column or box, and assigns the number to that space. We also tried to combine search and inference: when the search depth had reached a certain level, we created a new board from the current one, performed inference on it and then continued the search on the new board. However, though the number of state expansions was significantly reduced, the total time needed to solve the puzzles did not improve, so we did not include this method in the final version. This makes sense because the inference method we are using is not very deep, but the search method, with the variable heuristic and the domains, is fairly strong. If we had implemented a more complex inference method, maybe this sort of mixed method would have worked better. Late in the process we implemented a better inference step that looks for both spaces with only one possible number, and numbers with only one possible space. This methods turned out to be powerful and managed to solve many of the boards without resorting to search.

1.1.6 Other

One of the weaknesses of our solution is that it focuses on the fact that two neighbouring spaces cannot have the same value, but does not utilise the fact that each number must appear at least once in each row, column and box. For example, if there is a row where only one space can possibly have the value 1, that space can immediately be assigned the value of 1. A decent human Sudoku solver would easily spot this, but our solver does not, since it only focuses on the what values are possible for the spaces. We tried implementing this by using two sets of variables: spaces whose values are numbers, and numbers whose values are spaces, but we did not manage to do it efficiently. A solver that successfully used a method like that would be very efficient because it could, at each level of the recursion, both

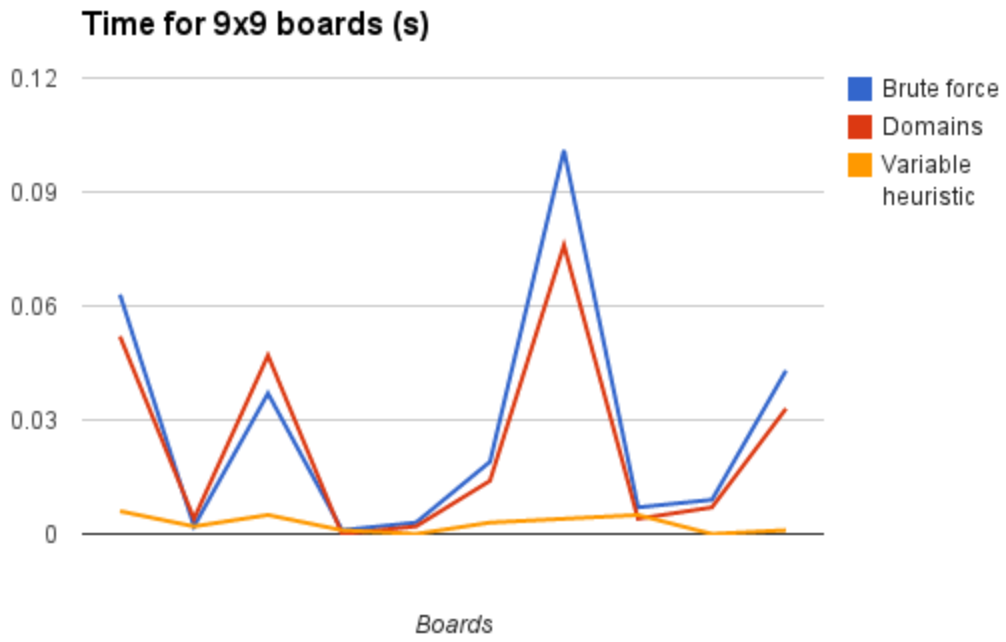
consider the spaces and the numbers in their domain, and the numbers and their spaces in their domain, so it could make a better decision about what assignment to make next.

1.2. Testing

Initially, we were going to make a sudoku generator, so that we would not have to hard code the test boards, but we had some problems with generating the larger boards, so we decided it was easier to just hard code the test boards and run them on our solver, Nilli, for testing.

2. Results

In the final version of the solver, we implemented a features that could be turned on and off with parameters. Those were **variable heuristics,** which turned out to be absolutely essential, **value heuristics** which turned out to be not that helpful, or maybe we did not figure out a good one, **simple inference,** which makes a few simple inferences based on the domains of the spaces before the search, and **extended inference** that works similarly to the previous one but also searches for values that have only one possible space to be placed in.



Graph 1. Solution times for 9x9 boards

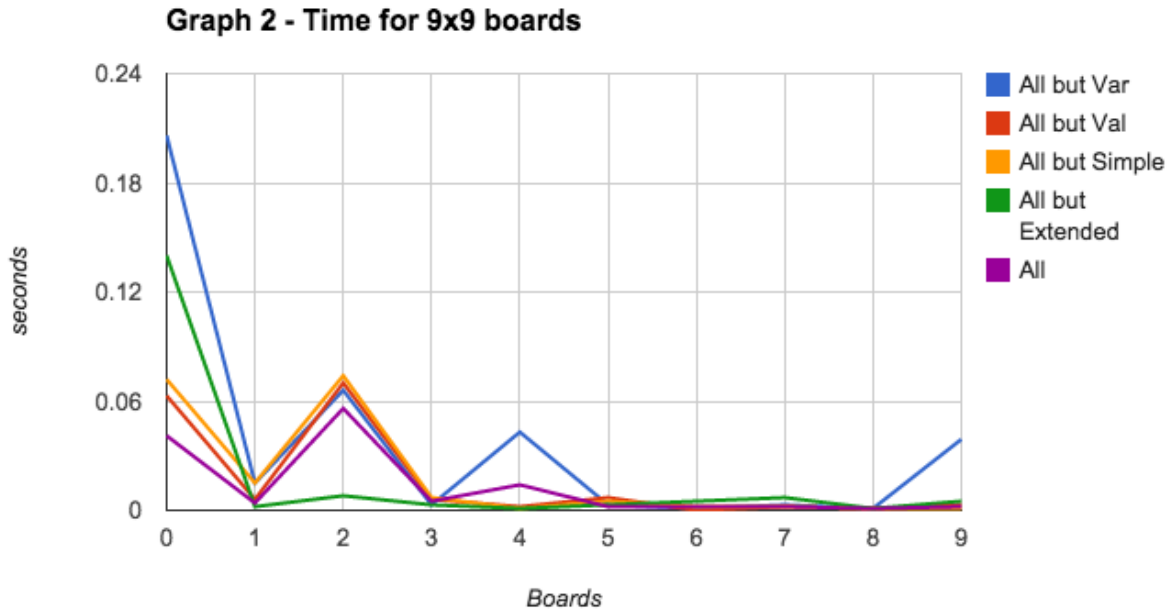
This graph compares three versions of the solver: A brute force versions that chooses variables and values without a heuristic, and checks if an assignment is valid afterwards, a version with domains that uses no heuristic but keeps track of each variables domain and

therefore only chooses valid values, and a version that uses domains and a variable heuristic so that the variable with the smallest domain is chosen.
As can be seen from the graph there is a small improvement in using domains, and a very large improvement in using a variable heuristic.
The table below shows the the number of state expansions for the same boards

Board number	Brute force / Domain	Variable heuristic
0	21628	323
1	1671	194
2	43489	1053
3	168	317
4	2844	56
5	27482	856
6	129967	715
7	7758	998
8	13175	82
9	68448	233

Table n. State expansions for 9x9 boards using different methods

One interesting thing was the difference between going immediately into the search and performing an initial inference. For simpler boards, though the inference significantly reduces the number of state expansions, the difference in time was not significant. For the more difficult boards however, making the initial inference was a huge improvement. For example, for one board (16x16, nr. 2) using the inference reduced the time from 10.3 sec. to 0.25 sec. The number of states went from 553087 to 1231. Two of the most difficult boards could not be solved within a reasonable amount of time without the inference.



Graph 2. Solution times for 9x9 boards using different heuristics
As we can see, the different heuristics did not make much of a difference between them when solving the regular sized boards, because the time taken to solve the boards is very little.

At the very last moment before turning the project in, we realized that mixing the simple inference step (where we assign values to spaces whose domain's size is one) and the extended inference step (where we assign numbers to spaces if there is only one valid space for it) before we start the search might get better results. This turned out to be true. We modified it so that when the extended inference parameter is on, we take turns running the extended and the simple inference until no more assignments have been made. This way we could solve 9 of the 16x16 boards using only inference and no search. When we made this breakthrough we did not have enough time to make new charts, so it is not included in the data above but the following table shows the results using the newest version. We were impressed to find out that board 7 could be solved using only inference, since it is a very difficult puzzle and was time consuming for the other versions of the solver. Puzzle 7 can be seen below.

Board	Time (s)	States
0	0.336	212
1	0.057	1
2	0.071	1

3	0.035	1
4	0.001	1
5	0.008	1
6	0.004	1
7	0.038	1
8	96.614	7023315
9	0.054	3469
10	0.013	223
11	0.008	1
12	0.13	1

Table 2. Time and state expansions for the new mixed inference version. (s.findSolution(true, false, true, true), uses var heuristic and both inferences)

	16	8	2			6	3				13				
10			9	1			8		16	3			15		
12		11	1										8		
	14		6					9							
					5		10								16
1	11	13		7		8					10			14	
		12		3	1	14			8		9	10		13	
4				9	2	15	16						3	7	
			11						2	10	12	4	5		
6								7	9		16				
							5		3	6				16	12
		3	5				14	4		13	5		7	9	
8		10								4		15	13		7
			16		14						6	2		4	1
						3	4			15		8	12	11	
	13		7					2					16		9

Figure 1. Board 7 (16x16)

We expected Nilli to be able to solve a regular sized sudoku in under 0.1 second and he managed to do that. We weren't quite sure about what to expect with the larger Sudoku, because it has a much bigger state space, but we hoped Nilli to be able to solve the hardest 16x16 puzzles, in a reasonable amount of time, which he did. In the final version, only one puzzle took him more than a second to solve.

3. Conclusion

This project allowed us to experiment with different heuristics, inference methods and other things to make the solver as efficient as possible. Thinking of new features, implementing them and then testing them to see if they were any good made this project different from the other projects we have done but overall it was a fun and educational experience.

The utility of the different features matched our expectations in most ways, except the value heuristic which turned out to not be very useful. The most important ones turned out to be the variable heuristic and the inference step before the search.

While doing this project we spent some time thinking about the search and inference and how they can be used for problems like Sudoku. A purely search-based approach, like our brute-force version is easy to make but inefficient, while a purely inference-based version would probably be very difficult to implement. Therefore a good Sudoku solving program would probably use a mixed approach. Our solver ended up being mostly search-based, the only inference it really does is the initial assignments, and keeping track of the domains, and choosing the variables with the smallest domains, but the performance difference between the brute-force version and the more sophisticated versions shows that those features are important.
Nilli met his goal of solving the hardest regular sized sudoku puzzles in under 0.1 seconds but it did take him more than a minute to solve the hardest larger puzzle. However, he was able to solve 11 test cases in under 0.1 secs, which was very pleasing.

4. Future work

There are many areas where our solver could be improved that would be interesting to explore in the future. For example, we still don't know if there exists a strong value heuristic that works for most Sudoku puzzles. Also, since we only came up with a strong inference step close to the deadline, we did not have time to experiment with it or try to integrate it with the search process. It would be interesting to study more complex inference methods for CSPs and try to find something that works well for Sudoku, and try to develop a stronger solver that uses the best of both search and inference.

5. References

- We used the Stopwatch.java file from algs4.edu for measuring time.
- Sudokus used for testing taken from
    - The daily sudoku: http://www.dailysudoku.com
    - Sudoku weekly: http://www.sudokuweekly.com
    - Sudoku-puzzles: http://sudoku-puzzles.merschat.com