Report practicum 2

Sunny Hsieh, Abel de Groote, Mika Sie

6534856, 6748236, 6603726

Findings

Chronological backtracking

In this report we describe our findings during practicum 2. In this practicum we programmed a chronological backtracking algorithm both with and without dynamic forward checking. Our problem domain was a sudoku existing of 9 by 9 squares, grouped in tiles of 3 by 3 squares. In every row and column each number from 1-9 can only be found once, conforming to the rules of sudoku. The task was to program a chronological backtracking algorithm both with and without dynamic forward checking.
The algorithm without dynamic forward checking works in the following way:
1. Go to the next box that has a non-fixed value.
2. Go through the values 1-9 until it finds a value that is considered safe according to the sudoku rules.
3. Go to step one and if at any point you have no safe values you go back to the last step 2 and try the next value which is considered safe.
This algorithm uses recursion to test all safe values and if none are found it goes to the last box. where it tries a value and tries a different one. It will eventually find a solution if there is one.

Chronological backtracking with forward checking

What is dynamic forward checking:
Dynamic forward checking is a technique used to decrease the search space for a constraint satisfaction problem. It goes over all elements with a variable value and for each element it checks if any values can be eliminated from the list of possible variable values that the element can take. It does this by checking if the value is consistent with the problem or with the constraints given.

Dynamic forward checking in our problem goes through all the elements with a variable value after these have been initialised and eliminates the values that the element cannot take due to the given initialization. This means that our chronological backtracking algorithm has less possible values to try whenever it goes through the possibilities, thus lowering the search space and sequentially the search time of the algorithm. The downside of dynamic forward checking is that it takes more time to initialize the algorithm before it can run as dynamic forward checking needs to go through all the possible values of every element with a variable value to see if they are consistent with the problem.

This is also where the difference in run time comes from between chronological backtracking with and without dynamic forward checking. One takes more time to search through the search space, the other takes less time to search through the search space but takes more time up front to decrease the size of the search space. We created these two algorithms in C#. The following is the main core on how we created the algorithm of backtracking with forward checking.

Core of our chronological backtracking with forward checking

We will focus here on our implementation of deleting domains, since this was the most difficult part, and also the part that contributes the most to the speed of solving the sudoku. We made a list for every box in the sudoku and we matched the 9 by 9 sudoku square to the numbers 0 to 80 as shown, this list keeps track of the possible values that the box can take (If the box has a value already, then the list contains just the number 0), in other words: the list keeps track of the domain of that box.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

The number can be calculated with the formula: row * 9 + col and is from now on referred to as that box's place.
Where row is the row number and col is the column number both ranging from 0 to 8.
As an example, if we have row = 2 and col = 4 we get 2 * 9 + 4 = 22.
As shown in the image we find that the row and col give the place 22.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

To delete or restore values in the domain of a box we have two methods.

The first method is DeleteDomain, which is given a box and a value.
Whenever any value is removed from a domain it marks the domain as being changed.
The rest of DeleteDomain is divided into three parts: row, column and block.

For the row:
Given a box and a value it goes through every domain that is further right in the row.
Every box for which it's place + 1 < (place/9 + 1) * 9 DeleteDomain goes through the box's domain and removes the value given if it is there. This ensures that if the value given is in a box's domain further right in the row it is removed. For example, given the box with place 22, the boxes with place 23,24,25 and 26 will be checked.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

For the column:
Given a box and a value it goes through every domain that is further down in the column. We take a variable a and set it to 9 as we only want to go through the boxes further down and thus never the box on the first row. Then while the place + a < 81 we go through the domain of the box with index place + a and then it adds 9 to a to go to the next box in the column. Same as for the row, when we go through the domain of a box we remove the given value if it is there. This ensures that if the value given is in a box's domain lower in the column it is removed. For example, given the box with place 22, the boxes with place 31,30,49,58,67 and 76 will be checked.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

For the block:
Given a box and a value it goes through every domain that is further right or down in the same 3 by 3 block. First the 3 by 3 block that the given box is in is determined by the two variables rowStart and colStart. These are calculated through row - row % 3 and col - col % 3 for rowStart and colStart respectively. This gives us the coordinate of the leftmost box in the 3 by 3 block that the given box is located in.
We then go through every box in that 3 by 3 block and check if it is further right or further down than the given box. This is done through seeing if the place of the to-check box is higher than the place of the given box. The place of the to-check box is calculated by it's row * 9 + it's column. If the place of the box is higher than the place of the given box we remove the value given if it is there. This ensures that if the value given is in a box's domain in the same block if the place is higher than the place of the given box. For example, given the box with place 40, the boxes with place 41, 48, 49 and 50 will be checked.

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

Since for a given box at this point both the row and column have already been checked. The boxes that are to the right and further down, if any, have already been checked if they contain the value and the value has been removed accordingly. So these are being checked twice, though this has very little effect on the performance.

The second method is RestoreDomain which is given a value and a list of boxes who have had the value from it's domain removed through DeleteDomain.
This method basically reverts the removal of values in DeleteDomain which is used after the algorithm was unable to find a solution given the values selected so far and thus needed to undo a value selection earlier.
This is done by going through the list of boxes and adding the value to their domain and sorting the domain.

Results

Chronological backtracking

Chronological backtracking	
Grid number	Average runtime (msec)
1	506,3
2	504,4
3	527,8
4	509,2
5	509,2

Table 1: Average runtime results for chronological backtracking

With chronological backtracking the solution was found very fast. The local search algorithm we previously made found a solution in a 20 second range. This algorithm will find it in about half a second which is a huge upgrade. As we can see the different grids do have some difference in their runtimes although these are very small. The biggest difference is between grid 1 and grid 3. This difference is only 21,5 milliseconds. In our previous report we found that grid 3 was probably the hardest one to solve and this algorithm proves the same.
We ran each grid 10 times in our algorithm and then calculated the average runtime in Excel. (All tables can be found in the syllabus of course.) By doing each grid multiple times our results would be more reliable and more precise.

Chronological backtracking with forward checking

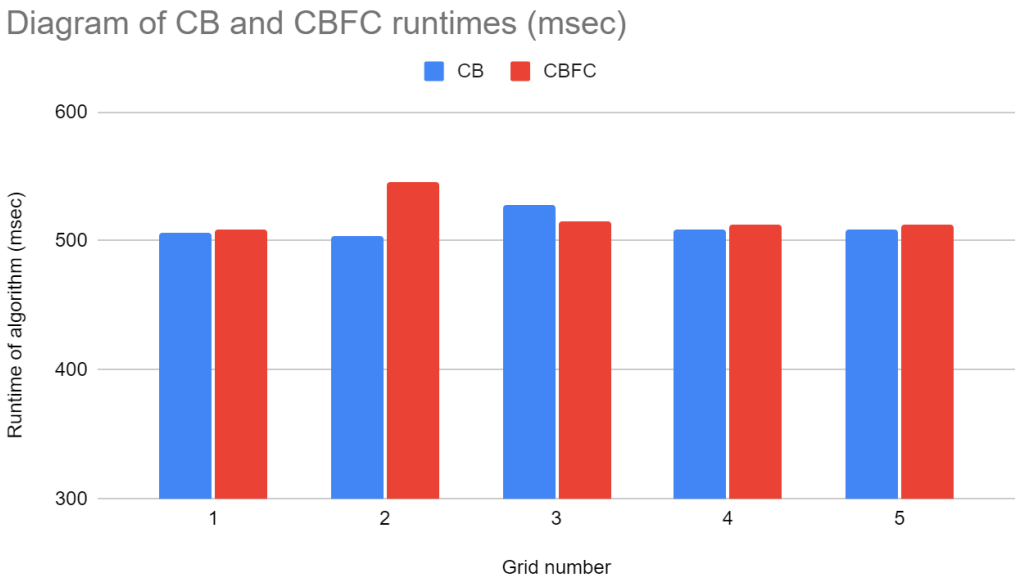
Chronological backtracking with forward checking	
Grid number	Average runtime (msec)
1	509,1
2	546,2
3	514,7
4	513,1
5	513,1

Table 1: Average runtime results for chronological backtracking with forward checking

The way of running chronological backtracking with forward checking was done the same as just chronological backtracking, we ran every grid 10 times again. We can see that the runtime is still pretty fast. But however, the runtime is a little bit slower when we use forward chaining especially for grid 2. All grids are solved slower when we didn’t use chronological backtracking except for grid 3, which was faster.

Conclusion

We can conclude that for the sudoku problem domain chronological backtracking without forward checking is slightly faster. We are talking in the realm of ten milliseconds which is very small but there was a consistent difference. In the following diagram you can see both the algorithms compared to each other.



Graph 1: Diagram of chronological backtracking and chronological backtracking with forward checking runtimes in milliseconds

For all grids except one, chronological backtracking was faster than chronological backtracking with forward checking. This is due to the fact that most sudoku's aren't that hard to solve, so forward checking isn't necessary to make the search process faster. By forward checking all the next values it will only take the algorithm longer to solve the problem. One of the reasons that decreases the speed of backward checking with forward checking is because the assignment asked us to assign the numbers in order from 1 to 9. By restoring the domain in our code, we called the sort method when we restored a number to a list. (this takes time!) If it wasn't necessary to assign the numbers of the domain in order, we expect that it would have been faster than chronological backtracking in almost all cases (not sure about case 2). This sort method is executed very quickly but it can explain the very small difference we can see between the two algorithms.

Chronological backtracking was slower than chronological backtracking with forward checking in one scenario however. This was the case in grid number 3. We can explain this because we know from the previous assignment that grid 3 is very hard to solve, the hardest one of all the grids. In this grid the forward checking was beneficial to solving the sudoku. This way the chronological backtracking algorithm with forward checking was faster for this grid.

To conclude we can say that for sudoku solving the chronological backtracking algorithm is better across the board. Only for really hard puzzles chronological backtracking with forward checking is a better option. But oftentimes sudokus aren't really hard but there are also easier sudokus. For these easier sudokus chronological backtracking is a better algorithm. But in the end we're talking about very tiny differences in runtime. If you know your problem set will contain all types of sudokus, easy to very hard, chronological backtracking is a much

better option. If you get a problem set which only contains very hard puzzles, it might be worth it to opt for chronological backtracking with forward checking.

Reflection on programming

The implementation of chronological backtracking went fast, it was written in around 1-2 hours. We started with processing the input: making a 2D array which represented the board (we have reused the code from assignment 1 here). But the modelling process took the longest. This is how we modelled the algorithm and what we had to do step by step. This was our thought process on creating the chronological backtracking algorithm:

1) Go through the board, left to right, up to down, and locate the first 0 we see. Remember this location. (If no 0's are found, it means that the board is solved)
2) Try to assign a number which is safe (need a method that checks if it is safe within the row, column and block), from 1 to 9.
3) If it's safe, assign the number to this place and go on a recursion. If there is no possible outcome for this assignment, we backtrack and assign the next possible number to it.

It took a few minutes about how to check if it appears within a block, but after modelling it, writing the code went very fast without struggles.

The implementation of chronological backtracking with forward checking took us a few days (around a week). First it was unclear how to start but after attending the seminar, the TA explained to us the concept of this algorithm. After this seminar we started modelling and our modelling process went like this:

1) Make an initial list for every location on the board, if the value of the location is 0, use the isSafe method we wrote for chronological backtracking to determine which values are possible for that location. If the value of that location is not 0 (so the value is fixed), the list just contains 0 (initially it was an empty list but we ran into some errors, so we fixed it by making the list containing just a 0).
2) After making the initial list, we go through almost the same steps as chronological backtracking: locate the first 0 we see on the board (if we don't see any 0's on the board means the board is solved).
3) We use the list of this location, (we call the possible numbers the domain of this location) and assign the first number of this domain (the domain is ordered from small to large).
4) We assign the number to this location and we delete this number from the domains that are in it's area. By area we mean: same row, same column, same block.
5) We go on a recursion. If there is no possible outcome for this assignment, we restore this number in the domains we just deleted it from, and try the next possible number.

The challenge was how to locate the domains and how to delete the domains (which is explained in detail before). This was for us the biggest struggle. However we still managed to finish the code 2 weeks before the deadline, so that we could write a detailed paper about our implementation.

Syllabus

Chronological backtracking

Grid 1	
Run number	**CPU time (msec)**
1	508
2	504
3	509
4	502
5	516
6	503
7	504
8	512
9	502
10	503

Grid 2	
Run number	**CPU time (msec)**
1	502
2	501
3	504
4	504
5	506
6	512
7	502
8	505
9	506
10	502

Grid 3	
Run number	**CPU time (msec)**
1	527
2	531
3	528
4	535
5	532
6	522

7	526
8	530
9	528
10	519

Grid 4	
Run number	**CPU time (msec)**
1	508
2	504
3	515
4	517
5	503
6	505
7	510
8	515
9	502
10	513

Grid 5	
Run number	**CPU time (msec)**
1	515
2	503
3	507
4	511
5	511
6	507
7	515
8	510
9	501
10	508

Chronological backtracking with forward checking

Grid 1	
Run number	**CPU time (msec)**
1	519
2	501

3	509
4	507
5	511
6	506
7	516
8	512
9	512
10	513

Grid 2	
Run number	**CPU time (msec)**
1	520
2	503
3	509
4	502
5	502
6	515
7	508
8	510
9	512
10	510

Grid 3	
Run number	**CPU time (msec)**
1	549
2	530
3	540
4	525
5	542
6	590
7	577
8	539
9	525
10	545

Grid 4	
Run number	**CPU time (msec)**
1	521
2	513
3	514
4	518
5	517
6	514
7	508
8	515
9	514
10	513

Grid 5	
Run number	**CPU time (msec)**
1	544
2	515
3	511
4	512
5	510
6	509
7	508
8	507
9	507
10	508