Simple-CSP Solver

T-768-SMAI, Search Methods in AI, 2023-08

Reykjavik University - School of Computer Science, Menntavegur 1, 102 Reykjavík, Iceland

Francesco Moschella francesco23@ru.is
McCord Murray dmurray3@umassd.edu

30 August 2023

1 Our Solution

In this assignment, we developed and tested the efficiency of three CSP-solving algorithms to solve Sudoku[1]. The core idea behind these algorithms is to test in an intelligent way the available configurations and find the most suitable to solve the problem. Of course, in our case, since a well-formed sudoku puzzle has only one solution, we had to find the only valid solution. The first algorithm is Chronological Backtracking, the simplest algorithm of the three. The second algorithm is Backjumping Search, an improvement of the Chronological Backtracking algorithm. The third algorithm is Conflict-directed Backjumping, a further improvement of Backjumping Search. Additionally, we implemented the Make Arc-Consistent-3 algorithm which optimizes the domains of each variable based on known constraints. The performances of all three algorithms were observed with arc-consistent and non-arc-consistent input domains.

1.1 Chronological Backtracking

The idea of this algorithm is that at each iteration, a partial configuration is created for the first "n" variables. The algorithm, after checking if the partial configuration is valid or violates any constraint up to that variable, can either backtrack to the previously assigned value and change the "n^{th}" value to look for a valid configuration up to that "n^{th}" variable or further move deep into the state-space search tree, and check for the "$n^{th} + 1$" value. While it's more efficient than randomly guessing, it can certainly be improved.

[1]Our solution source code, along with the other related files are available in the GitHub repository `https://github.com/HarlockOfficial/T-768-SMAI-Informed-Search-Methods-in-AI`

1.2 Backjumping Search

This algorithm is an improvement of the Chronological Backtracking algorithm which backtracks as far back as it can without missing any potential solutions. In other words, unlike the Chronological Backtracking algorithm, this algorithm can backtrack further up the state-space search tree. This leads to a lower amount of nodes visited and can help in finding an optimal solution in less computational time.

1.3 Conflict-direted Backjumping search

This algorithm further improves the concept behind the Backjumping Search, by also considering the conflicts detected by the child branches of the state-space search tree. Taking into account these data allows the search to be even faster and make more extended jumps, thus backtracking in a more efficient and effective way and visiting even fewer amount of nodes. Of course, as a side note, we have to remark that this algorithm pays the higher flexibility and "intelligence" with an amount of memory that is used to contain the conflict set, memory that was not necessary for the other two algorithms.

1.4 Arc-Consistency (AC-3)

The purpose of this algorithm is to optimise the domains of each variable. Allowing for a faster search in the state-space search tree and for a more rapid computation. The idea behind it is to reduce the available values that can be assigned to a specific parameter. Lowering the amount of possible values effectively reduces the computation time, making the aforesaid algorithms even faster. Although various implementations of the Make Arc-Consistent algorithm have been defined, in this project we are going to use the third version: Make Arc-Consistency 3 (AC-3), which has shown to have the best time complexity in the average case.

2 Benchmark

A total of 20 Sudoku problems were used as a benchmark to evaluate our results. In particular, for each algorithm, we will be evaluating the total nodes searched, the running time, and whether or not the algorithm found a solution. For each algorithm, we will also provide the result obtained when the arc consistency algorithm was applied and when it was not[2].

2.1 Efficency

Since the Conflict-directed Backjumping was created as an evolution of the Chronological Backjumping and that the latter was created as an evolution of the Backtracking, we expect this order to be reflected in their overall performances. Therefore, we expect the Conflict-directed Backjumping to be the more performant, in terms of efficiency (more efficient), amount of visited nodes (lower amount of visited nodes) and computation speed (faster computation and computation requiring lower time), with respect to the other two. Finally, we expect all algorithms to perform even better when the domain is Arc-Consistent.

For the specific data, we will refer to the Appendix section.

[2]The extensive results, along with other tests performed and their related plots, are available here: `https://osf.io/f49xq/?view_only=4ea1c2e14270468a98e4d473f92cf282`

2.1.1 Without Arc Consistency

In this section, we are going to present the overall results and performances for each of the three algorithms. The results have been computed without performing the Arc-Consistency step, therefore all the algorithms had to undergo a larger search (compared to the search performed when the domains were Arc-Consistent) where a huge amount of unnecessary nodes had to be visited. Just to avoid repetition in each section, all the algorithms successfully found the only valid solution to all the 20 sudoku puzzles that were part of the benchmark file.

Chronological Backtracking The performances of this algorithm were good, but not optimal, of course, the other two algorithms had lower runtimes and better performances in terms of visited nodes. For example, with this algorithm 12.6 million nodes were visited to find the last puzzle solution.

Backjumping Search In this case, the performances were better than the previously presented algorithm. Indeed, with respect to the Chronological Backtracking algorithm, in many cases, a number of nodes even less than half have been visited. Also, the overall runtime was far better, with puzzle computations requiring on average less time.

Conflict-directed Backjumping search In this last algorithm, as expected, on average, the results were the best among the described algorithms. This algorithm improved, in our opinion even with interesting results, the number of visited nodes and the search execution time.

Discussion In this paragraph, we want to show how the used algorithms impacted the amount of nodes visited and the average execution time, we will perform this comparison by using some data from the actual results. To assess the amount of nodes visited, we will use puzzle number 19, the last and most complicated one in the benchmark set. In this sudoku, the number of visited nodes fell from around 12.6 million in the Backtracking algorithm to 6.5 million in the Backjumping one, and finally to 1.7 million with the Backjumping. A similar relation is visible in the execution times, this value dropped significantly, with the puzzle computation requiring more than 3 seconds two times when the Backtracking algorithm was applied, and not even 0.3 seconds with the Conflict-directed Backjumping algorithm.

2.1.2 With Arc Consistency

In this section, we are going to present the overall results and performances for each of the three algorithms. The results have been computed after performing the Arc-Consistency step, therefore all the algorithms had to undergo a smaller search (compared to the search performed when the domains were not Arc-Consistent) where a low amount of nodes had to be visited. Just to avoid repetition in each section, all the algorithms successfully found the only valid solution to all the 20 sudoku puzzles that were part of the benchmark file. For the sake of remaining as much as possible within the page limitation, we avoid providing an extensive explanation for this part, we just want to remark that after performing the Arc-Consistency algorithm on the input domain, even the Chronological Backtracking, our least efficient algorithm visited no more than 1900 nodes per puzzle. This is evidently a good optimisation, especially when compared to a max of over 12.6 million nodes visited without it. Of course, the other algorithms reflect the same behaviour, with the Backjumping being more efficient than the Chronological backtracking (in terms of both spent time and visited nodes) and with the Conflict-directed Backjumping being the most efficient among the three algorithms. In the Appendix section, we will present the actual results obtained.

2.2 Potential Improvements

While developing the algorithms we thought of some improvements that may be applied to the implemented algorithms to simplify or speed up the state-space search. First of all, parallelization. It would be more efficient to search subtrees in parallel than to search the entire tree. For example, if the first variable has a domain with 8 elements and we have access to 4 processors, one processor could explore each of those subtrees with the CBJ algorithm. Then if one subtree is explored entirely, that processor starts exploring the next subtree. Then, we can think about similar or identical subtrees, an algorithm can save the results coming from specific subtrees, this approach, although will use some more memory, may improve the overall computation when we are in the presence of large trees. For example, if the algorithm stores that a specific subtree has an inconsistency, later, when exploring other nodes that are not related to the inconsistency, it will be possible to completely avoid the specific subtree, lowering the computational cost by paying with some memory. Another improvement that is applicable in a problem like sudoku, where the order in which the cells are filled is not important, can be ordering the cells by domain size, from the smallest domain to the biggest, when the domain has the same size, the order should follow the number of constraints that the cells have to adhere, from the largest number of constraint to the lowest, this way the tree will initially have an almost linear path, and then the convergence will be faster because the nodes with most rules will allow to rapidly cut the path to the correct solution.

A Configuration

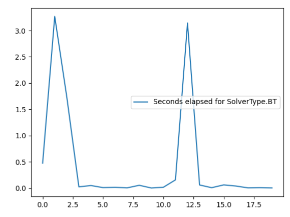
The tests have been performed on a Linux Mint Virtual Machine. All the detailed data are presented in the following listing, the file has been obtained by performing the bash command "inxi -Fxxxzr", some non-important data have been trimmed out of the file and have been exchanged with "...".

```
System:
  Kernel: 5.4.0-159-generic x86_64 bits: 64 compiler: gcc v: 9.4.0
  Desktop: Cinnamon 5.0.7 wm: muffin 5.0.2 dm: LightDM 1.30.0
  Distro: Linux Mint 20.2 Uma base: Ubuntu 20.04 focal
Machine:
  Type: Virtualbox System: innotek product: VirtualBox v: 1.2
  serial: <filter> Chassis: Oracle Corporation type: 1 serial: <filter>
  Mobo: Oracle model: VirtualBox v: 1.2 serial: <filter> BIOS: innotek
  v: VirtualBox date: 12/01/2006
Battery:
  ...
CPU:
  Topology: Quad Core model: Intel Core i7-10750H bits: 64 type: MCP
  arch: N/A L2 cache: 12.0 MiB
  flags: lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 bogomips: 11428
  Speed: 1429 MHz min/max: N/A Core speeds (MHz): 1: 1429 2: 1429 3: 1429
  4: 1429
Graphics:
  Device-1: VMware SVGA II Adapter driver: vmwgfx v: 2.15.0.0
  bus ID: 00:02.0 chip ID: 15ad:0405
  Display: x11 server: X.Org 1.20.13 driver: vmware
  unloaded: fbdev,modesetting,vesa resolution: 1920x967~60Hz
  OpenGL: renderer: SVGA3D; build v: 3.3 Mesa 21.2.6 direct render: Yes
Audio:
  ...
```

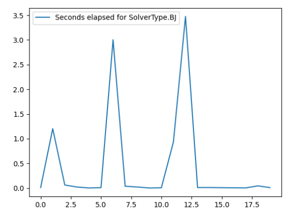
```
Network:
  ...
Drives:
  Local Storage: total: 35.00 GiB used: 26.59 GiB (76.0%)
  ID-1: /dev/sda vendor: VirtualBox model: VBOX HARDDISK size: 35.00 GiB
  speed: 3.0 Gb/s serial: <filter> rev: 1.0 scheme: MBR
Partition:
  ID-1: / size: 33.78 GiB used: 26.59 GiB (78.7%) fs: ext4 dev: /dev/sda5
Sensors:
  ...
Repos:
  ...
Info:
  Processes: 218 Uptime: 3h 00m Memory: 7.67 GiB used: 4.64 GiB (60.5%)
  Init: systemd v: 245 runlevel: 5 Compilers: gcc: 9.4.0 alt: 9 Shell: bash
  v: 5.0.17 running in: gnome-terminal inxi: 3.0.38
```

B Running Time

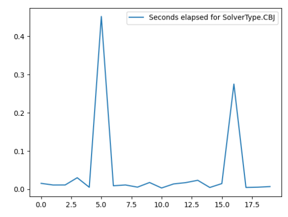
In this section, we will present the plots demonstrating time per puzzle for the benchmark puzzles. Actually, since the graphs show some peaks due to the machine usage, the small microseconds-related variations are slightly visible. But is possible to visualise the difference in the scale, with the BT having two values that are above 3.0 seconds, and the CBJ not reaching 0.5. Unfortunately, due to some issues with the laptop hosting the virtual machines we have some spikes in the BJ graph. We consider those values as outliers and think that running the tests again in a different environment or in a more stable desktop or server machine may produce more reliable results.



(a) Seconds elapsed for the BT Solver

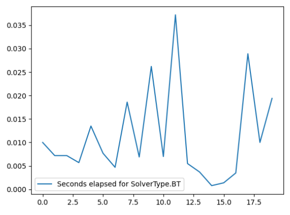


(b) Seconds elapsed for the BJ Solver

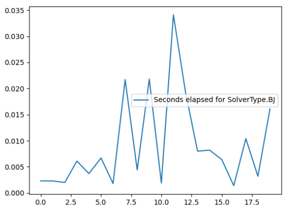


(c) Seconds elapsed for the CBJ Solver

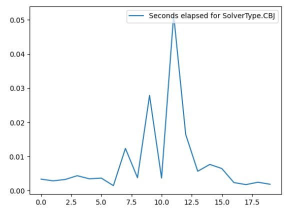
Figure 1: Run times in seconds (without arc consistency)



(a) Seconds elapsed for the BT Solver



(b) Seconds elapsed for the BJ Solver

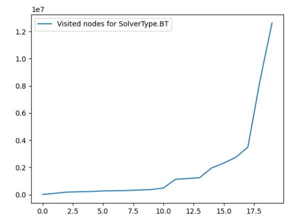


(c) Seconds elapsed for the CBJ Solver

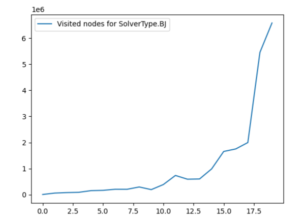
Figure 2: Run times in seconds (with arc consistency)

C Nodes Visited

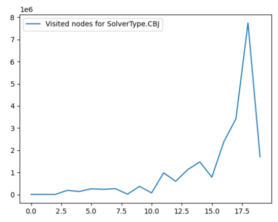
In this section, we will present the plots demonstrating nodes visited per puzzle for the benchmark puzzles.



(a) Nodes visited for the BT Solver

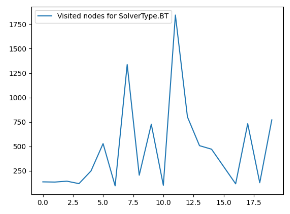


(b) Nodes visited for the BJ Solver

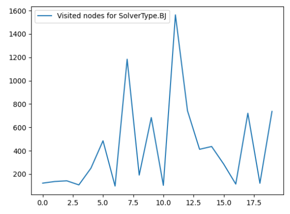


(c) Nodes visited for the CBJ Solver

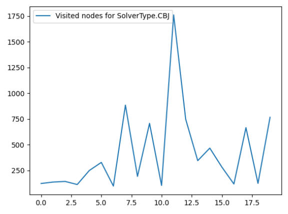
Figure 3: Puzzle Solving efficiency (without arc consistency)



(a) Nodes visited for the BT Solver



(b) Nodes visited for the BJ Solver



(c) Nodes visited for the CBJ Solver

Figure 4: Puzzle Solving efficiency (with arc consistency)

As is possible to notice, in both cases, the amount of visited nodes is significantly lower for the CBJ and BJ algorithms, with respect to the BT. Also, in the solving efficiency without arc consistency, we can notice an exponential-like curve, in our opinion, this is mainly related to the puzzle complexity rising through the test cases. In the plots presenting the results after the Arc-Consistency, we cannot appreciate the same exponential-like curve, mainly because the complexity of each problem changed, after the Arc-Consistency checks. In these graphs, we can see that in a specific case, the BJ performs better than BT and CBJ in terms of the number of visited nodes, we think that this is problem dependent and perhaps the specific puzzle was easily solvable with the aforementioned algorithm, instead of the other two.

D Results

In this section, we will list the raw data that have been obtained by the algorithms. The data are in the format[3]:

```
<Puzzle ID> <Used Agorithm> <Run time>
```

Listing 1: Results from the benchmark file (without Arc-Consistency)

```
0  SolverType.BT          10353      0.4740
1  SolverType.BT          95776      3.2690
```

[3]A more extensive version of these files can be found at the following link: `https://osf.io/f49xq/?view_only=4ea1c2e14270468a98e4d473f92cf282`

```
 2  SolverType.BT        185359     1.7352
 3  SolverType.BT        210449     0.0227
 4  SolverType.BT        228101     0.0478
 5  SolverType.BT        274481     0.0085
 6  SolverType.BT        284080     0.0138
 7  SolverType.BT        302347     0.0035
 8  SolverType.BT        339700     0.0525
 9  SolverType.BT        372342     0.0013
10  SolverType.BT        486885     0.0149
11  SolverType.BT       1123919     0.1553
12  SolverType.BT       1181915     3.1433
13  SolverType.BT       1247725     0.0583
14  SolverType.BT       1962927     0.0068
15  SolverType.BT       2320446     0.0599
16  SolverType.BT       2744247     0.0386
17  SolverType.BT       3494454     0.0037
18  SolverType.BT       8370964     0.0064
19  SolverType.BT      12632762     0.0025
 0  SolverType.BJ          6629     0.0129
 1  SolverType.BJ         60561     1.2044
 2  SolverType.BJ         78997     0.0635
 3  SolverType.BJ         90307     0.0231
 4  SolverType.BJ        153346     0.0040
 5  SolverType.BJ        165292     0.0099
 6  SolverType.BJ        206469     3.0036
 7  SolverType.BJ        206100     0.0392
 8  SolverType.BJ        294708     0.0231
 9  SolverType.BJ        193030     0.0035
10  SolverType.BJ        387858     0.0079
11  SolverType.BJ        736526     0.9350
12  SolverType.BJ        592239     3.4754
13  SolverType.BJ        602220     0.0127
14  SolverType.BJ        987279     0.0128
15  SolverType.BJ       1656166     0.0093
16  SolverType.BJ       1752927     0.0072
17  SolverType.BJ       1997680     0.0058
18  SolverType.BJ       5454935     0.0461
19  SolverType.BJ       6583895     0.0111
 0  SolverType.CBJ         6178     0.0155
 1  SolverType.CBJ         9638     0.0112
 2  SolverType.CBJ         4368     0.0113
 3  SolverType.CBJ       189761     0.0302
 4  SolverType.CBJ       140166     0.0053
 5  SolverType.CBJ       262097     0.4520
 6  SolverType.CBJ       236711     0.0092
 7  SolverType.CBJ       266971     0.0113
 8  SolverType.CBJ        19960     0.0056
 9  SolverType.CBJ       370137     0.0177
10  SolverType.CBJ        74918     0.0034
11  SolverType.CBJ       983611     0.0140
```

```
12  SolverType.CBJ      599178     0.0174
13  SolverType.CBJ     1127013     0.0235
14  SolverType.CBJ     1473791     0.0047
15  SolverType.CBJ      786157     0.0149
16  SolverType.CBJ     2391618     0.2752
17  SolverType.CBJ     3419290     0.0048
18  SolverType.CBJ     7743168     0.0055
19  SolverType.CBJ     1706872     0.0071
```

Listing 2: Results from the benchmark file (with Arc-Consistency)

```
 0  SolverType.BT           138     0.0100
 1  SolverType.BT           136     0.0072
 2  SolverType.BT           145     0.0072
 3  SolverType.BT           120     0.0057
 4  SolverType.BT           250     0.0135
 5  SolverType.BT           529     0.0077
 6  SolverType.BT            97     0.0047
 7  SolverType.BT          1337     0.0186
 8  SolverType.BT           206     0.0069
 9  SolverType.BT           727     0.0262
10  SolverType.BT           103     0.0070
11  SolverType.BT          1843     0.0372
12  SolverType.BT           802     0.0055
13  SolverType.BT           508     0.0037
14  SolverType.BT           472     0.0008
15  SolverType.BT           295     0.0014
16  SolverType.BT           118     0.0035
17  SolverType.BT           733     0.0289
18  SolverType.BT           129     0.0100
19  SolverType.BT           772     0.0194
 0  SolverType.BJ           122     0.0023
 1  SolverType.BJ           136     0.0023
 2  SolverType.BJ           142     0.0020
 3  SolverType.BJ           107     0.0061
 4  SolverType.BJ           250     0.0037
 5  SolverType.BJ           485     0.0067
 6  SolverType.BJ            97     0.0018
 7  SolverType.BJ          1184     0.0217
 8  SolverType.BJ           191     0.0044
 9  SolverType.BJ           684     0.0218
10  SolverType.BJ           103     0.0019
11  SolverType.BJ          1564     0.0341
12  SolverType.BJ           742     0.0193
13  SolverType.BJ           412     0.0080
14  SolverType.BJ           436     0.0082
15  SolverType.BJ           285     0.0064
16  SolverType.BJ           114     0.0014
17  SolverType.BJ           721     0.0104
18  SolverType.BJ           121     0.0032
19  SolverType.BJ           737     0.0161
 0  SolverType.CBJ          122     0.0034
 1  SolverType.CBJ          136     0.0029
 2  SolverType.CBJ          142     0.0033
 3  SolverType.CBJ          112      0.0044
 4  SolverType.CBJ          248      0.0035
 5  SolverType.CBJ          327      0.0037
 6  SolverType.CBJ           97      0.0015
 7  SolverType.CBJ          884      0.0124
 8  SolverType.CBJ          191      0.0038
 9  SolverType.CBJ          707      0.0279
10  SolverType.CBJ          103      0.0037
11  SolverType.CBJ         1762      0.0515
12  SolverType.CBJ          747      0.0165
13  SolverType.CBJ          344      0.0057
14  SolverType.CBJ          466      0.0077
15  SolverType.CBJ          282      0.0065
16  SolverType.CBJ          117      0.0024
17  SolverType.CBJ          665      0.0018
18  SolverType.CBJ          123      0.0025
19  SolverType.CBJ          766      0.0019
```

For completeness, we also run our algorithm with and without Arc-Consistency checks on the puzzles generated by our version of the sudoku domain and consistency checks generators. Instead of presenting the results here, we provide the link to the repository containing all our results and the related plots `https://osf.io/f49xq/?view_only=4ea1c2e14270468a98e4d473f92cf282`.