Assignment 1 – CS 686

Student id: 20696176

Name: Erlie Shang

1.1

1. Manhattan distance heuristic performs better. The reason is that both heuristics are less than the real cost. However, the Manhattan distance heuristic is always greater than or equal to the misplaced tile heuristic, which means the Manhattan distance heuristic is closer to the real cost.
2. They are both consistent.

Here’s the proof:

According to the definition, the heuristic cannot decrease by 2 or more after one move, which means the new heuristic can only decrease by 1 or stay still.

The cost for one move is 1.

Therefore, h(n) <= cost(n, n’) + h(n’) is valid for every node n.

So, both heuristics are consistent.

1.2

1. Time complexity: O(bm)

Space complexity: O(b\*m)

For this particular question, The average branching factor is 3, and the average cost (depth) is 22, so:

Time complexity: O(322)

Space complexity: O(3\*22)

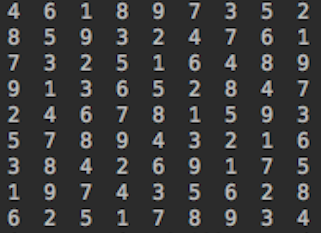
1. Yes, it’s complete. IDA\* is complete when the branching factor is finite. And for this question, the branching factor is 3, which is finite.
2. Yes, it’s optimal. IDA\* is optimal when the path cost is a non-decreasing function. And for this question, the path cost is the number of movement, which means the path cost is a non-decreasing function of the depth of the node.

2.1

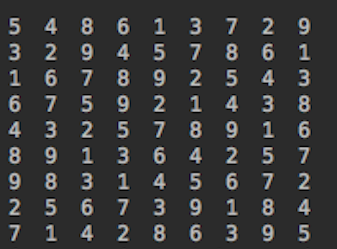
* 81 variables Vij, i = 0 to 8, j = 0 to 8
* Domain of each variable is integer from 1 to 9
* Constraints
* For all Vim, Vin, m≠n ⇒ Vim ≠ Vin
* For all Vmj, Vnj, m≠n ⇒ Vmj ≠ Vnj
* For all Vij, Vmn, (i / 3 = m / 3 ∧ j / 3 = n / 3) ⇒ Vij ≠ Vmn

2.2

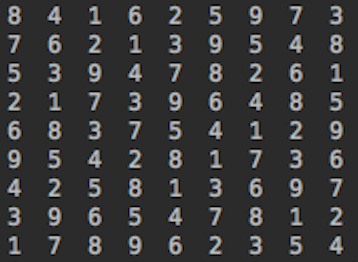
1. The code is in sudoku\_solver.py. Run this script directly, and you can see the required tables and the solution to each Sudoku in the terminal.
2. Easy



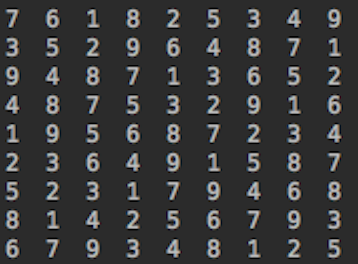
medium



hard



evil



Time (unit: ms)

|  |  |  |  |
| --- | --- | --- | --- |
|  | B | B + FC | B + FC + H |
| Easy | 35.14 ± 17.83 | 12.14 ± 6.89 | 3.98 ± 0.98 |
| Medium | 734.11 ± 544.29 | 119.16 ± 82.88 | 3.35 ± 0.67 |
| Hard | 609.71 ± 383.14 | 144.79 ± 67.38 | 3.81 ± 0.25 |
| evil | 272.31 ± 132.54 | 119.49 ± 60.47 | 7.67 ± 1.00 |

Node

|  |  |  |  |
| --- | --- | --- | --- |
|  | B | B + FC | B + FC + H |
| Easy | 967.08 ± 476.59 | 351.88 ± 209.80 | 60 ± 0 |
| Medium | 19447.26±14445.70 | 3240.40 ± 2207.71 | 53 ± 0 |
| Hard | 16714.7 ± 10373.78 | 4095.98 ± 1882.05 | 59 ± 0 |
| evil | 7649.16 ± 3635.36 | 3383.74 ± 1726.93 | 93 ± 0 |

Part (d) is on the next page.

1. Firstly, I use a 9 \* 9 matrix to denote a puzzle, and in each cell, digit 0 means this cell has not been assigned while digit 1-9 means this cell is assigned to this digit. And then, I create 3 classes for the solver with different algorithm.

As for the basic backtracking solver, I just utilize the constrains that I created in question 2.1 to implement the check method, which checks whether the current assignment violates any rule. In the construction method, I deep copy the puzzle matrix because I need the original puzzle to reinitialize the solver so that I can calculate the required table in one for loop. And, of course, I have excluded the time for deep copy by marking the start point in the solve method. The backtracking method checks whether the puzzle is solved by checking if there is any cell containing 0. If it’s solved, the result will be returned. If not, the program will pick up an unassigned cell and assign a value from 1 – 9 randomly to it, which also needs to meet the constrains. And then, start a new backtracking. I tried to pick up an unassigned cell randomly, but it will degrade the performance too much because it will pick up one cell repeatedly. Therefore, I remove this part.

As for the solver with forward checking, I generate a table to store possible candidate for each cell. Every time a cell has been assigned a value, this table will be updated. And if the current assignment does not work, the table will roll back. If there is one cell in the table containing no candidate, it means a conflict exists. The program will back track in this situation to save time. The other parts are similar to the basic one.

As for the solver with heuristic, I mainly modified the methods to find a proper unassigned cell and a proper value order, which means the find\_unassigned method returns the cell with least possible candidate and the order\_value method returns a value list ordered by the sum of possible values for all cells in the next backtracking decreasingly. These heuristics really improve the performance significantly. I also tried to add the most constraining variable heuristic, however, it will degrade the performance. So I removed this part. The other parts are similar to the solver with forward.

And then, in the main function, I use a for loop to calculate the data by running the solver 50 times and print them to the terminal.