CMPUT 657 Heuristic Search Assignment 2 Report

Talha Ibn Aziz
Student ID: 1669108

March 8, 2021

1 Theoretical Section

Section 1 describes the program and it's theories in details, where - section 1.1 gives a brief detail of the program structure, section 1.2 describes the minor enhancements of the search algorithm, section 1.3 details the different heuristic values experimented on and used, and section 1.4 explains the additional details of the program.

1.1 Program Description

The program consists of the main function and two classes - "state" and "SAS" (Single Agent Search). The main function checks for argument errors, keeps taking input until there is no input left, and calls the respective class functions. The description of the classes are mentioned in the following sections.

1.1.1 State Class (state.cpp and state.h)

The state class maintains and keeps information of the puzzle pieces, different moves, Transposition Tables, Zobrist Table, maximum limits of containers, move history, and a compressed representation of the puzzle grid (more in section 1.4). The class also has a number of functions which perform specific modifications on the state variables. The most important ones are detailed below:

- The 'set_state' and 'show_state' functions initialize the different state variables and print the current grid state on the console, respectively.
- The function 'move_range' calculates the maximum possible distance a specific piece can move in a given direction (up, down, left, or right).
- The function 'movePiece' performs the required changes to the state variables when a move is performed. The function 'undoMove' reverses the change of the most recent performed move.
- The functions 'saveTT' and 'checkTT' are used to save and check the entries of the Transposition Table.

There are a few heuristic functions which are defined in section 1.3.

1.1.2 SAS Class (SAS.cpp and SAS.h)

The Single Agent Search or SAS class contains the different search algorithm implementations and uses the state class to perform the searches. The SAS class also contains and maintains the different statistical variables of the searches. There are a few functions to initialize the class (init) and to implement the two searches - Depth First Iterative Deepening (DFID) and Iterative Deepening A-star (IDAstar). The two driver functions call the recursive functions Depth First Search (DFS) and Depth First A-star (DFAstar).

1.2 Search Enhancements:

IDAstar is used as the primary search function and there is very little enhancement added to the search. In order to avoid repetitive moves, any move performed by the same piece in the same axis (up-down or left-right) consecutively, is avoided. This is because the possible locations should already be covered by the earlier move level. For example, assume that the piece 'x' performs a two-step move towards the "up" direction in the current node. The next node must not be - 'x' performing any move towards the "up" or "down" direction. Any possible location reachable by two consecutive same-axis moves should be reachable by only one move.

The pieces are represented as a string of characters and the directions are stored using *enumeration*. Therefore, the moves are generated using three loops - outer loop for character string (piece), middle loop for direction (UP, DOWN, LEFT, RIGHT), and the innermost loop for number of steps in the selected direction. For each move performed, the state is changed accordingly and the recursive function is called. Once the recursive function returns, the "undo" action is again performed on the same state. In this way, instead of creating or duplicating states, the search uses the same state by modifying it iteratively.

1.3 Heuristic Function:

The heuristic function is a combination of two simple heuristic values. Experiments were performed using the following heuristics:

Manhattan: This well-known heuristic is calculated as - one move, if the rover is in the same row as the gate, and two moves otherwise.

Manhattan_2: This is a modified Manhattan Distance which considers all possible paths directly from the rover to the gate and calculates the minimum number of obstacles in the way as the heuristic value. Each obstacle adds a value of 1 (at least three moves to reach goal) to the original Manhattan distance heuristic value. However, including the heuristic made the program slower as calculating this value takes a lot of time.

Closed_h: This heuristic adds a value of 1 to the Manhattan heuristic if the rover is closed from all sides. If even one side is empty, it does not change the Manhattan distance. This heuristic offered little advantage as it was almost misleading when the case is difficult (more space but difficult positioning of pieces) and many cases needed to be handled as well.

Obstacle: This heuristic builds on the original Manhattan distance by considering obstacles in the shortest possible path. Manhattan heuristic assumes that there are no obstacles and calculates the shortest path to goal in the created abstract space. The 'obstacle' heuristic increases the value by 1 (at least 3 moves to reach goal) if there are any obstacles on the shortest path. As only the shortest path is considered, the value can be calculated easily.

The final heuristic value is calculated using 'Manhattan' and 'Obstacle' distance values.

1.4 Additional Details

1.4.1 State Compression

The puzzle positions in the board are represented in a compressed state using three integer variables. Although there are about 19 pieces, any position can have at most 7 possible values:

1. All 3-cell pieces can be considered the same. If any two such pieces are swapped, there is no change in the board state. However, their orientation can be changed to horizontal or vertical (two different piece types). Therefore, two possible piece types can be *TYPE_3x1* and *TYPE_1x3*.
2. Similarly, all 2-cell pieces can be considered same. If any two such pieces are swapped, there is no change in the board state. However, their orientation - again, can be changed to horizontal or vertical (two different piece types). Therefore, two possible piece types can be *TYPE_2x1* and *TYPE_1x2*.
3. Although both the rover and termite mounds have the same shape and features, the rover being near the gate is game over, but termite mounds being close to the gate does not mean anything (except an obstacle of course). These two can be considered as two pieces types - *TYPE_x* for rover and *TYPE_uv* for termite mounds.
4. The location can also be empty - piece *TYPE_EMP*.

Following the above-mentioned rules, the Zobrist Table can be formed using 6 pieces per location. As there are 49 possible locations for a 7×7 grid and each location can have 7 possible values (3 bits required), $49 \times 3 = 147$ bits can be used to represent the entire state. The gate location falls out of the 49 possible locations. However, as keeping the top-left location of the piece will suffice instead of keeping the entire shape, the gate location can be ignored (no need to store goal state). Two unsigned long long integers and one unsigned long integer $(64 + 64 + 16/32 = 160)$ is enough to store the compressed state.

1.4.2 Transposition Table

The compressed state was used to store Transposition Table Entries. As a result it was possible to store 6.1 million entries without exceeding the 200 MB memory limit (about 190 MB used in total). The Transposition Table is used to avoid repetition of states in the same recursive call. If the current call has lower depth than the entry, then the current call is stored in the table. Otherwise, the current call is returned - a cut-off occurs due to repetition. Furthermore, the table is used to store the most updated heuristic value regardless of test case. If any state matches with an entry of a previous case, the heuristic value can still be used.

2 Experimental Section

2.1 Experimental Data

The program solves the original dataset in about 86.44 seconds in the lab server ‘ug17’. The most time-consuming test case is “A23” which requires approximately 41.133 seconds. The time required and the number of moves for each case increases with difficulty. There are many factors which contribute to the increase in difficulty as is evident from the Figure 1. Some of these factors are - increasing number of large pieces (rhinos, elephants, termite mounds, etc.), distance of the rover from the gate, total number of pieces in the board, and the number of pieces surrounding the rover.

The cutoff values for each search are calculated for both - when calculated heuristic value or when the updated heuristic value from the transposition table - added with the depth/g-cost, exceeds the upper bound. For each iteration, the upper bound is then increased gradually until a solution is found.

The following table shows the changes in performance including and excluding different enhancements for test case "E31" of the original dataset. The execution time and number of nodes increases by about 10% and 25%, respectively, when consecutive repetitions are not avoided or only Manhattan is used as heuristic. The time increases by 5-folds without repetition checks using transposition tables and three times the nodes are evaluated. Without any kind of repetition check, the execution time becomes 9 times the original search time. The number of nodes evaluated is 6 times more than the original search. Therefore, these enhancements improved the search performance of the function "IDAstar" greatly.

2.2 Problems and Solutions

Some of the problems faced while implementing the program, along with the solutions are mentioned below:

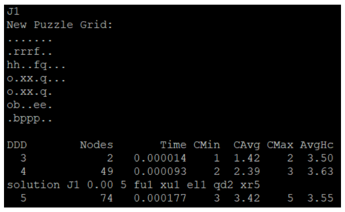
- The recursive functions are very difficult to debug. The related errors were solved by applying the program on small test cases and matching with manually calculated results.
- The state compression is difficult to get right in the first try. Once done, it is easy to duplicate or reverse (using functions). This problem can be solved by applying the compression and modification of the compressed states in a separate program for testing purposes.
- The most difficult problem was to come up with a heuristic which is both fast and useful. Various heuristics were tested to choose the appropriate one.
- The program was restructured continuously to reduce error and improve readability.

2.3 Future Work

The program can be further improved if a better heuristic function is used - separately or in combination with existing ones. The increase of transposition table size from 256K entries to 6.1 million entries drastically improved the execution time. Therefore, further increasing the size of transposition tables can improve the speed of execution.

Table 1: Improvement in terms of nodes evaluated and execution time

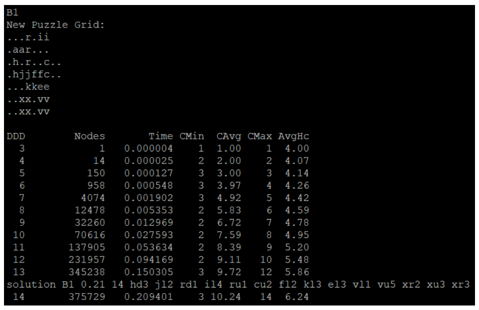
Category	**Nodes Evaluated**	**Time (s)**
Original Search	1,316,425	1.06
Without consecutive repetition check	1,622,496	1.14
With only Manhattan	1,564,324	1.15
Without TT repetition check	4,297,254	6.26
Without any repetition check	7,686,448	9.88



```
J1
New Puzzle Grid:
.......
.rrrf..
hh..fq...
o.xx.q...
o.xx.q.
ob..ee.
.bppp..

DDD        Nodes       Time CMin  CAvg CMax AvgHc
  3            2   0.000014    1  1.42    2  3.50
  4           49   0.000093    2  2.39    3  3.63
solution J1 0.00 5 fu1 xu1 el1 qd2 xr5
  5           74   0.000177    3  3.42    5  3.55
```

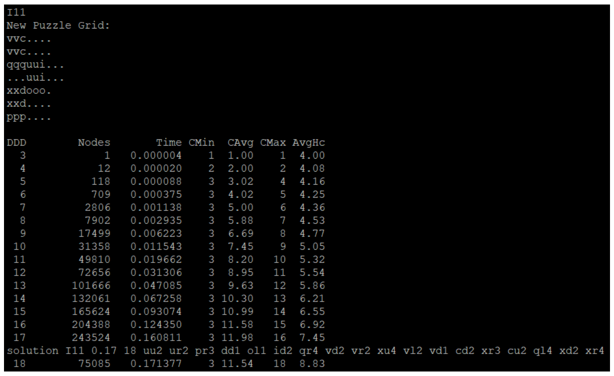
(a) First case of Junior data



```
B1
New Puzzle Grid:
...r.ii
.aar...
.h.r..c..
.hjjffc..
...kkee
..xx.vv
..xx.vv

DDD        Nodes       Time CMin  CAvg CMax AvgHc
  3            1   0.000004    1  1.00    1  4.00
  4           14   0.000025    2  2.00    2  4.07
  5          150   0.000127    3  3.00    3  4.14
  6          958   0.000548    3  3.97    4  4.26
  7         4074   0.001902    3  4.92    5  4.42
  8        12478   0.005353    2  5.83    6  4.59
  9        32260   0.012969    2  6.72    7  4.78
 10        70616   0.027593    2  7.59    8  4.95
 11       137905   0.053634    2  8.39    9  5.20
 12       231957   0.094169    2  9.11   10  5.48
 13       345238   0.150305    3  9.72   12  5.86
solution B1 0.21 14 hd3 jl2 rd1 il4 ru1 cu2 fl2 kl3 el3 vl1 vu5 xr2 xu3 xr3
 14       375729   0.209401    3 10.24   14  6.24
```

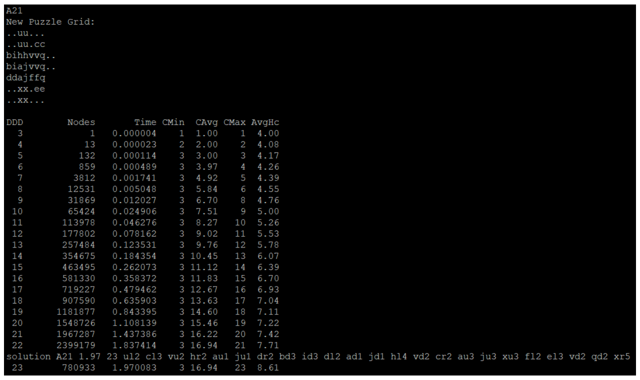
(b) First case of Beginner data



```
I11
New Puzzle Grid:
vvc....
vvc....
qqquui...
...uui...
xxdooo.
xxd....
ppp....

DDD        Nodes       Time CMin  CAvg CMax AvgHc
  3            1   0.000004    1  1.00    1  4.00
  4           12   0.000020    2  2.00    2  4.08
  5          118   0.000088    3  3.02    4  4.16
  6          709   0.000375    3  4.02    5  4.25
  7         2806   0.001138    3  5.00    6  4.36
  8         7902   0.002935    3  5.88    7  4.53
  9        17499   0.006223    3  6.69    8  4.77
 10        31358   0.011543    3  7.45    9  5.05
 11        49810   0.019662    3  8.20   10  5.32
 12        72656   0.031306    3  8.95   11  5.54
 13       101666   0.047085    3  9.63   12  5.86
 14       132061   0.067258    3 10.30   13  6.21
 15       165624   0.093074    3 10.99   14  6.55
 16       204388   0.124350    3 11.58   15  6.92
 17       243524   0.160811    3 11.98   16  7.45
solution I11 0.17 18 uu2 ur2 pr3 dd1 ol1 id2 qr4 vd2 vr2 xu4 vl2 vd1 cd2 xr3 cu2 ql4 xd2 xr4
 18        75085   0.171377    3 11.54   18  8.83
```

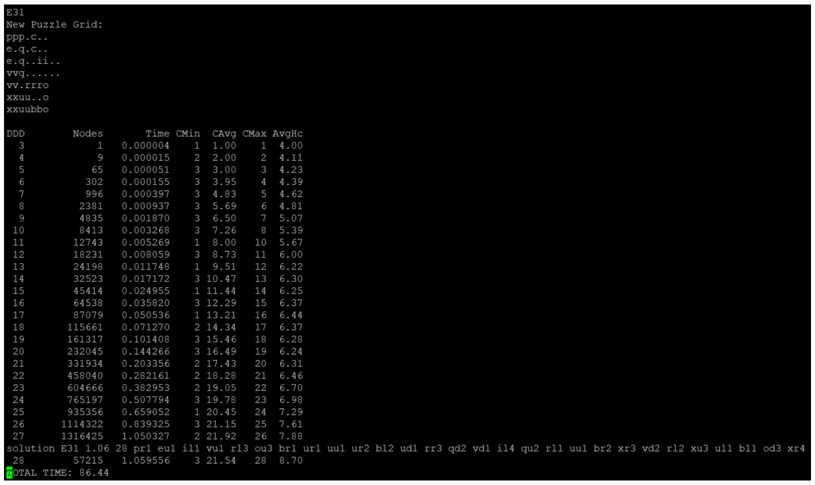
(c) First case of Intermediate data



```
A21
New Puzzle Grid:
..uu...
..uu.cc
bihhvvq..
biajvvq..
ddajffq
..xx.ee
..xx...

DDD        Nodes       Time CMin  CAvg CMax AvgHc
  3            1   0.000004    1  1.00    1  4.00
  4           13   0.000023    2  2.00    2  4.08
  5          132   0.000114    3  3.00    3  4.17
  6          859   0.000489    3  3.97    4  4.26
  7         3812   0.001741    3  4.92    5  4.39
  8        12531   0.005048    3  5.84    6  4.55
  9        31869   0.012027    3  6.70    8  4.76
 10        65424   0.024906    3  7.51    9  5.00
 11       113978   0.046276    3  8.27   10  5.26
 12       177802   0.078162    3  9.02   11  5.53
 13       257484   0.123531    3  9.76   12  5.78
 14       354675   0.184354    3 10.45   13  6.07
 15       463495   0.262073    3 11.12   14  6.39
 16       581330   0.358372    3 11.83   15  6.70
 17       719227   0.479462    3 12.67   16  6.93
 18       907590   0.635903    3 13.63   17  7.04
 19      1181877   0.843395    3 14.60   18  7.11
 20      1548726   1.108139    3 15.46   19  7.22
 21      1967287   1.437386    3 16.22   20  7.42
 22      2399179   1.837414    3 16.94   21  7.71
solution A21 1.97 23 ul2 cl3 vu2 hr2 au1 ju1 dr2 bd3 id3 dl2 ad1 jd1 hl4 vd2 cr2 au3 ju3 xu3 fl2 el3 vd2 qd2 xr5
 23       780933   1.970083    3 16.94   23  8.61
```

(d) First case of Advanced data



```
E31
New Puzzle Grid:
ppp.c..
e.q.c..
e.q..ii..
vvq......
vv.rrro
xxuu..o
xxuubbo

DDD        Nodes       Time CMin  CAvg CMax AvgHc
  3            1   0.000004    1  1.00    1  4.00
  4            9   0.000015    2  2.00    2  4.11
  5           65   0.000051    3  3.00    3  4.23
  6          302   0.000155    3  3.95    4  4.39
  7          996   0.000397    3  4.83    5  4.62
  8         2381   0.000937    3  5.69    6  4.81
  9         4835   0.001870    3  6.50    7  5.07
 10         8413   0.003268    3  7.26    8  5.39
 11        12743   0.005269    1  8.00   10  5.67
 12        18231   0.008059    3  8.73   11  6.00
 13        24198   0.011748    1  9.51   12  6.22
 14        32523   0.017172    3 10.47   13  6.30
 15        45414   0.024955    1 11.44   14  6.25
 16        64538   0.035820    3 12.29   15  6.37
 17        87079   0.050536    1 13.21   16  6.44
 18       115661   0.071270    2 14.34   17  6.37
 19       161317   0.101408    3 15.46   18  6.28
 20       232045   0.144266    3 16.49   19  6.24
 21       331934   0.203356    2 17.43   20  6.31
 22       458040   0.282161    2 18.28   21  6.46
 23       604666   0.382953    2 19.05   22  6.70
 24       765197   0.507794    3 19.78   23  6.98
 25       935356   0.659052    1 20.45   24  7.29
 26      1114322   0.839325    3 21.15   25  7.61
 27      1316425   1.050327    2 21.92   26  7.88
solution E31 1.06 28 pr1 eu1 il1 vu1 rl3 ou3 br1 ur1 uu1 ur2 bl2 ud1 rr3 qd2 vd1 il4 qu2 rl1 uu1 br2 xr3 vd2 rl2 xu3 ul1 bl1 od3 xr4
 28        57215   1.059556    3 21.54   28  8.70
TOTAL TIME: 86.44
```

(e) First and only case of Expert data

Figure 1: Gradually increasing time, number of nodes, and depth of search tree, with monotonically increasing difficulty from Figure 1a to Figure 1e. There are some exceptions like - (Beginner, Intermediate) and (Advanced, Expert)

.