

**T.C.**

**MARMARA UNIVERSITY**

**FACULTY of ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

CSE4082 – Project Design Document

**Group Members**

 150116012 – Rahim Gün

                               150116016 – Berk Köylü

        150116053 – Eray Ayaz

     03.01.2021

In our project we, we created a different class for each algorithm. In our main class, we run these algorithms using their objects.

**Classes**

1. **State**

We treated each puzzle as a state, so we created this class. In this class, we wanted to make things easier by keeping the information about the puzzle.

Fields

matrixPuzzle: Keeps the state’s puzzle

previousState: Keeps the previous state after the movement.

cost : Every movement has a cost so it variable keeps the total cost.

depth : Each movement has a depth in graph search, this variable holds this information.

heuristic : Keeps the heuristic value of the algorithm.

positions : Find the 0’s location on the puzzle.

blankTileCoordinateX : Keeps the 0’s location x coordinate.

blankTileCoordinateY: Keeps the 0’s location y coordinate.

Methods

findSpace : Find the 0’s location on the puzzle.

Get/Set methods of the given fields.

1. **Breadth First Search**

In the breadth-first search algorithm, we go to the children of the node where we can go from the node we are in and after visiting all the children, we choose one of these children and continue this process until we find our node of goal.

Fields

solution:  This 2 dimensional array contains our goal state.

visited: In this hashset we keep our expanded nodes

frontier: In this queue we keep child nodes which we did not expand before.

Methods

solve: We apply the algorithm and print the solution path, cost, number of expanded nodes and number of nodes stored in memory. Then  we add the child nodes to the frontier queue if there is no violation about the adding.

isSolution: Check each time if the visited node is the goal node.

searchVisited: Checking whether the child node is already visited or not.

searchFrontier: Checking whether the child node is already at frontier or not.

addQueue: If children nodes are in visited or frontier, we will not add to

frontier. We check it then add it to frontier if there is no problem.

printPuzzle: Print the solution path.

1. **Depth First Search**

The Depth first search algorithm provides an increasingly deeper search until the end of the first opened child node, then returns to the top node and continues in the same way as the other child node.

Fields

solution: This 2 dimensional array contains our goal state.

visited: In this hashset we keep our expanded nodes

frontier: In this stack we keep child nodes which we did not expand before.

Method

solve: We apply the algorithm and print the solution path, cost, number                       of expanded nodes and number of nodes stored in memory. Then we add the child nodes to the frontier queue if there is no violation about the adding.

isSolution: Check each time if the visited node is the goal node.

searchVisited: Checking whether the child node is already visited or not.

searchFrontier: Checking whether the child node is already at frontier or not.

addStack: If children nodes are in visited or frontier, we will not add to

frontier. We check it then add it to frontier if there is no problem.

printPuzzle: Print the solution path.

1. **Iterative Deepening Search**

Iterative Deepening Search algorithm works the same as the depth first search algorithm, but this time the height to go under the child node is limited.

Fields

solution: This 2 dimensional array contains our goal state.

visited: In this hashset we keep our expanded nodes

frontier: In this stack we keep child nodes which we did not expand before.

solvedFlag: This boolean variable tells the program to stop the project because the goal state was found.

Method

solve: We apply the algorithm and print the solution path, cost, number                       of expanded nodes and number of nodes stored in memory. Then we add the child nodes to the frontier queue if there is no violation about the adding.

isSolution: Check each time if the visited node is the goal node.

searchVisited: Checking whether the child node is already visited or not.

searchFrontier: Checking whether the child node is already at frontier or not.

addStack: If children nodes are in visited or frontier, we will not add to

frontier. We check it then add it to frontier if there is no problem.

printPuzzle: Print the solution path.

1. **Uniform Cost Search**

Uniform cost search algorithm tries to reach the target point with the  least cost by choosing the lowest cost of the points in frontier.

Fields

solution: This 2 dimensional array contains our goal state.

visited: In this hashset we keep our expanded nodes

frontier: In this priority queue we keep child nodes which we did not expand before.

Method

solve: We apply the algorithm and print the solution path, cost, number                       of expanded nodes and number of nodes stored in memory. Then we add the child nodes to the frontier queue if there is no violation about the adding.

isSolution: Check each time if the visited node is the goal node.

searchVisited: Checking whether the child node is already visited or not.

searchFrontier: Checking that the child node is not already at the frontier.

searchAtFrontier: Checking that the child node is already at the border.

swapFrontier: İf node to be added is already addded, then the lower cost must be added to frontier between these 2 same node. In this case, this method look to these two puzzle and take the puzzle with the smallest cost and if it needs to swap them.

addQueue: If children nodes are in visited or frontier, we will not add to

frontier. We check it then add it to frontier if there is no problem.

printPuzzle: Print the solution path

1. **Iterative Lengthening Search**

Iterative lengthening search works the same as uniform cost search algorithm but this time cost is limited. If the goal state is found in more than the limited cost, the goal state can not be found, only the limited cost can be looked at.

Fields

solution: This 2 dimensional array contains our goal state.

visited: In this hashset we keep our expanded nodes

frontier: In this priority queue we keep child nodes which we did not expand before.

Method

solve: We apply the algorithm and print the solution path, cost, number                       of expanded nodes and number of nodes stored in memory. Then we add the child nodes to the frontier queue if there is no violation about the adding.

isSolution: Check each time if the visited node is the goal node.

searchExploredAndVisited: Checking that the child node is not already at the frontier and visited

searchAtFrontier: Checking that the child node is already at the frontier.

swapFrontier: If the node to be added is already added, then the lower cost must be added to the frontier between these 2 same node. In this case, this method look to these two puzzle and take the puzzle with the smallest cost and if it needs to swap them.

addQueue: If children nodes are in visited or frontier, we will not add to

frontier. We check it then add it to frontier if there is no problem.

printPuzzle: Print the solution path

1. **A\* Heuristic Search**

The A \* algorithm is an intuitive algorithm. However, it is like the combination of two algorithms A \* algorithm. We use Uniform cost search to estimate the cost from the starting point to the destination point, and we use the greedy algorithm to estimate the least cost from the point to the end point.

H1**:**The number of misplaced tiles at node n

H2:The sum of the city-block distances of each misplaced tile from its current location to its goal location

Fields

solutionMatrix: This 2 dimensional array contains our goal state.

visited: In this hashset we keep our expanded nodes

frontier: In this priority queue we keep child nodes which we did not expand before.

maxNumberOfStoredInMemory: This variable keeps the maximum number of nodes stored in memory.

Method

solve: We apply the algorithm and print the solution path, cost, number                       of expanded nodes and number of nodes stored in memory. Then we add the child nodes to the frontier queue if there is no violation about the adding.

isSolution: Check each time if the visited node is the goal node.

searchVisited: Checking whether the child node is already visited or not.

searchFrontier: Checking that the child node is not already at the frontier.

searchAtFrontier: Checking that the child node is already at the frontier.

swapFrontier: If the node to be added is already added, then the lower cost must be added to the frontier between these 2 same node. In this case, this method look to these two puzzle and take the puzzle with the smallest cost and if it needs to swap them.

addQueue: If children nodes are in visited or frontier, we will not add to

frontier. We check it then add it to frontier if there is no problem.

calculateHeuristicH1: Count the misplaced tiles in initial puzzle.

calculateHeuristicH2: Calculate the of the city-block distances of each misplaced tile from its current location to its goal location.

printPuzzle: Print the solution path

1. **Puzzle Generator**

In this method, in order to compare performances of these algorithms, our program generate random puzzle instances with given depth.

Fields

lastMovedTile : It keeps the empty spaces on the puzzle.

puzzle : This matrix is our solution matrix.

Method

generatePuzzle : This method generating a puzzle with given depth and move can be done.

findPossibleMove : Find the possible move with given puzzle.

findSpace : Find 0’s place on the puzzle.

makeMove : Make the movement with the given place and 0’ location. So that we can make the changes on the puzzle.

1. **Move**

This interface provides all 8 directions to go and puzzles that can occur when these directions are applied.

Method

up : 0’s place in the puzzles moves to the up.

down : 0’s place in the puzzles moves to the down.

right : 0’s place in the puzzles moves to the right.

left : 0’s place in the puzzles moves to the left.

upAndRight : 0’s place in the puzzles moves diagonally to the right and up.

upAndLeft : 0’s place in the puzzles moves diagonally to the left and up.

downAndRight : 0’s place in the puzzles moves diagonally to the right and down.

downAndLeft : 0’s place in the puzzles moves diagonally to the left and down.

**Tables**

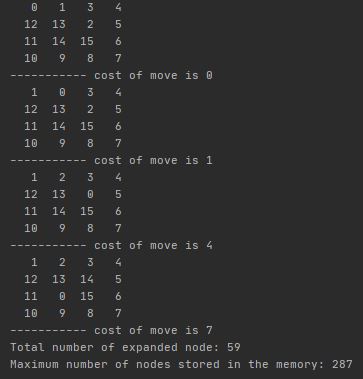
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| The total number of expanded nodes | | | | | | | |
| d | BFS | DFS | IDS | UCS | ILS | A\*(h1) | A\*(h2) |
| 2 | 13 |  | 9,8 | 16,6 | 19,6 | 2,6 | 2,2 |
| 4 | 328,6 |  | 220,6 | 6242 | 5845 | 28 | 12,8 |
| 6 | 3655,2 |  | 1870 | 9905,3 | 7345,6 | 35,6 | 19,6 |
| 8 | 4964,4 |  |  | 12878,4 | 10717,4 | 41,4 | 21 |
| 10 |  |  |  |  | 12568,6 | 53,4 | 29,7 |
| 12 |  |  |  |  |  | 721,7 | 102 |
| 16 |  |  |  |  |  | 5925,4 | 379,6 |
| 20 |  |  |  |  |  |  | 2031,9 |
| 24 |  |  |  |  |  |  | 5928,3 |
| 28 |  |  |  |  |  |  | 9385,6 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| The maximum number of nodes stored in memory | | | | | | | |
| d | BFS | DFS | IDS | UCS | ILS | A\*(h1) | A\*(h2) |
| 2 | 64,4 |  | 14,4 | 78,8 | 35,8 | 16,2 | 14 |
| 4 | 1367,8 |  | 241,4 | 24483,6 | 8133,6 | 137,8 | 68 |
| 6 | 14528,4 |  | 2134,4 | 39683,6 | 13682,8 | 181,8 | 109 |
| 8 | 21708,8 |  |  | 47766,8 | 25024,3 | 198,9 | 115,2 |
| 10 |  |  |  |  | 30243,5 | 210,8 | 185,2 |
| 12 |  |  |  |  |  | 3216,1 | 527,1 |
| 16 |  |  |  |  |  | 24522,3 | 1838 |
| 20 |  |  |  |  |  |  | 9403,7 |
| 24 |  |  |  |  |  |  | 26730,1 |
| 28 |  |  |  |  |  |  | 35543,3 |

**Puzzles**

Matrix a)

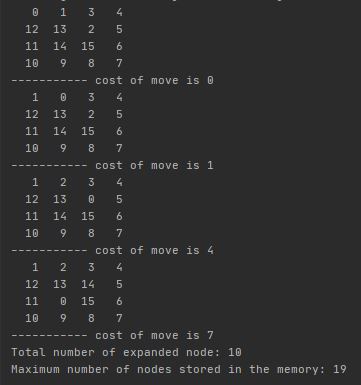
BFS



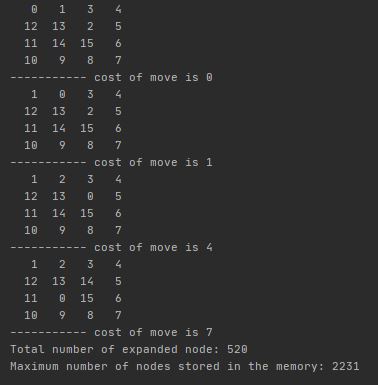
DFS

This algorithm can not find the solution in 15 minutes.

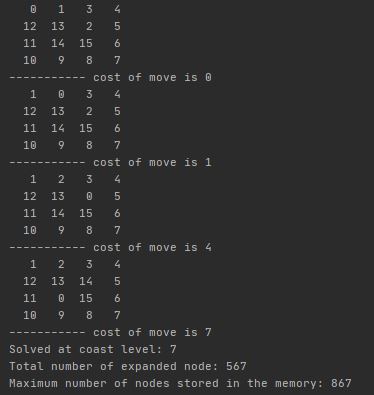
IDS



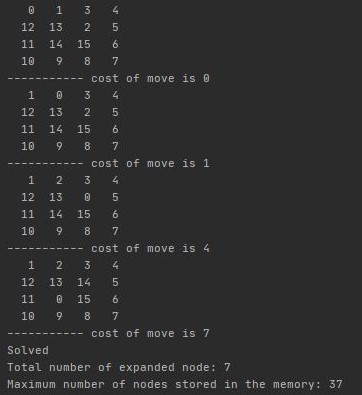
UCS



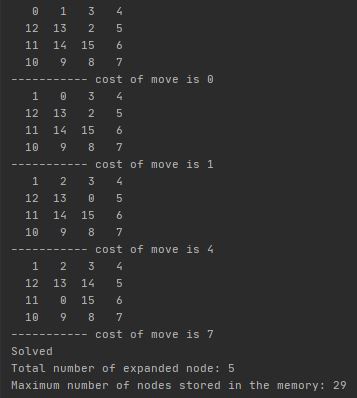
ILS



A\*H1

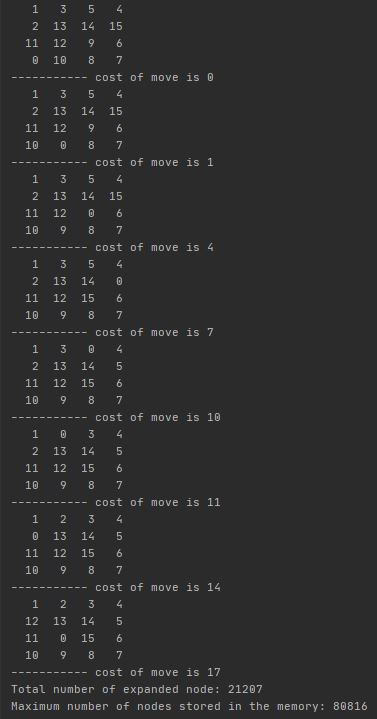


A\*H2



Matrix b)

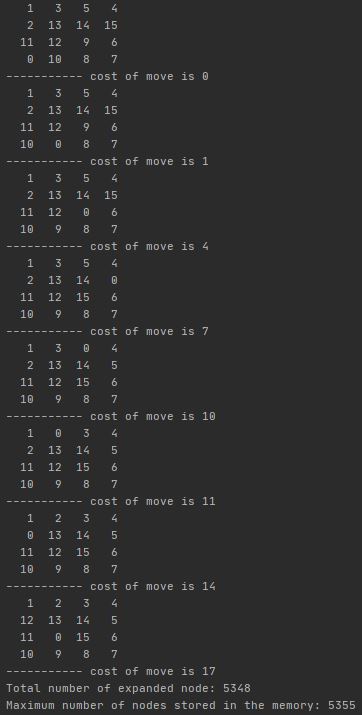
BFS



DFS

This algorithm can not find the solution in 15 minutes.

IDS



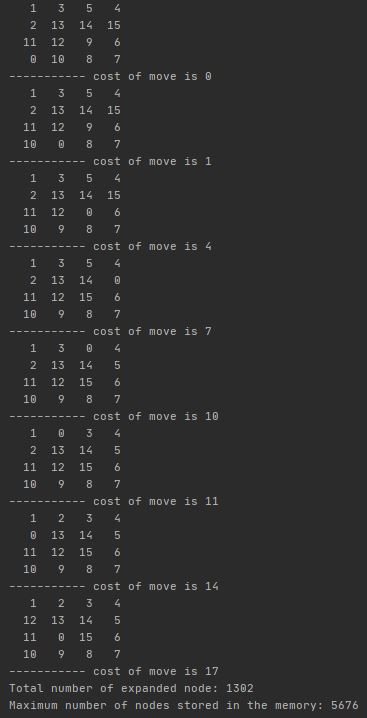
UCS

This algorithm can not find the solution in 15 minutes.

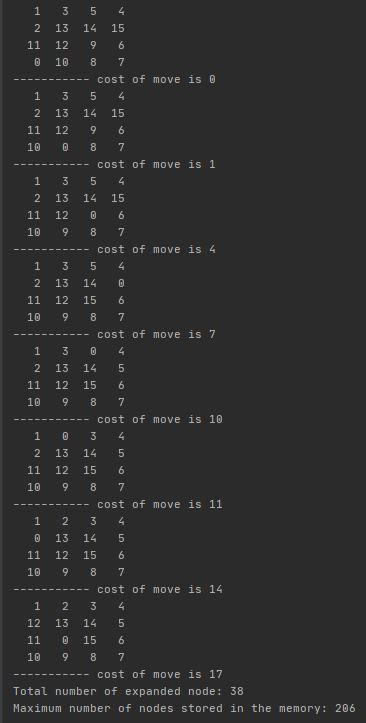
ILS

This algorithm can not find the solution in 15 minutes.

A\*H1



A\*H2



Matrix c)

BFS

This algorithm can not find the solution in 15 minutes.

DFS

This algorithm can not find the solution in 15 minutes.

IDS

This algorithm can not find the solution in 15 minutes.

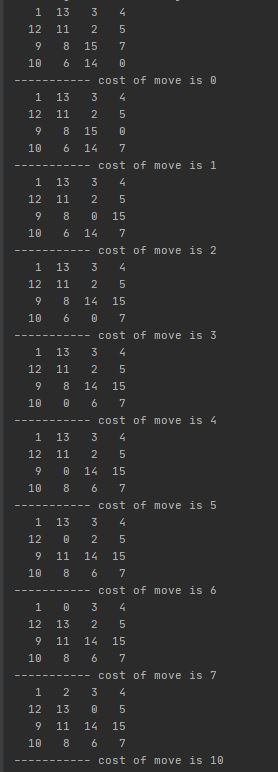
UCS

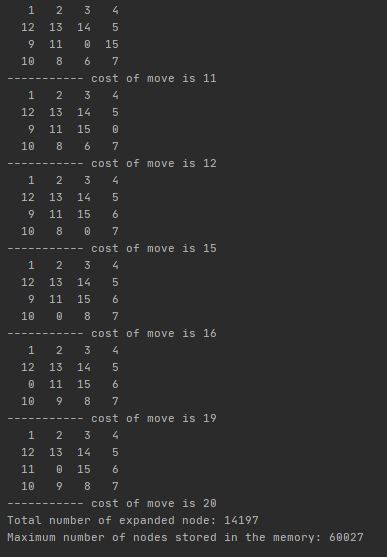
This algorithm can not find the solution in 15 minutes.

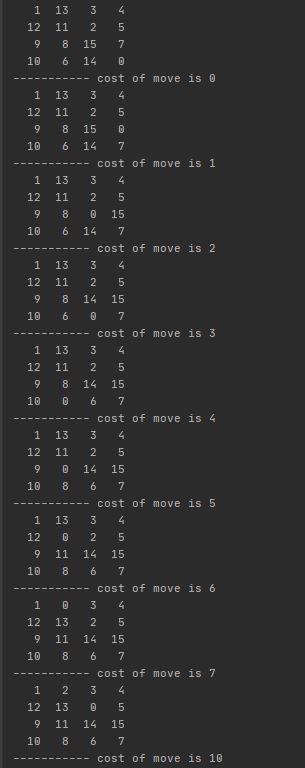
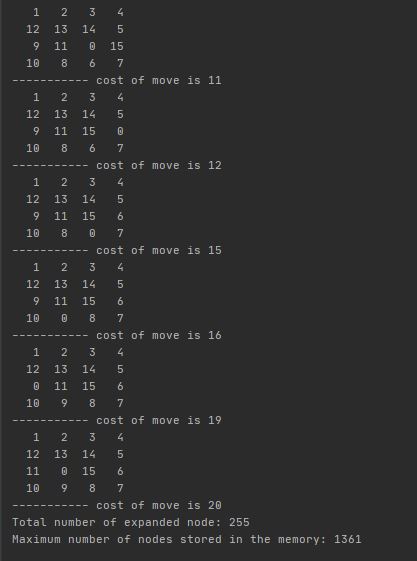
ILS

This algorithm can not find the solution in 15 minutes.

A\*H1





A\*H2

**Analysis**

A\* search algorithm with h2 gives the best result if we compare the total number of expanded nodes and the maximum number of nodes stored in memory between other algorithms. Also, the execution time of the A\* search with h2 is faster than any other algorithm. And, for both heuristic h1 and h2, A\* search algorithm gives the optimal solution because we implemented graph search manner so that we found optimal paths by tracking best f-values so far for all nodes visited and updating as you go. Also, UCS gives the optimal solution but, its exponential space complexity increases the execution time as we go through deeper levels and, UCS needs to visit all nodes before the optimal node to complete its search because of its cost function and frontier queue manner. ILS finds the optimal result and, as we can see in the tables, it has an advantage over UCS about a stored node in the memory. IDS finds goal state at lower levels and has an advantage over BFS about space complexity we can see in tables. Also, BFS finds the goal state at lower levels but cost steps are not identical so that result does not guarantee optimality. Finally, DFS is the worst algorithm because even for a simple goal state that can be reached by moving one upper move, it can go left first and continues deeper level and never comes back.