**EE586 Artificial Intelligence**

**Fall 2021-2022**

**Programming Take Home Assignment**

**Instructor: Prof. Afşar Saranlı**

**Sinan Çimen**

**2169001**

This programming assignment is done using C++ language. Code is compiled with Visual Studio Community 2013. Since C++ CLR support is not given after 2013, you may not be able to compile it in newer versions (I did not try). The solution folder of the project will be given alongside this document. A separate .exe file will also be given to run the program without compiling it. Windows OS is required(I only verified it on Windows 10 & Windows 11).

Please note that, exceptions are not covered by the code. Which means, entering wrong inputs in the GUI may lead to crashes. The way you should give inputs to the program will be written under the answers of each algorithm. Also, while the program is solving a puzzle or performing Monte-Carlo simulations, the program may freeze (if you click on another button or similar). Even if the program freezes, it will continue to try to solve the puzzle. When the puzzle is solved, it will recover from the frozen state.

1. A data structure that enables variable size puzzles to be stored is designed. This design is inspired from a incomplete github project; but, I made many modifications to it. (Link: <https://github.com/BytesClub/N-Puzzle-Solver/tree/master/cpp>).

One can see the data structure under the file “states.h” under the class *Node.* You can see the screenshot of the definition of that class below.

The state(position of tiles) are stored using a variable size 2-D vector, which is a class of C++ Standard Library. Also, there are several properties. Parent property is defined to store from which Node, the corresponding Node is expanded. Misplaced\_tiles and Manhattan\_distance properties are defined to store heuristics of a state. Note that, the class definition is the same for A\*, BFS, DFS and ID algorithms; however, these heuristics are only calculated when we are using A\* algorithm.

Several getters and setters are writing to retrieve and write a node’s properties. Steps taken property is automatically generated by incrementing that property of the parent node by one. Therefore, only getter of that property is available. Blank tile of the puzzle is stored by variables blank\_x and blank\_y just to make it easier for the solver.

To begin a puzzle, the initial node is created with NULL parent.

metin içeren bir resim

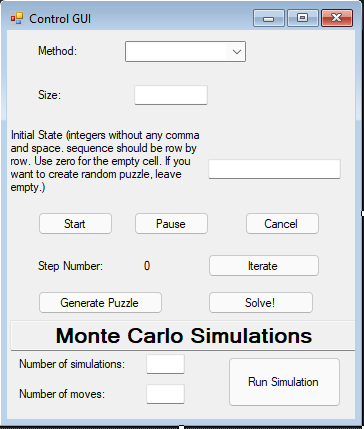
Açıklama otomatik olarak oluşturuldu

1. A basic graphical interface is designed to retrieve inputs and show results of simulations. Windows Forms are used to achieve this. You can see the file structure of the project below.

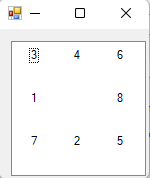
metin, ekran, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

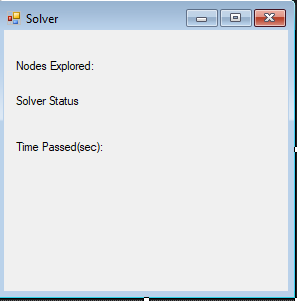
4 separate graphical interfaces are present. The first one is the main interface named Control GUI (MyForm). Only this interface is seen when you run the program. You can see the screenshot of Control GUI below. In this interface method, size, initial state and parameters for Monte-Carlo simulations are retrieved. You can generate the puzzle and solve it and iterate the solution from this GUI.



When you enter an initial state, or the size if you want to generate a random puzzle; and, click on Generate Puzzle, the generated puzzle will be seen in a new window named Puzzle Visual (MyForm1).



When you click on solve after generating a puzzle, another window will appear named Solver(MyForm2). In this window, you will see the solver state, number of explored nodes; and, if the solver succeed to solve the puzzle, you will see the time passed while solving the puzzle. If the program does not freeze out somehow, nodes explored will be continuously updated while solving the puzzle.

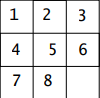


When you run a Monte-Carlo simulation, after the simulation is complete, another window will appear named Monte-Carlo Simulation(MyForm3). In this window, you will be able to see the average number of nodes explored and average number of nodes stored.

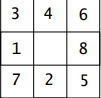
metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Note that, the usage of GUI will be written in the next parts. Also, note that, the goal state cannot be changed with the GUI and it is always the same. The goal state is given below. For different puzzle sizes, the distribution of tiles of the goal state is in the same manner.



In the GUI, if the initial state textbox is empty and the size is given, program will generate a random puzzle. Note that the size must given as 3 for 8-Puzzle, 4 for 15-Puzzle etc. Initial state must be given as a sequence of integers without any commas or spaces. The blank tile must be denoted as zero. For example, such a puzzle below needs to be written as **346108725**.



1. Successors() function is developed to generate children of a node. This function mainly uses the position of the blank tile to determine which moves can be generated. For example, if the blank tile is on the upper left corner; then, there are 2 possible moves. If the blank tile is on the middle of left edge, there are 3 possible moves. If the blank tile is not on any edge, then, there are 4 possible moves.

The successors() function takes a node as an input and returns a vector of nodes. You can see the successors function below.

std::vector<Node\*> successors(Node\* configuration)

{

std::vector<std::vector<int>> parentConfiguration = configuration->getState();

std::vector<int> blankPosition= configuration->getBlankPosition();

int size = parentConfiguration.size();

std::vector<Node\*> successors\_list;

if (blankPosition.at(0) == 0)

{

if (blankPosition.at(1) == 0)

{

successors\_list.push\_back(moveUp(configuration));

successors\_list.push\_back(moveLeft(configuration));

//move up and left

}

else if (blankPosition.at(1) == size - 1)

{

successors\_list.push\_back(moveDown(configuration));

successors\_list.push\_back(moveLeft(configuration));

//move left and down

}

else

{

successors\_list.push\_back(moveDown(configuration));

successors\_list.push\_back(moveLeft(configuration));

successors\_list.push\_back(moveUp(configuration));

//move left, up, down

}

}

else if (blankPosition.at(1) == 0)

{

if(blankPosition.at(0) == size - 1)

{

successors\_list.push\_back(moveRight(configuration));

successors\_list.push\_back(moveUp(configuration));

//move left and up

}

else

{

successors\_list.push\_back(moveRight(configuration));

successors\_list.push\_back(moveUp(configuration));

successors\_list.push\_back(moveLeft(configuration));

//move right, left, up

}

}

else if (blankPosition.at(0) == size - 1)

{

if (blankPosition.at(1) == size - 1)

{

successors\_list.push\_back(moveRight(configuration));

successors\_list.push\_back(moveDown(configuration));

// move down and right

}

else

{

successors\_list.push\_back(moveRight(configuration));

successors\_list.push\_back(moveDown(configuration));

successors\_list.push\_back(moveUp(configuration));

//move up down right

}

}

else if (blankPosition.at(1) == size - 1)

{

successors\_list.push\_back(moveRight(configuration));

successors\_list.push\_back(moveDown(configuration));

successors\_list.push\_back(moveLeft(configuration));

//move left right down

}

else

{

successors\_list.push\_back(moveRight(configuration));

successors\_list.push\_back(moveDown(configuration));

successors\_list.push\_back(moveUp(configuration));

successors\_list.push\_back(moveLeft(configuration));

//move up down left right

}

return successors\_list;

}

Note that successors function use moveRight, moveDown, moveUp and moveLeft functions to generate childs. An example of these, moveRight function is given below. It simply exchanges moving tile with the empty tile.

Node\* moveRight(Node\* configuration)

{

std::vector<int> blankPosition;

blankPosition = configuration->getBlankPosition();

std::vector<std::vector<int>> state = configuration->getState();

state[blankPosition.at(0)][blankPosition.at(1)] = state.at(blankPosition.at(0) - 1).at(blankPosition.at(1));

state[blankPosition.at(0) - 1][blankPosition.at(1)] = 0;

Node\* resultNode = new Node(configuration, state, 0);

resultNode->setBlankPosition(blankPosition.at(0) - 1, blankPosition.at(1));

return resultNode;

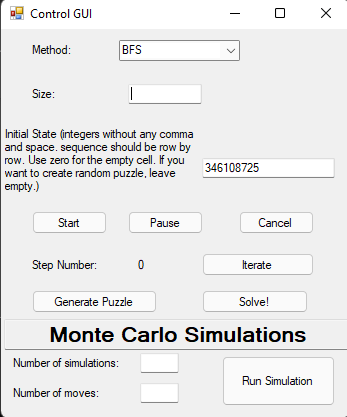
}

1. I am not used to the 8-Puzzle game; so, I am not able to solve it by myself. However, the optimal sequence of moves will be obtained when we run the BFS algorithm for the puzzle.

All algorithms used to solve puzzles are defined under solver.cpp. BFS is implemented with the function find\_solution\_bfs(). This function takes the initial state as a 1-D vector; then, it arranges the initial state to convert it into a 2-D vector. The function returns a vector of 2-D vectors; which is the sequence of solution.

BFS uses a queue which is a vector of nodes. At each iteration, it expands the first element in the queue. Then, algorithm checks if the successors are already labelled (already expanded). Successors that are not labelled previously are added at the end of queue.

To solve this puzzle with BFS, inputs to the GUI should be given as;



Note that size is not required since we defined the initial state. After selecting the method and writing the initial state, we should first click on ‘Generate Puzzle’ and then ‘Solve!’.

4.a. Number of node expansion: 1637

4.b. Number of moves: 12

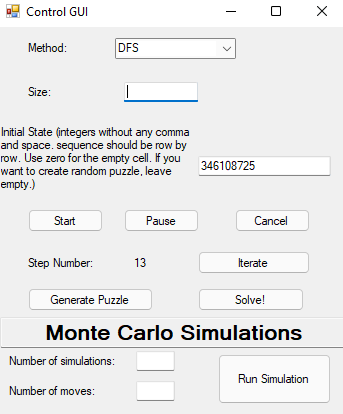
4.c. Time required:

One can see the sequence of moves by using both ‘Iterate’ button to take one step, or one can use ‘Start’ to see the sequence continuously (1 step per second).

1. I tried the simplest form of DFS to solve the puzzle; however, it was not able to solve the puzzle. Then, I added loop-avoiding feature to the DFS. Also, it was not able to solve the problem (I waited for 4-5 hours). It is probably because after each iteration, the computational load of the algorithm increases because of checking for loops.

Then, I limited the depth. Since I knew that the optimal solution requires 12 moves, I made an educated guess of depth. I have chosen the maximum depth as 20, which I think is not a perfect guess such that someone who is accustomed to 8-puzzle game may be able to predict.

DFS is implement with the function find\_solution\_dfs(). Inputs and outputs of that function are the same as BFS. The way I limited the depth is using the stepsTaken property of node class which is automatically generated when a node is created. If stepsTaken>20; then, the node will not be added to the queue.



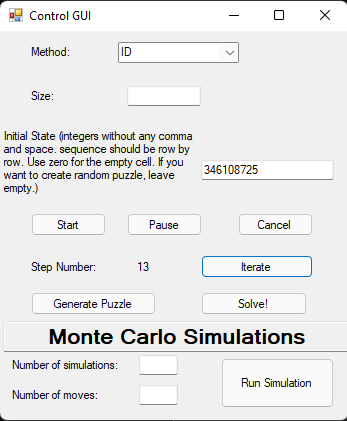
5.a. Number of node expansion: 2337

5.b. Number of moves: 14

5.c. Time required:

Note that since the depth is not perfectly limited and DFS is not an optimal algorithm, number of moves required to reach the goal is more than the optimal number(12).

1. Iterative deepening search is implement with the function find\_solution\_id(). Since this algorithm does not check for loops, the number of node expansions are quite high. However, memory requirement is low since we do not store the labelled states.



6.a. Number of node expansion: 81942

6.b. Number of moves: 12

6.c. Time required:

1. a. The Manhattan distance heuristics contains more information about the state of the puzzle than the Misplaced tiles heuristics. Misplaced tiles heuristic is a much more optimistic estimation of the number of moves that needs to be taken; therefore, it results in a less accurate estimation.

7.b. The two heuristic function are implemented as w.

void calculateManhattanDistance(Node\* node)

{

std::vector<std::vector<int>> state = node->getState();

int size = state.size();

int manhattan\_distance = 0;

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

int number\_on\_state = state.at(i).at(j);

if (number\_on\_state == 0)

number\_on\_state = 9;

manhattan\_distance += abs((int)(number\_on\_state / size) - (int)((size \* i + j + 1) / size)) + abs((int)(number\_on\_state % size) - (int)((size \* i + j + 1) % size));

}

}

manhattan\_distance += node->getStepsTaken();

node->setManhattanDistance(manhattan\_distance);

}

void calculateMisplacedTiles(Node\* node)

{

std::vector<std::vector<int>> state = node->getState();

int size = state.size();

int misplaced\_tiles = 0;

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

{

if (state.at(i).at(j) != size \* i + j + 1)

misplaced\_tiles++;

}

}

misplaced\_tiles += node->getStepsTaken();

node->setMisplacedTiles(misplaced\_tiles);

}

Note that, after the calculation of the manhattan distance and misplaced tiles, stepsTaken property is added to the heuristic function. This is done to force the algorithm to find the optimal path.

7.c. For Misplaced tiles;

Number of node expansion: 77

Number of moves: 12

Time required:

For Manhattan distance:

Number of node expansion: 46

Number of moves: 12

Time required: