

CS205: Artificial Intelligence, Dr. Eamonn Keogh

Project 1 – Eight Puzzle Solver using Uniform Cost Search and A* search with heuristics.

Name: Suryaa Charan Shivakumar

Student ID: 862395094

Email: sshiv012@ucr.edu

Date: May-15-2023

In completing this project I have consulted the following:

- My friend Bhavya S Gada who also happens to be my classmate in CS205 this quarter.
- Java Documentation from W3 Schools - <https://www.w3schools.com/java/>
- Stack Overflow for various stupid questions that clouded my mind during the course of programming, here are a few links I recalled from history -
<https://stackoverflow.com/questions/13198703/java-convert-listinteger-to-string>
<https://stackoverflow.com/questions/20252727/is-not-an-enclosing-class-java>
<https://stackoverflow.com/questions/17578820/accessing-outer-class-variable-in-inner-class>
<https://stackoverflow.com/questions/40741742/how-should-i-add-my-intellij-project-to-an-already-existing-github-repository>
- Chat GPT when that did not help, and for debugging when I was not able to figure out what went wrong (Attached Screenshots of the conversation at the end of the report)
- The Blind Search and Heuristic Search lecture slides and notes annotated from lecture and Professor Eamonn's project report template.

I acknowledge that all important code is original. Unimportant subroutines that are not completely original are –

- List functions from java.util, including ArrayList and its functions – used to perform operations and maintain state ordering
- Java Math and all its subroutines to avoid writing functions for mathematical calculations like sqrt, or absolute value or max or floor value.
- Priority queue from java.util and all its subroutines for minheap in search algorithm
- Comparator and its functions for sorting objects of a custom class
- Java Map and HashMap from java.util to maintain list of visited nodes
- Java Collectors API from java.util.stream to convert a list of Integers to String (Useful for hashing)

Outline of the report –

- Cover page (this page) Page 1
- Report, pages 2 - 7
- Sample Trace on Easy problem, pages 8-9
- Sample Trace on Hard problem, pages 10-12
- Code(<https://github.com/sshib012/CS205Project>), pages 12-17

CS205: Project 1 The Eight Puzzle Problem

Suryaa Charan Shivakumar, Student ID – 862395094, May-15-2023

Introduction

The 8-puzzle problem^[1] is a classic and well-known puzzle in the field of artificial intelligence and computer science. It involves a 3x3 grid containing eight numbered tiles and one blank tile. The goal is to rearrange the tiles from a given initial configuration to a desired target configuration by sliding the tiles into the blank space. The puzzle poses an interesting challenge due to the immense number of possible configurations and the need to find an optimal solution.

The 8-puzzle problem serves as a representative example of a broader class of sliding tile puzzles. Variants of this problem extend beyond the original 3x3 grid size and can include different numbers of tiles. One such variant is the 15 puzzles, which uses a 4x4 grid and consists of 15 numbered tiles and one blank tile. Another variant is the 24 puzzles, which employs a 5x5 grid and involves 24 numbered tiles and one blank tile. Examples are illustrated in figure 1.

Furthermore, there are variations that introduce additional complexities to the puzzle. For instance, the torus puzzle considers a wrap-around effect where tiles on the left edge are connected to those on the right edge, and tiles on the top edge are connected to those on the bottom edge. This introduces a cyclic property to the puzzle, expanding the possible configurations and challenging the solver to find an optimal solution.

In this project (project-1 in Dr. Eamonn Keogh's Introduction to AI course at the University of California, Riverside), we aim to address the 8-puzzle problem by implementing three different search algorithms: Uniform Cost Search, A* Search with misplaced tile heuristics, and A* Search with Manhattan distance heuristic. By comparing the performance and efficiency of these algorithms, we hope to gain insights into their effectiveness in solving the 8-puzzle problem and potentially extend these findings to other related problems. My language of choice was Java (more specifically Java 8) and the full code for the project is included.

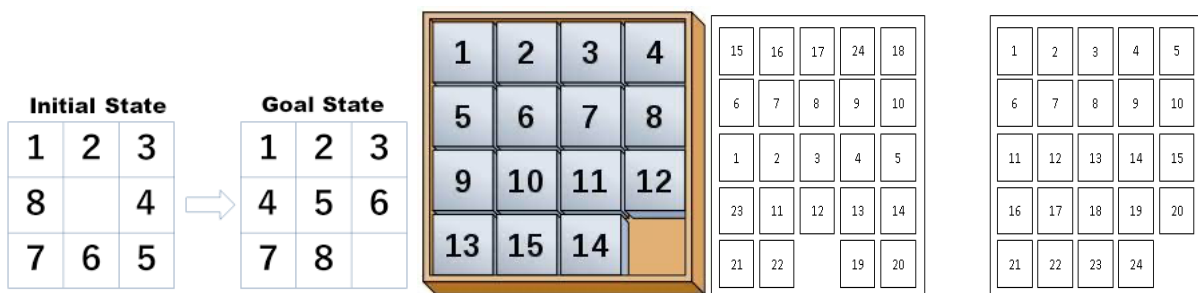


Figure 1 – Puzzle problem and it's variants (Image Courtesy – Wikipedia, Stackoverflow)

Comparison of algorithms

The three algorithms implemented are as follows:

1. Uniform Cost Search
2. A* using the Misplaced Tile heuristic.
3. A* using the Manhattan Distance heuristic.

Uniform Cost Search:

Uniform Cost Search is an uninformed search algorithm that aims to find the optimal solution by considering the cost of each action taken. In the context of the 8 puzzle problem, it explores the search space by expanding states with the lowest cumulative cost, starting from the initial state. This algorithm guarantees finding the optimal solution, but its efficiency depends on the branching factor and cost of actions. In this project, Uniform Cost Search was implemented as one of the search algorithms to solve the 8 puzzle problem, allowing for an analysis of its performance and comparison with other algorithms. It can be considered as a variant of A* search with no heuristic, therefore $f(n)=g(n)$ where $g(n)$ is simply the depth of the node.

A* search using the Misplaced Tile heuristic (Figure 2):

A* Search with Misplaced Tile Heuristic is an informed search algorithm commonly employed in solving problems such as the 8 puzzles. It utilizes a heuristic function, which estimates the cost from the current state to the goal state based on the number of misplaced tiles. The algorithm maintains a priority queue of states, prioritizing the states with the lowest sum of the cost to reach that state ($g(n)$) and the heuristic value ($h(n)$) i.e., $f(n) = g(n) + h(n)$. At each step, the algorithm selects the state with the lowest priority and expands it by generating successor states. The cost to reach each successor state is calculated ($g(n)$) along with the heuristic value ($h(n)$), based on the number of misplaced tiles. The successor states are then added to the priority queue. By integrating the misplaced tile heuristic, A* Search with Misplaced Tile Heuristic efficiently guides the search towards the goal state, faster than Uniform cost search.

A* search using the Manhattan Distance heuristic:

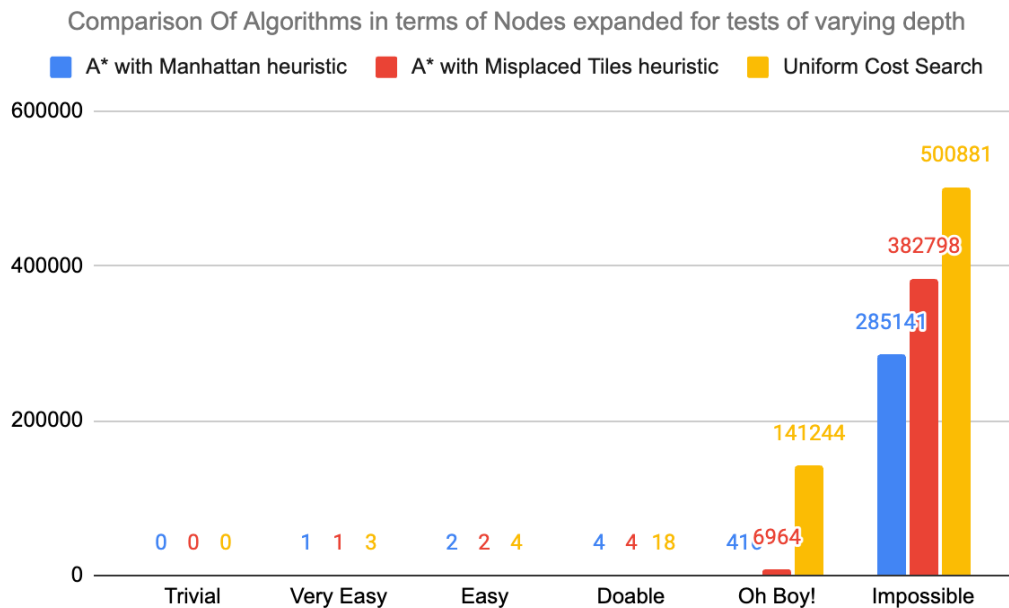
A* Search with Manhattan Distance Heuristic is an informed search algorithm commonly utilized to solve problems like the 8 puzzle. It incorporates a heuristic function that estimates the cost from the current state to the goal state based on the Manhattan distance of the misplaced tiles. The algorithm maintains a priority queue of states, prioritizing the states with the lowest sum of the cost to reach that state ($g(n)$) and the heuristic value ($h(n)$). i.e., $f(n) = g(n) + h(n)$. During each step, the algorithm selects the state with the lowest priority from the priority queue and expands it by generating successor states. The cost to reach each successor state ($g(n)$) and the heuristic value ($h(n)$), computed based on the Manhattan distance, are then added to determine the priority. By leveraging the Manhattan distance heuristic, A* Search with Manhattan Distance Heuristic effectively guides the search towards the optimal solution, faster and saving more space as compared Misplaced tile heuristic and Uniform Cost Search. Using the example initial state shown in Figure 2, not counting the position of 0, it can be seen that tiles 4,3, and 5 are out of place. Based on their positions in the puzzle and their goal state positions, $g(n)=8$.

1	2	4	1	2	3
3	0	6	4	5	6
7	8	5	7	8	0

Figure 2: Misplaced Tile heuristic, with Initial State (red) and Goal State (green)

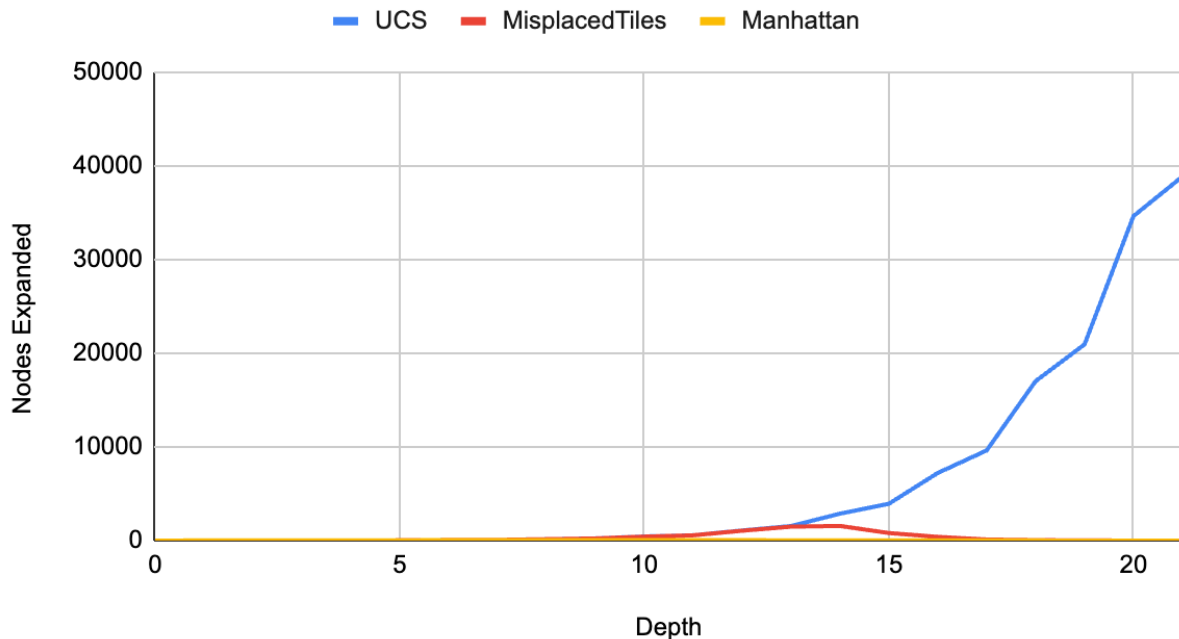
Algorithms		Uniform Cost Search		A* with Misplaced Tile Heuristic		A* with Manhattan Distance Heuristic	
Sample Test	Depth	Nodes Expanded	Max Queue Size	Nodes Expanded	Max Queue Size	Nodes Expanded	Max Queue Size
Trivial	0	0	0	0	0	0	0
Very Easy	1	3	5	1	3	1	3
Easy	2	4	6	2	3	2	3
Doable	4	18	16	4	4	4	4
Oh Boy!	22	141244	59800	6964	4247	416	253
Impossible	-	500881	71273	382798	48879	285141	31126

Table 1: Results of Sample Test Puzzles



Plot 1: Results of Comparison of nodes expanded for various algorithms and test cases.

Nodes Expanded at every depth For Oh Boy (depth 22) example



Plot 2: Results of Comparison of nodes expanded at every depth level for various algorithms and oh boy test case of diameter 22.

Comparison of algorithms on Sample Puzzles

The test samples provided by Dr. Eamonn Keogh for the project are as follows,

1. (1, 2, 3, 4, 5, 6, 7, 8, 0) // Level 0 - trivial.
2. (1, 2, 3, 4, 5, 6, 7, 0, 8) // Level 1 - very easy
3. (1, 2, 0, 4, 5, 3, 7, 8, 6) // Level 2 - easy
4. (0, 1, 2, 4, 5, 3, 7, 8, 6) // Level 3 - doable
5. (8, 7, 1, 6, 0, 2, 5, 4, 3) // Level 4 - oh boy
6. (1, 2, 3, 4, 5, 6, 8, 7, 0) // Level 5 – impossible

For simplicity, the game states are represented in 1-D array, here the first 3 elements belong to first row, next three belong to second row and the last three belong to third row. This representation was chosen as it will help save space. (Please let me know if you would like a 2D visualization).

In this project, we compared and contrasted three search algorithms—Uniform Cost Search, A* Search with Misplaced Tile Heuristic, and A* Search with Manhattan Distance Heuristic—for solving the 8-puzzle problem. By using various initial puzzle states with different solution depths as mentioned above, we analyzed the time complexity (number of nodes expanded) and space complexity (maximum number of nodes in the queue) of each algorithm.

Through our experiments, we executed each algorithm on multiple initial puzzle states, measuring the number of nodes expanded and the maximum queue size. We aimed to identify patterns and determine the most efficient algorithm for solving the 8 puzzle problem, considering the given puzzle configurations. The results and analysis as mentioned in Table 1, Plot 1 and Plot 2 will provide valuable insights into the performance and suitability of each algorithm. By conducting a thorough evaluation of time and space complexities across different puzzle states, we will draw conclusions about the optimal algorithm.

Table 1 summarizes the results obtained from our experiments. The table clearly demonstrates that the heuristic searches (A* with Misplaced Tiles and A* with Manhattan Distance) perform significantly better than the blind Uniform Cost Search. Additionally, the Manhattan Distance heuristic outperforms the Misplaced Tiles heuristic, as expected, due to the additional information it provides to the search function. The results indicate that the Manhattan Distance heuristic offers a sense of proximity between tiles and their goal positions, resulting in improved performance. Notably, the puzzles labeled "Impossible" and "Trivial" do not expand any nodes or enqueue the initial state due to pre-search checks for solvability and the goal state. Table 1 reveals that the Uniform Cost Search algorithm expands a significantly larger number of nodes compared to both heuristic searches. This stark difference highlights the effectiveness of the informed search algorithms in efficiently exploring the search space.

To provide visual clarity to the data, Plot 1 displays bar graphs representing the number of expanded nodes for each of the six sample initial states of the test puzzles. The plot further illustrates the superior performance of the heuristic searches compared to Uniform Cost Search, confirming our earlier observation. The results from our experiments and the corresponding analysis confirm that the heuristic searches, particularly A* with Manhattan Distance heuristic, are superior to Uniform Cost Search in terms of time complexity, as indicated by the significantly reduced number of expanded nodes. These findings highlight the importance of incorporating informed heuristics for optimizing the solution process in the 8-puzzle problem.

Based on Plot 2, which illustrates the number of nodes expanded at each depth for the "Oh boy" test sample with a diameter of 22, we can make several observations. Initially, up to depth 10, Uniform Cost Search (UCS) and A* Search with Misplaced Tile Heuristic exhibit similar expansion patterns, while A* Search with Manhattan Distance Heuristic requires a significantly lower number of nodes. This highlights the advantage of the Manhattan Distance heuristic in guiding the search towards the goal state more efficiently.

However, beyond depth 10, a notable change occurs. UCS experiences exponential growth in the number of nodes expanded at each level, while the number for Misplaced Tile Heuristic remains constant or even decreases slightly. In comparison, A* Search with Manhattan Distance Heuristic consistently maintains a significantly lower number of nodes compared to the other two algorithms. This demonstrates the superior performance of the Manhattan Distance heuristic in reducing the search effort, particularly as the depth increases.

An important observation is that UCS, Misplaced Tile and Manhattan Heuristic initially have similar node expansion numbers up to level 5, after which their patterns diverge. This suggests that the influence of algorithm choice on performance is more pronounced in deeper levels of the puzzle.

Furthermore, it is worth noting that for simpler puzzles with lower diameters, the number of nodes expanded remains relatively constant across the three algorithms. This indicates that the choice of algorithm has less impact on performance for puzzles with lower complexity.

Overall, Plot 2 provides empirical evidence supporting the efficacy of the Manhattan Distance heuristic in reducing the number of nodes expanded at each depth compared to UCS and the Misplaced Tile heuristic. This information underscores the importance of selecting an appropriate algorithm and heuristic based on the complexity of the puzzle to achieve optimal performance.

Test for a simple 16 puzzle game

To test how generic the code is I tested it for a couple of 16 puzzle games. For a game with low diameter, it was able to easily give the solution, however for a game with very high diameter it ran overnight and resulted in out of heap memory. Here's the input to use it for 16 puzzle game,

1. Select 2 to create your own.
2. Size of the puzzle = 16
3. Here's the input: Row 1 – 1 2 3 4; Row 2 5 6 7 8; Row 3 – 9 10 11 12; Row 4 – 13 14 0 15
4. Select an algorithm, 3 for A* with Manhattan Distance Heuristic
5. Enable backtracking if you wish

Output

Solution depth was 1; Number of nodes expanded: 1; Max Queue Size: 3

Conclusion

- The heuristic searches, A* Search with Misplaced Tile Heuristic and A* Search with Manhattan Distance Heuristic, outperform Uniform Cost Search (UCS) in terms of time complexity (number of nodes expanded) and space complexity (Queue Size) for the 8 puzzle problem.
- The Manhattan Distance heuristic significantly reduces the number of nodes expanded compared to the Misplaced Tile heuristic, indicating its superiority in providing more informed search decisions.
- UCS and Misplaced Tile Heuristic exhibit similar performance up to a certain depth, after which UCS experiences exponential growth in the number of nodes expanded, while the number for Misplaced Tile Heuristic remains relatively constant or even decreases slightly.
- A* Search with Manhattan Distance Heuristic consistently maintains a significantly lower number of nodes expanded at each depth compared to UCS and Misplaced Tile Heuristic, demonstrating its efficiency in guiding the search towards the goal state.
- The choice of algorithm has a greater impact on performance for deeper levels of the puzzle, while for simpler puzzles with lower diameters, the algorithm choice has minimal influence on the number of nodes expanded.

Sample Trace on an Easy Problem

Welcome to an 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own

2

Enter the size of the puzzle:

9

Enter the puzzle in the order asked

Enter the values for row 1

1

2

3

Enter the values for row 2

4

0

6

Enter the values for row 3

7

5

8

Select algorithm. (1) for Uniform Cost Search, (2) for A* with Misplaced Tile Heuristic, or (3) for A* with Manhattan Distance Heuristic

3

Do you want to see back tracking? (1) Yes or (2) No

1

The best state to expand with a $g(n) = 0$ and $h(n) = \text{Infinity}$ is

Depth: 0 Node Cost: Infinity

1 2 3

4 0 6

7 5 8

The best state to expand with a $g(n) = 1$ and $h(n) = 2.0$ is

Depth: 1 Node Cost: 2.0

1 2 3

4 5 6

7 0 8

The best state to expand with a $g(n) = 2$ and $h(n) = 2.0$ is

Depth: 2 Node Cost: 2.0

1 2 3

4 5 6

7 8 0

Solution depth was 2

Number of nodes expanded: 2

Max Queue Size: 5

Backtracking solution from Goal State to Initial State, State at Depth- 2

Depth: 2

1 2 3

4 5 6

7 8 0

Backtracking solution from Goal State to Initial State, State at Depth- 1

Depth: 1

1 2 3

4 5 6

7 0 8

Backtracking solution from Goal State to Initial State, State at Depth- 0

Depth: 0

1 2 3

4 0 6

7 5 8

Process finished with exit code 0

Sample Trace of Hard (Depth 16) Problem –

Welcome to an 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own

2

Enter the size of the puzzle:

9

Enter the puzzle in the order asked

Enter the values for row 1

1

6

7

Enter the values for row 2

5

0

3

Enter the values for row 3

4

8

2

Select algorithm. (1) for Uniform Cost Search, (2) for A* with Misplaced Tile Heuristic, or (3) for A* with Manhattan Distance Heuristic

3

Do you want to see back tracking? (1) Yes or (2) No

1

The best state to expand with a $g(n) = 0$ and $h(n) = \text{Infinity}$ is

Depth: 0 Node Cost: Infinity

1 6 7

5 0 3

4 8 2

The best state to expand with a $g(n) = 1$ and $h(n) = 12.0$ is

Depth: 1 Node Cost: 12.0

1 6 7

0 5 3

4 8 2

The best state to expand with a $g(n) = 1$ and $h(n) = 12.0$ is

Depth: 1 Node Cost: 12.0

1 0 7

5 6 3

4 8 2

The best state to expand with a $g(n) = 2$ and $h(n) = 12.0$ is

Depth: 2 Node Cost: 12.0

1 6 7

4 5 3

0 8 2

*//Here I deleted about 5 pages of trace to save space, this can easily be replicated by running
//the program and verified*

The best state to expand with a $g(n) = 6$ and $h(n) = 14.0$ is

Depth: 6 Node Cost: 14.0

1 3 0

5 7 6

4 8 2

4 0 8

The best state to expand with a $g(n) = 5$ and $h(n) = 16.0$ is

Depth: 5 Node Cost: 16.0

1 3 6

5 8 7

4 0 2

Depth = No. of nodes expanded at given depth

{0=1, 1=4, 2=8, 3=7, 4=9, 5=11, 6=15, 7=13, 8=12, 9=7, 10=3, 11=3, 12=2, 13=1, 14=1, 15=1}

Solution depth was 16

Number of nodes expanded: 98

Max Queue Size: 66

Backtracking solution from Goal State to Initial State, State at Depth- 16

Depth: 16

1 2 3

4 5 6

7 8 0

Backtracking solution from Goal State to Initial State, State at Depth- 15

Depth: 15

1 2 3

4 5 6

7 0 8

Backtracking solution from Goal State to Initial State, State at Depth- 14

Depth: 14

1 2 3

4 5 6

0 7 8

Backtracking solution from Goal State to Initial State, State at Depth- 13

Depth: 13

1 2 3

0 5 6

4 7 8

.

.

*//Here I deleted about 3 pages of trace to save space, this can easily be replicated by running
//the program and verified*

Backtracking solution from Goal State to Initial State, State at Depth- 3

Depth: 3

1 0 6

5 3 7

4 8 2

Backtracking solution from Goal State to Initial State, State at Depth- 2

Depth: 2

1 6 0

5 3 7

4 8 2

Backtracking solution from Goal State to Initial State, State at Depth- 1

Depth: 1

1 6 7

5 3 0

4 8 2

Backtracking solution from Goal State to Initial State, State at Depth- 0

Depth: 0

1 6 7

5 0 3

4 8 2

Process finished with exit code 0

Code – How to run ?

Prerequisites –

1. Java 1.8,
2. bash/shell terminal to run the script,
3. Maven (optional, in case you want to build the project)

Steps –

1. Download the project from the GITHUB URL <https://github.com/ss Shiv012/CS205Project>
2. You can just run the command the below command and you will be able to run the program
`sh start.sh`
3. In case you would like to build the project, you need to have maven. Use the below command,
`mvn clean install`
4. Follow the instructions by the driver program, you can also create a custom puzzle of custom size (it is generic).

Code – URL : <https://github.com/sshiv012/CS205Project>

```
package com.ucr.edu.cs205.project1;
import java.util.ArrayList;
import java.util.List;
//Represents a node in the search tree for the 8 puzzle problem.
public class Node {
    private final List<Integer> state;
    private final Node parentNode;
    private final int depth;
    //Methods removed to save space
}
```

Here's how I implemented the Operators as functions using Java

```
package com.ucr.edu.cs205.project1;

import java.util.ArrayList;
/**
 * Class containing operators for shifting tiles in the 8 puzzle problem.
 */
public class Operators {
    /**
     * Shifts a tile to the left in the given current node's state.
     *
     * @param currNode The current node.
     * @param position The position of the tile to be shifted.
     * @param boardEdge The edge length of the puzzle board.
     * @return The child node after shifting the tile to the left, or null if
     not possible.
     */
    public Node shiftTileLeft(Node currNode, int position, int boardEdge) {
        // If the position is on the extreme left, return null
        if (position % boardEdge == 0)
            return null;
        else {
            Node childNode = new Node(new ArrayList<>(currNode.getState()),
currNode);
            childNode.swapTiles(position, position - 1);
            return childNode;
        }
        //Abstracted Below methods to save space

    public Node shiftTileRight(Node currNode, int position, int boardEdge) {

    }

    public Node shiftTileUp(Node currNode, int position, int boardEdge) {

    }

    public Node shiftTileDown(Node currNode, int position, int boardEdge) {

    }
}
```

Here's how I implemented the Search Algorithm and the driver program using Java

```
package com.ucr.edu.cs205.project1;

import java.util.*;
import java.util.stream.Collectors;

/**
 * The main class implementing the 8 puzzle problem solving algorithm.
 */
public class EamonnsGodAlgorithm {

    /**
     * Enumeration representing the queue function for the search algorithm.
     */
    public enum QueueFunction {UCS, AStarMisplaced, AStarManhattan}

    private static int boardEdge = 0;
    private static int boardSize = 9;
    private boolean isBacktrackingEnabled = false;
    private static final List<Integer> goalState = new ArrayList<>();

    /**
     * Class representing the cost of a node in the priority queue.
     */
    private class NodeCost {
        float cost;
        Node node;

        public NodeCost(float cost, Node node) {
            this.cost = cost;
            this.node = node;
        }
    }

    /**
     * Expands the given node by generating its child nodes based on valid moves.
     *
     * @param node The node to expand.
     * @return The list of child nodes.
     */
    public List<Node> expand(Node node) {
        //Abstracted to save space, uses all 4 operators to expand the nodes.
    }
}
```

```

    /**
     * Performs the ultimate search algorithm given by Dr.Eamonn Keogh to
     solve the 8 puzzle problem.
     *
     * @param initialState The initial state of the puzzle.
     * @param queueFunction The queue function for the search algorithm.
     */
    public void ultimateSearch(Node initialState, QueueFunction
queueFunction) {
        int noOfExpandedNodes = 0;
        int maxQSize = 0;

        if (isGoalState(initialState)) {
            System.out.println("The puzzle is already solved");
            return;
        }
        PriorityQueue<NodeCost> nodes = new PriorityQueue<>(100, comparator);
        Map<String, Boolean> visited = new HashMap<>();
        nodes.add(new NodeCost(Float.POSITIVE_INFINITY, initialState));
        while (true) {

            if (nodes.isEmpty()) {
                System.out.println("No Solution found after expanding " +
noOfExpandedNodes + " with Queue Size " + maxQSize);
                return;
            }
            maxQSize = Math.max(maxQSize, nodes.size());
            NodeCost nodeCost = nodes.poll();
            if (nodeCost.node.getDepth() != localDepth)
            {
                localExpandedNodesMap.put(localDepth,
localExpandedNodesMap.getOrDefault(localDepth,0)+localExpandedNodes);
                localDepth=nodeCost.node.getDepth();
                localExpandedNodes=0;
            }

            if (isGoalState(nodeCost.node)) {
//Print Details of the solution
            }
            visited.put(getListOfNumbersAsString(nodeCost.node.getState()),
true);
            for (Node child : expand(nodeCost.node)) {
                if
(!visited.containsKey(getListOfNumbersAsString(child.getState()))) {
                    if (queueFunction.equals(QueueFunction.UCS))
                        nodes.add(new NodeCost(child.getDepth(), child));
                    else if
(queueFunction.equals(QueueFunction.AStarMisplaced))
                        nodes.add(new NodeCost(child.getDepth() +
calculateMisplacedTilesCount(child.getState()), child));
                    else
                        nodes.add(new NodeCost(child.getDepth() +
calculateTotalManhattanDistance(child.getState()), child));
                }
            }
        }
    }

```

```

        noOfExpandedNodes += 1;
        localExpandedNodes+=1;
    }

}

/**
 * Converts a list of integers to a string representation.
 *
 * @param intList The list of integers.
 * @return The string representation.
 */
public static String getListOfNumbersAsString(List<Integer> intList) {
    //Abstracted to save space
}

/**
 * Checks if a given number is a perfect square.
 *
 * @param number The number to check.
 * @return True if the number is a perfect square, false otherwise.
 */
public static boolean checkPerfectSquare(double number) {
    //Abstracted to save space,
}

/**
 * Calculates the count of misplaced tiles in the puzzle state.
 *
 * @param gameState The state of the puzzle.
 * @return The count of misplaced tiles.
 */
public int calculateMisplacedTilesCount(List<Integer> gameState) {
    //Abstracted to save space,
}

/**
 * Calculates the total Manhattan distance of tiles in the puzzle state.
 *
 * @param gameState The state of the puzzle.
 * @return The total Manhattan distance.
 */
public int calculateTotalManhattanDistance(List<Integer> gameState) {
    //Abstracted to save space,
}

/**
 * Checks if the given state is the goal state.
 *
 * @param currState The current state of the puzzle.
 * @return True if the state is the goal state, false otherwise.
 */
public Boolean isGoalState(Node currState) {
    return goalState.equals(currState.getState());
}

/**

```



```

    * Generates the goal state of the puzzle.
    */
    public void generateGoalState() {
        for (int i = 1; i < boardSize; i++)
            goalState.add(i);
        goalState.add(0);
    }
    /**
     * The main method/entrypoint to run the 8 puzzle problem solver.
     *
     * @param args The command line arguments.
     */
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Welcome to an 8-Puzzle Solver. Type '1' to use a
default puzzle, or '2' to create your own");
        int choice1 = sc.nextInt();
        int choice2;
        int choice3;
        QueueFunction algoChoice = QueueFunction.AStarManhattan;
        List<Integer> inputPuzzle = new ArrayList<>();
        Map<Integer, List<Integer>> eightPuzzle = new HashMap<>();
        //Abstracted to save space, add all sample test cases to eightPuzzle

        boardEdge = (int) Math.sqrt(boardSize);

        //Removed code to save space

        System.out.println("Select algorithm. (1) for Uniform Cost Search,
(2) for A* with Misplaced Tile Heuristic, or (3) for A* with Manhattan
Distance Heuristic");
        switch (sc.nextInt()) {
            case 1: {
                algoChoice = QueueFunction.UCS;
                break;
            }

            case 2: {
                algoChoice = QueueFunction.AStarMisplaced;
                break;
            }

            }

        System.out.println("Do you want to see back tracking? (1) Yes or (2)
No");
        choice3 = sc.nextInt();
        EamonnsGodAlgorithm search = new EamonnsGodAlgorithm();

        if (choice3 == 1)
            search.isBacktrackingEnabled = true;

        Node testGame = new Node(inputPuzzle, null);
        search.generateGoalState();
        search.ultimateSearch(testGame, algoChoice);
    }
}

```

ChatGPT Conversations 😊

S

```
eightpuzzle = [[1, 2, 3, 4, 5, 6, 7, 8, 0], # 0 - trivial
               [1, 2, 3, 4, 5, 6, 8, 7, 0], # 1 - impossible
               [1, 2, 3, 4, 5, 6, 7, 0, 8], # 2 - very easy
               [1, 2, 0, 4, 5, 3, 7, 8, 6], # 3 - easy
               [0, 1, 2, 4, 5, 3, 7, 8, 6], # 4 - doable
               [8, 7, 1, 6, 0, 2, 5, 4, 3]] # 5 - oh boy
```

Create a hashmap in java with indices 0-5 as key and the list of integers as value



Here is how you can create a HashMap in Java that maps Integer keys to a List of Integers:



```
java
Copy code

import java.util.*;

public class Main {
    public static void main(String[] args) {
        Map<Integer, List<Integer>> eightPuzzle = new HashMap<>();

        eightPuzzle.put(0, Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 0)); // 0 - trivial
        eightPuzzle.put(1, Arrays.asList(1, 2, 3, 4, 5, 6, 8, 7, 0)); // 1 - impossible
        eightPuzzle.put(2, Arrays.asList(1, 2, 3, 4, 5, 6, 7, 0, 8)); // 2 - very easy
        eightPuzzle.put(3, Arrays.asList(1, 2, 0, 4, 5, 3, 7, 8, 6)); // 3 - easy
        eightPuzzle.put(4, Arrays.asList(0, 1, 2, 4, 5, 3, 7, 8, 6)); // 4 - doable
        eightPuzzle.put(5, Arrays.asList(8, 7, 1, 6, 0, 2, 5, 4, 3)); // 5 - oh boy
    }
}
```

S

```
public int compareTo(NodeCost other) {
    return Float.compare(this.cost, other.cost);
}
```

is this comparator ascending or descending



The `compareTo` method as you've written it will result in an ascending order when used for sorting. This is because `Float.compare(f1, f2)` returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second, respectively.

In a sorting operation, a negative return value from `compareTo` means that the object being compared should come before the other object, zero means they are equal, and a positive return value means that the object should come after the other object.

So, `Float.compare(this.cost, other.cost)` will result in smaller `cost` values appearing earlier in the sorted order, which means the order is ascending.

S

I am getting higher depth than expected for all uniform cost search, A* with manhattan search and A* with misplaced tile heuristic, do you know where I could be wrong



There could be several reasons for getting a higher depth than expected in your search algorithms. It's hard to pinpoint the exact issue without seeing your specific code, but here are a few possibilities:



- Incorrect Heuristic Function:** If you're using A* with a heuristic function (either Manhattan Distance or Misplaced Tiles), make sure your heuristic function is implemented correctly. The heuristic function should never overestimate the cost to reach the goal (this property is known as being "admissible"). If it does, it could lead to higher-than-expected search depths.
- Inefficient Search Algorithm:** If you're using uniform cost search (also known as Dijkstra's algorithm), keep in mind that this algorithm can be quite inefficient, as it doesn't take into account the direction of the goal. This could result in exploring many more nodes than necessary, leading to a higher search depth.
- Incorrect Goal Test:** Make sure your goal test is correctly implemented. If the goal test is incorrect, the algorithm might miss the goal state and continue searching, resulting in a higher depth.
- Incorrect Expansion of Nodes:** generates the successors of a