Assignment 1 **CS205:** Artificial Intelligence, Dr. Eamonn Keogh

Harsh Gunwant
SID: 862393331
Email: hgunw001@ucr.edu
Date: 15th May 2023

In completing this assignment I consulted:

- The Blind search and Heuristic Search lecture slides and notes annotated from lecture.
- Java Documentation: https://www.tutorialspoint.com/java/java_documentation.htm
- What is 8 piece puzzle. (https://youtu.be/OXG04zPBsZ8)
- Solving 8 piece puzzle with Heuristic. (https://youtu.be/nmWGhb9E4es)
- Uniform Cost search (https://youtu.be/dRMvK76xQJI) (https://youtu.be/dvWk0vgHijs)
- A* with the Misplaced Tile heuristic (https://youtu.be/oAye4hI_sis)
- A* with the Euclidean Distance heuristic (https://youtube.com/shorts/N0FNwaJ7htA?feature=share)
- A* with Euclidean and Manhattan comparison. (https://youtu.be/YooweZVcNpE)

Outline of this report:

- Cover Page(This page)
- My report: Pages 2 to 9
- Sample trace on an easy problem, page 10
- Sample trace on an easy problem, page 10 and 11
- My code, pages to 11,12 and 13

CS250: Assignment 1: The Eight-Puzzle

Harsh Gunwant, SID 862393331, May-15-2021

Sliding tile puzzles require players to slide flat pieces on a board to create a specified end-configuration. The pieces to move may be simple shapes, colors, patterns, parts of a bigger picture (like a jigsaw puzzle), numbers, or letters. The "15 puzzle" is a frame of numbered square tiles with one tile missing. The problem involves sliding tiles to arrange them. The riddle can be textual, graphical, or tactile. Even if they use three dimensions or mechanically coupled components like Rubik's Cubes, sliding tile puzzles are two-dimensional.

Although there are variants, the puzzle usually involves finding the fewest moves to reach a goal state from a start state. The variant in this assignment to be tested was a 8-puzzle. The 8-puzzle is a variation of the 15-puzzle.

The 8-puzzle is a 3x3 grid with 9 squares. Eight numbered tiles occupy these squares, leaving one free for neighbouring tiles to go into. Tiles might be numbered 1–8 or contain a pattern or picture to reassemble. The challenge requires reaching a target configuration from an arbitrary configuration. Target configuration:

1. Ascending order: 1, 2, 3, 4, 5, 6, 7, 8, with the bottom right corner unfilled.
2. A picture, pattern, or zero when the tiles are properly aligned.
3. Slide a neighboring tile into the empty square to create a new empty space and slide a fresh tile into it.

Find the fewest moves to solve the puzzle from a given initial condition. Only half of the 362,880 8-puzzle configurations can be accessed from any given arrangement.

A puzzle game was available on the Play Store that help me build my intuition for sliding the puzzles.
https://play.google.com/store/apps/details?id=com.dopuz.klotski.riddle&hl=en_US&gl=US&pli=1

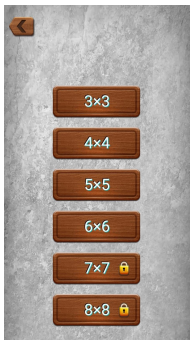


Fig.1 This represent the n-puzzle game that the user wants to play

Fig. 2 This represents the initial state

Fig. 3 The goal/final/target state

In the above figure 1, the n*n represents the (n*n - 1) puzzle game. For example, 3*3 represents the 8 puzzle game, and 4*4 represents the 15 puzzle game. Both figures 2 and 3 represent the initial state(randomized/shuffled) and the goal state, respectively.

INTRODUCTION

The following is a rundown of the three algorithms that must be used in order solve the 8-puzzle properly: Search with uniform cost, A* with the misplaced tile heuristic, and A* with the manhattan distance heuristic.

BACKGROUND

1. UNIFORM COST SEARCH

UCS is an artificial intelligence and computer science tree traversal or graph search algorithm. A Dijkstra version. UCS is used on trees/graphs with a target node, while Dijkstra's approach finds the shortest path from the starting node to all other nodes.
UCS is an informed best-first search algorithm that finds the lowest-cost path first. A priority queue selects the next node to examine based on travel costs from the start node.

Path cost in the 8-puzzle is the number of movements from the initial state to a state. Reach the target configuration with the fewest moves.
UCS always grows the lowest-path-cost node in the 8-puzzle, ensuring the lowest cost solution. If the 8-puzzle is enormous or the goal state is remote from the initial state, UCS may not be the most efficient algorithm. UCS searches in all directions without a heuristic because it has no goal state.

A*, which we will see later in the report uses UCS's say of optimality plus a heuristic function to direct the search, is generally more efficient for large or difficult 8-puzzles.
Below figure 4 shows an example of uniform cost search and how it reaches from the initial state S to the goal state G by finding the lowest cost path.

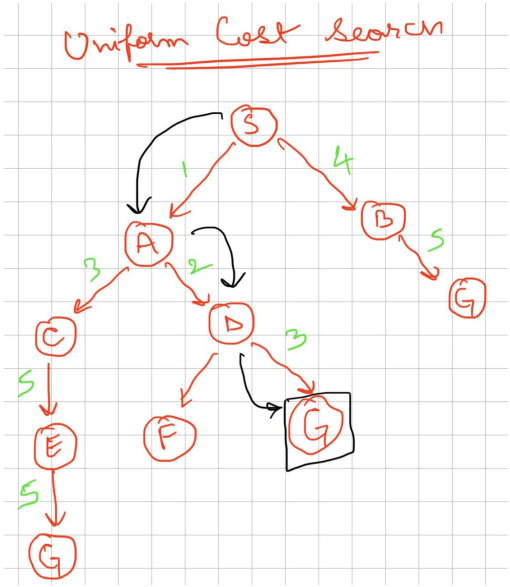


Fig. 4 Uniform Cost search

2. A* ALGORITHM

For pathfinding and graph traversal, including 8-puzzles, the A* technique is popular. It combines the benefits of Uniform Cost Search with Greedy benefits-First-Search.

A* utilizes a heuristic function to guide its search, which helps it find the target more efficiently than algorithms that lack this help. Heuristic function h(n) estimates the cost to reach the goal from node n. A* considers the cost to reach the current node from the start node, g(n), in addition to the heuristic function. f(n) = g(n) + h(n) estimates the optimal solution's cost via node n.

Two frequent 8-puzzle heuristics:

Misplaced Tiles: This heuristic counts misplaced tiles. It's simple yet inaccurate because it thinks each misplaced tile can be moved in one move.

Manhattan Distance: This heuristic is the sum of all tiles' distances from their destination places, calculated in moves along the grid. The misplaced tiles heuristic is less accurate because it doesn't account for the number of moves needed to move each tile.

A* solves the 8-puzzle faster by going through states that are more likely to lead to the objective. However, the heuristic function should not be overestimated (which could lead to non-optimal answers) or too complex to compute (which would negate the efficiency gained by directing the search). The Manhattan distance and misplaced tiles heuristics meet these requirements for the 8-puzzle and are discussed further below.

3. A* WITH THE MISPLACED TILE HEURISTIC

The Misplaced Tiles heuristic is a straightforward method that is frequently implemented in the process of solving sliding-tile puzzles like the 8-puzzle. It functions according to a simple premise, which is to count the number of tiles that are placed in the incorrect location.

The heuristic known as Misplaced Tiles is an easy one to understand and implement, and it functions adequately for the 8-puzzle. It determines how many tiles are placed incorrectly and tallies the total amount. This heuristic is acceptable, which means that it does not overstate the actual cost of achieving the goal. This is a fundamental criterion for A* to fulfill in order to guarantee the discovery of an optimal solution.

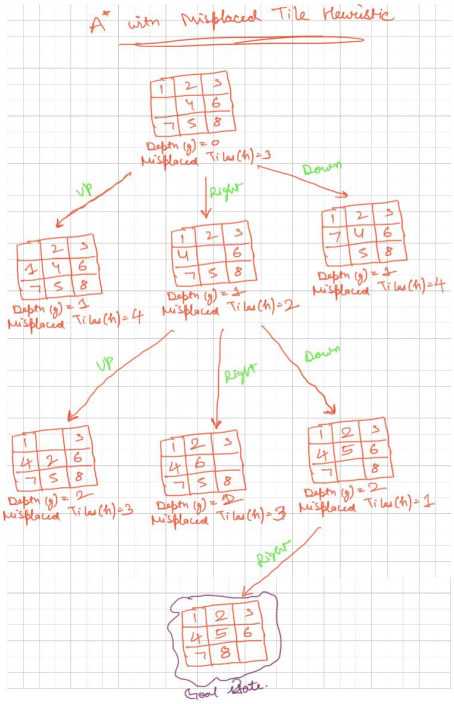


Fig. 5: A* with Misplaced Tile heuristic

The above figure 5 shows the initial state of the 8 puzzle and how it comes to the goal state by looking at each tile in the current state of the puzzle. If a tile is not in its goal position, count it as a misplaced tile. The total number of misplaced tiles is the value of the heuristic.

4. A* WITH THE MANHATTAN DISTANCE HEURISTIC

Here, we need to understand what manhattan distance is.
Manhattan Distance is a distance metric(like Euclidean Distance) also known as the L1 norm or the taxi-cab distance. Lets take an example to understand it, Imagine that you're a taxi driver in a city with a grid-like street layout (like Manhattan), and you need to get from point A to point B. The shortest distance is not a straight line, as you're limited by the grid layout of the streets. You have to move horizontally and vertically along the grid lines. The total number of blocks you would need to cross to get from A to B is the Manhattan distance.

With respect to the 8-puzzle the Manhattan Distance heuristic is the sum of the distances that all tiles are from their respective goal positions, with each distance being evaluated in terms of the number of moves that are taken along the grid. The distance between each tile and its target place is determined by adding the number of rows and columns that separate it from that position.

IMPLEMENTATION

This is a brief description of how we have implemented the 3 algorithms, the detailed description is commented in the code itself.

In the given Java class **Node**, an instance of **Node** represents a state of the 8-puzzle. Each Node keeps track of its parent node, the configuration of its puzzle, the coordinates of the empty tile(**xCord and yCord**), the cost to reach this state, and whether the Manhattan Distance heuristic is being used(**boolean**).

The class includes methods for calculating the heuristic value (using either the Misplaced Tiles heuristic or the Manhattan Distance heuristic, depending on the ManhattanUsed flag), for calculating the Manhattan Distance and the number of misplaced tiles, and for checking if given coordinates are valid within the puzzle grid.

Let's go over how the provided class incorporates uniform-cost search, A* with Misplaced Tile heuristic, and A* with Manhattan Distance heuristic:

Uniform-Cost Search: The costs variable in the **Node** class represents the cost to reach the current state from the start state (i.e., the number of moves made so far). When creating a new Node (i.e., moving to a new state), this cost is incremented by the cost of the move, which is assumed to be 1. The **compareTo** method compares nodes based on their costs, so if we use a priority queue to store nodes and always dequeue the node with the smallest cost, we are effectively implementing uniform-cost search.

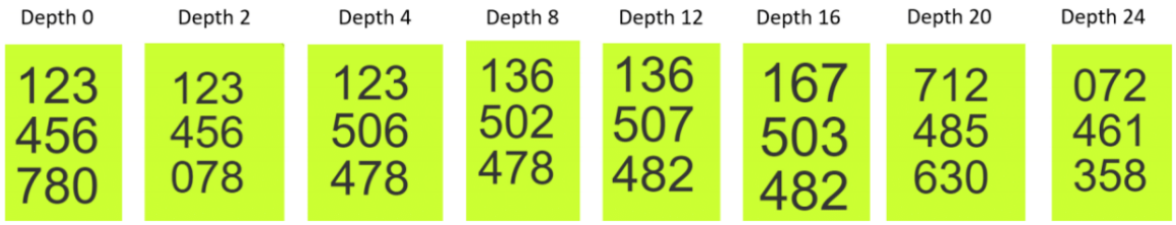
A* with Misplaced Tile Heuristic: If we set **useHeuristic** to true and **ManhattanUsed** to false when creating a new Node, the constructor will add the number of misplaced tiles (calculated by **calculateMisplacedTiles**) to the costs. This means costs will represent the sum of the actual cost to reach the current state and the estimated cost to reach the goal state, making this an implementation of the A* algorithm with the Misplaced Tile heuristic.

A* with Manhattan Distance Heuristic: If we set **useHeuristic** to true and **ManhattanUsed** to true when creating a new Node, the constructor will add the Manhattan Distance (calculated by calculateManhattanDistance) to the costs. This means costs will represent the sum of the actual cost to reach the current state and the estimated cost to reach the goal state, making this an implementation of the A* algorithm with the Manhattan Distance heuristic.

It's also worth noting that the **isValid** method is useful for generating valid neighboring states of the current state by ensuring we don't try to move the empty tile outside of the puzzle grid.

COMPARISON OF ALGORITHMS ON SAMPLE PUZZLES

To evaluate the results of these search algorithms, we can consider several factors. Firstly, we can measure the number of nodes expanded or generated during the search process, as it indicates the efficiency of the algorithm in exploring the search space. Secondly, the time taken (in java it was in milliseconds) to find a solution can be measured, which reflects the algorithm's computational efficiency. Additionally, we can compare the path cost of the solution found by each algorithm to determine if it is optimal. By considering these factors, we can assess the performance and efficiency of each algorithm in solving the 8 puzzle problem at different depths.



As shown in the figure 6 above, Dr. Keogh provided us with following test cases, sorted by depth of optimal solution. In the figure 7 and figure 8 given below, there is a comparison of all the 3 algorithms.

Uniform Cost Search:

For the 8 puzzle problem with various depths, UCS will find the optimal solution by exploring all possible states. Since UCS considers only the cost of reaching each state, it does not use any heuristic information. Therefore, the performance of UCS will depend on the branching factor and the specific configuration of the puzzle.

If the depth of the puzzle is 0, UCS will terminate immediately since the goal state is already reached. For depths 4, 8, 12, 16, 20, and 24, UCS will continue exploring the search space until it finds the goal state, expanding nodes based on the cumulative cost.

A* Search with Misplaced Tile Heuristic:

For depths 0, 4, 8, 12, 16, 20, and 24, A* search with the misplaced tile heuristic will perform better than UCS. The misplaced tile heuristic provides admissible and consistent estimates, guiding the search towards the goal state. A* search will prioritize expanding the nodes with the lowest estimated cost (heuristic value plus cumulative cost), leading to a more efficient search compared to UCS.

A* Search with Manhattan Distance Heuristic:

For depths 0, 4, 8, 12, 16, 20, and 24, A* search with the Manhattan distance heuristic will also perform better than UCS. The Manhattan distance heuristic is admissible and consistent, providing a more accurate estimate of the cost to reach the goal state. A* search using this heuristic will guide the search more effectively and find the optimal solution more efficiently compared to UCS.

In summary, both A* search algorithms with the misplaced tile heuristic and Manhattan distance heuristic outperforms Uniform Cost Search for the 8 puzzle problem at depths 0, 4, 8, 12, 16, 20, and 24. The specific performance depends on the branching factor and the particular configuration of the puzzle, but A* search with informed heuristics will generally provide faster and more efficient solutions.

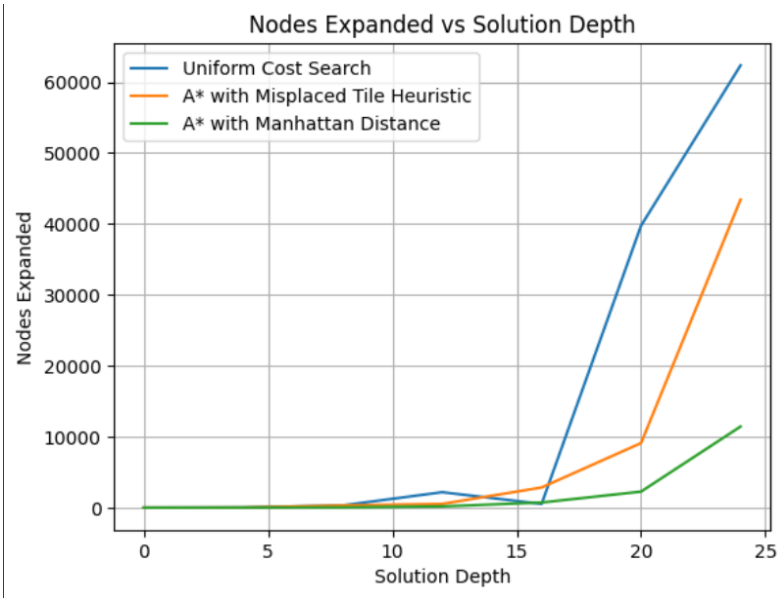


Fig. 7: Nodes Expanded vs Solution Depth

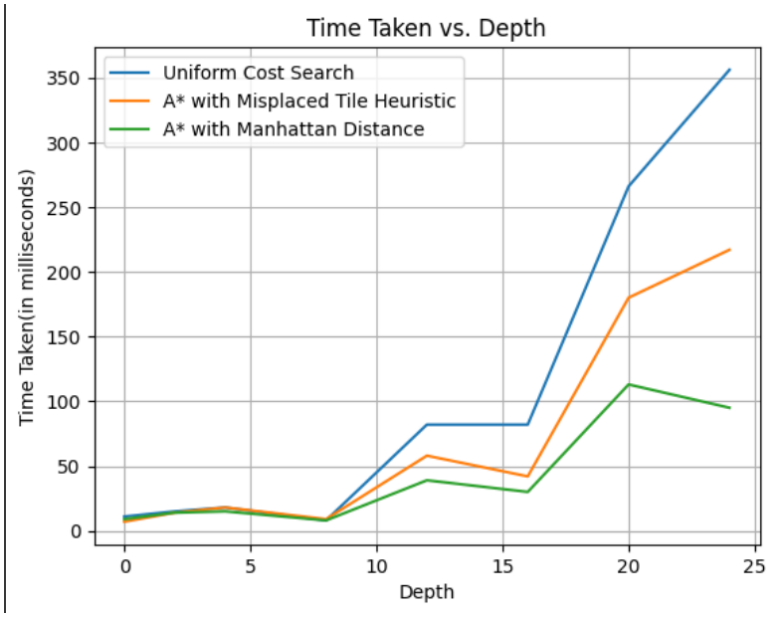


Fig. 8:Time taken vs Depth

CONCLUSION

We may summarize the three algorithms, Uniform Cost Search (UCS), A* with misplaced tile heuristic, and A* with Manhattan distance heuristic, as follows:

Depth: The 8 puzzle becomes more complex with depth. Depth exponentially increases the number of states and search tree branching factor. This expands the search space, making it harder for all three algorithms to solve. A* search algorithms with informed heuristics (misplaced tile and Manhattan distance) can lead the search more efficiently toward the goal state than UCS (see graphs above), perhaps managing deeper depths better.

Time vs Depth Graph: For all three techniques, solution time grows with depth. A* search algorithms with informed heuristics (misplaced tile and Manhattan distance) have a better time-depth trade-off than UCS. A*

search uses informed heuristics to pick promising paths, lowering search time compared to UCS, especially as depth increases.

Nodes Expanded versus Depth Graph: All three techniques expand or generate more nodes as depth increases. A* algorithms with informed heuristics expand fewer nodes than UCS. Informed heuristics help A* search prioritize desirable states and avoid less likely pathways. A* with informed heuristics expands fewer nodes, improving efficiency.

Comparing A* with Manhattan Distance with A* with Misplaced Tile Heuristic

The Manhattan distance heuristic provides a more informed estimate of the cost to reach the goal state. It takes into account the actual distances between each tile and its goal position, considering only horizontal and vertical movements. This heuristic tends to guide the search more effectively, leading to faster convergence and fewer unnecessary explorations of the search space.

On the other hand, the misplaced tile heuristic simply counts the number of tiles that are not in their goal positions. While this heuristic can still provide useful guidance, it may be less precise than the Manhattan distance heuristic. It does not consider the specific distances between the tiles and their goal positions, potentially leading to less optimal paths being explored.

Therefore, in most cases of the different depths, A* search with the Manhattan distance heuristic is expected to outperform A* search with the misplaced tile heuristic.

The Following is a traceback of an easy puzzle:

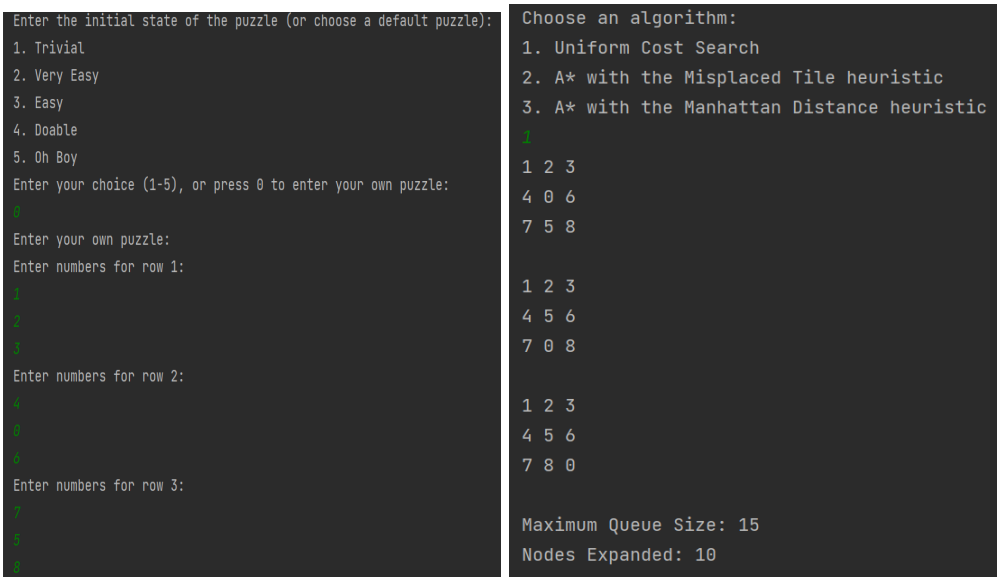


Fig. 9: This is a traceback of an easy puzzle.

In figure 9 we have implemented an easy puzzle using uniform cost search where 10 nodes were expanded and maximum queue size was 15.

The Following is a traceback of a hard puzzle (depth 16):

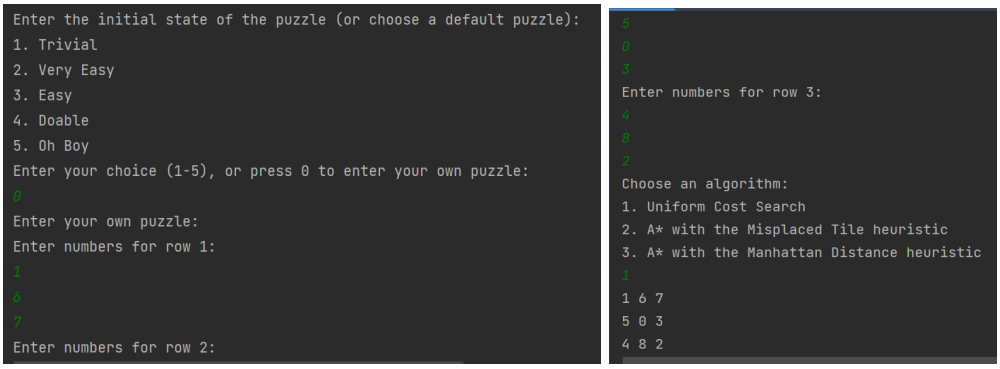


Fig 10. This is a traceback of a hard puzzle with depth =16.

In figure 10 and we have implemented a hard puzzle using uniform cost search (option 1) where 21946 nodes were expanded and the maximum queue size was 12041 as can be seen in figure 11 below.

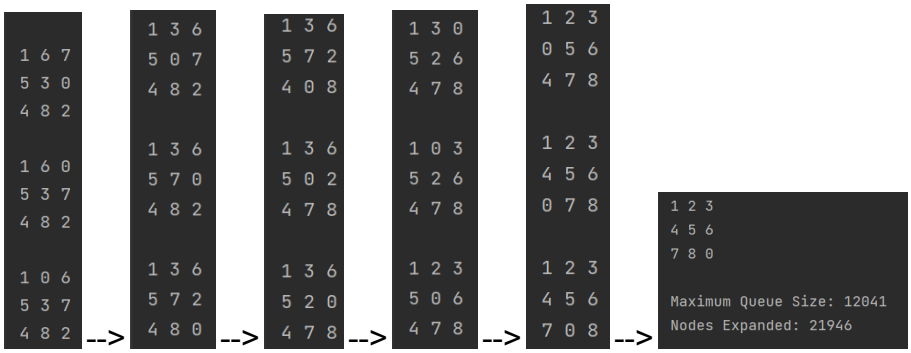
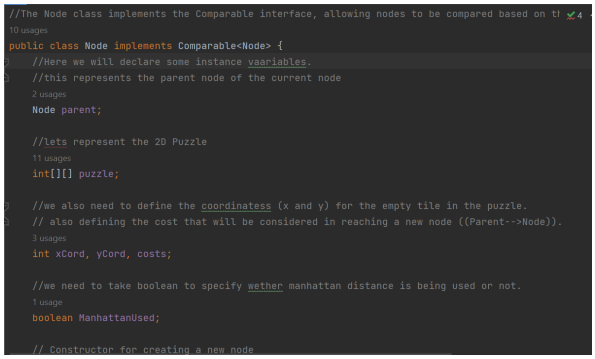
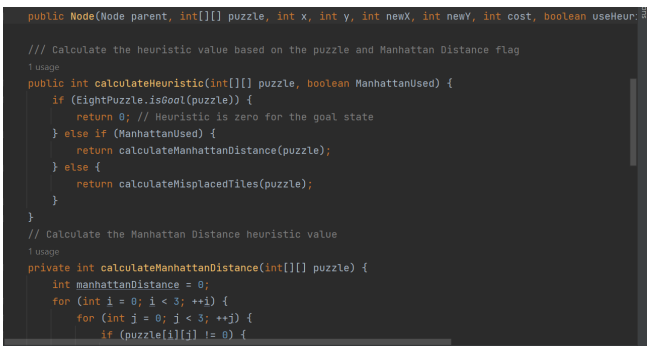


Fig 11: This is a traceback of a hard puzzle(continued) with depth =16.

I have also provided my code URL → https://github.com/harshgunwant/AI_Project_SearchAlgo

SNAPSHOTS OF CODE:





```java
private int calculateMisplacedTiles(int[][] puzzle) {
    int misplacedTiles = 0;
    int tile = 1;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (puzzle[i][j] != 0 && puzzle[i][j] != tile) {
                misplacedTiles++;
            }
            tile++;
        }
    }
    return misplacedTiles;
}

@Override
public int compareTo(Node other) {
    // Compare nodes based on their costs
    return this.costs - other.costs;
}

// Check if the given coordinates are valid within the puzzle grid
1 usage
```

The above code is for the Node class.

```java
public class EightPuzzle {
    no usages
    public static int maxQueueSize; // Variable to track the maximum size of the priority queue
    no usages
    public static int nodesExpanded; // Variable to track the number of nodes expanded

    // Method to check if the puzzle is solvable
    1 usage  new *
    public static boolean isSolvable(int[][] puzzle) {
        int inversions = 0; // Variable to count the inversions
        List<Integer> puzzleNumbers = new ArrayList<>(); // List to store the numbers of the puzzle

        // Iterate over the puzzle and add the numbers to the list
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                puzzleNumbers.add(puzzle[i][j]);
            }
        }
```

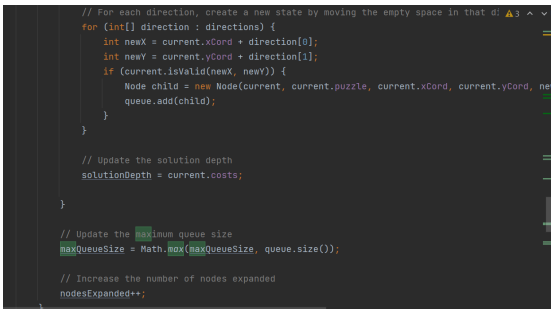
```java
    // Convert the list to an array
    Integer[] array = new Integer[9];
    puzzleNumbers.toArray(array);

    // Count the inversions
    for (int i = 0; i < 9; ++i) {
        for (int j = i + 1; j < 9; ++j) {
            if (array[i] != 0 && array[j] != 0 && array[i] > array[j]) {
                inversions++;
            }
        }
    }

    // Return true if the number of inversions is even, false otherwise
    return inversions % 2 == 0;
}

// Method to check if the current state of the puzzle is the goal state
```

```java
public static boolean isGoal(int[][] puzzle) {
    // If the last position is not 0, return false
    if (puzzle[2][2] != 0) {
        return false;
    }

    int count = 1; // Initialize count with 1
    // Iterate over the puzzle and check if the numbers are in order
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            // If count is less than or equal to 8 and the number in the current position is not eq
            if (count <= 8 && puzzle[i][j] != count) {
                return false;
            }
            count++;
        }
    }
```

The above and below code are from the Eight Puzzle Class.

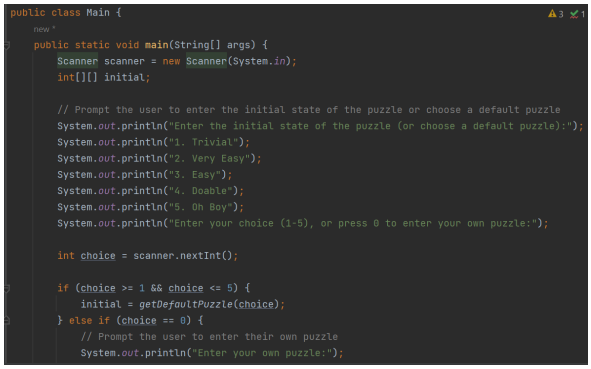
```java
public static void solve(int[][] initial, int x, int y, boolean useHeuristic, boolean useManhattan
    // Create a priority queue and add the initial state to the queue
    PriorityQueue<Node> queue = new PriorityQueue<>();
    queue.add(new Node( parent: null, initial, x, y, x, y,  cost: 0, useHeuristic, useManhattan));

    // Array to represent the four directions (right, down, left, up)
    int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    // Set to store the states that have already been explored
    Set<String> explored = new HashSet<>();

    // Variables to store the solution depth, maximum queue size and number of nodes expanded
    int solutionDepth = 0;
    int maxQueueSize = 0;
    int nodesExpanded = 0;

    // Iterate until the queue is empty
    while (!queue.isEmpty()) {
        // Remove the node with the highest priority from the queue
```

```java
// Iterate until the queue is empty
while (!queue.isEmpty()) {
    // Remove the node with the highest priority from the queue
    Node current = queue.poll();

    // If the current state is the goal state, print the solution and the statistics and return
    if (isGoal(current.puzzle)) {
        printSolution(current);
        printStatistics(maxQueueSize, nodesExpanded, solutionDepth);
        return;
    }
    // Convert the current state to a string
    String currState = Arrays.deepToString(current.puzzle);

    // If the current state has not been explored yet, add it to the set of explored states
    if (!explored.contains(currState)) {
        explored.add(currState);

        // For each direction, create a new state by moving the empty space in that direction a
        for (int[] direction : directions) {
            int newX = current.xCord + direction[0];
```



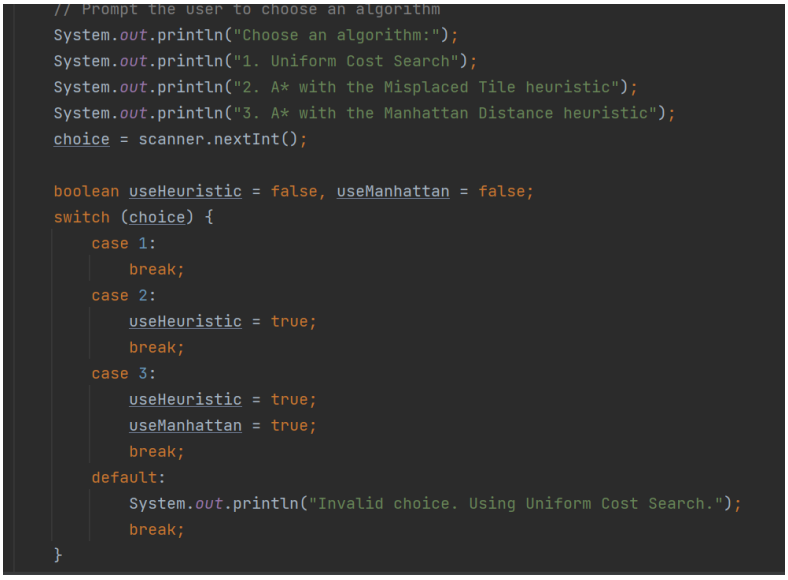
```
        // For each direction, create a new state by moving the empty space in that d
        for (int[] direction : directions) {
            int newX = current.xCord + direction[0];
            int newY = current.yCord + direction[1];
            if (current.isValid(newX, newY)) {
                Node child = new Node(current, current.puzzle, current.xCord, current.yCord, ne
                queue.add(child);
            }
        }

        // Update the solution depth
        solutionDepth = current.costs;

    }

    // Update the maximum queue size
    maxQueueSize = Math.max(maxQueueSize, queue.size());

    // Increase the number of nodes expanded
    nodesExpanded++;
}
```

The above code is implemented as present in Slides of Dr. Keogh for General Search Algorithm.



```
public class Main {
    new *
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int[][] initial;

        // Prompt the user to enter the initial state of the puzzle or choose a default puzzle
        System.out.println("Enter the initial state of the puzzle (or choose a default puzzle):");
        System.out.println("1. Trivial");
        System.out.println("2. Very Easy");
        System.out.println("3. Easy");
        System.out.println("4. Doable");
        System.out.println("5. Oh Boy");
        System.out.println("Enter your choice (1-5), or press 0 to enter your own puzzle:");

        int choice = scanner.nextInt();

        if (choice >= 1 && choice <= 5) {
            initial = getDefaultPuzzle(choice);
        } else if (choice == 0) {
            // Prompt the user to enter their own puzzle
            System.out.println("Enter your own puzzle:");
```

```
            System.out.println("Enter your own puzzle:");
            initial = new int[3][3];
            for (int i = 0; i < 3; ++i) {
                System.out.println("Enter numbers for row " + (i + 1) + ":");
                for (int j = 0; j < 3; ++j) {
                    initial[i][j] = scanner.nextInt();
                }
            }
        } else {
            System.out.println("Invalid choice. Using the default puzzle.");
            initial = getDefaultPuzzle( choice: 1);
        }

        int x = -1, y = -1;
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                if (initial[i][j] == 0) {
                    x = i;
                    y = j;
                    break;
                }
            }
```



```
// Prompt the user to choose an algorithm
System.out.println("Choose an algorithm:");
System.out.println("1. Uniform Cost Search");
System.out.println("2. A* with the Misplaced Tile heuristic");
System.out.println("3. A* with the Manhattan Distance heuristic");
choice = scanner.nextInt();

boolean useHeuristic = false, useManhattan = false;
switch (choice) {
    case 1:
        break;
    case 2:
        useHeuristic = true;
        break;
    case 3:
        useHeuristic = true;
        useManhattan = true;
        break;
    default:
        System.out.println("Invalid choice. Using Uniform Cost Search.");
        break;
}
```

This is the code for the main class