Student ID: 30012627
Intelligent Systems Coursework

Approach:

This investigation explores several uninformed and informed search methods(Breadth-first, Depth-First, Iterative Deepening, Breadth First graph implementation as well as Depth First Search graph implementation) to solve the n * n grid problem, using programming language Java. The approach to the exploration is as follows: four classes have been created; Program.java, Node.java, Search.java as well as NodePriorityComparator.java. The code can be found in section 5 of the Appendices.

The Program class contains the main method, which calls a search method on the instance of the Search class to launch the program. Each node is represented by a 2D puzzle of size n*n, where n can be set to any number. Tile A is represented by Integer 1, B by 2, C by 3 and the agent by 4. The start state is illustrated in Figure 1. Each search method takes the root node and its configuration puzzle as a parameter. The Node class handles and defines all properties of a node whenever it is created. It contains the goal configuration, methods to set and get the depth, total cost(A* search) of each node, as well as the expandNode method which produces all possible children of a currentNode; by calling methods moveRight , moveReft, moveUp or moveDown. Each move method takes the puzzle configuration and the agent as parameters, and only creates a new node with the new respective puzzle configuration if the agent can move accordingly on the board. Then, this node is added to the list of children of the current node and the parent of each of these children is set to the current node. Additionally, the Node class contains the GoalTest method which is called every time a node is popped from the stack/queue to check whether its puzzle configuration matches the goal puzzle configuration. Heuristic one and two for A* Search are also defined in this class. The NodePriorityComparator class extends the Comparator class. The comparator is used by the priority queue in A* search to store the nodes according to cost values.

BreadthFirstSearch has been implemented using a Queue(FIFO), using the Java Linked List. The depth of the root node is set to 0 and the number of generated nodes is initially set to 0. Reference to the root node is passed onto the current node, which is then pushed into the queue. Now, a while loop is used to guard that the program only enters to loop when the goal is not found and while the queue is not empty. The head of the queue is removed from the queue and set to the current node. If the current node matches the configuration of the goal node, then the program exits the loop and prints the path to the solution. Otherwise, the program expands the current node, and retrieves all its children. The number of generated nodes is the running count of generated nodes + number of newly created children. Now, each child is then pushed into the queue and the program loops again. This implementation ensures that all the nodes in a particular level of the tree are explored before the next depth level is explored.

DepthFirstSearch is implemented in a very similar way. However, a stack(LIFO) was used rather than a queue. As a result, the last child that is pushed into the stack is popped first. This ensures that the nodes on the furthest right of the tree are explored in increasing depth. The initial limitation to this search strategy was that nodes with the same puzzle configuration were being explored repeatedly. The order in which a node was expanded was right, left, up and then down. Then, using the given start state,the agent will move left rather than move up, and then right. With the current implementation, the agent

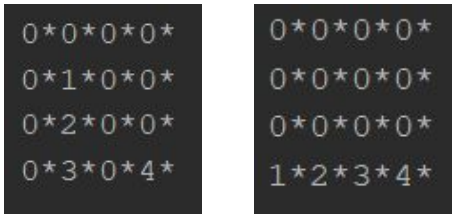
would move back left and then right again. This sequence of actions will continue indefinitely due to the uninformed nature of the search. The approach taken to handle this limitation was to shuffle the children before adding them to the queue one by one. In this way the order at which the children are picked is not maintained. This new approach is labelled as RandomisedDepthFirstSearch.

Iterative Deepening search is analogous to conducting DepthFirstSearch from a depth limit of 0 to n, where n is the depth of the goal configuration.While the goal configuration is not found, if the stack is not empty and the depth of the current node is below the limit, the node is expanded. If the goal configuration is found, the path to the goal solution is returned. If the stack is emptied, the search progresses to the next depth limit.

A priority queue was used to implement the A* heuristic(informed) search algorithm. The priority queue contains nodes sorted according to their total cost. The total cost of each node is the sum of the depth of the node and the result of applying a heuristic. Hence, the node with the least total cost is dequeued first. Two heuristics were implemented for this algorithm- The Manhattan Distance and the number of misplaced tiles.The number of misplaced tiles(Heuristic 1) compares each tile in current puzzle configuration with the goal puzzle configuration. If they don't match then the running total is incremented by 1. The approach to calculating the Manhattan Distance(Heuristic 2) was initially finding the indices in which tiles A,B, and C were located in the current puzzle and goal puzzle. If these indices did not match, the sum of vertical and horizontal distances between indices of the corresponding tiles were calculated and added to the running total.

Evidence:
Two scenarios have been used to illustrate the four search methods in operation. The first scenario is shown in Figure 1, below. This is the start and end state given in the problem specification. The latter is a 3*3 grid with problem size 3, as shown in Figure 2. A smaller problem size has been chosen to trace all the possible moves by BFS and DFS, to the end solution.

Figure 1:



Start state End state

Scenario 1

Figure 2:

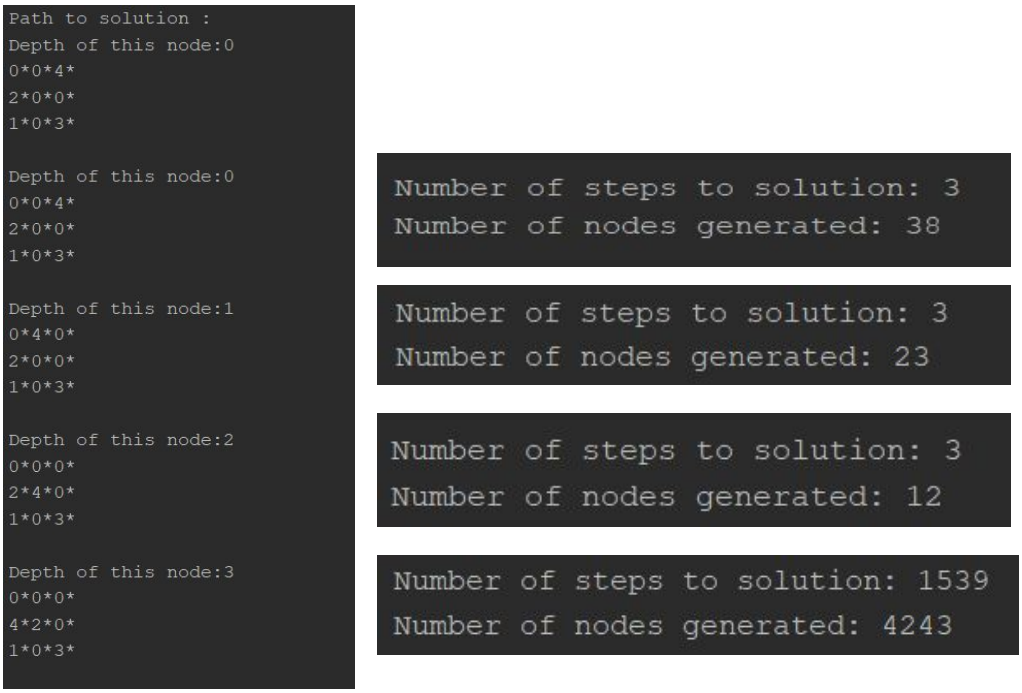
Start state End state

Scenario 2

Since both IDS and BFS iterate through all the nodes at each depth level, the path to solution by these methods is always optimal. Additionally, in this case, since the heuristics 1 and 2 are admissible and do not overestimate the cost, the path returned by A* is optimal as well. Hence, all three search algorithms have the same path to solution. The path to solution taken by BFS, IDS, and A* for Scenario 1 has been illustrated in Figure 3. On the right side of the figure, the number of steps taken from the start state to the goal state(path to solution) and the number of nodes generated by BFS, IDS, A* and Random Depth-First are shown respectively, from top to bottom. Similarly, this is shown for scenario 2 in Figure 4. Please refer to the Appendices (Sections 1 and 2) for the debugging output of both scenarios. For each search the name of the search appears in the beginning of the debugging output. In addition to the current configuration of the puzzle, the depth of the node is printed for ease in understanding. The output for Breadth First and Iterative Deepening Searches have been limited to the depth 3 for Scenario 1. Heuristic one has been chosen to illustrate the debugging output of A*search.In addition to displaying the depth of the picked node, the total cost associated with each node is displayed too. The output should be read down a column, and across the page.

Figure 3:

Figure 4:



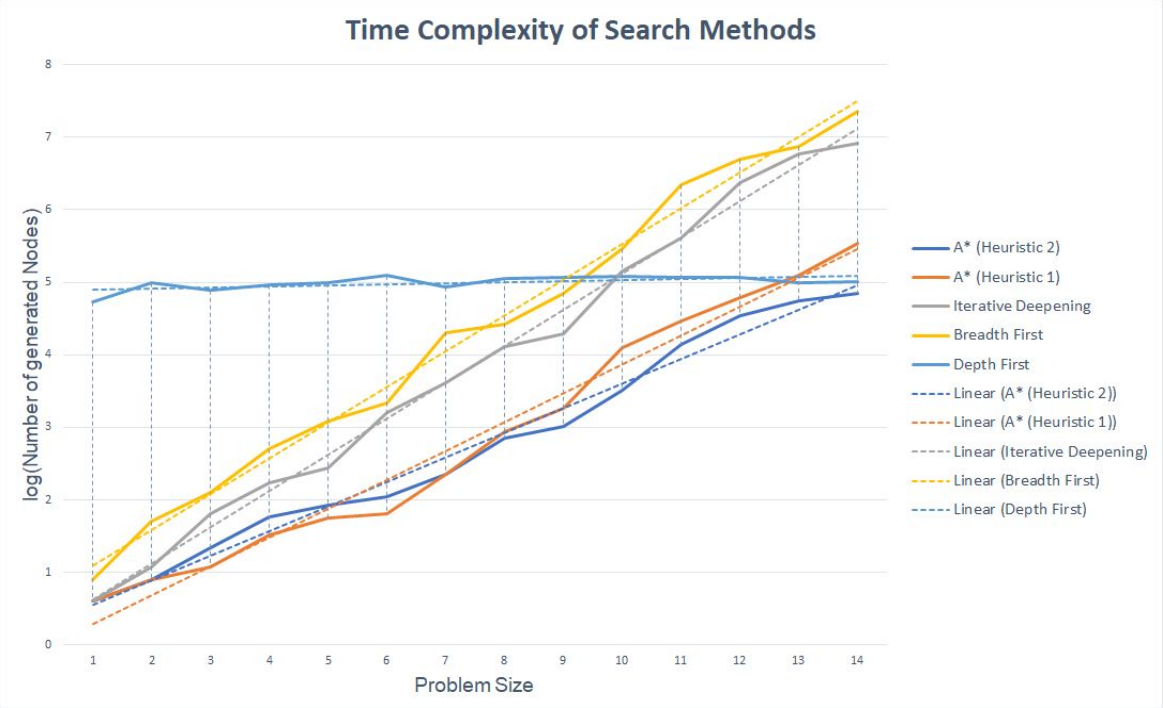
Scalability Study:

The graph below(Figure 5) shows the relationship between the problem size and log(generated nodes) for the four different search methods. The log values were plotted instead for the ease of comparison between the different search methods, on one scale, since the number of generated nodes varies significantly. The plot of number of generated nodes versus problem size can be found in Section 3(Figure 7) of the Appendices. The problem size has been altered by making each node configuration in the path from the start state to the end state (as shown in Figure 3) the start state. For example, the state in position 14 is of problem size 1, whereas the state at position 0 is of problem size 14.

Let us first compare BFS, IDS, and A*. All the graphs have a strong linear relationship between the problem size and log(number of generated nodes). As a result, it can be deduced that the number of generated nodes grows exponentially with the problem size. Now, there is a significant difference between the number of generated nodes for each search method. The number of nodes generated for problem size 14 by BFS is above 23 million, compared to just above 8 million for IDS only around 70,000 for A* search(Heuristic 1), 340735 (Heuristic 2).The number of nodes generated for problem size 7 by BFS is above 200 000, compared to just above 4000 for IDS and only 228 for A*(both heuristics). This is reflected by the graph below. Breadth First Search has the worst time complexity, followed by Iterative Deepening. A* heuristic search is a much better search method to tackle the problem. More specifically, Heuristic 2 gives better results for a greater problem size. However, if we consider the space complexities of these search methods, A* has the worst space complexity as all the nodes generated are all stored in memory.

Linear trends have been found for all search algorithms except Randomised DFS, in the graph below. The results for Randomised DFS have been plotted after calculating the average of 50 trails. As shown in the graph below, the change in problem size does not affect the time complexity of the search method. The number of generated nodes nodes does not exceed 130,000, as a result, one may argue that DFS is a good search strategy. However, due to the random nature of DFS in this case, it is difficult to make such an assumption.

Figure 5:



For example, for problem size 2, BFS generated 51 nodes whereas DFS generated 97060 nodes in one trial and 8 in another.

Limitations:
There are several limitations to this investigation. All the search methods contain some duplicate code; for example: setting the root in every method, updating its depth to 0 and setting the initial current node to the root, then checking if the current node has the goal configuration etc. Especially the implementation of BFS and DFS are almost analogous, except a stack is used for DFS whereas a queue has been used for BFS. In hindsight, this code duplication could be avoided by creating a class for each search method and having them extend a superclass Search class. The advantage of using encapsulation here is that common fields and methods do not have to be declared repeatedly, they can be defined once in the Search class and all the different Search classes can either add the fields or methods in the superclass or override its methods.

It is also important to note that the number of generated nodes for any particular problem size is dependant on the start state configuration and end state configuration. Given two different start configurations for the same goal configuration (with the same problem size), it is not necessary that the number of nodes generated in reaching goal state from start state is the same for both problems. The investigation would be more comprehensive, had there been graphs (as shown above) plotted for multiple same problem size scenarios for comparison or a single graph displaying the average results. Similarly, the results shown above can be compared with scenarios where an obstacle is placed on the board or with different board sizes.

The number of generated nodes may also vary depending on the order in which the agent moves right, left, up or down. This is because a node configuration is only compared with the goal configuration after it has been dequeued/popped from the queue/stack. The current order in which agent is moved is right, left,up then down. Let's assume that the agent were to reach the goal state upon moving down.With the current implementation, this would take 4 moves, however, if the agent were to move down first, it would only take 1 move to reach the goal state. Hence, all the four possible orders can be compared to find the least number of generated nodes.

Given the nature of the search algorithms, it is not feasible to track the generated or expanded nodes for bigger problem sizes. This makes it difficult to debug the output of the program. Similarly, due to the random nature of Random Depth First Search, the output changes every time the program is run. Due to the tree structure of the uninformed search algorithm, the algorithm ends up indefinitely expanding the depth of the tree. Both these factors contribute to the infeasibility of output debugging in Depth First Search.

Extensions:
Breadth First and Depth First Search algorithms have been implemented using Tree search previously in this investigation. Now, we will explore these search methods through Graph search. The Graph search methods can be found towards the end of the Search class in Section 5 of the Appendices. The difference between Tree search and Graph search is that there is an additional list that keeps track of all the nodes that have been visited to make sure they are not explored again - this refers to the expandedNodesList in GraphBreadthFirstSearch and GraphDepthFirstSearch. The approach has also been altered slightly, instead of using a stack/ queue, a list (nodesToExpandList) has been used to keep

track of all the expanded nodes. The new currentNode can be retrieved every time by retrieving the node at index 0 in this list.
A child node is only added to the nodesToExpandList if no other node with the same configuration is inside nodesToExpandList and expandedNodesList.

In addition to the two lists mentioned above another list nodesToExpandList1 is used to store the children of each node in GraphDepthFirstSearch. After all the children have been added to this list, it is appended to the beginning of the nodesToExpandList. This ensures that each currentNode is a child of the previous currentNode and keeps the LIFO property of the queue implemented earlier.

This approach would eliminate the problem we face when Depth First Search was initially implemented and children were not shuffled. Using the graph method, the children of each node do not need to be shuffled, as their corresponding configuration can only be visited once. Additionally, this prevents the search algorithm from creating nodes deeper and deeper in the same branch.The graphs in Appendices Section 3 (Figures 8 and 9) depict plots of the number of nodes generated in Graph Implementation of Breadth First Search and Depth First Search. Upon comparison with the tree implementation, we can easily deduce that these have much better time complexity. In fact for problem size 14 the graph implementation of BFS and DFS give 3100 and 53464 generated nodes compared to tree implementation of BFS and DFS; which are above 23 million and 100000 respectively. These are great advantages to graph search, however it is necessary to note its limitation - much more memory is allocated as the nodes explored are stored in a list (increase in space complexity).

The program is also implemented so that it can find the path to solution for any n*n tile configuration. This ease of adjustment allowed the tracking of the generated nodes and the path to solution for the start and end states shown in figure 2.

Obstacles can easily be put on the board, such that the agent cannot exchange positions with the obstacles. Assume a number 8 is an obstacle in the Scenario 2, with new start and end states as shown in the figure below. This constraint is ensured in all the move methods of the Node class. Consider the moveRight method for example: an additional constraint can be added which makes sure that the node can only move right if the tile with which the position of the agent is exchanged, is not the number 8 tile. This additional constraint has been placed in the move methods as commented code. The output debugging and the path to the solution for BFS are in section 4 of the Appendices.

Figure 6:

Start state End state

Scenario 3

Appendices

Table of Contents

Section 1: Debugging output for Scenario 1

Breadth First Search - Scenario 1

```
Breadth First Search
Current node depth: 0
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth: 1
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Current node depth: 1
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth: 2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*
```

```
Current node depth: 2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*4*2*3*

Current node depth: 2
0*0*0*0*
0*0*0*0*
0*0*4*0*
1*2*0*3*

Current node depth: 2
0*0*0*0*
0*0*0*0*
0*0*4*0*
1*2*3*0*

Current node depth: 2
0*0*0*0*
0*0*0*4*
0*0*0*0*
1*2*3*0*
```

```
Current node depth: 2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*
```

```
Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*0*
4*1*2*3*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*4*0*0*
1*0*2*3*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*0*3*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*4*0*0*
1*2*0*3*
```

```
Current node depth: 3
0*0*0*0*
0*0*4*0*
0*0*0*0*
1*2*0*3*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*4*0*0*
1*2*3*0*
```

```
Current node depth: 3
0*0*0*0*
0*0*4*0*
0*0*0*0*
1*2*3*0*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*3*0*
1*2*4*0*

Current node depth: 3
0*0*0*0*
0*0*4*0*
0*0*0*0*
1*2*3*0*

Current node depth: 3
0*0*0*4*
0*0*0*0*
0*0*0*0*
1*2*3*0*
```

```
Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*
```

Depth First Search - Scenario 1

```
Depth First Search
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth: 0
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth: 1
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth: 2
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*
```

```
Current node depth: 3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth: 4
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth: 5
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth: 6
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*
```

Random Depth First Search - Scenario 1

```
Random Depth First Search
Current node depth:0
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth:1
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*
```

```
Current node depth:4
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*4*2*3*

Current node depth:5
0*0*0*0*
0*0*0*0*
0*0*0*0*
4*1*2*3*

Current node depth:6
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*4*2*3*

Current node depth:7
0*0*0*0*
0*0*0*0*
0*0*0*0*
4*1*2*3*
```

Iterative Deepening Search - Scenario 1.

In case of 2 rows of columns, please read through the debugging output down each column and across the row and then move on to the second row of columns.

```
Iterative Deepening Search
Current node depth:0
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current limit is:1
Current node depth:0
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth:1
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth:1
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*
```

```
Current limit is:2
Current node depth:0
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth:1
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth:2
0*0*0*0*
0*0*0*4*
0*0*0*0*
1*2*3*0*
```

```
Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*4*0*
1*2*3*0*

Current node depth:1
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*4*0*
1*2*0*3*

Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*4*2*3*
```

```
Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current limit is:3
Current node depth:0
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth:1
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*
```

```
Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Current node depth:2
0*0*0*0*
0*0*0*4*
0*0*0*0*
1*2*3*0*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*
```

```
Current node depth:3
0*0*0*0*
0*0*4*0*
0*0*0*0*
1*2*3*0*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*4*0*0*
1*2*3*0*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth:1
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*
```

```
Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*4*0*
1*2*0*3*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Current node depth:3
0*0*0*0*
0*0*4*0*
0*0*0*0*
1*2*0*3*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*4*0*0*
1*2*0*3*
```

```
Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Current node depth:2
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Current node depth:3
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*
```

A* Search - Scenario 1

Note : In A* search nodes with equal priority can be dequeued in any order from the priority queue

```
A* Search
Node picked:
Current node depth:0
Cost of Node:5
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Cost of current child:7
-----
-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Cost of current child:8
-----
```

```
-----
Node picked:
Current node depth:1
Cost of Node:7
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

Cost of current child:7
-----
-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*4*2*3*

Cost of current child:8
-----
```

```
-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*4*0*
1*2*0*3*

Cost of current child:8
-----
Node picked:
Current node depth:2
Cost of Node:7
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*3*4*

-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Cost of current child:9
-----
```

```
-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*3*0*

Cost of current child:10
-----
Node picked:
Current node depth:2
Cost of Node:8
0*0*0*0*
0*0*0*0*
0*0*4*0*
1*2*0*3*

-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*0*4*
1*2*0*3*

Cost of current child:9
-----
```

```
-----
Current child:
0*0*0*0*
0*0*0*0*
0*4*0*0*
1*2*0*3*

Cost of current child:8
-----
-----
Current child:
0*0*0*0*
0*0*4*0*
0*0*0*0*
1*2*0*3*

Cost of current child:9
-----
-----
Current child:
0*0*0*0*
0*0*0*0*
0*0*0*0*
1*2*4*3*

Cost of current child:9
-----
```

Section 2: Debugging output for Scenario 2

Breadth First Search - Scenario 2

```
Breadth First Search
Current node depth: 0
0*0*4*
2*0*0*
1*0*3*

Current node depth: 1
0*4*0*
2*0*0*
1*0*3*

Current node depth: 1
0*0*0*
2*0*4*
1*0*3*

Current node depth: 2
0*0*4*
2*0*0*
1*0*3*

Current node depth: 2
4*0*0*
2*0*0*
1*0*3*
```

```
Current node depth: 2
0*0*0*
2*4*0*
1*0*3*

Current node depth: 2
0*0*0*
2*4*0*
1*0*3*

Current node depth: 2
0*0*4*
2*0*0*
1*0*3*

Current node depth: 2
0*0*0*
2*0*3*
1*0*4*

Current node depth: 3
0*4*0*
2*0*0*
1*0*3*
```

```
Current node depth: 3
2*0*0*
4*0*0*
1*0*3*

Current node depth: 3
0*0*0*
2*0*4*
1*0*3*

Current node depth: 3
0*0*0*
4*2*0*
1*0*3*
```

Random Depth First Search - Scenario 2

```
Random Depth First Search
Current node depth:0
0*0*4*
2*0*0*
1*0*3*

Current node depth:1
0*0*0*
2*0*4*
1*0*3*

Current node depth:2
0*0*4*
2*0*0*
1*0*3*

Current node depth:3
0*0*0*
2*0*4*
1*0*3*

Current node depth:4
0*0*4*
2*0*0*
1*0*3*
```

```
Current node depth:5
0*0*0*
2*0*4*
1*0*3*

Current node depth:6
0*0*0*
2*0*3*
1*0*4*

Current node depth:7
0*0*0*
2*0*4*
1*0*3*

Current node depth:8
0*0*0*
2*4*0*
1*0*3*

Current node depth:9
0*0*0*
4*2*0*
1*0*3*
```

Iterative Deepening Search - Scenario 2

```
Iterative Deepening Search
Current node depth:0
0*0*4*
2*0*0*
1*0*3*

Current limit is:1
Current node depth:0
0*0*4*
2*0*0*
1*0*3*

Current node depth:1
0*0*0*
2*0*4*
1*0*3*

Current node depth:1
0*4*0*
2*0*0*
1*0*3*

Current limit is:2
Current node depth:0
0*0*4*
2*0*0*
1*0*3*
```

```
Current node depth:1
0*0*0*
2*0*4*
1*0*3*

Current node depth:2
0*0*0*
2*0*3*
1*0*4*

Current node depth:2
0*0*4*
2*0*0*
1*0*3*

Current node depth:2
0*0*0*
2*4*0*
1*0*3*

Current node depth:1
0*4*0*
2*0*0*
1*0*3*
```

```
Current node depth:2
0*0*0*
2*4*0*
1*0*3*

Current node depth:2
4*0*0*
2*0*0*
1*0*3*

Current node depth:2
0*0*4*
2*0*0*
1*0*3*

Current limit is:3
Current node depth:0
0*0*4*
2*0*0*
1*0*3*

Current node depth:1
0*0*0*
2*0*4*
1*0*3*
```

```
Current node depth:2
0*0*0*
2*0*3*
1*0*4*

Current node depth:3
0*0*0*
2*0*4*
1*0*3*

Current node depth:3
0*0*0*
2*0*3*
1*4*0*

Current node depth:2
0*0*4*
2*0*0*
1*0*3*

Current node depth:3
0*0*0*
2*0*4*
1*0*3*
```

```
Current node depth:3
0*4*0*
2*0*0*
1*0*3*

Current node depth:2
0*0*0*
2*4*0*
1*0*3*

Current node depth:3
0*0*0*
2*0*0*
1*4*3*

Current node depth:3
0*4*0*
2*0*0*
1*0*3*

Current node depth:3
0*0*0*
4*2*0*
1*0*3*
```

A* Search - Scenario 2

```
A* Search
Node picked:
Current node depth:0
Cost of Node:4
0*0*4*
2*0*0*
1*0*3*

-----
Current child:
0*4*0*
2*0*0*
1*0*3*

Cost of current child:5
-----
-----
Current child:
0*0*0*
2*0*4*
1*0*3*

Cost of current child:5
-----
```

```
-----
Node picked:
Current node depth:1
Cost of Node:5
0*4*0*
2*0*0*
1*0*3*

-----
Current child:
0*0*4*
2*0*0*
1*0*3*

Cost of current child:6
-----
-----
Current child:
4*0*0*
2*0*0*
1*0*3*

Cost of current child:5
-----
```

```
-----
Current child:
0*0*0*
2*4*0*
1*0*3*

Cost of current child:5
-----
Node picked:
Current node depth:1
Cost of Node:5
0*0*0*
2*0*4*
1*0*3*

-----
Current child:
0*0*0*
2*4*0*
1*0*3*

Cost of current child:5
-----
```

```
-----
Current child:
0*0*4*
2*0*0*
1*0*3*

Cost of current child:6
-----
-----
Current child:
0*0*0*
2*0*3*
1*0*4*

Cost of current child:7
-----
Node picked:
Current node depth:2
Cost of Node:5
0*0*0*
2*4*0*
1*0*3*

-----
```

```
-----
Current child:
0*0*0*
2*0*4*
1*0*3*

Cost of current child:7
-----
-----
Current child:
0*0*0*
4*2*0*
1*0*3*

Cost of current child:5
-----
-----
Current child:
0*4*0*
2*0*0*
1*0*3*

Cost of current child:7
-----
```

```
-----
Current child:
0*0*0*
2*0*0*
1*4*3*

Cost of current child:5
-----
Node picked:
Current node depth:2
Cost of Node:5
0*0*0*
2*4*0*
1*0*3*

-----
Current child:
0*0*0*
2*0*4*
1*0*3*

Cost of current child:7
-----
```

```
-----
Current child:
0*0*0*
4*2*0*
1*0*3*

Cost of current child:5
-----
-----
Current child:
0*4*0*
2*0*0*
1*0*3*

Cost of current child:7
-----
Current child:
0*0*0*
2*0*0*
1*4*3*

Cost of current child:5
-----
```

```
-----
Node picked:
Current node depth:3
Cost of Node:5
0*0*0*
2*0*0*
1*4*3*

-----
Current child:
0*0*0*
2*0*0*
1*3*4*

Cost of current child:8
-----
-----
Current child:
0*0*0*
2*0*0*
4*1*3*

Cost of current child:8
-----
```

```
-----
Current child:
0*0*0*
2*4*0*
1*0*3*

Cost of current child:7
-----
Node picked:
Current node depth:3
Cost of Node:5
0*0*0*
4*2*0*
1*0*3*

Goal found
```

Section 3: Additional Graph results

Figure 7:

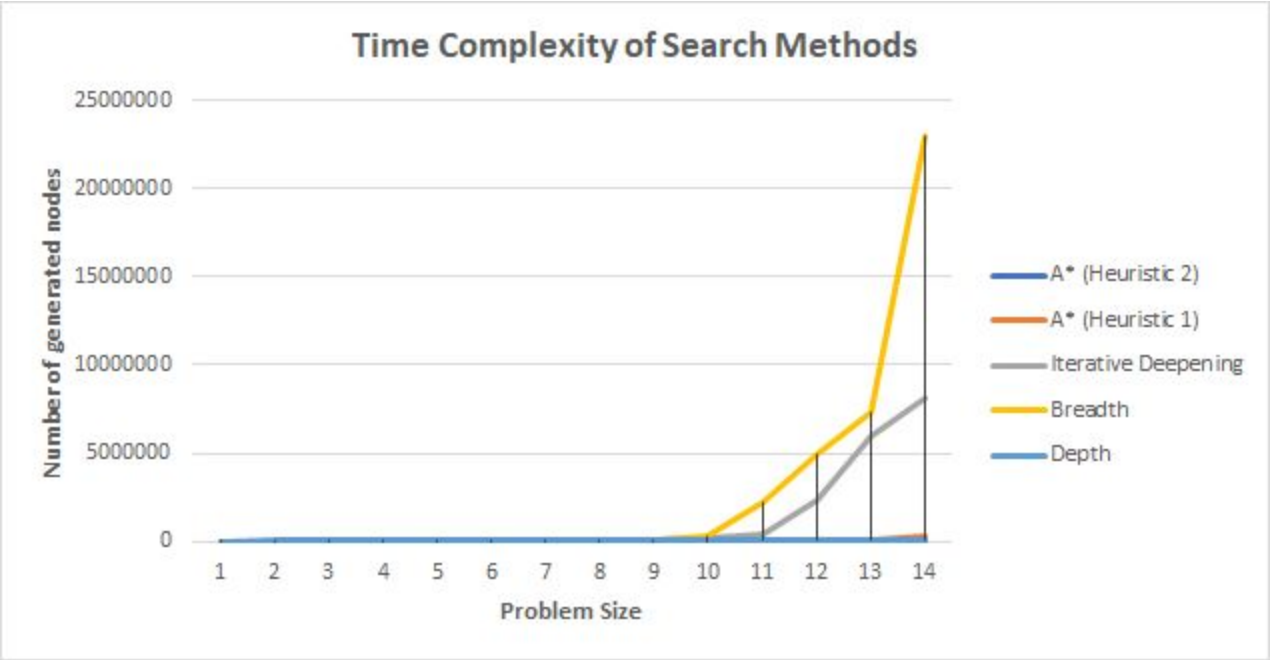


Figure 8:

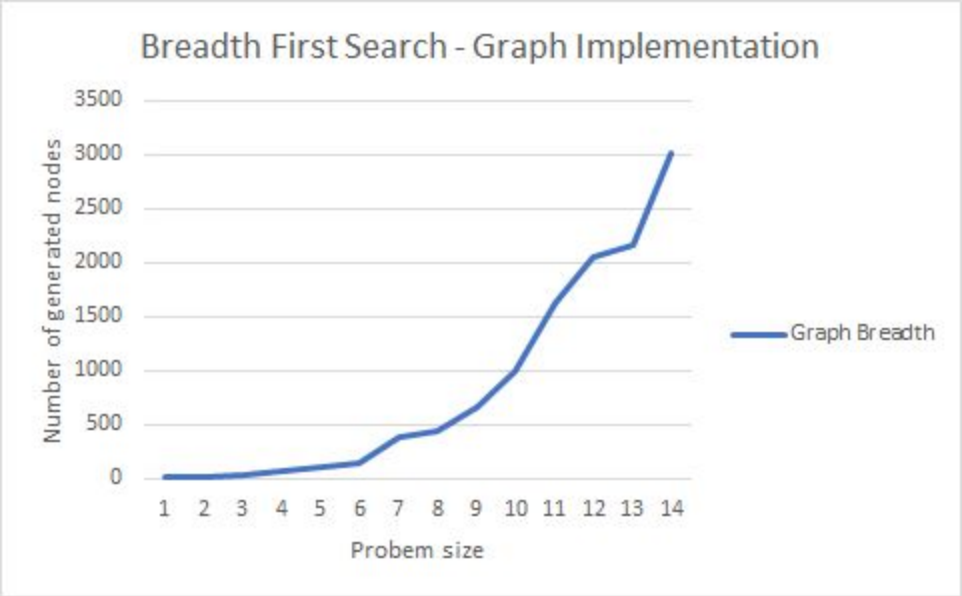
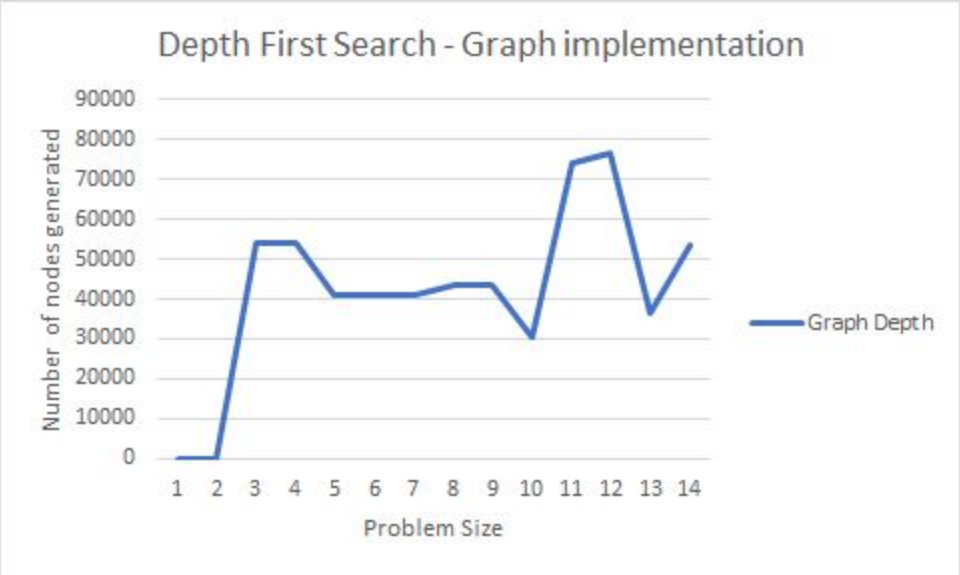


Figure 9:


Depth First Search - Graph implementation
Number of nodes generated
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14
Problem Size
Graph Depth

Section 4: Debugging output, Path to solution for Scenario 3

```
Breadth First Search
Current node depth: 0
0*8*4*
2*0*0*
1*0*3*

Current node depth: 1
0*8*0*
2*0*4*
1*0*3*

Current node depth: 2
0*8*0*
2*4*0*
1*0*3*

Current node depth: 2
0*8*4*
2*0*0*
1*0*3*

Current node depth: 2
0*8*0*
2*0*3*
1*0*4*

Current node depth: 3
0*8*0*
2*0*4*
1*0*3*

Current node depth: 3
0*8*0*
4*2*0*
1*0*3*

Goal found
```

```
Goal found
Path to solution :
Depth of this node:0
0*8*4*
2*0*0*
1*0*3*

Depth of this node:0
0*8*4*
2*0*0*
1*0*3*

Depth of this node:1
0*8*0*
2*0*4*
1*0*3*

Depth of this node:2
0*8*0*
2*4*0*
1*0*3*

Depth of this node:3
0*8*0*
4*2*0*
1*0*3*

Number of steps to solution: 3
Number of nodes generated: 13
```

Section 5: Code in Human-Readable Format

Note: The NodePriorityComparator.java class was adapted from the code found on the internet, hence it has not been included in the report. However, the class can be found with all the source code in the zipped directory.

Node.java

```
import java.util.*;

public class Node {

   List<Node> children = new ArrayList<>();
   public Node parent;
   //position of the agent
   public int x = 4;
   //the value of n in n*n matrix can be set here
   public int col = 4;
   //size of new array is dependant on column/row size
   int size = col*col;
   public int[] puzzle = new int[size];
   private int totalCost;
   private int depth;
   int[] goalPuzzle =
         {
               0, 0, 0, 0,
               0, 1, 0, 0,
               0, 2, 0, 0,
               0, 3, 0, 4
         };

   int[] list =
         {1,2,3};


   public Node(int[] p){
      setPuzzle(p);
   }

   public int getDepth() {
      return depth;
   }

   public Node getParent() {
      return parent;
   }
```

```
public void setDepth(int depth) {
    this.depth = depth;
}


public int getTotalCost() {
    return this.totalCost;
}

public void setTotalCost(int totalCost) {
    this.totalCost = totalCost;
}



//Copy the initial (original) puzzle to this copy
public void setPuzzle(int[] p){
    for(int i = 0; i < puzzle.length; i++)
        this.puzzle[i] = p[i];
}


public void expandMove(){
    for(int i = 0; i < puzzle.length; i++){
        //Apply the legal action with 0
        if(puzzle[i] == 4) {
            x = i;
        }
    }
    moveRight(puzzle, x);
    moveLeft(puzzle, x);
    moveUp(puzzle, x);
    moveDown(puzzle, x);
}


public int heuristicOne(){
    int difference = 0;
    for(int i = 1; i < goalPuzzle.length; i++){
        if(puzzle[i] != goalPuzzle[i]){
            difference += 1;
        }
    }
    return difference;
}
```

```
public int heuristicTwo() {
    int difference = 0;
    for(int i = 0; i < list.length; i++){
        for (int j = 0; j < puzzle.length; j += 1){
            if(puzzle[j] == list[i]){
                int k = findIndex(puzzle[j]);
                difference = difference + ((Math.abs(j % 4 - k % 4)) + Math.abs(j / 4 - k / 4));
            }
        }


    }
    return difference;
}


public int findIndex(int t)
{
    int len = this.goalPuzzle.length;
    int i = 0;

    // traverse in the array
    while (i < len) {
        if (goalPuzzle[i] == t) {
            return i;
        }
        else {
            i = i + 1;
        }
    }
    return -1;
}

public boolean GoalTest(){
    boolean isGoal = true;
    for(int i = 0; i < list.length; i++) {
        for (int j = 0; j < puzzle.length; j += 1) {
            if (puzzle[j] == list[i]) {
                int k = findIndex(puzzle[j]);
                if (k != j) {
                    isGoal = false;
                }
            }
        }
    }
```

```
    return isGoal;
}

//You can expand a node by moving right, left, down or up
public void moveRight(int[] p, int i){

    //this constraint makes sure the following operation is not applied to the last(third) column
    if(i % col < col - 1){
        //the if constraint commented out below takes into account an obstacle numbered 8
        //if(p[i+1] != 8){
            int[] pc = new int[size];
            //a new copy of the puzzle is created every time a move method is called
            copyPuzzle(pc, p);
            int temp = pc[i+1];
            pc[i+1] = pc[i];
            pc[i] = temp;

            Node child = new Node(pc);
            children.add(child);
            child.parent = this;

    }
}

public void moveLeft(int[] p, int i){
    if(i % col > 0) {
        //if(p[i-1] != 8) {
            int[] pc = new int[size];
            copyPuzzle(pc, p);
            int temp = pc[i - 1];
            pc[i - 1] = pc[i];
            pc[i] = temp;

            Node child = new Node(pc);
            children.add(child);
            child.parent = this;
        //}

    }
}

public void moveUp(int[] p, int i){
    if(i - col > 0){
        //if(p[i-col] != 8) {
            int[] pc = new int[size];
            copyPuzzle(pc, p);
```

```
            int temp = pc[i - col];
            pc[i - col] = pc[i];
            pc[i] = temp;

            Node child = new Node(pc);
            children.add(child);
            child.parent = this;
        //}
    }

}

public void moveDown(int[] p, int i){
    if(i + col < puzzle.length){
        //if(p[i+col] != 8) {
            int[] pc = new int[size];
            copyPuzzle(pc, p);
            int temp = pc[i + col];
            pc[i + col] = pc[i];
            pc[i] = temp;

            Node child = new Node(pc);
            children.add(child);
            child.parent = this;
        //}

    }
}

public void printPuzzle(){
    //System.out.println(puzzle[0]);
    int m = 0;
    for(int i = 0; i < col; i++){
        for(int j = 0; j < col; j++){
            System.out.print(puzzle[m] + "*");
            m++;
        }
        System.out.println(" ");
    }
    System.out.println(" ");
}

//This method is used when things are moved
public void copyPuzzle(int[] a, int[] b){
    for(int i = 0; i < b.length; i++){
        a[i] = b[i];
```

```
        }
    }

    public boolean isSamePuzzle(int[] p) {
        boolean samePuzzle = true;
        for (int i = 0; i < p.length; i++) {
            if (this.puzzle[i] != p[i]) {
                samePuzzle = false;
            }
        }

        return samePuzzle;

    }

}
```

Search.java

```
import java.util.List;
import java.util.*;
import java.util.Stack;
import java.util.PriorityQueue;

public class Search {

    public Search() {

    }

    public void BreadthFirstSearch(Node root) {
        System.out.println("Breadth First Search");
        List<Node> states = new ArrayList<>();
        int time = 0;
        Boolean goalFound = false;
        Queue<Node> queue = new LinkedList<>();
        root.setDepth(0);
        Node currentNode = root;
        queue.add(currentNode);
        while ((goalFound==false) && !queue.isEmpty()) {
            currentNode = queue.poll();
            System.out.println("Current node depth: "+ currentNode.getDepth());
            currentNode.printPuzzle();

            if (currentNode.GoalTest()) {
                System.out.println("Goal found");
                goalFound = true;
                printSolution(currentNode);
            }
            else{
                currentNode.expandMove();
                states.add(currentNode);
                time = currentNode.children.size() + time;
                for (Node currentChild : currentNode.children) {
                    currentChild.setDepth(currentNode.getDepth()+1);
                    ((LinkedList<Node>) queue).add(currentChild);
                }
            }

        }

        System.out.println("Number of nodes generated: " + time);
```

```
    /*Number of generated nodes can be confirmed by counting
    the number of elements remaining in queue and in ArrayList states */
    int count = 0;
    while(!queue.isEmpty()){
        queue.poll();
        count = count + 1;
    }
    int size = states.size()+count;
    System.out.println("Check: " + size);

}

public void DepthFirstSearch(Node root) {
    System.out.println("Depth First Search");
    List<Node> states = new ArrayList<>();
    int time = 0;
    Boolean goalFound = false;
    Stack<Node> stack = new Stack<>();
    root.setDepth(0);
    Node currentNode = root;
    stack.push(currentNode);

    while (!goalFound && !stack.isEmpty()) {
        currentNode = stack.pop();
        currentNode.printPuzzle();
        System.out.println("Current node depth: "+ currentNode.getDepth());
        if (currentNode.GoalTest()) {
            System.out.println("Goal found");
            goalFound = true;
            printSolution(currentNode);
        }
        else{
            currentNode.expandMove();
            states.add(currentNode);
            time = currentNode.children.size() + time;
            for (Node currentChild : currentNode.children) {
                currentChild.setDepth(currentNode.getDepth()+1);
                stack.push(currentChild);
            }
        }

    }
    System.out.println("Number of nodes generated: " + time);

    /*Number of generated nodes can be confirmed by counting
```

```
    the number of elements remaining in queue and in ArrayList states */
    int count = 0;
    while(!stack.isEmpty()){
        stack.pop();
        count = count + 1;
    }
    int size = states.size()+count;
    System.out.println("Check: " + size);
}


public int RandomDepthFirstSearch(Node root) {
    System.out.println("Random Depth First Search");
    int time = 0;
    Boolean goalFound = false;
    Stack<Node> stack = new Stack<>();
    root.setDepth(0);
    Node currentNode = root;
    stack.push(currentNode);

    while (!goalFound && !stack.isEmpty()) {
        currentNode = stack.pop();
        System.out.println("Current node depth:" + currentNode.getDepth());
        currentNode.printPuzzle();

        if (currentNode.GoalTest()) {
            System.out.println("Goal found");
            goalFound = true;
            printSolution(currentNode);
        }
        else{
            currentNode.expandMove();
            time = currentNode.children.size() + time;
            Collections.shuffle(currentNode.children);
            for (Node currentChild : currentNode.children) {
                currentChild.setDepth(currentNode.getDepth()+1);
                stack.push(currentChild);
            }
        }

    }

    System.out.println("Number of nodes generated: " + time);

    return time;
}
```

```
/* Method to call RandomDepthFirst search 50 times
and take the average of number of generated nodes */
public void DepthFirstSearchNTimes(Node root){
    int i = 0;
    int totalTime = 0;
    while (i < 50){
        totalTime   = totalTime + RandomDepthFirstSearch(root);
        i++;
    }
    System.out.println("The average number of nodes generated: " + (totalTime/50));
}


public void iterativeDeepening(Node root) {
    System.out.println("Iterative Deepening Search");
    List<Node> states = new ArrayList<>();
    int time = 0;
    int limit = 0;
    Boolean goalFound = false;
    Stack<Node> stack = new Stack<>();
    root.setDepth(0);
    Node currentNode = root;
    stack.push(currentNode);
    int count = 0;

    while (!goalFound) {
        if (stack.empty() == false) {
            currentNode = stack.pop();
            System.out.println("Current node depth:" + currentNode.getDepth());
            currentNode.printPuzzle();
            if (currentNode.GoalTest()) {
                System.out.println("Goal found");
                printSolution(currentNode);
                goalFound = true;
            } else if (currentNode.getDepth() < limit) {
                states.add(currentNode);
                currentNode.expandMove();
                time = currentNode.children.size() + time;
                for (Node currentChild : currentNode.children) {
                    currentChild.setDepth(currentNode.getDepth()+1);
                    stack.push(currentChild);
                }
                currentNode.children.clear();
            }
            if(currentNode != root){
```

```
            count++;
          }
        } else  {
          limit = limit + 1;
          System.out.println("Current limit is:" + limit);
          stack.push(root);
        }
      }

      System.out.println("Number of nodes generated: " + time);
}

public void aStar(Node root) {
    System.out.println("A* Search");
    boolean goalFound = false;
    List<Node> states = new ArrayList<>();
    int time = 0;
    NodePriorityComparator nodePriorityComparator = new NodePriorityComparator();
    PriorityQueue<Node> nodePriorityQueue = new PriorityQueue(10, nodePriorityComparator);
    root.setDepth(0);
    root.parent = null;
    root.setTotalCost(root.heuristicOne());
    Node currentNode = root;
    nodePriorityQueue.add(currentNode);

    while (!goalFound && !nodePriorityQueue.isEmpty()) {
        currentNode = nodePriorityQueue.poll();
        System.out.println("Node picked:" );
        System.out.println("Current node depth:" + currentNode.getDepth());
        System.out.println("Cost of Node:" + currentNode.getTotalCost() );
        currentNode.printPuzzle();
        if(currentNode.GoalTest()){
            System.out.println("Goal found");
            printSolution(currentNode);
            goalFound = true;
        }
        else{
            currentNode.expandMove();
            states.add(currentNode);
            time = currentNode.children.size() + time;
            for(Node currentChild : currentNode.children){
                currentChild.setDepth(currentNode.getDepth()+1);
                //

                int h = currentChild.heuristicOne();
                currentChild.setTotalCost(currentChild.getDepth() + h);
```

```
                System.out.println("-----");
                System.out.println("Current child:" );
                currentChild.printPuzzle();
                System.out.println("Cost of current child:" + currentChild.getTotalCost());
                System.out.println("-----");
                nodePriorityQueue.add(currentChild);
            }
        }

    }
    System.out.println("Number of nodes generated: " + time);
}

//method to print path to solution
public static void printSolution(Node goalNode) {
    Stack<Node> solutionStack = new Stack<>();

    solutionStack.push(goalNode);
    Node current = goalNode;
    while (current.parent != null) {
        solutionStack.push(current.getParent());
        current = current.getParent();
    }
    System.out.println("Path to solution : ");
    solutionStack.push(current);
    int i = 0;
    while (solutionStack.isEmpty()==false){
        Node thisNode = solutionStack.pop();
        System.out.println("Depth of this node:" + thisNode.getDepth());
        thisNode.printPuzzle();
        i++;

    }

    System.out.println("Number of steps to solution: " + (i-2));

}

//Extensions
public void GraphBreadthFirstSearch(Node root) {
    //List of nodes that need to be expanded
    List<Node> nodesToExpandList = new ArrayList<>();
    int time = 0;
    //List of nodes that have already been expanded
    List<Node> expandedNodesList = new ArrayList<>();
    root.setDepth(0);
```

```
        nodesToExpandList.add(root);
        boolean goalFound = false;

        while (nodesToExpandList.size() > 0 && !goalFound) {
            Node currentNode = nodesToExpandList.get(0);
            currentNode.printPuzzle();
            expandedNodesList.add(currentNode);
            nodesToExpandList.remove(0);
            if (currentNode.GoalTest()) {
                System.out.println("Goal found");
                goalFound = true;
                printSolution(currentNode);
            }
            else{
                currentNode.expandMove();
                //currentNode.printPuzzle();
                for (int i = 0; i < currentNode.children.size(); i++) {
                    Node currentChild = currentNode.children.get(i);
                    currentChild.setDepth(currentNode.getDepth()+1);
                    //if current child configuration is not in either list, add child to the nodesToExpandList
                    //this ensures that a node configuration that has been visited is not visited again
                    if (!Contains(nodesToExpandList, currentChild) && !Contains(expandedNodesList,
currentChild)) {
                        time = time + 1;
                        nodesToExpandList.add(currentChild);
                    }
                }
            }
        }
        System.out.println("Number of nodes generated:" + time);
    }

    public void GraphDepthFirstSearch(Node root) {
        List<Node> nodesToExpandList = new ArrayList<>();
        int time = 0;
        //Additional list to keep track of all the children nodes
        List<Node> nodesToExpandList1 = new ArrayList<>();
        List<Node> expandedNodesList = new ArrayList<>();
        root.setDepth(0);
        nodesToExpandList.add(root);
        boolean goalFound = false;

        while (nodesToExpandList.size() > 0 && !goalFound) {
            Node currentNode = nodesToExpandList.get(0);
            currentNode.printPuzzle();
            expandedNodesList.add(currentNode);
```

```
            nodesToExpandList.remove(0);
            currentNode.expandMove();
            if (currentNode.GoalTest()) {
                System.out.println("Goal found");
                goalFound = true;
                printSolution(currentNode);
            }
            else{
                currentNode.expandMove();
                for (int i = 0; i < currentNode.children.size(); i++) {
                    Node currentChild = currentNode.children.get(i);
                    currentChild.setDepth(currentNode.getDepth()+1);
                    if (!Contains(nodesToExpandList, currentChild) && !Contains(expandedNodesList,
currentChild)) {
                        time = time + 1;
                        nodesToExpandList1.add(currentChild);
                    }
                }
                nodesToExpandList.addAll(0,nodesToExpandList1);
                nodesToExpandList1.clear();

            }



        }

        System.out.println("Number of nodes generated:" + time);

    }

    /* method to check whether the current child configuration
    has been visited already in Graph search */
    public static boolean Contains(List<Node> list, Node c){
        boolean contains = false;
        for(int i = 0; i < list.size();i++){
            if(list.get(i).isSamePuzzle(c.puzzle))
                contains = true;
        }
        return contains;
    }



}
```

Program.java

```
public class Program {

    //methods in Search.java care called through this class.
    public static void main(String[] args){
        int[] puzzle =
                {
                        0, 0, 0, 0,
                        0, 0, 0, 0,
                        0, 0, 0, 0,
                        1, 2, 3, 4
                };



        Node root = new Node(puzzle);
        Search search = new Search();
        //search.BreadthFirstSearch(root);
        //search.DepthFirstSearch(root);
        //search.RandomDepthFirstSearch(root);
        //search.iterativeDeepening(root);
        //search.aStar(root);
        //search.GraphBreadthFirstSearch(root);
        //search.GraphDepthFirstSearch(root);

    }
}
```