Project 1: The Eight Puzzle

CS205 Artificial Intelligence (Winter 2019)
Instructor: Dr. Eamonn Keogh

Lokesh Koppaka
ID#: 862123164
lkopp001@ucr.edu
12-Febuary-2019

In completing the project, I have consulted

- Amazing class presentations of Heuristic Search - A* Algorithm provided by Dr. Eamonn Keogh
- The format of the report is taken from sample report provided
- Used same puzzle examples from sample report provided to perform analysis in comparison graph section

The code and the documentation are entirely original

- Used Collections in Java like ArrayList in order to maintain queue
- Used Collections standard sort and compare methods
- The standard sort and compare methods are overridden to perform custom sort and comparations

NOTE: As an initiative to save Trees, the report is printed in both sides of the paper

Introduction:

The project is designed to implement the Eight Puzzle problem using various search algorithms in order to understand how heuristic function can help in improving the search. Uniform Cost Search, A* with heuristics as misplaced tile, A* with heuristics as Manhattan distance algorithms are used to implement Eight puzzle. To compare how well the algorithms perform among each other the report compares attributes such as how many nodes are expanded to reach the goal state, the max queue size and the depth of the goal state which are necessary in understanding the time and space complexities.

The Eight Puzzle:

Eight Puzzle is a sliding tile-based problem in which the tiles numbered from 1 to 7 are placed in random position on a square shaped background along with a miss tile. The task of the problem solver (i.e. smart human beings/ Program) is to get all the misplaced tiles to their correct position in as less moves as possible. As a point of note, the eight puzzle is a specific consideration of n-puzzle problem we can have puzzle across various ranges like 15-puzzle, 24-puzzle etc..

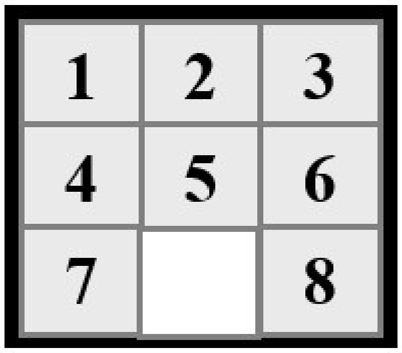


Fig1.0 Initial State

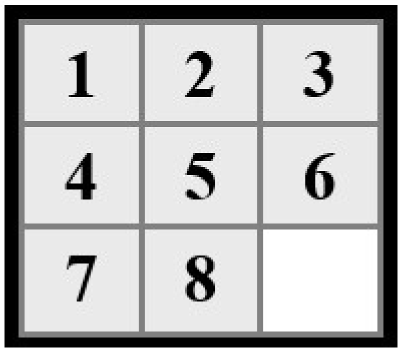


Fig1.1 Goal State

The following represents the important aspects that needs to be defined for any Search Problem

- Problem Space – i.e Initial State, Goal State, Operations
- Search Algorithm – Uniform Cost Search, A*

The following table (Table 1.0) demonstrates specific details of Problem Space for 8-puzzles

State Representation	The state is represented as nxn matrix
Initial State	The start state of the puzzle (Fig 1.0)
Goal State	The desired state of the puzzle (Fig 1.1)
Operations	Move miss tile to top Move miss tile to bottom Move miss tile to left Move miss tile to right

Table 1.0 8-puzzle Problem Space details

Search Algorithms:

Eight puzzle is implemented using search with notion of heuristic using the following algorithms

1. Uniform Cost Search
2. A* with misplaced tile heuristic
3. A* with Manhattan distance heuristic

Before discussing about the algorithms lets define some important terms (Table 1.1) which are necessary in understanding them

Heuristic $h(n)$	Approximate cost to reach the goal state from the current state
Cost to node $g(n)$	Cost taken to reach the current state
Estimated Cost $f(n)$	Estimated cost of the solution from current node n. i.e $g(n) + h(n)$

Table 1.1 A* key terms

Uniform Cost Search

Uniform Cost Search is a simple version of A* with heuristic $h(n) = 0$. The algorithm proceeds by checking the cheapest node and expanding them till the goal state is reached. It is important to note that the cost taken to move from one node to its immediate node is always equal to 1(since we move only one tile at a time) resulting the algorithm to work the same as Breadth first Search with time and space complexity as $O(b^d)$ where b is the branching factor and d is the depth.

A* with the Misplaced Tile heuristic

Misplaced tile heuristic algorithms work the same as the Uniform Cost Search except that the heuristic is computed using how many tiles are misplaced at the current state. At each instance it picks the node with less estimated cost $f(n)$. The estimated cost $f(n)$ at each node is computed as cost taken to get to that node i.e. $g(n)$ + heuristic $h(n)$. It continues the same till the goal state is reached.

A* with the Manhattan Distance heuristic

Manhattan Distance heuristic algorithms is also like Uniform Cost Search except that the heuristic is computed as the cumulative sum of the Manhattan distance for every misplaced tile to reach to its correct position. At each node it computes the estimated cost $f(n)$ as cost taken to get to that node i.e. $g(n)$ + heuristic $h(n)$ and picks the node that has less estimated cost. It continues the same till the goal state is reached.

Implementation Details:

Eight Puzzle is implement in Java programming Language. The program included two essential classes

a) State Class (State.java)

State class maintains the properties/attributes of the current state. It includes the following properties/attributes (Table 1.2) which have an essential role such as holding the scramble order of tiles in that state, the miss tile location, heuristic cost, cost taken to reach that state.

Attribute/Property Name	**Description**
puzzleState	Holds scramble order of tiles in that state
uniformCost	Cost taken to reach that state
heuristicCost	The heuristic Cost
blankLocation	The Miss tile location

Table 1.2 State Class Attributes/Properties

b) AStar Class (AStar.java)

AStar Class is the primary class that starts the program. The class holds the following functionality.

- User preferences – either to choose default / custom puzzle
- Generic Search – takes in the type of algorithm and perform the search

Apart from the above, AStar holds the following helper function which are essential for performing generic search

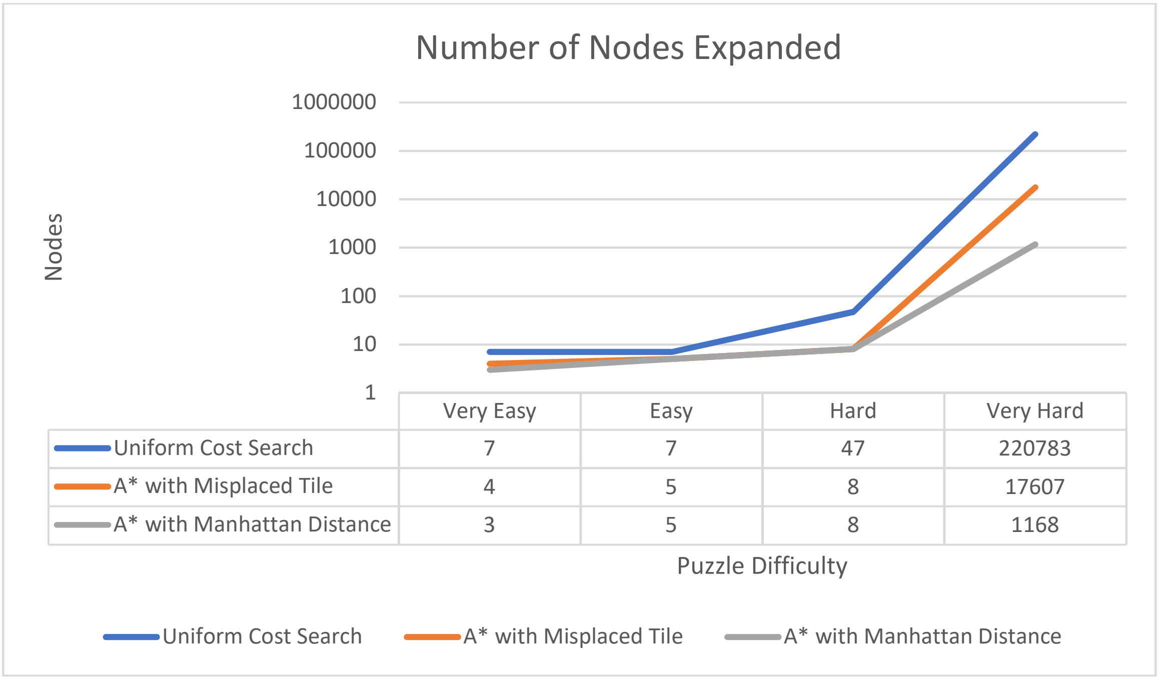
Helper Name	**Description**
heuristicUsingManhattenDist	Computes Manhattan distance heuristic
heuristicUsingMisplaceTiles	Computes Misplace tile heuristic
isGoalState	Checks if the current state is the goal state
constructGoalState	Constructs the goal state of size n
expand	Expands to all the possible states from the current state
appendValidStatesToQueue	Appends valid states to queue
isStateVisited	Checks if the current state is visited or not
printPuzzle	Prints the current state to standard output
addRowToPuzzle	Adds user inputted row to the puzzle

Table 1.3 AStar helper functions

Comparison Graphs:
In order to compare the performance of the algorithms, various initial puzzle states with respect to difficulty are taken and the following plots are generated.

Difficulty	**Puzzle Instance**
Very Easy	[1, 2, 3] [4, 5, 6] [7, 0, 8]
Easy	[1, 2, 0] [4, 5, 3] [7, 8, 6]
Hard	[0, 1, 2] [4, 5, 3] [7, 8, 6]
Very Hard	[8, 7, 1] [6, 0, 2] [5, 4, 3]

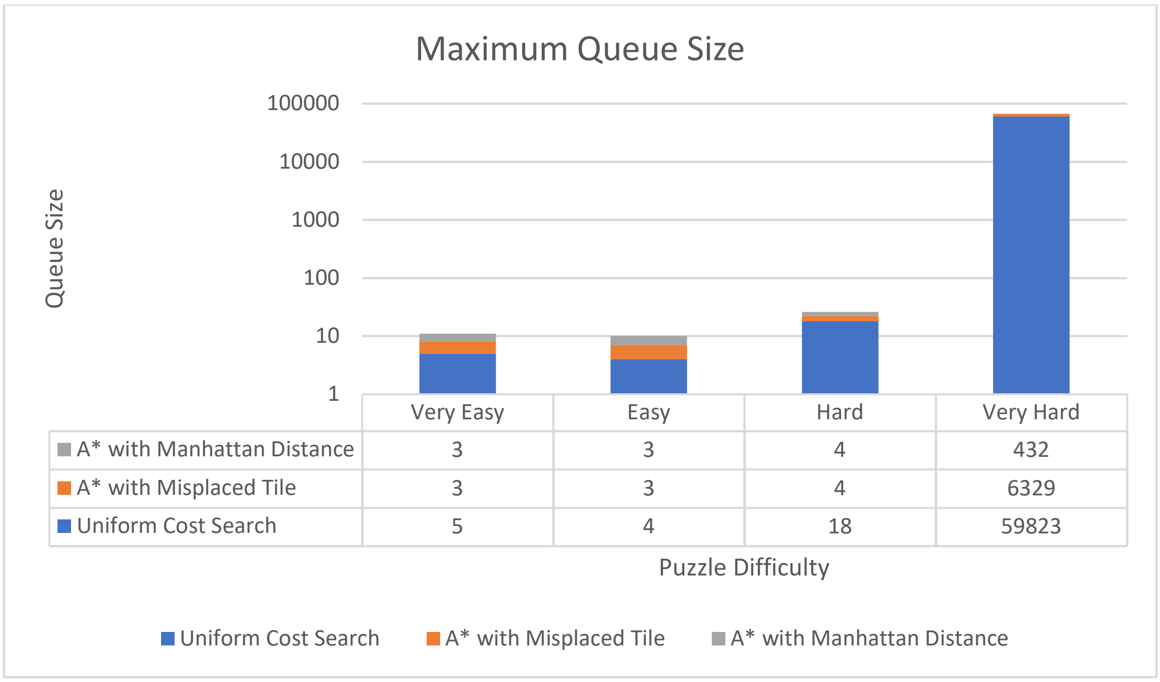
Table 1.4 Puzzle difficulty



	Very Easy	Easy	Hard	Very Hard
Uniform Cost Search	7	7	47	220783
A* with Misplaced Tile	4	5	8	17607
A* with Manhattan Distance	3	5	8	1168

Fig 1.3 Nodes expansion across algorithms and puzzle difficulty level

Note: Since the values are very much skewed towards very hard, the y axis scale is changed to logarithmic scale of base 10



	Very Easy	Easy	Hard	Very Hard
▨ A* with Manhattan Distance	3	3	4	432
🟧 A* with Misplaced Tile	3	3	4	6329
🟦 Uniform Cost Search	5	4	18	59823

Fig 1.4 Max Queue sizes across algorithms and puzzle difficulty level

Note: Since the values are very much skewed towards very hard, the y axis scale is changed to logarithmic scale of base 10

Conclusions:

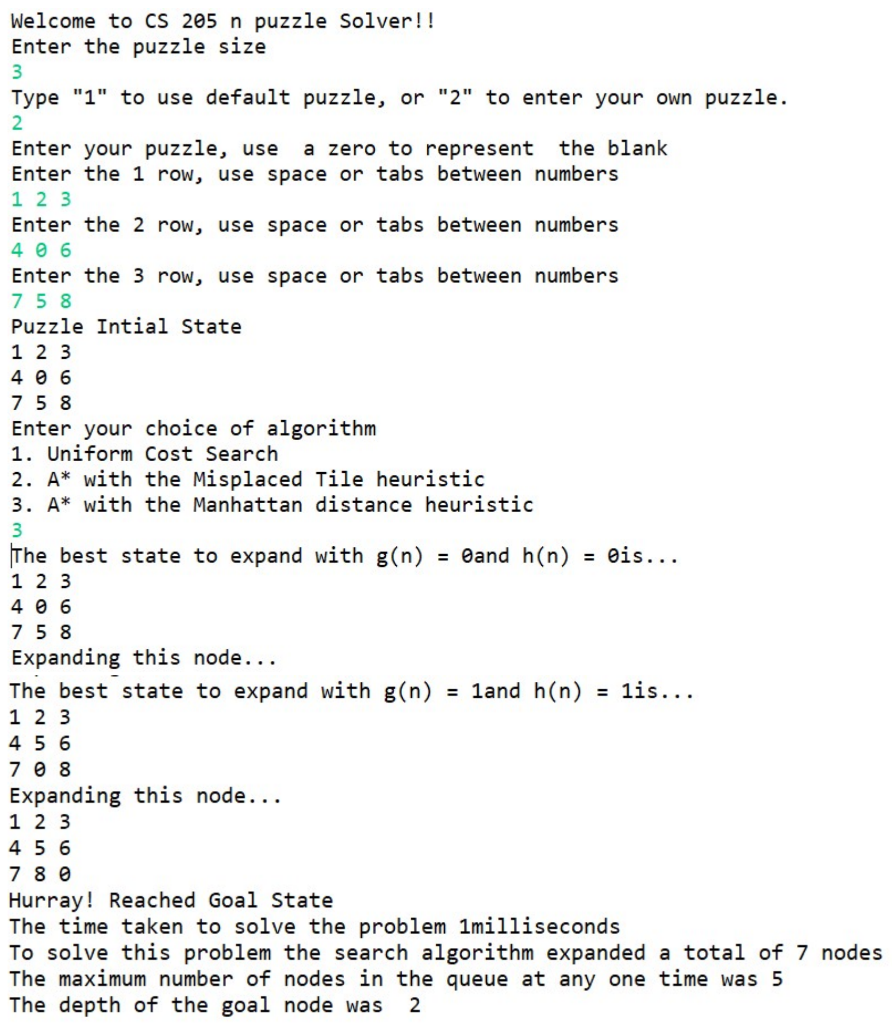
It is observed from the above comparison graphs(Fig 1.3, Fig 1.4)

- Searching with heuristic functions improves the performance by reducing the number of states to examine in the search space which indirectly reduces the time and space complexity
- Heuristic function quality plays an important role in improving the search this can be observed by comparing how A* performed with different heuristics i.e. Manhattan distance, Misplaced tile
- Manhattan heuristic leads in performance compared to Misplaced tile and uniform cost search
- Uniform cost search makes it position to the last in terms of performance it is because of the heuristic being 0 leading to behave like Breadth first search resulting in low time and space complexity
- A* with the Misplaced Tile heuristic and A* with the Manhattan Distance heuristic takes the best from both the worlds i.e. complete and optimal from Uniform Cost Search and fastness from Hill climbing resulting in a better version than Uniform Cost Search.

Take Away Note:

Search with the right heuristic function can significantly improve searching by leading the searching algorithm to pick up only the state that cost less to reach the goal state.

Traceback of the Manhattan Distance A*:



```
Welcome to CS 205 n puzzle Solver!!
Enter the puzzle size
3
Type "1" to use default puzzle, or "2" to enter your own puzzle.
2
Enter your puzzle, use  a zero to represent  the blank
Enter the 1 row, use space or tabs between numbers
1 2 3
Enter the 2 row, use space or tabs between numbers
4 0 6
Enter the 3 row, use space or tabs between numbers
7 5 8
Puzzle Intial State
1 2 3
4 0 6
7 5 8
Enter your choice of algorithm
1. Uniform Cost Search
2. A* with the Misplaced Tile heuristic
3. A* with the Manhattan distance heuristic
3
The best state to expand with g(n) = 0and h(n) = 0is...
1 2 3
4 0 6
7 5 8
Expanding this node...
The best state to expand with g(n) = 1and h(n) = 1is...
1 2 3
4 5 6
7 0 8
Expanding this node...
1 2 3
4 5 6
7 8 0
Hurray! Reached Goal State
The time taken to solve the problem 1milliseconds
To solve this problem the search algorithm expanded a total of 7 nodes
The maximum number of nodes in the queue at any one time was 5
The depth of the goal node was  2
```

Code:
State.java

```
/* @Author : Lokesh Koppaka, Student ID: 862123164
 * Class : State
 * Description : This class holds the information about the current state of the n puzzle.
 *  The following are the information it stores
 *  1. puzzleState - the current state tiles arrangement
 *  2. puzzleSize - holds the size of the puzzle
 *  3. uniformCost - the uniform cost for the state i.e g(n)
 *  4. heuristicCost - the heuristicCost for the state i.e h(n)
 *  5. blankLocation - the row and col index of blank
 */

public class State implements Comparable<State>{
	int[][] puzzleState; // instance variable - holds current tiles arrangement
	int puzzleSize; // instance variable - holds the size of the puzzle
	int uniformCost; // instance variable - holds uniform cost
	int heuristicCost; // instance variable - holds heuristic cost
	int[] blankLocation; // instance variable - holds row and col index of blank
	// Constructor - performs instance variable initializations
	public State(int[][] puzzleState, int puzzleSize,int uniformCost,int heuristicCost, Boolean
visited, int[] blankLocation){
			this.puzzleState = puzzleState;
			this.uniformCost = uniformCost;
			this.heuristicCost = heuristicCost;
			this.blankLocation = blankLocation;
			this.puzzleSize = puzzleSize;
		}
	// Overridden standard equals method to check equality of two objects based on puzzle
arrangement
	public boolean equals(Object o) {
		// Logic to check if two puzzle arrangements are same
		if(o instanceof State) {
			State currentState = (State) o;
			for(int i = 0; i < puzzleSize; i++) {
				for(int j = 0; j < puzzleSize; j++) {
				 if(puzzleState[i][j] != currentState.puzzleState[i][j]) {
							return false;
					}
				}
			}
			return true;
		}
		return false;
	}
	// Overridden standard compareTo method inorder to perform sort based on totalCost = g(n) +
h(n)
	@Override
	public int compareTo(State currentState) {
		// TODO Auto-generated method stub
		return (this.uniformCost + this.heuristicCost) - (currentState.uniformCost +
currentState.heuristicCost);
	}

	}
```

AStar.java

```
/* @Course : CS205 Artificial Intelligence
 * @Author: Lokesh Koppaka, Student ID: 862123164
 * ClassName: AStar
 * Description : This class computes the solution for n-puzzle problem using A* Algorithm with
Misplaced Tile & Manhatten Distance heuristics
 */

import java.util.ArrayList; // Imported to implement dynamic Queue
import java.util.Collections; // Imported to perform sort
import java.util.List; // Super class for ArrayList hence required to import
import java.util.Scanner; // Imported to scan input from user

public class AStar {

	public int puzzleSize; // instance variable - holds size of the puzzle
	private int maxQueueLen = 0; // instance variable - keeps track of max Queue length
	private int currentQueueLen; // instance variable - keeps track of current size of the
Queue
	private int expandedNodes = 1;// instance variable - keeps track of No of expanded nodes
	private List<State> visitedList; // List - holds list of visited states, this is necessary
to avoid search on already visited states
	private List<State> queue; // List - used to implement queue
	private int[][] deafultPuzz = {{8,7,1},{6,0,2},{5,4,3}}; // Default State for puzzle size
3x3
	private int[][] deafultPuzzFour = {{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,0}}; //
Default State for puzzle size 4X4
	private int[][] deafultPuzzFive =
{{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15},{16,17,18,19,20},{21,22,23,24,0}}; // Default State for
puzzle size 5X5
	private int[][] GoalState;
	// Constructor - performs instantiation of instance variables
	AStar(){
		visitedList = new ArrayList<State>();
		queue = new ArrayList<State>();
	}
	/* Method Name - constructGoalState
	 * Description - construct goal state for given puzzle size
	 */
	private void constructGoalState() {
		GoalState = new int[puzzleSize][puzzleSize];
		int i = 1;
		for(int j = 0;  j < puzzleSize; j++) {
			for(int k = 0; k < puzzleSize; k++) {
				GoalState[j][k] = i;
				i++;
			}
		}
		GoalState[puzzleSize - 1][puzzleSize - 1] = 0;
	}
	/* Method Name - addRowsToPuzzle
	 * Parameters  - puzzle(The puzzle grid), row(row index where the vals need to added),
vals(the values to be added)
	 * Description - Takes the puzzle grid and add the values in order at specified row index
	 * Return      - returns the puzzle grid with added values to specified row
	 */
	private int[][] addRowToPuzzle(int[][] puzzle, int row, String[] vals) {
		int len = vals.length;
		for(int i = 0; i< len; i++) {
			puzzle[row][i] = Integer.valueOf(vals[i]);
		}
```

```java
        return puzzle;
}
/* Method Name - printPuzzle
 * Parameters  - puzzle(The puzzle grid)
 * Description - prints the values in the grid to standard output
 */
private void printPuzzle(int[][] puzzle) {
        for(int i = 0; i < puzzleSize; i++) {
                for(int j = 0; j < puzzleSize; j++) {
                        System.out.print(puzzle[i][j] + " ");
                }
                System.out.println();
        }
}
/* Method Name - isStateVisited
 * Parameters  - currentState(The current puzzle state)
 * Description - checks if the currentState is visited by looking into visitedList
 * Return      - returns true if it is visited else false
 */
private boolean isStateVisited(State currentState) {
        if(visitedList.contains(currentState)) {
                return true;
        }
        return false;
}
/* Method Name - genericSearch
 * Parameters  - puzzle(The puzzle grid), queuingFunction
 *                        queuingFunction
 *              1 - Uniform Cost Search
 *              2 - A* with misplaced Tiles as heuristic
 *              3 - A* with manhatten Dist as heuristic
 * Description - performs generic search on the puzzle
 */
private void genericSearch(int[][] puzzle, int queuingFunction) {
 int uCost = 0; // holds uniform cost
 int hCost = 0; // holds heuristic cost
 int[] blankLoc = new int[2]; // holds row and column index of the blank
 // Logic to compute blank location indexes
 for(int i = 0; i < puzzleSize; i++) {
        for(int j = 0; j < puzzleSize; j++) {
                if(puzzle[i][j] == 0) {
                        blankLoc[0] = i;
                        blankLoc[1] = j;
                        break;
                }
        }
 }
 // Logic to create current state and add it to queue
 State currentState = new State(puzzle,puzzleSize, uCost, hCost, false, blankLoc);
 queue.add(currentState);
 maxQueueLen = 1;
 currentQueueLen = 1;
 long startTime = System.nanoTime();
 long endTime;
 while(!queue.isEmpty()) {
        // Logic to pop the current State and add it to visitedList
        currentState = queue.remove(0);
        currentQueueLen --;
        visitedList.add(currentState);
        // Logic to check if the current State is the Goal State
        if(isGoalState(currentState)) {
```

```
                    // Logic - if the current state is goal state
                    endTime = System.nanoTime();
                    long timeElapsed = endTime - startTime;
                    printPuzzle(currentState.puzzleState); // Print the puzzle
                    System.out.println("Hurray! Reached Goal State");
                    System.out.println("The time taken to solve the problem " + timeElapsed/
1000000 + "milliseconds");
                    System.out.println("To solve this problem the search algorithm expanded a
total of " + expandedNodes + " nodes" );
                    System.out.println("The maximum number of nodes in the queue at any one time
was " + maxQueueLen);
                    System.out.println("The depth of the goal node was  " +
currentState.uniformCost);
                    return;
              }else {
                    // Logic - if the current state is not the goal state
                    System.out.println("The best state to expand with g(n) = " +
currentState.uniformCost + "and h(n) = " + currentState.heuristicCost + "is...");
                    printPuzzle(currentState.puzzleState); // Print the puzzle
                    System.out.println("Expanding this node...");
                    expand(currentState, queuingFunction); // Logic to expand the current node
and add it queue
                    Collections.sort(queue); // Sort the queue - this cases the least cost state
to be in front of the queue
              }
        }
        System.out.println("Its impossible to find the goal state!");
        System.out.println("To solve this problem the search algorithm expanded a total of " +
expandedNodes + " nodes" );
        System.out.println("The maximum number of nodes in the queue at any one time was " +
maxQueueLen);
        System.out.println("The depth of the goal node was  " + currentState.uniformCost);

       }
       /* Method Name - expand
        * Parameters  - currentState(The current puzzle state),queuingFunction
        *                         queuingFunction
        *               1 - Uniform Cost Search
        *               2 - A* with misplaced Tiles as heuristic
        *               3 - A* with manhatten Dist as heuristic
        * Description - expands to all the possible states from the current state
        */
       private void expand(State currentState, int queuingFunction) {
             // Logic to get the row and column index
             int blankI = currentState.blankLocation[0];
             int blankJ = currentState.blankLocation[1];
             //queue = new ArrayList<State>(); // added new!
             // Logic if blank can Move Top
             if(blankI > 0){
                   appendValidStatesToQueue(currentState,blankI- 1,blankJ, blankI, blankJ,
queuingFunction);
             }
             // Logic if blank can Move bottom
             if(blankI < puzzleSize - 1) {
                   appendValidStatesToQueue(currentState,blankI + 1,blankJ, blankI, blankJ,
queuingFunction);
             }
             // Logic if blank can Move left
             if(blankJ < puzzleSize - 1) {
                   appendValidStatesToQueue(currentState,blankI,blankJ + 1, blankI, blankJ,
queuingFunction);
```

```
			}
			// Logic if blank can Move right
			if(blankJ > 0) {
				appendValidStatesToQueue(currentState,blankI,blankJ - 1, blankI, blankJ, 
queuingFunction);
			}

		}
		/* Method Name - appendValidStatesToQueue
		 * Parameters  - currentState(The current puzzle state)
		 *                    - newI, newJ (The new indexes where the blank can be moved)
		 *                    - curI, curJ (The current indexes of the blank )
		 *                        queuingFunction
		 *              1 - Uniform Cost Search
		 *              2 - A* with misplaced Tiles as heuristic
		 *              3 - A* with manhatten Dist as heuristic
		 * Description - Creates a new State, checks if it is not visited then computes g(n) and 
h(n) and add it to the queue
		 */
		private void appendValidStatesToQueue(State curState, int newI, int newJ, int curI, int 
curJ, int queuingFunction) {
			int[][] puzzleState = new int[puzzleSize][puzzleSize];
			// Logic to copy the current puzzle state
			for(int i = 0; i < puzzleSize; i++) {
				for(int j = 0; j < puzzleSize; j++) {
					puzzleState[i][j] = curState.puzzleState[i][j];
				}
			}
			// Logic to create the new puzzle state
			puzzleState[curI][curJ] = puzzleState[newI][newJ];
			puzzleState[newI][newJ] = 0;
			// Logic to create new state check if it is not visited, then compute g(n), h(n) 
based on queuing Function
				State newState = new State(puzzleState,puzzleSize,0,0,false,new int[] {newI, 
newJ});
				if(!isStateVisited(newState)) {
					newState.uniformCost = curState.uniformCost + 1;
					if(queuingFunction == 2) {
						newState.heuristicCost = heuristicUsingMisplaceTiles(newState);
					}else if(queuingFunction == 3) {
						newState.heuristicCost = heuristicUsingManhattenDist(newState);
					}
					expandedNodes ++;
					queue.add(newState);
					currentQueueLen ++;
					// Logic to compute max Queue length
					if(currentQueueLen > maxQueueLen) {
						maxQueueLen = currentQueueLen;
					}
				}
		}
		/* Method Name - heuristicUsingMisplaceTiles
		 * Parameters  - currentState(The current puzzle state)
		 * Description - computes the heuristic cost by counting number of misplaced tiles
		 * Return      - returns heuristic cost
		 */
		private int heuristicUsingMisplaceTiles(State currentState){
			int[][] stateVals = currentState.puzzleState;
			int count = 0;
			// Logic to compute number of tiles that are misplaced
			for(int i = 0; i < puzzleSize; i++) {
				for(int j = 0; j < puzzleSize; j++) {
					if(stateVals[i][j] != GoalState[i][j]) {
						count ++;
					}
				}
			}
			return count;
		}
		/* Method Name - heuristicUsingManhattenDist
		 * Parameters  - currentState(The current puzzle state)
		 * Description - computes the heuristic cost by calculated the mahantten distance of each
tile from its goal state
		 * Return      - returns heuristic cost
		 */
		private int heuristicUsingManhattenDist(State currentState) {
			int[][] stateVals = currentState.puzzleState;
			int count = 0;
			// Logic to compute heuristic
			for(int i = 0; i < puzzleSize; i++) {
				for(int j = 0;j < puzzleSize; j++) {
					// if it is blank skip it
					if(stateVals[i][j] == 0) {
						continue;
					}else if(stateVals[i][j] != GoalState[i][j]) {
						// Logic to compute mahantten distance
						int dist[] = returnLocation(stateVals[i][j]);
						count += Math.abs(i - dist[0]);
						count += Math.abs(j - dist[1]);
					}
				}
			}
			System.out.println("****** In Manhatten ************\n");
			printPuzzle(currentState.puzzleState);
			System.out.println("Hanhatten Dist = " + count);
			return count;
		}
		/* Method Name - returnLocation
		 * Parameters  - tile value
		 * Description - returns the location in of the tile value in terms i and j index
		 * Return      - array of size 2 holding the ith and jth index of the passed tile value
		 */
		private int[] returnLocation(int val) {
			for(int i = 0; i < puzzleSize; i++) {
				for(int j = 0; j < puzzleSize; j++) {
					if(val == GoalState[i][j]) {
						return new int[] {i,j};
					}
				}
			}
			return null;
		}
		/* Method Name - isGoalState
		 * Parameters  - currentState(The current puzzle state)
		 * Description - checks if the current state is a goal state
		 * Return      - returns true if current state is goal state else false
		 */
		private Boolean isGoalState(State currentState) {
			int[][] stateVals = currentState.puzzleState;
			// Logic to check if current state is a goal state
			for(int i = 0; i < puzzleSize; i++) {
				for(int j = 0; j < puzzleSize; j++) {
```

```
                        if(stateVals[i][j] != GoalState[i][j]) {
                              return false;
                        }
                  }
            }
            return true;
      }
      public static void main(String[] args) {
            AStar eightPuzzle =  new AStar(); // AStar instance
            //State
            Scanner sc = new Scanner(System.in); // used to get user input from standard input
            int algoChoice; // holds the user algorithm choice
            // Logic for prompting user preferences
            System.out.println("Welcome to CS 205 n puzzle Solver!!");
            System.out.println("Enter the puzzle size");
            eightPuzzle.puzzleSize = sc.nextInt();
            sc.nextLine();
            eightPuzzle.constructGoalState();
            System.out.println("Type \"1\" to use default puzzle, or \"2\" to enter your own
puzzle.");
          int puzzleChoice = sc.nextInt();
          sc.nextLine();
          int puzzle[][] = new int[eightPuzzle.puzzleSize][eightPuzzle.puzzleSize]; // holds the
puzzle
          // Logic for default puzzle choice
          if(puzzleChoice == 1) {
            if(eightPuzzle.puzzleSize == 3) {
                  puzzle = eightPuzzle.deafultPuzz;
            }else if(eightPuzzle.puzzleSize == 4){
                  puzzle = eightPuzzle.deafultPuzzFour;
            }else if(eightPuzzle.puzzleSize == 5) {
                  puzzle = eightPuzzle.deafultPuzzFive;
            }
          }else if(puzzleChoice == 2) {
            // Logic for user inputing puzzle
            String[] input = new String[eightPuzzle.puzzleSize];
            puzzle = new int[eightPuzzle.puzzleSize][eightPuzzle.puzzleSize];
            System.out.println("Enter your puzzle, use  a zero to represent  the blank");
            for(int i = 0; i < eightPuzzle.puzzleSize; i++) {
                  System.out.println("Enter the "+ (i+1) +" row, use space or tabs between
numbers");
                  String inputStr = sc.nextLine();
                  input = inputStr.split("\\s+");
                  puzzle = eightPuzzle.addRowToPuzzle(puzzle,i,input);
            }
          }else {
            return;
          }
          System.out.println("Puzzle Intial State");
          eightPuzzle.printPuzzle(puzzle); // print the puzzle
          // User algo choice prompt
          System.out.println("Enter your choice of 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");
          algoChoice = sc.nextInt();// read user algo choice
          eightPuzzle.genericSearch(puzzle, algoChoice); // call to generic search
      }

}
```