Hoshank Mali
801254416

Intelligent Systems
Project 1 – Sliding Puzzle using A*

1) An overview of the problem being solved

An eight puzzle is a simple game with a 3 x 3 grid (containing 9 squares). There is an empty square in one of the squares. The purpose is to move the squares around into various positions while keeping the numbers in the "target/goal state”. The image on the left represents the "3 x 3" board and 8 puzzle tiles in its unsolved initial state. The sliding puzzle is said to be like the eight-puzzle problem. The start state might be thought of as the image on the left below. The graphic on the right represents the desired state. We can think about the following things based on the project question:

Start: 7,2,4,5,0,6,8,3,1
Goal: 0,1,2,3,4,5,6,7,8

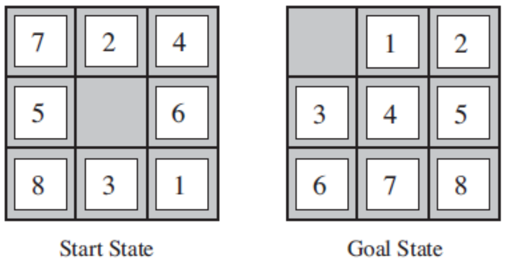


Fig: 1.1

The motions for the 8-puzzle can be easily interpreted as follows.
-Move empty space (blank) to the “Left”, move blank “Up”, move blank to the “Right” and move blank “Down”.
-Production rules are used to model these actions, which operate on the state descriptions in the proper way.
-The termination is based on the goal condition.

The sliding puzzle can also have N-tiles. But for the purpose of this project, we consider 8 tiles and the problem formulation for the same can be given as below:

- **States**: A state description describes each of the eight tiles, as well as the blank in one of the nine spaces.
- **Starting point**: Any state can be used as the starting point. It's worth mentioning that each goal can be accomplished by starting from half of the possible states.
- **Actions**: At their most basic level, actions are movements of the blank space to the left, right, up, or down. Different subsets of these are possible depending on where the blank is.
- **Transition model**: This produces the next state given a state and an action; for example, applying Left to the start state swaps the 5 and the blank.
- **Goal test**: The goal test examines whether the present state matches the goal configuration.

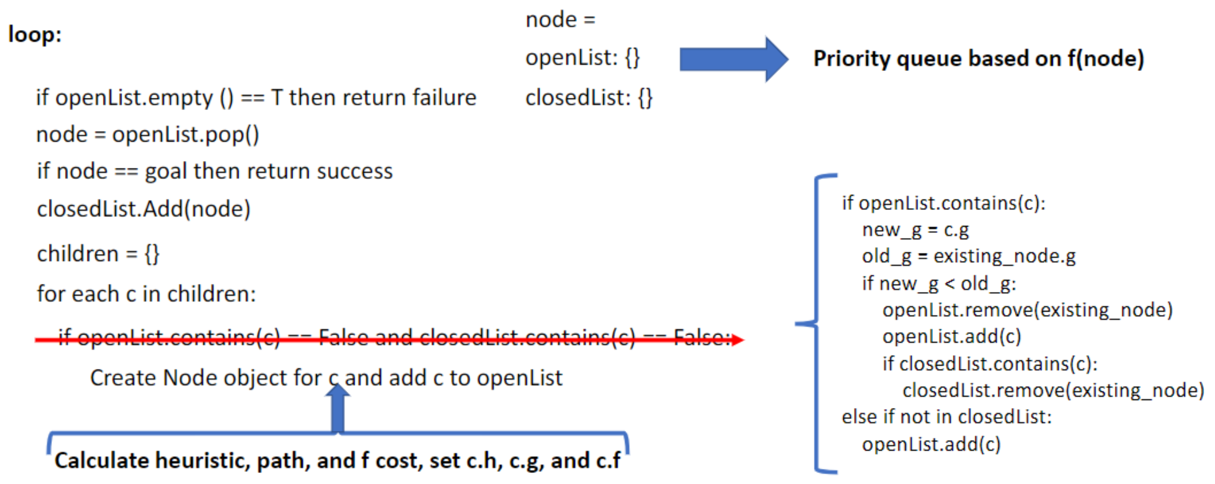
Hoshank Mali
801254416

- **Path cost**: The path cost is the number of steps along the path multiplied by one, i.e. each step costs one.

2) An overview of the algorithm used

A* is a path finding and graph traversal algorithm that uses informed search. It's a hybrid of uniform cost search and best first search that saves costly paths from being expanded. A* star employs acceptable heuristics, which is ideal because the path to goal is never overestimated. A* star employs the evaluation function:
f(n) = g(n) + h to calculate distance (n)
g(n) = the total cost of getting to n thus far
h(n) = cost estimate from n to objective
f(n) = total estimated cost of journey from n to objective

Heuristic function: The heuristic function is a technique of informing the search about a goal's direction. It gives information that can be used to estimate which neighboring node will lead to the desired destination. The heuristic function considered for this project is the number of misplaced tiles in the board.
Below is the pseudocode of **how A* algorithm works**:
We use open and closed list, where the newly generated states are pushed into the **open** list and after expanding the current state, it is pushed into the **closed** list. A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state. The algorithm chooses the best possible action and proceeds in that path.


Priority queue based on f(node)
Calculate heuristic, path, and f cost, set c.h, c.g, and c.f

```
node =
openList: {}
closedList: {}

loop:

    if openList.empty () == T then return failure
    node = openList.pop()
    if node == goal then return success
    closedList.Add(node)
    children = {}
    for each c in children:
        ~~if openList.contains(c) == False and closedList.contains(c) == False:~~
            Create Node object for c and add c to openList
```

```
if openList.contains(c):
    new_g = c.g
    old_g = existing_node.g
    if new_g < old_g:
        openList.remove(existing_node)
        openList.add(c)
        if closedList.contains(c):
            closedList.remove(existing_node)
else if not in closedList:
    openList.add(c)
```

Time and space complexity of A* can be given as follows:

$$\text{Time complexity: } O\left(b^{\epsilon d}\right), \epsilon = \frac{h^{*} - h}{h^{*}} \text{ (relative error)}$$

$$\text{Space complexity: } O\left(b^{\epsilon d}\right), \epsilon = \frac{h^{*} - h}{h^{*}}$$

h^{*} = true cost to goal
h = estimated cost to goal
d = depth of goal
b = branching factor

Hoshank Mali
801254416

3) An overview of how your code implements the algorithm

- Which functions/methods correspond to which parts of the algorithm?
  - **isSolvable()**: Function to check if the given board is solvable. It checks the number of inversions in the board. If the number of inversions are even then the function returns true or else it returns false.
  - **getHeuristic()**: returns the heuristic cost based on the number of misplaced tiles for a given board.
  - **qContains()**: checks if the given state is already present in the open list i.e priority queue. Returns true if present, false if not.
  - **LContains()**: checks if the given state is already present in the closed list ie ArrayList. Returns true if present, false if not.
  - **qContains1()**: If the board is present in open list, function compares the path cost of the current board and the board in the open list. If the path cost of current is less then removes the existing board and adds the current board in open list.
  - **getNeighbours()**: the function is used to get neighboring boards of given board and decides if it is needed to add the boards in the open list or not using the functions qContains(), LContains() and qContains1().
  - **Expand()**: Expands the boards from the initial board to the goal board with the help of getNeighbours() function. Uses a priority queue and arrayList as open list and closed list.
  - **isGoal()**: checks if the given board is equal to the goal board. Returns true if goal board is found.
  - **getPath()**: Prints the path from the initial path to the goal board using the parent attribute of the aStar class.

- How are the termination conditions implemented?
  - The Algorithm checks if the board is solvable or not. If a board is not solvable the program terminates or checks for the next board.
  - If the board is solvable, then the program terminates if goal board is found and returns the path, path cost and the number of nodes expanded.
  - The program also terminates if the open list (Priority Queue) becomes empty.

- An overview of the results

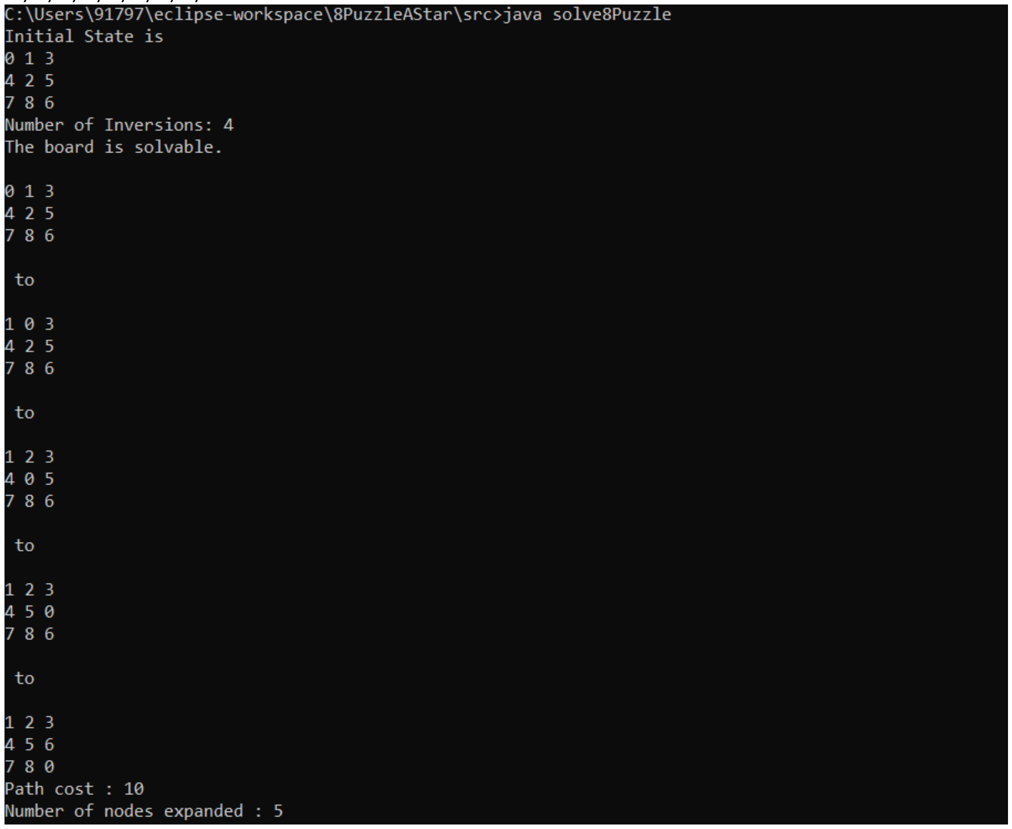
  All of the test cases given in the project assignment give the goal board except the last one which is not solvable and as displayed in the output.

  Only one test case (8,3,0,5,6,1,7,4,2) takes around 5 minutes due to large number of expanding nodes in this case.

Hoshank Mali
801254416

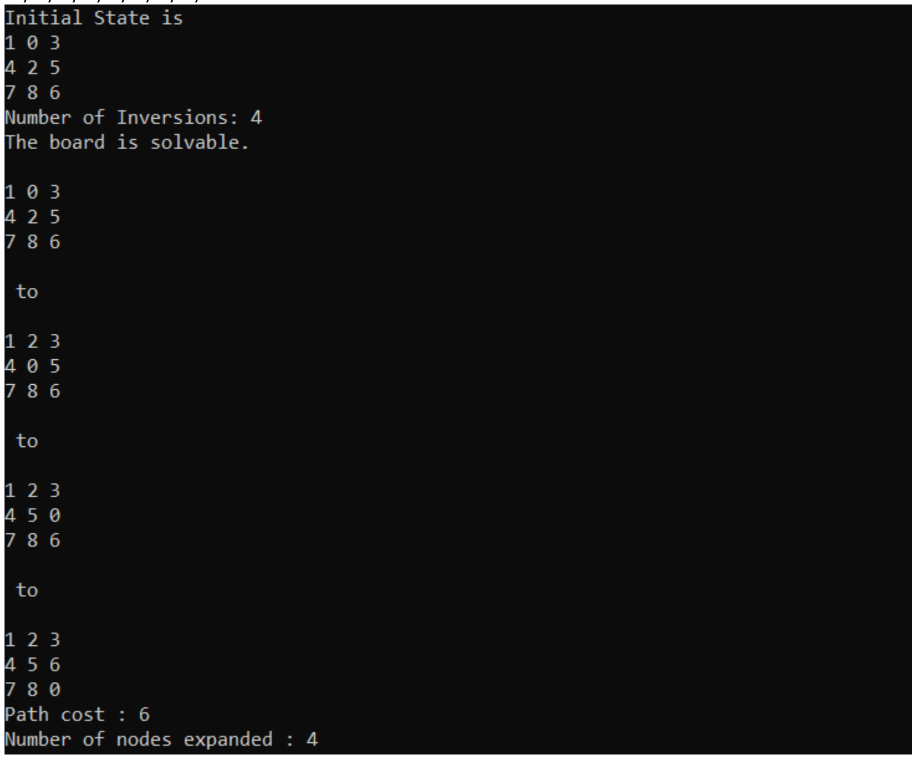
Please find attached screenshot of the output/results:

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



```
C:\Users\91797\eclipse-workspace\8PuzzleAStar\src>java solve8Puzzle
Initial State is
0 1 3
4 2 5
7 8 6
Number of Inversions: 4
The board is solvable.

0 1 3
4 2 5
7 8 6

 to

1 0 3
4 2 5
7 8 6

 to

1 2 3
4 0 5
7 8 6

 to

1 2 3
4 5 0
7 8 6

 to

1 2 3
4 5 6
7 8 0
Path cost : 10
Number of nodes expanded : 5
```

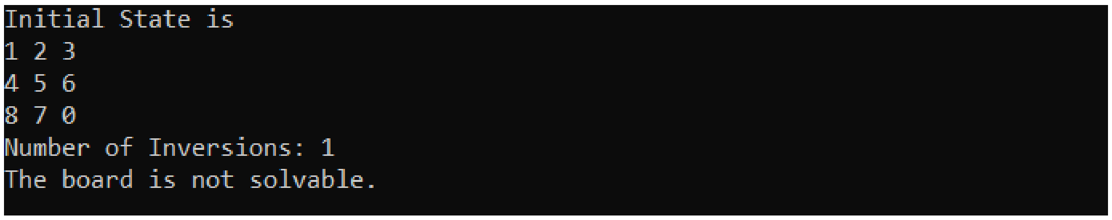
1,0,3,4,2,5,7,8,6



```
Initial State is
1 0 3
4 2 5
7 8 6
Number of Inversions: 4
The board is solvable.

1 0 3
4 2 5
7 8 6

 to

1 2 3
4 0 5
7 8 6

 to

1 2 3
4 5 0
7 8 6

 to

1 2 3
4 5 6
7 8 0
Path cost : 6
Number of nodes expanded : 4
```

Hoshank Mali
801254416

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



```
Initial State is
1 2 3
4 5 6
8 7 0
Number of Inversions: 1
The board is not solvable.
```