# 1. Problem

8 - Puzzle game solver, the user writes the initial state to show steps to the final state: 0,1,2,3,4,5,6,7,8

# 2. Program Description

The program is used to solve the 8-puzzle problem. the user enters the initial state then the program checks that no duplicates and that the numbers entered are from 0:8 then the solving operation takes place based on storing all possible moves and searching with an uninformed search like Breadth-first and Depth-first searches and with an informed Search like A* and showing all possible moves according to the way of search and also give the user path to goal, cost of the path, nodes expanded from the tree, search depth and running time of the program.

# 3. Algorithms

## a) BFS Algorithm

- It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node. And the structure used in this algorithm is Queue.

**BFS search**

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

## b) DFS Algorithm

- a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph. And the structure used in this algorithm is Stack.

**DFS search**

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

Taken from the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI)

## c) A* Algorithm

- It is a searching algorithm that is used to find the shortest path between an initial and a final point, also It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

**A* search**

Taken from the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI)

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

**To calculate the heuristic,** it uses the Manhattan or Euclidean distance:

**a) Manhattan Distance**

- It is the sum of absolute values of differences in the goal's x and y coordinates and
- the current cell's x and y coordinates respectively,
- h = abs(current cell:x - goal:x) + abs(current cell:y - goal:y)

**b) Euclidean Distance**

- It is the distance between the current cell and the goal cell using the distance formula
- h = sqrt((current cell:x - goal:x)^2 + (current cell.y - goal:y)^2).

# 4. Libraries used

- java. util.*
- java.io. *
- java. util. ArrayList
- java. util. Queue
- java. util. Stack
- javax.swing.*;
- java.awt. *

# 5. Use Data Structures

- Stack
- Queue
- HashSet
- Tree
- Map. Entry

# 6. Contained classes:

## a. BFS

- insert two array lists: path list- move list
- (bfs)Method:

- It takes the initial state from the user
- Start time to calculate the time of runtime
- Set goal “012345678”
- Call node and set previous goal and null for parent and move
- Declaring Queue to work with BFS, which contains a search way and checks if it reaches the goal
- Declaring all data structure, we will use in the search
- Start the Search by add and poll from queue according to BFS algorithm (that explain in algorithms section)
- Add moves and new parent to print and boarder functions at work_space
- Get the node’s parent and print the path to the goal and all the wanted information.
- Get the index of 0 to start, get neighbors and add the move in the tree, and undo the last moves
- As it has two axis x, y:
    - if it subtracts one from the x-axis value, it will move to the left, and the number of moves = 1 and add to node1
    - if it adds one to the x-axis, it will move to the right, and the number of moves = 4 and add to node4
    - If y if it subtracts one from the y-axis value, it will move up, and the number of moves = 2 and add to node2
    - if it adds one to the y-axis, it will move down, and the number of moves = 3 and add to node3
- Undo the last move through the assign the value by adding 1 if it is subtracted and subtracting if it is added.
- Print Not solvable if it doesn’t find a solution by BFS search

## b. DFS

- insert two array lists: (path list - move list)
- (dfs)Method:
  - It takes the initial state from the user
  - Start time to calculate the time of runtime
  - Set goal “012345678”
  - Call node and set previous goal and null for parent and move
  - Declaring Stack to work with DFS, which contains a search way and checks if it reaches the goal
  - Declaring all data structure, we will use in the search
  - Start the Search by push and pop from stack according to DFS algorithm (that explain in algorithms section)
  - Add moves and new parent to print and boarder functions at work_space
  - Get the node’s parent and print the path to the goal and all the wanted information.
  - Get the index of 0 to start, get neighbors and add the move in the tree, and undo the last moves
  - As it has two axis x, y:
    - if it subtracts one from the x-axis value, it will move to the left, and the number of moves = 1 and add to node1
    - if it adds one to the x-axis, it will move to the right, and the number of moves = 4 and add to node4
    - If y if it subtracts one from the y-axis value, it will move up, and the number of moves = 2 and add to node2
    - if it adds one to the y-axis, it will move down, and the number of moves = 3 and add to node3
  - Undo the last move through the assign the value by adding 1 if it is subtracted and subtracting if it is added.
  - Print Not solvable if it doesn’t find a solution by DFS search

## c. EuclideanAStar

- Euclidean (method):
  - it takes current x, current y, goal x and goal y then it calculates Euclidean distance
- GetHEuclidean (method):
  - to get the heuristic with Euclidean Distance
- Insert two array lists: (path list - move list)
- AStarEuclidean(method):
  - Take initial state from user
  - Start time to calculate the time of runtime
  - Set goal “012345678”
  - Call node and set a previous goal and null for parent and move
  - Declaring priority Queue by Use the Map. entry to work with A*
  - Declaring all data structures, we will use in the search
  - Start the Search by poll from priority Queue according to the A* algorithm (that explain in the algorithms section)
  - Add moves and new parent to print and boarder functions at work_space
  - Get the node’s parent and print the path to the goal and all the wanted information.
  - Get the index of 0 to start, get neighbors and add the move in the tree, and undo the last moves
  - As it has two axis x, y:
    - if it subtracts one from the x-axis value, it will move to the left, and the number of moves = 1 and add to node1
    - if it adds one to the x-axis, it will move to the right, and the number of moves = 4 and add to node4
    - If y if it subtracts one from the y-axis value, it will move up, and the number of moves = 2 and add to node2
    - if it adds one to the y-axis, it will move down, and the number of moves = 3 and add to node3
  - Undo the last move through the assign the value by adding 1 if it is subtracted and subtracting if it is added.
  - Print Not solvable if it doesn’t find a solution by A * search

## d. ManhattanAStar

- Manhattan (method):
  - it takes current x, current y, goal x, and goal y then calculate Manhattan distance and returns its value
- GetHManhattan(method):
  - to get the heuristic with Manhattan Distance
- Insert two array lists: (path list - move list)
- AStarManhattan(method):
  - Start time to calculate the time of runtime
  - Set goal “012345678”
  - call node and set a previous goal and null for parent and move
  - declaring priority Queue by Use the Map. entry to work with A*
  - Declaring all data structures, we will use in the search
  - Start the Search by poll from priority Queue according to the A* algorithm (that explain in the algorithms section)
  - Add moves and new parent to print and boarder functions at work_space
  - Get the node’s parent and print the path to the goal and all the wanted information.
  - Get the index of 0 to start, get neighbors and add the move in the tree, and undo the last moves
  - it has two axis x, y:
    - if it subtracts one from the x-axis value, it will move to the left, and the number of moves = 1 and add to node1
    - if it adds one to the x-axis, it will move to the right, and the number of moves = 4 and add to node4
    - If y if it subtracts one from the y-axis value, it will move up, and the number of moves = 2 and add to node2
    - if it adds one to the y-axis, it will move down, and the number of moves = 3 and add to node3
  - Undo the last move through the assign the value by adding 1 if it is subtracted and subtracting if it is added.
  - Print Not solvable if it doesn’t find a solution by A * search

## e. Node

- Declaring all the nodes and moves that will enter in the tree.
- Making constructors receive all the data about the node and give every node null values

## f. Main

- Ask for Initial state
- Take Initial state and do operation to check if its length 9 and the numbers only from 0 to 8 without duplicated
- After checking Initial State is qualified turns to choose search algorithm
- Choose between DFS, BFS and A* with two types (Euclidean and Manhattan)
- After user choose search algorithm, call function of algorithm and print require information
- Ask for Another operation with same Initial state if not reset old Initial state and ask for Initial state again

## g. Tree

- Declaring node root.
- Make a constructor that return the root with null.
- Method (add):
  - add the Node with its data in the tree by the parameters:
    - root
    - board (location of index)
    - number (every number link with specific node)
    - parent
    - move

## h. Work_Space

- Method (printBoard):
  - Print numbers from 2D array line by line and split with | to show 8 puzzle shape with border
- Method (toBoardArray):
  - Return String Explored in 2D array
- Method (toBoardString):
  - Return the 2D array puzzle to string
- Method (print):
  - Print moves
  - Show path of goal
  - Print: Depth - Nodes Expanded - Execution time of Euclidean distance
- Method (GoalX):
  - Get the index of x for the chosen element in the goal
- Method (GoalY):
  - Get the index of Y for the chosen element in the goal

## i. Start (JFrame)

- Use java swing to create interfaces to the program
- Start graphical user interface (GUI)
- Create frame to take the Initial State
- Constructor (Start):
  - To set locations of button, label, and text field
- Method (actionPerformed):
  - Open Another Frame to show the puzzle and its solver
- Methods (callBFS – callDFS – callManhattan – callEuc )
  - To show steps with information in console
  - To give (path to goal) to the GUI
  - To give (movement) to the GUI

## j. Puzzle (JFrame)

- Create the interface of the puzzle with the initial value taken from user in start window
- Declaration buttons and labels
- Declare 2D array, one for algorithms and other for path list and move list
- Method showPuzzle ():
  - Set location, added the buttons and titles to panels, set fonts and color
- Methods (mouseEntered()- mouseExited()-mouseClicked-mousePressed()-mouseReleased()):
  - To express when the mouse moves
- Methods (setNumbers):
  - To change the button value which the user enters
- Methods (actionPerformed):
  - To perform the function when the user clicks on the button of the algorithms then it will apply the algorithm technique by calling method of it which user click on its button.

# 7. Simple Run with Console

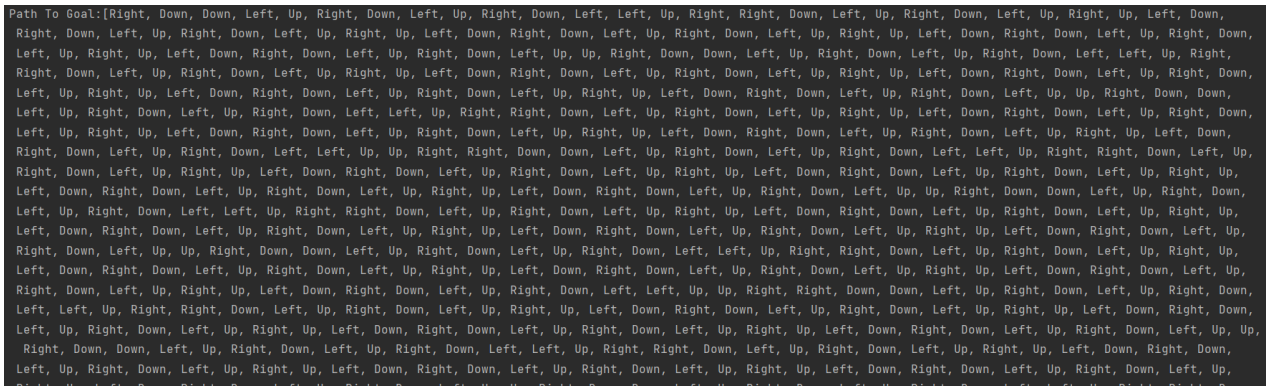
## i. Welcome Screen, Check the Initial State and choose between different algorithms

```
------------WELCOME TO 8 PUZZLE------------
Please Enter Initial state : 102365874
The Initial State is :102365874
Duplicate Characters in above string are :
1 is TRUE
0 is TRUE
2 is TRUE
3 is TRUE
6 is TRUE
5 is TRUE
8 is TRUE
7 is TRUE
4 is TRUE
Choose :
1. DFS
2. BFS
3. A*
Please Choose one type of search algorithms : 1
```

## ii. Print the steps and directions and final shape (Goal)

```
Move 61355 to Goal: Up
Up
|1|2|5|
|3|0|4|
|6|7|8|
Move 61356 to Goal: Right
Right
|1|2|5|
|3|4|0|
|6|7|8|
Move 61357 to Goal: Up
Up
|1|2|0|
|3|4|5|
|6|7|8|
Move 61358 to Goal: Left
Left
|1|0|2|
|3|4|5|
|6|7|8|
Move 61359 to Goal: Left
Left
|0|1|2|
|3|4|5|
|6|7|8|
```

## iii. Print the path of goal

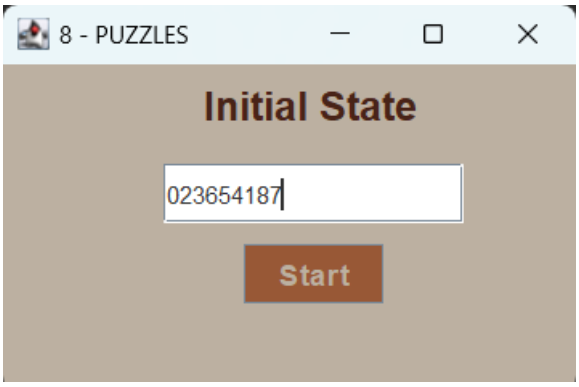



## iv. Print Cost, Depth, Node Expanded and Run time the Ask if do another operation or not

```
Cost of path= 61359
Depth= 61359
Nodes Expanded= 114909
Execution time of Euclidean distance: 17.715953 seconds
Do you want another operation with same Initial State (y/n)
```

# 8. Simple Run with GUI

## i. Ask to Enter Initial State




## ii. Then go to 8 puzzles interface with algorithms to solve

# THANK YOU