Name: Amr Mohamed Gaber
ID: 4818

LAB 1

To run the program, type:

```
python3 searchAlg.py n,n,n,n,n,n,n,n,n searchType
```

i.e:

```
>> python3 searchAlg.py 3,1,2,6,4,5,0,7,8 bfs
```

Global Assumptions:

- The order in which the nodes will enter the data structures, will follow the reverse order of the board state when moving the empty tile Upwards, Downwards, to the Left, and to the Right.
- The running time will be calculated starting from creating the first instance of the object puzzleBoard with our initial/input.
- The total cost will be equal to the number of nodes traversed after our parent root to reach our solution node.
- If the code takes a long time to run, it's because I created the “explored” hashset as a list by accident and couldn’t change it back unless i make drastic changes to the code structure.

Name: Amr Mohamed Gaber
ID: 4818

1) BFS:

```
python3 ai_lab1/searchAlg.py 3,1,2,6,4,5,0,7,8 bfs
```

```
._____________
| 03| 01| 02|
|___|___|___|
| 06| 04| 05|
|___|___|___|
| 00| 07| 08|
|___|___|___|
DIRECTION:  Start
._____________
| 03| 01| 02|
|___|___|___|
| 00| 04| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Up
._____________
| 00| 01| 02|
|___|___|___|
| 03| 04| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Up
*** DONE ***

Path Taken:   -> Start -> Move Up -> Move Up
Cost of that path:  2
Nodes Expanded:  4
Search Depth:  2
Running Time: 0.0015 sec
```

Data structures used:

a List which is used as a (FIFO) Queue by using List.pop(0) to remove the first element in the List.

Name: Amr Mohamed Gaber
ID: 4818

Assumptions:

We will use a None value inserted in the List to indicate the end of each level, hence the depth of our solution.
After finishing each level, the next List.pop(0) will be equal to None, in which case we will increase our current depth as we reached the end of the level and we still couldn't find a solution.

Code:

```
def bfsSearch(initialState):
    '''
    The main idea here is that we will use the None value to
    indicate the end of each level hence the depth of our solution.

    After finishing a level, the next state will be equal to None,
    in which case we are able to easily identify the end of the current level.

    We will use a list as a queue for holding our states.
    '''
    # initializing our root (depth=0)
    frontier = [initialState, None]
    explored = []
    depth = 0
    # to find when to add None (end of level) to our queue
    flag = 0
    while len(frontier) > 0:
        # .pop(0) -> remove the first element of the list
        state = frontier.pop(0)
        # if None then we reached the end of our current level and raise flag
        if not state:
            depth += 1
            flag = 1
            continue
```

Name: Amr Mohamed Gaber
ID: 4818

```
    # add None to our queue and reset the flag value
    if flag:
        frontier.append(None)
        flag = 0

    # add the current state to our explored list
    explored.append(state.tilesToList())

    # check if the current state is the solution to our puzzle
    if goalTest(state):
        return success(state, len(explored), depth)

    # add the neighbours to our current node
    # only if they are not in (explored U frontier)
    for neighbour in state.children():
        if not checkExp(neighbour, explored):
            if not checkFront(neighbour, frontier):
                frontier.append(neighbour)

# in case we couldn't solve the puzzle
return False
```

we start at the initialState, and explore all of the neighbour nodes at the current depth that aren’t explored or in our frontier Queue prior to moving on to the nodes at the next depth level.

Name: Amr Mohamed Gaber
ID: 4818

2) DFS:

```
python3 ai_lab1/searchAlg.py 3,1,2,6,4,5,0,7,8 dfs
```

```
.___________
| 03| 01| 02|
|___|___|___|
| 06| 04| 05|
|___|___|___|
| 00| 07| 08|
|___|___|___|
DIRECTION:  Start
.___________
| 03| 01| 02|
|___|___|___|
| 06| 04| 05|
|___|___|___|
| 07| 00| 08|
|___|___|___|
DIRECTION:  Move Right
.___________
| 03| 01| 02|
|___|___|___|
| 06| 04| 05|
|___|___|___|
| 07| 08| 00|
|___|___|___|
DIRECTION:  Move Right
.___________
| 03| 01| 02|
|___|___|___|
| 06| 04| 00|
|___|___|___|
| 07| 08| 05|
|___|___|___|
DIRECTION:  Move Up
```

```
.___________
| 03| 01| 02|
|___|___|___|
| 06| 00| 04|
|___|___|___|
| 07| 08| 05|
|___|___|___|
DIRECTION:  Move Left
.___________
| 03| 01| 02|
|___|___|___|
| 00| 06| 04|
|___|___|___|
| 07| 08| 05|
|___|___|___|
DIRECTION:  Move Left
.___________
| 03| 01| 02|
|___|___|___|
| 07| 06| 04|
|___|___|___|
| 00| 08| 05|
|___|___|___|
DIRECTION:  Move Down
.___________
| 03| 01| 02|
|___|___|___|
| 07| 06| 04|
|___|___|___|
| 08| 00| 05|
|___|___|___|
DIRECTION:  Move Right
```

```
.___________
| 03| 01| 02|
|___|___|___|
| 07| 06| 04|
|___|___|___|
| 08| 05| 00|
|___|___|___|
DIRECTION:  Move Right
.___________
| 03| 01| 02|
|___|___|___|
| 07| 06| 00|
|___|___|___|
| 08| 05| 04|
|___|___|___|
DIRECTION:  Move Up
.___________
| 03| 01| 02|
|___|___|___|
| 07| 00| 06|
|___|___|___|
| 08| 05| 04|
|___|___|___|
DIRECTION:  Move Left
.___________
| 03| 01| 02|
|___|___|___|
| 00| 07| 06|
|___|___|___|
| 08| 05| 04|
|___|___|___|
DIRECTION:  Move Left
```

```
.___________
| 03| 01| 02|
|___|___|___|
| 08| 07| 06|
|___|___|___|
| 00| 05| 04|
|___|___|___|
DIRECTION:  Move Down
.___________
| 03| 01| 02|
|___|___|___|
| 08| 07| 06|
|___|___|___|
| 05| 00| 04|
|___|___|___|
DIRECTION:  Move Right
.___________
| 03| 01| 02|
|___|___|___|
| 08| 07| 06|
|___|___|___|
| 05| 04| 00|
|___|___|___|
DIRECTION:  Move Right
.___________
| 03| 01| 02|
|___|___|___|
| 08| 07| 00|
|___|___|___|
| 05| 04| 06|
|___|___|___|
DIRECTION:  Move Up
```

...

(MANY NODES LATER)

...

Name: Amr Mohamed Gaber
ID: 4818

```
.___________
| 01| 04| 02|
|___|___|___|
| 03| 07| 05|
|___|___|___|
| 06| 00| 08|
|___|___|___|
DIRECTION:  Move Left
.___________
| 01| 04| 02|
|___|___|___|
| 03| 00| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Up
.___________
| 01| 00| 02|
|___|___|___|
| 03| 04| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Up
.___________
| 00| 01| 02|
|___|___|___|
| 03| 04| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Left
*** DONE ***
```

```
Cost of that path:  50
Nodes Expanded:  2977
Search Depth:  50
Running Time: 9.5191 sec
```

```
Path Taken:   -> Start -> Move Right -> Move Right -> Move Up -> Move Left -> Move Left
-> Move Down -> Move Right -> Move Right -> Move Up -> Move Left -> Move Left -> Move Do
wn -> Move Right -> Move Right -> Move Up -> Move Left -> Move Left -> Move Down -> Move
 Right -> Move Right -> Move Up -> Move Left -> Move Left -> Move Down -> Move Right ->
Move Right -> Move Up -> Move Left -> Move Down -> Move Right -> Move Up -> Move Left ->
 Move Left -> Move Up -> Move Right -> Move Right -> Move Down -> Move Left -> Move Up -
> Move Right -> Move Down -> Move Left -> Move Up -> Move Right -> Move Down -> Move Dow
n -> Move Left -> Move Up -> Move Up -> Move Left
```

Name: Amr Mohamed Gaber
ID: 4818

Data structures used:

a List which is used as a (LIFO) Stack by using List.pop() to remove the last element in the List.

Assumptions:

We will use a None value inserted in the List to indicate the current depth, by finding the number of None values remaining in our Stack we will get the current depth for our node.
After finishing each node, if the next List.pop() equals to None,this means we reached the maximum end for this node and we will pop another node to expand, otherwise, we will keep expanding our current node adding a None value to our stack before adding its children to our frontier stack and explored list.
Since we don’t have unlimited Stack (Memory) to hold all the expanded nodes till we reach our maximum depth,and to avoid memory leaks/errors, we will Limit our depth to 50 for simplicity.

Name: Amr Mohamed Gaber
ID: 4818

Code:

```
def dfsSearch(initialState):
    '''
    The main idea here is that we will use the None value to
    calculate our current depth.

    if the current state has a neighbour not in (explored U frontier),
    we will add None, otherwise we will not add None, because the current state
    has no children and we will have to pop another state from the stack.

    We will use a list as a stack for holding our states.
    '''
    # initializing our root (depth=0)
    frontier = [initialState]
    explored = []
    # not used
    maxDepth = 0
    depth = 0
    # to find when to add None (node expanded) to our stack
    flag = 0

    while len(frontier) > 0:
        # .pop() -> remove the last element of the list
        state = frontier.pop()

        # if None then we are going to the parent
        if not state:
            continue

        # the number of None in our list will be our depth
        depth = frontier.count(None)
```

Name: Amr Mohamed Gaber
ID: 4818

```
        # not used but can calculate our max Depth
        maxDepth = max(maxDepth, depth)

        # add the current state to our explored list
        explored.append(state.tilesToList())

        # check if the current state is the solution to our puzzle
        if goalTest(state):
            return success(state, len(explored), depth)

        # limit our search to a specific depth to not have the states expand to
reach our max Depth,
        # which can cause Memory &/or deepcopy errors
        depthLimit = 50
        if depth < depthLimit:
            # find where is the start of our expansion nodes
            # to add our None before them
            row = len(frontier)
            # add the neighbours to our current node
            # only if they are not in (explored U frontier)
            for neighbour in state.children():
                if not checkExp(neighbour, explored):
                    if not checkFront(neighbour, frontier):
                        # raise the flag if we found a neighbour
                        flag = 1
                        frontier.append(neighbour)

            # if we didn't expand, we will not add None
            if flag == 1:
                # insert None before the added neighbours
                frontier.insert(row, None)
                # reset flag
                flag = 0
```

Name: Amr Mohamed Gaber
ID: 4818

```
    # in case we couldn't solve the puzzle
    return False
```

we start at the initialState, then explore to our depthLimit along each branch before backtracking.

Name: Amr Mohamed Gaber
ID: 4818

3) A*:

<u>Manhattan:</u>

```
python3 ai_lab1/searchAlg.py 3,1,2,6,4,5,0,7,8 astar,manhattan
```

```
.____________
| 03| 01| 02|
|___|___|___|
| 06| 04| 05|
|___|___|___|
| 00| 07| 08|
|___|___|___|
DIRECTION:  Start
.____________
| 03| 01| 02|
|___|___|___|
| 00| 04| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Up
.____________
| 00| 01| 02|
|___|___|___|
| 03| 04| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Up
*** DONE ***

Path Taken:   -> Start -> Move Up -> Move Up
Cost of that path:  2
Nodes Expanded:  3
Search Depth:  3
Running Time: 0.0010 sec
```

Name: Amr Mohamed Gaber
ID: 4818

Euclidian:

```
python3 ai_lab1/searchAlg.py 3,1,2,6,4,5,0,7,8 astar,euclician
```

```
.____________
| 03| 01| 02|
|___|___|___|
| 06| 04| 05|
|___|___|___|
| 00| 07| 08|
|___|___|___|
DIRECTION:  Start
.____________
| 03| 01| 02|
|___|___|___|
| 00| 04| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Up
.____________
| 00| 01| 02|
|___|___|___|
| 03| 04| 05|
|___|___|___|
| 06| 07| 08|
|___|___|___|
DIRECTION:  Move Up
*** DONE ***

Path Taken:   -> Start -> Move Up -> Move Up
Cost of that path:  2
Nodes Expanded:  3
Search Depth:  3
Running Time: 0.0009 sec
```

Name: Amr Mohamed Gaber
ID: 4818

Data structures used:

a min heap which will take a node (cost, boardState) as input, which will heapify it according to the cost calculated depending on the type of the Heuristic algorithm.

Code:

```
def astarSearch(initialState, type):
    frontier = []
    explored = []
    # calculate the cost depending on the type of Heuristics
    cost = manhattanDistance(
        initialState) if type == "manhattan" else euclideanDistance(initialState)
    # create a tuple for the node to hold the cost which will be used to by the
 heapsort,
    # and its state
    node = (cost, initialState)
    heapq.heappush(frontier, node)
    while len(frontier) > 0:
        state = heapq.heappop(frontier)[1]

        # add the current state to our explored list
        explored.append(state.tilesToList())

        #  check if the current state is the solution to our puzzle
        if goalTest(state):
            # find the depth by tracing back the root parent of the state
            depth = 0
            temp = state
            while temp:
                # print(temp)
                depth += 1
```

Name: Amr Mohamed Gaber
ID: 4818

```
            temp = temp.parent
        return success(state, len(explored), depth)


    # add the neighbours to our current node
    # only if they are not in (explored U frontier)
    for neighbour in state.children():
        if not checkExp(neighbour, explored):
            if not checkFront(neighbour, [state for cost, state in frontier]):
                # calculate the cost depending on the type of Heuristics
                cost = manhattanDistance(
                    neighbour) if type == "manhattan" else
 euclideanDistance(neighbour)
                node = (cost, neighbour)
                heapq.heappush(frontier, node)
  # in case we couldn't solve the puzzle
  return False


# calculate the Manhattan Distance
# h=abs(currentcell.x-goal.x)+abs(currentcell.y-goal.y)
def manhattanDistance(state):
  tiles = state.tilesToList()
  h = state.cost
  for i in range(len(tiles)):
      if tiles[i] != 0:
          currentCellX, currentCellY, goalCellX, goalCellY = getXY(
              i, tiles[i], state.size)
          h += (abs(currentCellX - goalCellX) +
                abs(currentCellY - goalCellY))
  return h
```

Name: Amr Mohamed Gaber
ID: 4818

```
# calculate the Euclidean Distance
# h=sqrt((currentcell.x-goal.x)^2+sqrt((currentcell.y-goal.y)^2)
def euclideanDistance(state):
   tiles = state.tilesToList()
   h = state.cost
   for i in range(len(tiles)):
       if tiles[i] != 0:
           currentCellX, currentCellY, goalCellX, goalCellY = getXY(
               i, tiles[i], state.size)
           h += ((currentCellX - goalCellX)**2 +
                 (currentCellY - goalCellY)**2)**0.5
   return h


# get x, y values for the currentCell and the goalCell
def getXY(i, tile, size):
   return i // size, i % size, tile // size, tile % size
```

By checking our running time, we find out that euclidean's algorithm for A* search is a bit faster compared to manhattan's algorithm.