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.

ID: 4818

1) BFS:

```
python3 ai_lab1/searchAlg.py 3,1,2,6,4,5,0,7,8 bfs
  03 01 02
     04 05
  06
  00 07 08
DIRECTION: Start
 03 01 02
 00 04 05
 06 07 08
DIRECTION: Move Up
 00| 01| 02|
 031
     04 | 05 |
     07 08
  06
DIRECTION: Move Up
*** DONE ***
Path Taken: -> Start -> Move Up -> Move Up
Cost of that path: 2
Nodes Expanded: 4
```

Data structures used:

Running Time: 0.0015 sec

Search Depth: 2

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

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.
  frontier = [initialState, None]
  explored = []
  depth = 0
  flag = 0
  while len(frontier) > 0:
      state = frontier.pop(0)
      if not state:
           depth += 1
           flag = 1
```

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)

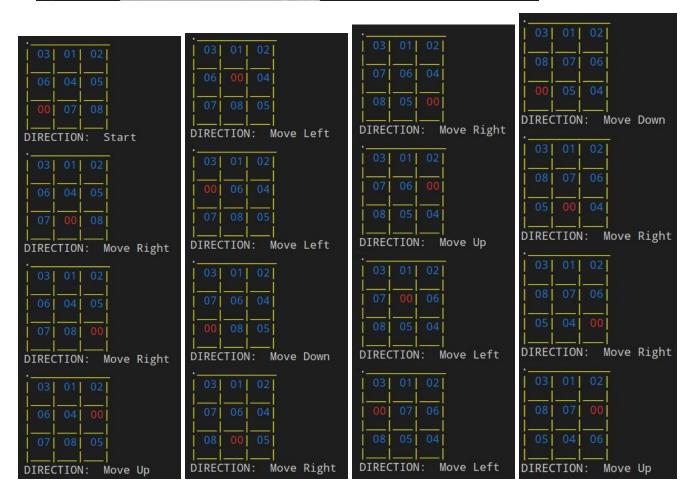
# in case we couldn't solve the puzzle
return False
```

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.

ID: 4818

2) DFS:

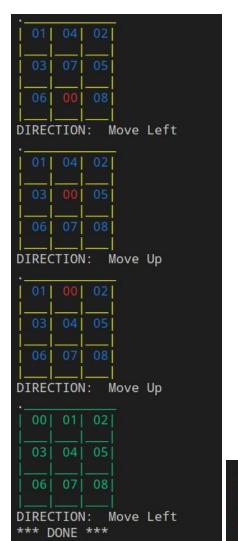
python3 ai_lab1/searchAlg.py 3,1,2,6,4,5,0,7,8 dfs



(MANY NODES LATER)

- - -

ID: 4818



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 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 Right -> Move Up -> Move Left -> Move Right -> Move Up -> Move Left -> Move Right -> Move Up -> Move Left -> Move Down -> Move Left -> Move Up -> Move Right -> Move Down -> Move

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.

ID: 4818

Code:

```
def dfsSearch(initialState):
   frontier = [initialState]
  explored = []
  maxDepth = 0
  depth = 0
  flag = 0
  while len(frontier) > 0:
      state = frontier.pop()
       if not state:
      depth = frontier.count(None)
```

ID: 4818

```
maxDepth = max(maxDepth, depth)
explored.append(state.tilesToList())
if goalTest(state):
    return success(state, len(explored), depth)
depthLimit = 50
if depth < depthLimit:</pre>
    row = len(frontier)
    for neighbour in state.children():
        if not checkExp(neighbour, explored):
            if not checkFront(neighbour, frontier):
                flag = 1
                frontier.append(neighbour)
    if flag == 1:
        frontier.insert(row, None)
        flag = 0
```

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.

ID: 4818

3) A*:

Manhattan:

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

```
01 02
     04 05
 06
 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
```

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
```

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 = []
  cost = manhattanDistance(
      initialState) if type == "manhattan" else euclideanDistance(initialState)
  node = (cost, initialState)
  heapq.heappush(frontier, node)
  while len(frontier) > 0:
      state = heapq.heappop(frontier)[1]
      explored.append(state.tilesToList())
      if goalTest(state):
           depth = 0
           temp = state
          while temp:
               depth += 1
```

ID: 4818

```
temp = temp.parent
          return success(state, len(explored), depth)
      for neighbour in state.children():
           if not checkExp(neighbour, explored):
               if not checkFront(neighbour, [state for cost, state in frontier]):
                   cost = manhattanDistance(
                       neighbour) if type == "manhattan" else
 euclideanDistance(neighbour)
                   node = (cost, neighbour)
                  heapq.heappush(frontier, node)
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))
```

ID: 4818

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