Introduction:

8-puzzle game using formed and informed search.

Programming language used: Python.

The problem will be discussed through Uninformed search criteria:

1-Breadth-First-Search (BFS)

2-Depth-First-Search (DFS)

And informed search criteria:

1- A* search using:

-Manhattan heuristic

-Euclidean heuristic

Discussion:

- First check that the input is solvable or not
  Unsolvable test case:

```
Please Enter Your puzzle: 812043765
This Puzzle can't be solved
Please Enter Your puzzle:
```

- **BFS test cases:**

1st test case 125340678:

```
Please Enter Your puzzle: 125340678
1.BFS 2.DFS 3.A star with Manhattan Heuristic 4.A star with Euclidean Distance 5.EXIT
Please Enter Your Choice(1-4): 1
Path To Goal:
1 2 5
3 4 0
6 7 8

1 2 0
3 4 5
6 7 8

1 0 2
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Explored Nodes:
1 0 5
3 2 4
6 7 8
```

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

1 0 2
3 4 5
6 7 8

1 2 5
3 7 4
6 0 8

1 2 5
3 0 4
6 7 8

1 2 5
3 4 8
6 0 7

1 2 5
3 4 8
6 7 0

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

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

1 2 0
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Depth is  4
cost= 3
Time elapsed:  0.0005002021789550781
```

2nd test case 312045678:

```
Please Enter Your puzzle: 312045678
1.BFS 2.DFS 3.A star with Manhattan Heuristic 4.A star with Euclidean Distance 5.EXIT
Please Enter Your Choice(1-4): 1
Path To Goal:
3 1 2
0 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Explored Nodes:
0 1 2
3 4 5
6 7 8

3 1 2
0 4 5
6 7 8

Depth is  1
cost= 1
Time elapsed:  0.0
```

3rd test case 087654321:

Note: The path to goal and explored nodes in this test case is very big.

```
Depth is  31
cost= 30
Time elapsed:  7.736814975738525
```

- **DFS test cases:**

Note: The path to goal and explored nodes in DFS is very big.

1st test case 125340678:

```
Depth is  66123
cost= 59123
Time elapsed:  6.452682256698608
```

2nd test case 312045678:

```
Depth is  66123
cost= 1
Time elapsed:  6.145670652389526
```

3rd test case 087654321:

```
Depth is  62856
cost= 62856
Time elapsed:  4.869813919067383
```

- **A* test cases:**

Using Manhattan heuristic:

1st test case 125340678:

Note: The path to goal in this test case is very big.

```
Explored Nodes:
1 0 2
3 4 5
6 7 8

1 2 5
3 4 0
6 7 8

1 2 0
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Depth is  3
cost= 3
Time elapsed:  0.0
```

2nd test case 312045678:

```
Please Enter Your puzzle: 312045678
1.BFS 2.DFS 3.A star with Manhattan Heuristic 4.A star with Euclidean Distance 5.EXIT
Please Enter Your Choice(1-4): 3
Path To Goal:
3 1 2
0 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Explored Nodes:
3 1 2
0 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Depth is  1
cost= 1
Time elapsed:  0.0
```

3rd test case 087654321:

Note: The path to goal and explored nodes in this test case is very big.

```
Depth is  30
cost= 30
Time elapsed:  3.749133586883545
```

<u>Using Euclidean heuristic:</u>

1st test case 125340678:

Note: The path to goal in this test case is very big.

```
Explored Nodes:
1 0 2
3 4 5
6 7 8

1 2 5
3 4 0
6 7 8

1 2 0
3 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Depth is  3
cost= 3
Time elapsed:  0.0
```

2nd test case 312045678:

```
Please Enter Your puzzle: 312045678
1.BFS 2.DFS 3.A star with Manhattan Heuristic 4.A star with Euclidean Distance 5.EXIT
Please Enter Your Choice(1-4): 4
Path To Goal:
3 1 2
0 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Explored Nodes:
3 1 2
0 4 5
6 7 8

0 1 2
3 4 5
6 7 8

Depth is  1
cost= 1
Time elapsed:  0.0004925727844238281
```

3rd test case 087654321:

Note: The path to goal and explored nodes in this test case is very big.

```
Depth is  30
cost= 30
Time elapsed:  7.740764856338501
```

Conclusion:

- Manhattan heuristic is more admissible than Euclidean heuristic.
- Informed search criteria A* with Manhattan heuristic is the most efficient algorithm.

Data structures used:

Dictionary, set, list.

Code:

The code is separated into 5 files.

Main:

C: > Users > Mariam Mohamed > PycharmProjects > aiProject > main.py

```python
import time
from a_star import A_star
from bfs import bfs
from dfs import dfs
from functions import validateInput, checkIfSolvable, printNodes

if __name__ == '__main__':
    stop = 0
    while stop == 0:
        try:
            inputPuzzle = input("Please Enter Your puzzle: ")
            if validateInput(inputPuzzle) and checkIfSolvable(inputPuzzle):
                print("1.BFS 2.DFS 3.A star with Manhattan Heuristic 4.A star with Euclidean Distance 5.EXIT")
                choice = input("Please Enter Your Choice(1-4): ")
                t0 = time.time()  # start timer
                choice = int(choice)
                current_State = inputPuzzle
                parents = {}
                expanded = set()
                maxDepth = 0

                if choice == 1:
                    parents, current_State, maxDepth, expanded = bfs(inputPuzzle, "012345678")
                elif choice == 2:
                    parents, current_State, maxDepth, expanded = dfs(inputPuzzle, "012345678")
                elif choice == 3:
                    parents, current_State, maxDepth, expanded = A_star(0, inputPuzzle, "012345678")
                elif choice == 4:
                    parents, current_State, maxDepth, expanded = A_star(1, inputPuzzle, "012345678")
                elif choice == 5:
                    stop = 1
                trace_state = current_State  # This is used to trace the path to goal
                cost = 0
                pathToGoal = []  # Used to trace the path to goal
                while parents[trace_state]:  # while not none
                    cost += 1  # increasing cost as we go up
                    pathToGoal.append(trace_state)  # adding element to pathToGoal list to be printed
                    trace_state = parents[trace_state]  # Going to the parent
                pathToGoal.append(trace_state)  # adding last element which doesn't have a parent
                printNodes(pathToGoal, 0)  # printing path To Goal
                printNodes(list(expanded), 1)  # printing explored nodes after turning them to list of string
                print("Depth is ", maxDepth)
                print("cost=", cost)
                t1 = time.time() - t0  # stopping timer

                print("Time elapsed: ", t1)  # printing CPU seconds elapsed (floating point)
                print()

        except:
            stop = 1
```

Functions:

functions.py ×

functions.py > ...

```python

def testGoal(explored, goalTest):
    if explored == goalTest:
        return True
    else:
        return False


def getNeighbors(currentPuzzle):  # This function is used to find the neighbors (children) of the current puzzle
    state = str(currentPuzzle)
    index = currentPuzzle.find('0')  # finding index of zero to see available slides
    neighbors = []
    if index > 2:  # move up
        neighbors.append(swap(state, index - 3))
    if index < 6:  # move down
        neighbors.append(swap(state, index + 3))
    if index % 3 > 0:  # move left
        neighbors.append(swap(state, index - 1))
    if index % 3 < 2:  # move right
        neighbors.append(swap(state, index + 1))
    return neighbors


def swap(currentPuzzle, neighbor):  # This function is used to swap 0 with the available neighbor
    list1 = list(currentPuzzle)  # converting string (puzzle) to list
    b = neighbor  # index of neighbor
    a = list1.index('0')  # getting index of 0
    list1[a], list1[b] = list1[b], list1[a]  # swapping
    return list1
```

functions.py ×

C: > Users > Mariam Mohamed > PycharmProjects > aiProject > functions.py

```python

def commonCode(explored, state, depth, parent, frontier_state):
    for neighbour in getNeighbors(state):  # iterating through neighbors
        neighbour = ''.join(neighbour)  # joining list of characters to get a string
        if neighbour not in frontier_state and neighbour not in explored:
            # checking if string exists in frontier or explored
            parent[neighbour] = state  # the parent of the neighbor is the state
            depth[neighbour] = depth[state] + 1  # depth is increased by 1
            frontier_state[neighbour] = True  # inserting new neighbor in frontier_state
    return depth, parent, frontier_state


def printNodes(printingNodes, option):
    # printingNodes is list of string that has either explored nodes or has nodes from Path to goal
    if option == 0:
        print("Path To Goal:")
        printingNodes.reverse()  # reversing to start from original puzzle to goal not vice versa
    else:
        print("Explored Nodes:")
    for word in range(len(printingNodes)):  # word is the index of string inside printingNodes
        for i in range(0, 10, 3):  # "i" is the index of letters in the string
            print(" ".join(printingNodes[word][i:i + 3]))  # adding space between each number

```

functions.py ×

C: > Users > Mariam Mohamed > PycharmProjects > aiProject > functions.py

```python


def validateInput(inputPuzzleState):  # validate that input is 9 digits from 0 to 9 with no duplicates
    if len(inputPuzzleState) != 9:
        print("Incorrect Puzzle(It should 9 digits)")
        return False
    try:
        int(inputPuzzleState)  # used in try except to see if the input is only integer or not
        duplicates = [number for number in list(inputPuzzleState) if
                      list(inputPuzzleState).count(number) > 1]  # count duplicates and save them
        if len(duplicates) != 0:
            print("Incorrect Puzzle (Repeated Digit)")
            return False
        return True
    except(Exception,):
        print("Incorrect Puzzle (Not integer)")
        return False


def checkIfSolvable(inputPuzzleState):  # check if the puzzle is solvable.....This can be done
    # by checking for even inversions or odd inversions. If it is even then it can be solved because every slide (
    # change in puzzle) add two inversions or remove 2 inversions So if it is odd inversions it won't be solved This
    # can be done by simply counting how many large digit come before a smaller digit
    inversions = 0
    for i in range(0, 9):  # iterating through all elements
        for j in range(i + 1, 9):
            if int(inputPuzzleState[i]) > int(inputPuzzleState[j]) and int(inputPuzzleState[i]) != 0 and int(
                    inputPuzzleState[j]) != 0:  # 0 isn't counted in inversions
                inversions += 1
    if inversions % 2 != 0:  # if it is odd, it can't be solved
        print("This Puzzle can't be solved")
        return False
    return True

```

BFS:

bfs.py ×

C: > Users > Mariam Mohamed > PycharmProjects > aiProject > bfs.py

```python
from functions import testGoal, commonCode


def bfs(initialState, goalTest):
    explored = set()  # defining empty set that will contain the expanded (explored) states
    parent = {initialState: None}  # parent is used to track path to goal
    frontier_state = {
        initialState: True}  # hashing data for faster performance , key is initial state
    # , value is true means neighbor found
    depth = {initialState: 0}  # will be used to find the longest depth
    while frontier_state:  # while there is state in frontiers explore
        state = next(iter(frontier_state))  # converting frontier dict to iterable and finding first state
        frontier_state.pop(state)  # removing element from frontier
        explored.add(state)  # adding state to explored
        if testGoal(state, goalTest):  # checking if goal was reached
            return parent, state, depth[max(depth, key=depth.get)], explored
        depth, parent, frontier_state = commonCode(explored, state, depth, parent, frontier_state)
    return False


```

A*:

a_star.py ×

a_star.py > ...

```python
import math

from functions import testGoal, getNeighbors


def euclideanDistance(prev_row, goal_row, prev_col, goal_col):  # Euclidean Heuristic function
    # h = sqrt((current cell.x – goal.x)**2 + sqrt((current cell.y – goal.y)**2)
    return math.sqrt((prev_row - goal_row) ** 2 + (prev_col - goal_col) ** 2)


def manhattanDistance(prev_row, goal_row, prev_col, goal_col):  # Manhattan Heuristic function
    # h(n) = abs(currentState.x - goal.x) + abs(currentState.y - goal.y)
    return abs(prev_row - goal_row) + abs(prev_col - goal_col)


def heuristicDecider(option, state, goal):  # Decides which heuristic to use
    state = list(state)  # converting state(string) to list of characters '0' -> '9'
    goal = list(goal)  # converting goal (string) to list of characters '0' -> '9'
    H = 0
    for i, item in enumerate(
            state):  # iterating through the state's characters  ("i" is the index, "item" is the character(Digit))
        if item == '0':
            i = 0  # Removing zero as it isn't counted in the heuristic functions
            item = 0
        else:  # any digit other than '0'
            # i = int(i)
            item = goal.index(item)  # Modification for any goal.
        prev_row, prev_col = int(i / 3), i % 3  # i/3 get the row , i%3 get the column
        goal_row, goal_col = int(item / 3), int(item) % 3
        if option == 0:
            H += manhattanDistance(prev_row, goal_row, prev_col, goal_col)
        else:
            H += euclideanDistance(prev_row, goal_row, prev_col, goal_col)
    return H


def A_star(option, initialState, goalState):
    # f = G + h
    G = 0  # G is the cost between states
    parent = {initialState: None}  # parent is used to track path to goal
    frontier = {initialState: heuristicDecider(option, initialState, goalState) + G}
    # hashing data for faster performance , key is initial state , value is F=G+H
    explored = set()  # defining empty set that will contain the expanded (explored) states
    depth = {initialState: 0}  # will be used to find the longest depth
    while frontier:
        state = min(frontier, key=frontier.get)  # finding minimum heuristic value (just like a priority queue)
        # state = next(iter(frontier))
        G = frontier[state] - heuristicDecider(option, state, goalState)  # Tracking G F=G+H G=F-H
        frontier.pop(state)  # removing element from frontier
        explored.add(state)  # adding state to explored
        # print(state)
        G += 1  # increasing cost by 1 (we went down in our tree)
        if testGoal(state, goalState):  # checking if goal was reached
            return parent, state, depth[max(depth, key=depth.get)], explored
        for neighbour in getNeighbors(state):  # iterating through neighbors
            neighbour = ''.join(neighbour)  # joining list of characters to get a string
            if neighbour not in frontier and neighbour not in explored:
                # checking if string exists in frontier or explored
                parent[neighbour] = state  # the parent of the neighbor is the state
                depth[neighbour] = depth[state] + 1  # depth is increased by 1
                frontier[neighbour] = heuristicDecider(option, neighbour,
                                                       goalState) + G  # inserting new neighbor in frontier_state
    return False
```

DFS:

```
dfs.py
C: > Users > Mariam Mohamed > PycharmProjects > aiProject > dfs.py
from functions import commonCode, testGoal


def dfs(initialState, goalTest):
    explored = set()  # defining empty set that will contain the expanded (explored) states
    parent = {initialState: None}  # parent is used to track path to goal
    frontier_state = {
        initialState: True}  # hashing data for faster performance , key is initial state , value is true means
    # neighbor found
    depth = {initialState: 0}  # will be used to find the longest depth
    while frontier_state:
        # state = next(iter(reversed(frontier_state)))
        # frontier_state.pop(state)
        state = frontier_state.popitem()[0]  # pop last element in frontier_state (just like a stack)
        explored.add(state)  # adding state to explored
        if testGoal(state, goalTest):  # checking if goal was reached
            return parent, state, depth[max(depth, key=depth.get)], explored
        depth, parent, frontier_state = commonCode(explored, state, depth, parent, frontier_state)
        # function that has common code which get neighbor of a state
    return False
```