8 Puzzle Game

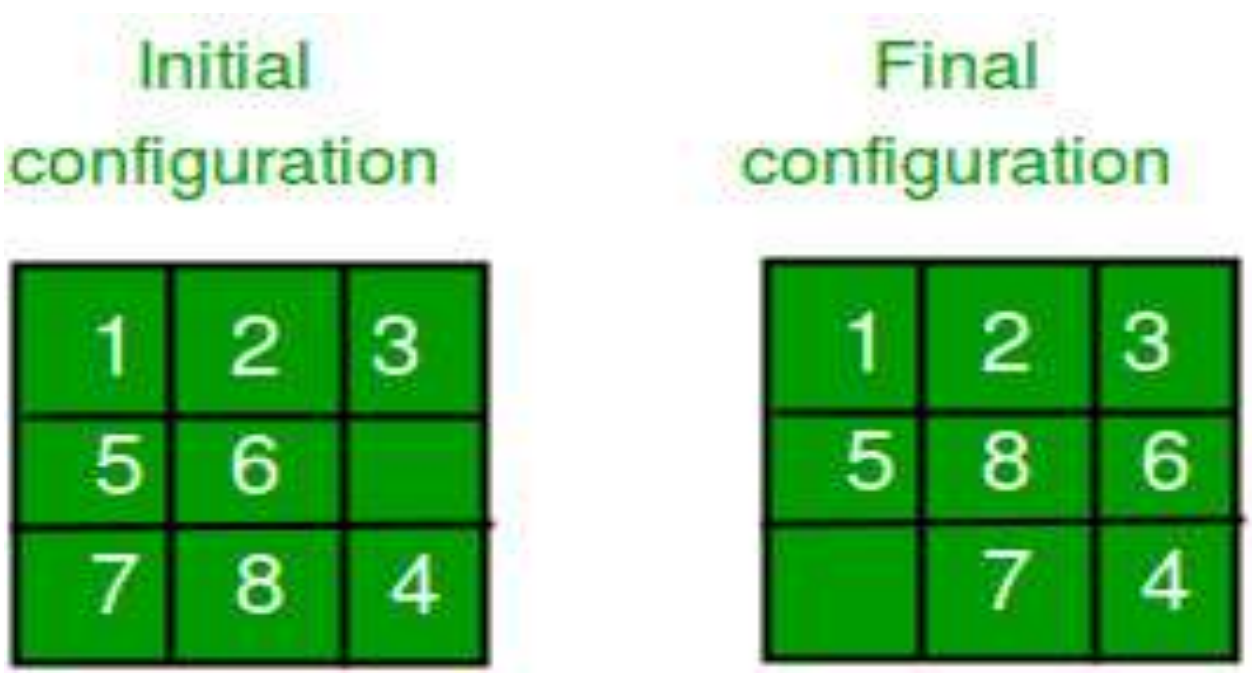
Presented to: Dr. Amira Youssef and Eng. Heba Abd El Atty

Done by:

Mennatallah abdelmeguid ID: 6538
Martina Mamdouh ID:6150

Mariette Magdy ID: 6147

Theodora George ID:6628



$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & 5 \\ \hline 3 & 4 & \\ \hline 6 & 7 & 8 \\ \hline \end{array} \quad \Longrightarrow \quad child = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \quad \Longrightarrow \quad child = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \quad \Longrightarrow \quad child = \begin{array}{|c|c|c|}\hline & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

Functions us

```
def get_neighbours(state):
i    = 0     k = 0
    neighbours = []
     for item in state:
j    = 0          for element in item:            if element == 0:
k    = 1                  break
            j = j + 1        if k == 1:            break        i = i + 1

    neighbour = list(state)     n = 0
     for item in neighbour:
        neighbour[n] = list(item)        n = n + 1
     if i < 2:
        neighbour[i][j] = state[i + 1][j]        neighbour[i + 1][j] = 0
         n = 0         for item in neighbour:
            neighbour[n] = tuple(item)          n = n + 1
        neighbour = tuple(neighbour)
         neighbours.append(neighbour)

    neighbour = list(state)     n = 0     for item in neighbour:
        neighbour[n] = list(item)        n = n + 1
     if j < 2:
        neighbour[i][j] = state[i][j + 1]        neighbour[i][j + 1] = 0        n = 0
      for item in neighbour:
            neighbour[n] = tuple(item)          n = n + 1
        neighbour = tuple(neighbour)        neighbours.append(neighbour)
    neighbour = list(state)     n = 0     for item in neighbour:
        neighbour[n] = list(item)        n = n + 1
     if i > 0:
        neighbour[i][j] = state[i - 1][j]        neighbour[i - 1][j] = 0        n = 0
      for item in neighbour:
            neighbour[n] = tuple(item)          n = n + 1
        neighbour = tuple(neighbour)        neighbours.append(neighbour)

    neighbour = list(state)     n = 0     for item in neighbour:
        neighbour[n] = list(item)        n = n + 1
     if j > 0:
        neighbour[i][j] = state[i][j - 1]        neighbour[i][j - 1] = 0        n = 0
      for item in neighbour:
            neighbour[n] = tuple(item)          n = n + 1
        neighbour = tuple(neighbour)        neighbours.append(neighbour)

    return neighbours
```

Get neighbours

Get neighbours functions is used to get all the states I can move to from my current state by checking the position of the zero in input state, it checks all the possible moves from a state and puts them all in a list.

Goal test

```
def goal_test(state):
    goal = ((0, 1, 2), (3, 4, 5), (6, 7, 8))     result = 1     if state != goal:
result = 0     return result
```

Sets the goal we want to reach

If the state is not the same as the end goal, it will return 0

Solution

```
def solution(state, parent_map):
    sol = []     cost_to_goal = 0
    while parent_map[state]:          sol.append(state)          state = parent_map[state]
        cost_to_goal += 1

    sol.reverse()     sol.append(cost_to_goal)
    return sol
```

This function generates the solution of the puzzle. The loop starts with the leaf nodes going up through all parents and ends at the root . At each iteration it appends the current state to the solution, adds one to the total cost and update the current state to its parent. Finally the array is reversed and the final cost is appended to the array.

Get cost

```
def get_cost(state, parent_map, l):     h1 = 0  #manhattan heuristic cost     h2 = 0
#euclidean heuristic cost     for row in state:         for element in row:
            #calculating the heuristic according to manhattan distance            h1 +=
abs(state.index(row) - int(element / 3)) + abs(row.index(element) - element % 3)     for
row in state:         for element in row:
            #calculating the heuristic according to euclidean distance            h2 +=
math.sqrt((state.index(row) - int(element / 3)) ** 2 + (row.index(element) - element % 3)
** 2)

    g = 0
    #calculating the distance from current node to root node     while parent_map[state]:

        g = g + 1
        state = parent_map[state]
    #value returned if the chosen heuristic is manhattan         if l == '1':
        return g + h1
    #value returned if the chosen heuristic was euclidean     elif l == '2':
        return g + h2
```

This function is used with the A* search to calculate the total cost of path to goal, it depends on whether the heuristic chosen is Manhattan distance(h1) or Euclidean distance (h2), the cost is calculated by adding the distance from current node to root node (g) to either h1 or h2 depending on the chosen heuristic, then this cost is used to sort the states in the priority queue to add the state with the least cost to the start of the frontier list in the process of searching for the goal.

Get inverse count

```
def getInvCount(arr):
    l = []
     for item in arr:          for element in item:
           l.append(element)
    inv_count = 0
     for i in range(0, 9):         if l[i] == 0:             continue          for j in
range(i + 1, 9):            if l[j] == 0:                 continue            if l[i] >
l[j]:                 inv_count += 1

    return inv_count
```

This function calculates the number of inversion in the initial state of the puzzle

Number of inversion=number of unsorted pair in the array isSolvable

```
def isSolvable(puzzle):
    invCount = getInvCount(puzzle)

    return (invCount % 2 == 0)
```

Checks whether the puzzle is solvable or not

If the number of inversions in the initial state is even then the puzzle is not solvable , if it's odd the puzzle is solvable

<u>Choose algorithm</u>

```
def choose_algo(z):
    sol = []     if z == '1':
        start_time = time.time()         sol = BFS(puzzle)
        print("--- %s seconds ---" % (time.time() - start_time))     elif z == '2':

        start_time = time.time()         sol = DFS(puzzle)
        print("--- %s seconds ---" % (time.time() - start_time))     elif z == '3':

        m = input("CHOOSE THE Heuristics \n1-Manhattan Distance\n2-Euclidean Distance\n")

        start_time = time.time()         sol = A_star(puzzle, m)
        print("--- %s seconds ---" % (time.time() - start_time))
```

Will chose the algorithm upon the user’s choice whether it’s BFS, DFS or A* and calls the specified function

Search algorithms

BFS (uses a queue as a frontier list)

```
def BFS(init_state):
    frontier = [init_state]     explored = []     exploredNodes = 0

    parent_map = {init_state: 0} #initializing a map to know the parent of each state

     while frontier:
        current_state = frontier.pop(0)   #get the state from the frontier list
        explored.append(current_state)    #add the state to the explored states
        exploredNodes += 1

        if goal_test(current_state):   #check if the current state is the goal
            print("Explored nodes: ", exploredNodes - 1)            sol =
solution(current_state, parent_map)            return sol
         for neighbour in get_neighbours(current_state):
            isFrontier = 0             isExplored = 0

            for item in frontier:  #if neighbour is found in the frontier list set flag
isFrontier to 1 and break                 if neighbour == item:
isFrontier = 1                     break
             for item in explored:  #if neighbour is explored set flag isExplored to 1
and break                 if neighbour == item:                    isExplored = 1
              break
            # if the neighbour is not found in neither the frontier list nor the explored
states           if isFrontier == 0 and isExplored == 0:
                frontier.append(neighbour)    #add neighbour to the frontier list
                parent_map[
```

The Breadth First Search is an algorithm for traversing or searching tree or graph data structures. It explores all the nodes at the present depth before moving on to the nodes at the next depth level.

The steps of the algorithm are: picking a node and enqueue all its adjacent nodes into a queue, dequeue a node from the queue, marking it as visited and enqueue all its adjacent nodes into a queue and repeating this process until the queue is empty or you meet a goal.

The program can be stuck in an infinite loop if a node is revisited and was not marked as visited before. Hence, prevent exploring nodes that are visited by marking them as visited.

<u>DFS (uses a stack as a frontier list)</u>

```
def DFS(init_state):
    frontier = [init_state]     stack = []     exploredNodes = 0

    parent_map = {init_state: 0}
     while frontier:
        current_state = frontier.pop()  #get the state from the frontier list
stack.append(current_state)     #add the state to the explored states
exploredNodes += 1          if goal_test(current_state):
            print("Explored nodes: ", exploredNodes - 1)            sol =
solution(current_state, parent_map)            return sol
         for neighbour in get_neighbours(current_state):
            isFrontier = 0             isExplored = 0

            for item in frontier:   #if neighbour is found in the frontier list set flag
isFrontier to 1 and break                 if neighbour == item:
isFrontier = 1                     break

            for item in stack:    #if neighbour is explored set flag isExplored to 1 and
break              if neighbour == item:                  isExplored = 1
          break
            # if the neighbour is not found in neither the frontier list nor the explored
states            if isFrontier == 0 and isExplored == 0:
                frontier.append(neighbour)    #add neighbour to the frontier list
                parent_map[neighbour] = current_state
```

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

A* search (uses a priority queue -heap- as a frontier list)

```
def A_star(init_state, n): #takes a puzzle and the heuristic type as input     explored =
[]

    state = State(init_state, 0)    #initializing an object of class state and giving it
cost = 0
    frontier = [state]    #frontier list initialized with the initial state
parent_map = {init_state: 0}     exploredNodes = 0
     while frontier:
        # sorting the frontier list according to the cost with giving a higher priority
to the state with the smallest cost         frontier.sort(key=lambda x: x.cost)

        current_state = frontier.pop(0)      #pop frontier[0] (the state with the
smallest cost)
        explored.append(current_state.state) #add the popped (current) state to the
explored states         exploredNodes += 1
         if goal_test(current_state.state):    #check if this state is the goal state

            print("Explored nodes: ", exploredNodes - 1)            sol =
solution(current_state.state, parent_map)            return sol
         for neighbour in get_neighbours(current_state.state):
            isFrontier = 0
```

A * algorithm is a searching algorithm that searches for the shortest path between the initial and the final state.

A* algorithm has 3 parameters:

g: the cost of moving from the initial cell to the current cell. Basically, it is the sum of all the cells that have been visited since leaving the first cell.

h: also known as the heuristic value, it is the estimated cost of moving from the current cell to the final cell. The actual cost cannot be calculated until the final cell is reached. Hence, h is the estimated cost. We must make sure that there is never an over estimation of the cost.

f: it is the sum of g and h. So, **f = g + h**

The way that the algorithm makes its decisions is by taking the f-value into account. The algorithm selects the smallest f-valued cell and moves to that cell. This process continues until the algorithm reaches its goal cell.

Test cases:

Test case 1: Initial state = 0, 8, 7, 6, 5, 4, 3, 2, 1

BFS

```
Enter first row: 0,8,7
Enter second row: 6,5,4
Enter third row: 3,2,1
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
1
Explored nodes:  181392
--- 3829.7382237911224 seconds ---
(6, 8, 7)
(0, 5, 4)
(3, 2, 1)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cost of Path = Depth =  30

Process finished with exit code 0
```

A* search using Manhattan

```
Enter first row: 8,0,7
Enter second row: 6,5,4
Enter third row: 3,2,1
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
3
CHOOSE THE Heuristics
1-Manhattan Distance
2-Euclidean Distance
1
Explored nodes:  15569
--- 63.870593309402466 seconds ---
(8, 0, 7)
(6, 5, 4)
(3, 2, 1)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cost of Path = Depth =  30

Process finished with exit code 0
```

A* search using Euclidean

```
Enter first row: 8,0,7
Enter second row: 6,5,4
Enter third row: 3,2,1
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
3
CHOOSE THE Heuristics
1-Manhattan Distance
2-Euclidean Distance
2
Explored nodes:  33264
--- 438.6220033168793 seconds ---
(8, 0, 7)
(6, 5, 4)
(3, 2, 1)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cost of Path = Depth =  30

Process finished with exit code 0
```

Test case 2: Initial state = 3, 1, 2, 0, 4, 5, 6, 7, 8

BFS

```
Enter first row: 3,1,2
Enter second row: 0,4,5
Enter third row: 6,7,8
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
1
Explored nodes:  3
--- 0.0009999275207519531 seconds ---
(0, 1, 2)
(3, 4, 5)
(6, 7, 8)

Cost of Path = Depth =  1

Process finished with exit code 0
```

DFS

```
Enter first row: 3,1,2
Enter second row: 0,4,5
Enter third row: 6,7,8
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
2
Explored nodes:  1
--- 0.0 seconds ---
(0, 1, 2)
(3, 4, 5)
(6, 7, 8)

Cost of Path = Depth =  1

Process finished with exit code 0
```

A* search using Manhattan

```
Enter first row: 3,1,2
Enter second row: 0,4,5
Enter third row: 6,7,8
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
3
CHOOSE THE Heuristics
1-Manhattan Distance
2-Euclidean Distance
1
Explored nodes:  1
--- 0.0 seconds ---
(0, 1, 2)
(3, 4, 5)
(6, 7, 8)

Cost of Path = Depth =  1

Process finished with exit code 0
```

A* search using Euclidean

```
Enter first row: 3,1,2
Enter second row: 0,4,5
Enter third row: 6,7,8
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
3
CHOOSE THE Heuristics
1-Manhattan Distance
2-Euclidean Distance
2
Explored nodes:  1
--- 0.0 seconds ---
(0, 1, 2)
(3, 4, 5)
(6, 7, 8)

Cost of Path = Depth =  1

Process finished with exit code 0
```

Test case 3: Initial state = 1, 2, 5, 3, 4, 0, 6, 7, 8

BFS

```
Enter first row: 1,2,5
Enter second row: 3,4,0
Enter third row: 6,7,8
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
1
Explored nodes:  12
--- 0.0 seconds ---
(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)

Cost of Path = Depth =  3

Process finished with exit code 0
```

DFS

```
Enter first row: 1,2,5
Enter second row: 3,4,0
Enter third row: 6,7,8
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
2
Explored nodes:  27
--- 0.0 seconds ---
(1, 2, 5)
(3, 0, 4)
(6, 7, 8)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Cost of Path = Depth =  27

Process finished with exit code 0
```

A* search using Manhattan

```
Enter first row: 1,2,5
Enter second row: 3,4,0
Enter third row: 6,7,8
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
3
CHOOSE THE Heuristics
1-Manhattan Distance
2-Euclidean Distance
1
Explored nodes:  3
--- 0.0 seconds ---
(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)

Cost of Path = Depth =  3

Process finished with exit code 0
```

A* search using Euclidean

```
Enter first row: 1,2,5
Enter second row: 3,4,0
Enter third row: 6,7,8
Solvable
CHOOSE THE SEARCH Algorithms
1-BFS
2-DFS
3-A*
3
CHOOSE THE Heuristics
1-Manhattan Distance
2-Euclidean Distance
2
Explored nodes:  3
--- 0.0010037422180175781 seconds ---
(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)

Cost of Path = Depth =  3

Process finished with exit code 0
```