Artificial Intelligence

Assignment #1 Report

by

Ahmed El-Ramady - 4298

Abdelaziz Hussein - 4126

Objective

Our goal in this project was to implement a simple 8 Puzzle application using informed and uninformed search methods that include Breadth-First Search (BFS), Depth-First Search (DFS), and A* Search.

Overview

The implementation was carried out through Python and the user interface was designed using a library called PyQt.

Its simplicity enabled us to design a versatile and dynamic version of the 8 Puzzle game that can be illustrated through puzzle pictures or the standard numbers.

A shuffle feature was added to the program so that the user can generate a random layout of the numbers on the tiles displayed.

The user can also determine the style of tiles displayed, by choosing either a numbers puzzle or a picture puzzle.

Description

The application will prompt the user to choose one of three search algorithms. Once the algorithm is selected, a thread of the search algorithm will begin execution and the final puzzle (goal state) will be displayed upon completion with each step shown throughout the process.

A simple node implementation is created to identify the data, parent, and cost (only used in A* search) and as every algorithm

runs, the nodes in a tree are created according to the algorithm's constraints.

```
class Node:
    def __init__(self, data, parent, cost):
        self.data = data
        self.parent = parent
        self.cost = cost
```

We also opted to use a regular one-dimensional array (list) to store the puzzle data for more flexibility when developing the search algorithms.

For moving tiles in the puzzle, we created four simple methods to handle the indexes within array and to make sure the right movement is made.

```
    def movePosition(self, array, action):
        if action == "Left":
            return self.move_left(array)
        elif action == "Right":
            return self.move_right(array)
        elif action == "Up":
            return self.move_up(array)
        elif action == "Down":
            return self.move_down(array)
        else:
            return
```

```
    def move_down(self, array):
        arr = deepcopy(array)
        i = self.findIndex(arr, 0)
        down_index = i + 3
        if (down_index > 8):
            return
        else:
            temp = arr[i]
            arr[i] = arr[down_index]
            arr[down_index] = temp
            return arr

    def move_left(self, array):
        arr = deepcopy(array)
        i = self.findIndex(arr, 0)
        left_index = i - 1
        if (left_index == 5 or left_index == 2 or left_index == -1):
            return
        else:
            temp = arr[i]
            arr[i] = arr[left_index]
            arr[left_index] = temp
            return arr
```

```
def move_right(self, array):
    arr = deepcopy(array)
    i = self.findIndex(arr, 0)
    right_index = i + 1
    if (right_index == 3 or right_index == 6 or right_index == 9):
        return
    else:
        temp = arr[i]
        arr[i] = arr[right_index]
        arr[right_index] = temp
        return arr

def move_up(self, array):
    arr = deepcopy(array)
    i = self.findIndex(arr, 0)
    up_index = i - 3
    if (up_index < 0):
        return
    else:
        temp = arr[i]
        arr[i] = arr[up_index]
        arr[up_index] = temp
        return arr
```

For the A* search, we also included a get_cost() function to find the cost and determine the distance needed (Manhattan or Euclidean) for the algorithm to operate.

```
def get_cost(self, input_array, goal_array):
    for i in range(0, 9):

        if i % 3 == 0:
            cCol = 0

        elif (i - 1) % 3 == 0:
            cCol = 1

        else:
            cCol = 2

        if i < 3:
            cRow = 0
        elif i < 6:
            cRow = 1
        else:
            cRow = 2

        if goal_array[input_array[i]] % 3 == 0 and (input_array[i] - 1) > -1:
            Gcol = 0
        elif (goal_array[input_array[i]] - 1) % 3 == 0 and (input_array[i] - 1) > -1:
            Gcol = 1
```

```
            else:
                Gcol = 2

            if goal_array[input_array[i]] < 3 and (input_array[i] - 1) > -1:
                GRow = 0
            elif goal_array[input_array[i]] < 6 and (input_array[i] - 1) > -1:
                GRow = 1
            # elif Goal[Current[i]-1] < 9 and (Current[i]-1) > -1:
            else:
                GRow = 2

            # print(cRow, GRow, cCol, Gcol)
            hEuclidean[i] = math.sqrt((cRow - GRow) ** 2 + (cCol - Gcol) ** 2)
            hManhattan[i] = abs(cRow - GRow) + abs(cCol - Gcol)
            if input_array[i] == 0:
                hEuclidean[i] = 0
                hManhattan[i] = 0
        # print("Manhattan distance", hManhattan)
        # print("Euclidean distance", hEuclidean)
        # print(sum(hManhattan))
        # print(sum(hEuclidean))
        return sum(hManhattan)
```

The BFS Search

```
    def buildBFSTree(self):
        global start_time
        if (node.data == goal_array):
            self.setStatusText("Goal already reached")
            return
        iteration = 0
        explored = []
        explored.append(input_array)
        self.current_node = [node]
        currentIteration = 0
        tempArray = deepcopy(input_array)
        while self.current_node:
            current_root = deepcopy(self.current_node.pop(0))

            for move in range(4):
                tempArray = self.movePosition(current_root.data, actions[move])
                if tempArray is not None:
                    self.child_Node = Node(tempArray, current_root, 0)
                    if self.child_Node.data not in explored:
                        if tempArray is not None:
                            self.setStatusText('Parent Node')
                            self.assignTiles((current_root.data))
                            time.sleep(delay)
                            time.sleep(delay)
```

```
                    time.sleep(delay)
                    self.setStatusText('Child Node')
                    #self.displayPuzzle(current_root.data)
                    #print("Child Nodes")
                    #print("--------")
                self.current_node.append(self.child_Node)
                explored.append(deepcopy(self.child_Node.data))
                self.statuslineedit.setText("Moving " + actions[move])
                #self.displayPuzzle(child_Node.data)
                self.assignTiles((self.child_Node.data))
                self.displayPuzzle((self.child_Node.data))
                time.sleep(delay)
                time.sleep(delay)

                currentIteration += 1
                if self.child_Node.data == goal_array:
                    #self.assignTiles(deepcopy(self.child_Node.data))
                    self.setStatusText('Successfully completed!')
                    self.shufflelineedit.setEnabled(True)
                    self.startlineedit.setEnabled(True)
                    self.tilesradiolineedit.setEnabled(True)
                    self.numbersradiolineedit.setEnabled(True)
                    print("Cost to goal is: {}".format(len(explored)))
```

The DFS Search

```
def buildDFSTree(self):
    global start_time
    nodeCost = []
    rootCost = []

    if (node.data == goal_array):
        self.setStatusText("Goal already reached")
        return
    iteration = 0
    explored = []
    explored.append(input_array)
    self.current_node = [node]
    currentIteration = 0
    while self.current_node:
        current_root = [self.current_node.pop(0)]
        rootCost.append(current_root)
        #print(current_root)
        while current_root:
            current_ext_node = current_root.pop(0)
            #print(current_ext_node.data)
            for move in range(4):
                if deepcopy(current_ext_node.data) is not None:
                    # self.setStatusText('Parent Node')
```

```
                # self.assignTiles(current_ext_node.data)
                time.sleep(delay)
                self.setStatusText('Child Node')
            tempArray = deepcopy(current_ext_node.data)
            tempArray = self.movePosition(deepcopy(tempArray), actions[move])
            if tempArray is not None:
                if tempArray not in explored:
                    #current_ext_node.data = deepcopy(tempArray)
                    # tempArray = deepcopy(current_ext_node.data)
                    self.child_Node = Node(tempArray, current_ext_node, 0)
                    nodeCost.append(current_ext_node)
                    # self.displayPuzzle(current_root.data)
                        # print("Child Nodes")
                        # print("--------")
                    #current_ext_node.append(self.child_Node)
                    current_root.append(self.child_Node)
                    explored.append(deepcopy(self.child_Node.data))
                    self.statuslineedit.setText("Moving " + actions[move])
                    self.displayPuzzle(self.child_Node.data)
                    self.assignTiles(deepcopy(self.child_Node.data))
                    time.sleep(delay)
                    if self.child_Node.data == goal_array:
                        # self.assignTiles(deepcopy(self.child_Node.data))
                        self.setStatusText('Successfully completed!')
```

```
                        self.setStatusText('Successfully completed!')
                        self.startlineedit.setEnabled(True)
                        self.shufflelineedit.setEnabled(True)
                        self.tilesradiolineedit.setEnabled(True)
                        self.numbersradiolineedit.setEnabled(True)
                        self.current_node = []
                        self.child_Node = []
                        current_root = []
                        # print("GOAL REACHED")
                        print("Cost to goal is: {}".format(len(rootCost) + len(nodeCost)))
                        end_time = time.time()
                        print("Running time = {}".format(end_time - start_time))
                        rootCost = []
                        nodeCost = []
                        return
```

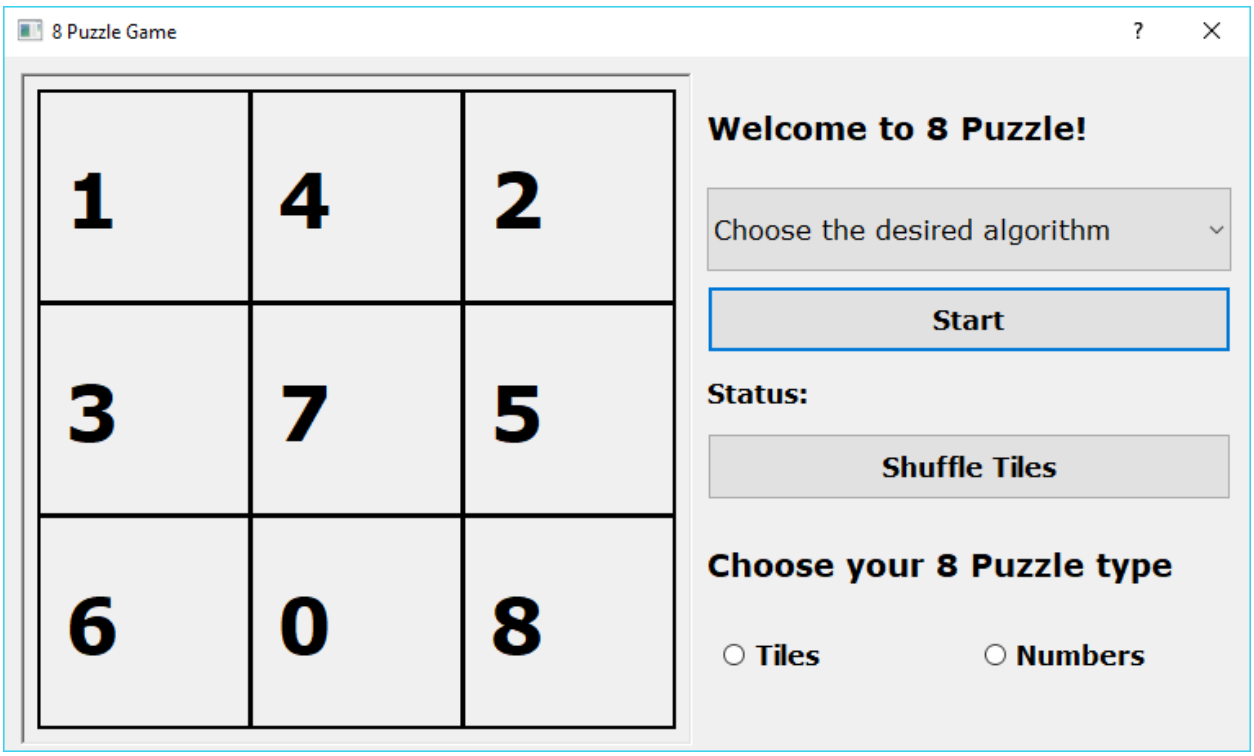
A* Search

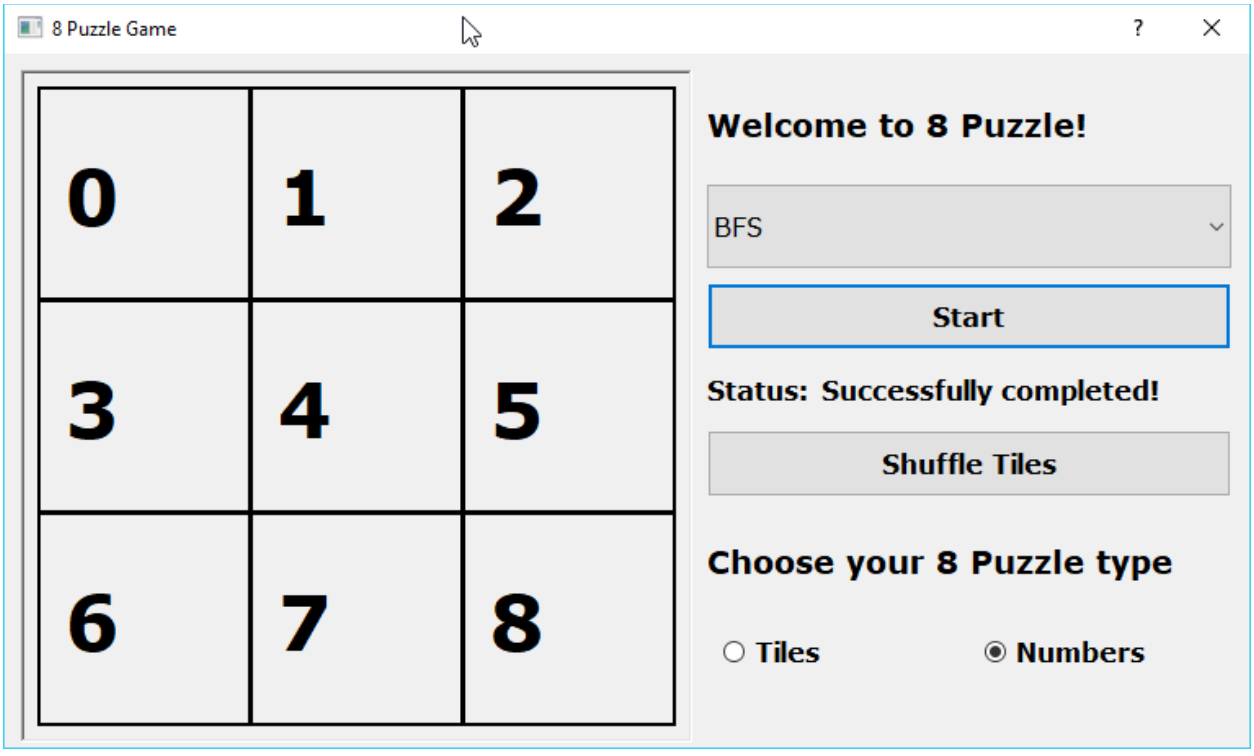
```
def buildAStarTree(self):
    global start_time
    if(node.data == goal_array):
        self.setStatusText("Goal already reached")
        return
    explored = []
    node.cost = 0
    path_to_goal = []
    cost = 0
    cost_arr = []
    level_nodes = []
    explored.append(input_array)
    current_node = [node]
    currentIteration = 0
    tempArray = deepcopy(input_array)
    while current_node:
        path_to_goal.append(current_node)

        current_root = (deepcopy(current_node.pop(0)))

        for move in range(4):
            tempArray = self.movePosition(current_root.data, actions[move])
            if tempArray is not None:
                child_Node = Node(tempArray, current_root, cost)
```

```
            child_Node = Node(tempArray, current_root, cost)
            if child_Node.data not in explored:
                if tempArray is not None:
                    print("Parent Node")
                    self.setStatusText('Parent Node')
                    self.assignTiles(current_root.data)
                    self.displayPuzzle(current_root.data)
                    time.sleep(delay)
                    print("Child Nodes")
                    self.setStatusText('Child Node')
                    print("--------")

                explored.append(deepcopy(child_Node.data))
                print("Moving " + actions[move])
                cost = self.get_cost(child_Node.data, goal_array)
                child_Node.cost = cost
                print("Cost of node")
                print(child_Node.cost)
                cost_arr.append(child_Node.cost)
                print("Cost array ", cost_arr)
                min_child_Node = child_Node.cost
                print(child_Node.cost)
                print("the minimum node of all nodes     ", child_Node.data)
```

```
                self.assignTiles(child_Node.data)
                self.displayPuzzle(child_Node.data)
                time.sleep(delay)
                currentIteration += 1
                if child_Node.data == goal_array:
                    print("GOAL REACHED")
                    self.setStatusText('Successfully completed!')
                    self.startlineedit.setEnabled(True)
                    self.shufflelineedit.setEnabled(True)
                    self.tilesradiolineedit.setEnabled(True)
                    self.numbersradiolineedit.setEnabled(True)
                    print("Cost to path is {}".format(len(path_to_goal)))
                    self.current_node = []
                    self.child_Node = []
                    explored = []
                    end_time = time.time()
                    print("Running time = {}".format(end_time - start_time))
                    return
                level_nodes.append(child_Node)
        if len(cost_arr) > 0:
            minCost_ele = min(cost_arr)
            print("the minimium node is     ", minCost_ele)
            tempNode = self.find_node(level_nodes, minCost_ele)
            current_node.append(tempNode)
```

Sample Runs


8 Puzzle Game
1 4 2
3 7 5
6 0 8
Welcome to 8 Puzzle!
Choose the desired algorithm
Start
Status:
Shuffle Tiles
Choose your 8 Puzzle type
Tiles
Numbers


8 Puzzle Game
0 1 2
3 4 5
6 7 8
Welcome to 8 Puzzle!
BFS
Start
Status: Successfully completed!
Shuffle Tiles
Choose your 8 Puzzle type
Tiles
Numbers

Cost to goal is: 19

Running time = 0.8595011234283447

Detailed nodes expansion:

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 7, 5]

[6, 8, 0]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 7, 5]

[0, 6, 8]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 8, 0]

Child Node

[1, 4, 2]

[3, 7, 0]

[6, 8, 5]

Parent Node

[1, 4, 2]

[3, 7, 5]

[0, 6, 8]

Child Node

[1, 4, 2]

[0, 7, 5]

[3, 6, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 7, 0]

[6, 8, 5]

Child Node

[1, 4, 2]

[3, 0, 7]

[6, 8, 5]

Parent Node

[1, 4, 2]

[3, 7, 0]

[6, 8, 5]

Child Node

[1, 4, 0]

[3, 7, 2]

[6, 8, 5]

Parent Node

[1, 4, 2]

[0, 7, 5]

[3, 6, 8]

Child Node

[1, 4, 2]

[7, 0, 5]

[3, 6, 8]

Parent Node

[1, 4, 2]

[0, 7, 5]

[3, 6, 8]

Child Node

[0, 4, 2]

[1, 7, 5]

[3, 6, 8]

Parent Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Child Node

[1, 4, 0]

[3, 5, 2]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Child Node

[1, 4, 2]

[3, 5, 8]

[6, 7, 0]

Parent Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Child Node

[0, 4, 2]

[1, 3, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[6, 3, 5]

[0, 7, 8]

Parent Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

Child Node

[1, 2, 0]

[3, 4, 5]

[6, 7, 8]

Parent Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

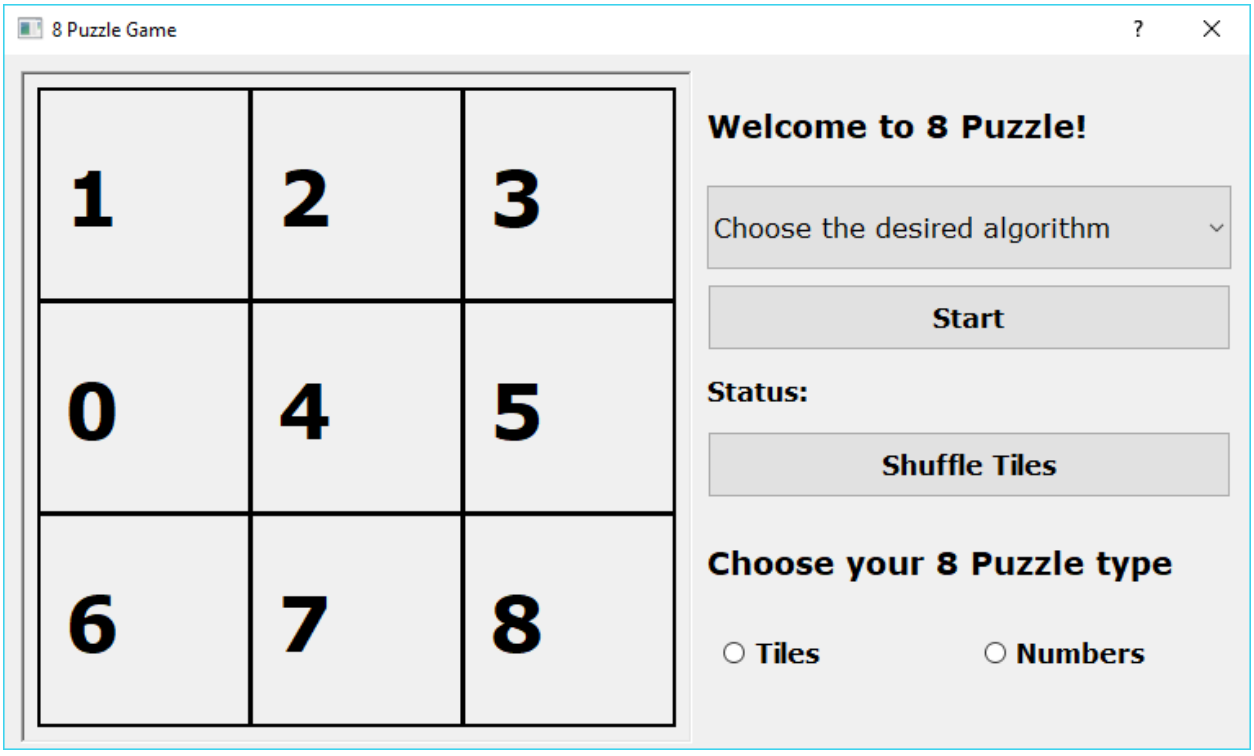
Child Node

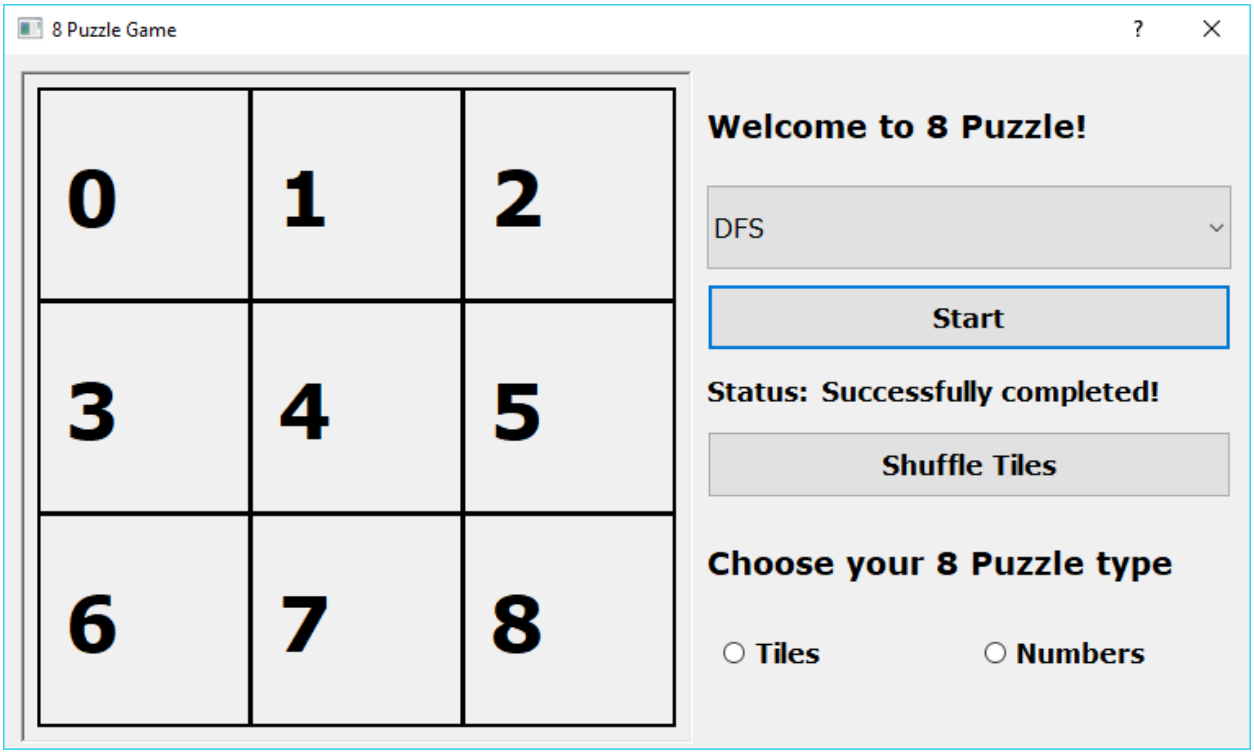
[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

GOAL REACHED!





Cost to goal is: 20

Running time = 0.8554904460906982

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 7, 5]

[6, 8, 0]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 7, 5]

[0, 6, 8]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 8, 0]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 8, 0]

Child Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 8, 0]

Child Node

[1, 4, 2]

[3, 7, 0]

[6, 8, 5]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 8, 0]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[0, 6, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[0, 6, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[0, 6, 8]

Child Node

[1, 4, 2]

[0, 7, 5]

[3, 6, 8]

Parent Node

[1, 4, 2]

[3, 7, 5]

[0, 6, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 0]

[6, 8, 5]

Child Node

Parent Node

[1, 4, 2]

[3, 7, 0]

[6, 8, 5]

Child Node

[1, 4, 2]

[3, 0, 7]

[6, 8, 5]

Parent Node

[1, 4, 2]

[3, 7, 0]

[6, 8, 5]

Child Node

[1, 4, 0]

[3, 7, 2]

[6, 8, 5]

Parent Node

[1, 4, 2]

[3, 7, 0]

[6, 8, 5]

Child Node

Parent Node

[1, 4, 2]

[0, 7, 5]

[3, 6, 8]

Child Node

[1, 4, 2]

[7, 0, 5]

[3, 6, 8]

Parent Node

[1, 4, 2]

[0, 7, 5]

[3, 6, 8]

Child Node

Parent Node

[1, 4, 2]

[0, 7, 5]

[3, 6, 8]

Child Node

[0, 4, 2]

[1, 7, 5]

[3, 6, 8]

Parent Node

[1, 4, 2]

[0, 7, 5]

[3, 6, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Child Node

Parent Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Child Node

[1, 4, 0]

[3, 5, 2]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Child Node

[1, 4, 2]

[3, 5, 8]

[6, 7, 0]

Parent Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Child Node

Parent Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Child Node

Parent Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Child Node

[0, 4, 2]

[1, 3, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[6, 3, 5]

[0, 7, 8]

Parent Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

Child Node

[1, 2, 0]

[3, 4, 5]

[6, 7, 8]

Parent Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

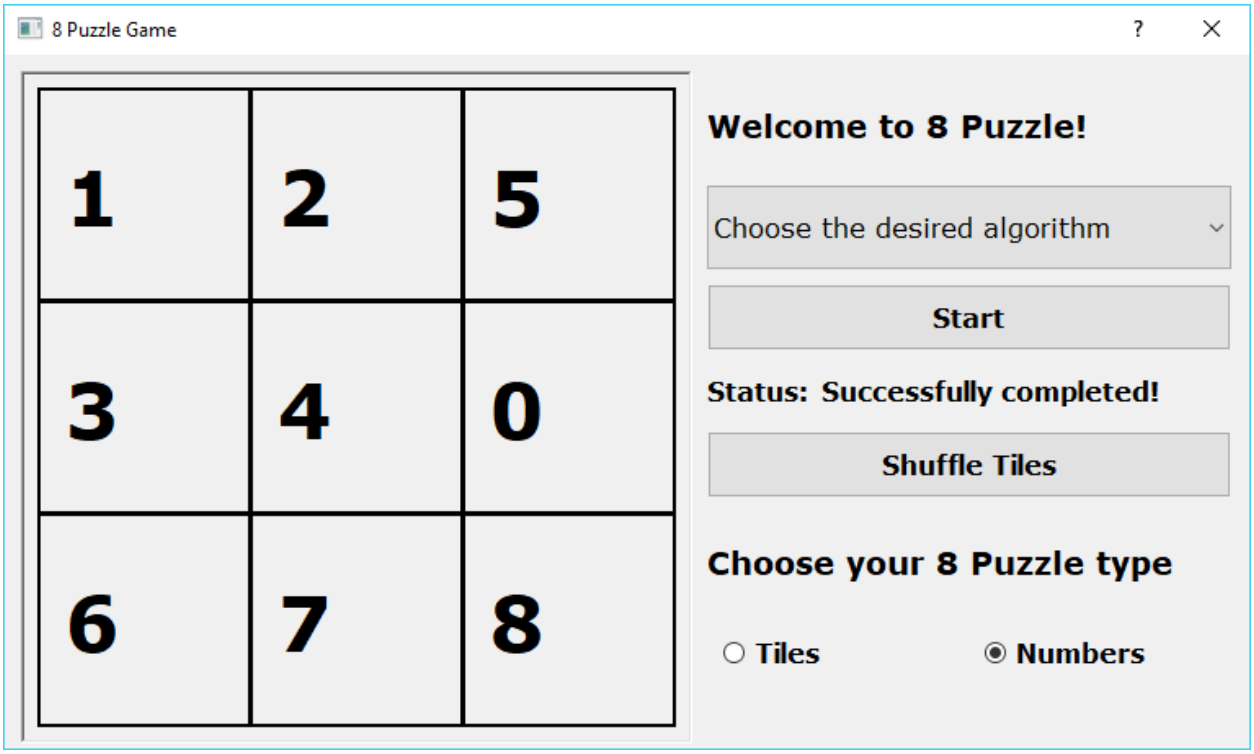
Child Node

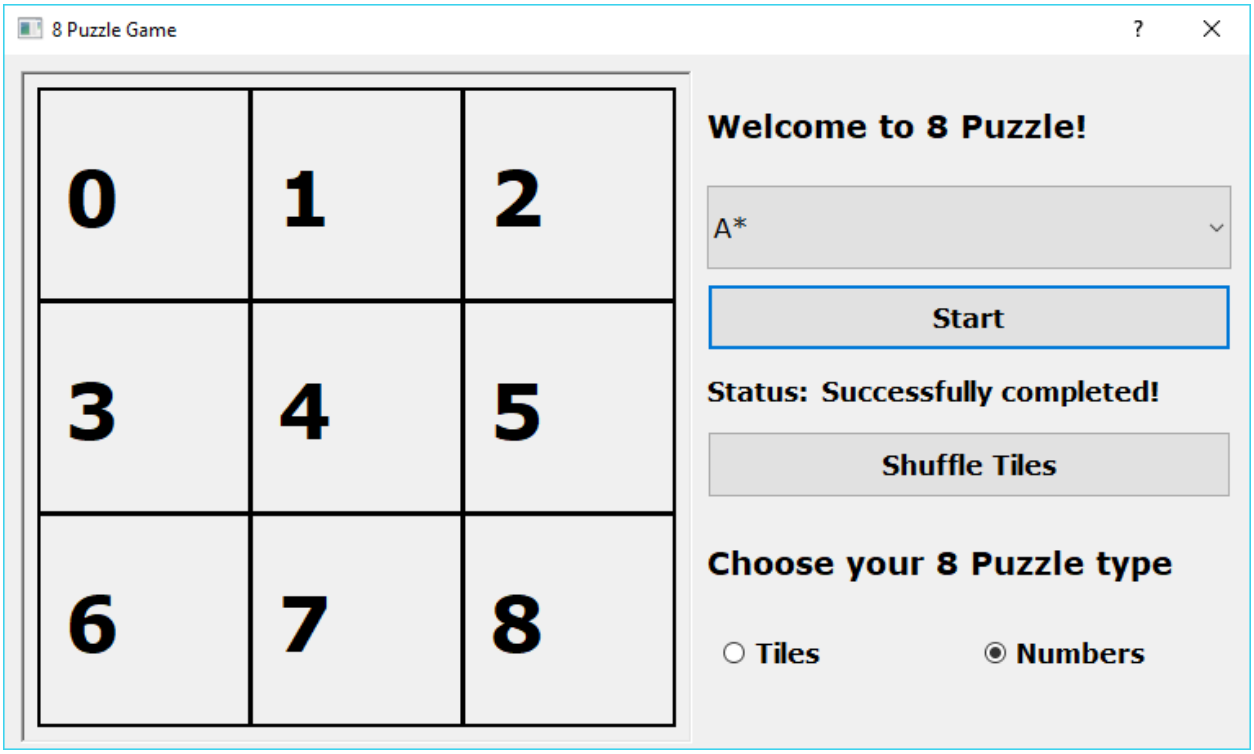
[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

GOAL REACHED!


8 Puzzle Game
1 2 5
3 4 0
6 7 8
Welcome to 8 Puzzle!
Choose the desired algorithm
Start
Status: Successfully completed!
Shuffle Tiles
Choose your 8 Puzzle type
Tiles
Numbers


8 Puzzle Game
0 1 2
3 4 5
6 7 8
Welcome to 8 Puzzle!
A*
Start
Status: Successfully completed!
Shuffle Tiles
Choose your 8 Puzzle type
Tiles
Numbers

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 7, 5]

[6, 8, 0]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 7, 5]

[0, 6, 8]

Parent Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Child Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[3, 5, 0]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[0, 3, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

Parent Node

[1, 4, 2]

[3, 0, 5]

[6, 7, 8]

Child Node

[1, 4, 2]

[3, 7, 5]

[6, 0, 8]

Parent Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

Child Node

[1, 2, 0]

[3, 4, 5]

[6, 7, 8]

Parent Node

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

Child Node

[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

GOAL REACHED

Cost to path is 3

Running time = 0.8565013408660889

Extra Feature