CC482: AI

Assignment 1

8-puzzle Solver

October 27, 2018

Team

Fares Mehanna 52 **Dahlia Chehata 27**

Table Of Contents

Overview

- 8 puzzle solver is a python application that uses different searching techniques to achieve the predetermined perfect state .
- The techniques are BFS, DFS, UCS, A* with 3 different heuristic types: Manhattan distance, Euclidean Distance and Misplaced Tiles
- The application computes the time, cost, depth, number of visited nodes for each search method and plot graphs comparing these methods
- The visited path is printed step by step till reaching the goal state
- If the puzzle is unsolvable, a message is shown
- Slow motion feature allows to keep track of the puzzle different states to the solution
- The user can enter the initial State of the puzzle or choose a random state to begin with

Data Structures and Algorithms used

➢ DFS

- DS: stack,set

```
def search(state, goal_state, yield_after):
   cur_node = Node(state)
   explored = set()
   stack = list([cur_node])
      while stack:
       cur_node = stack.pop()
       explored.add(cur_node.map)
       if cur_node.is_goal(goal_state):
           break
       cur_node.expand()
       for child in reversed(cur_node.children):
           if child.map not in explored:
               stack.append(child)
               explored.add(child.map)
```

➢ BFS

- DS: queue, set

```
def search(state, goal_state, yield_after):
   cur_node = Node(state)
```

```
    explored = set()
    queue = deque([cur_node])
      while len(queue) != 0:
        cur_node = queue.popleft()
        explored.add(cur_node.map)
        if cur_node.is_goal(goal_state):
            break
        cur_node.expand()
        for child in cur_node.children:
            if child.map not in explored:
                queue.append(child)
                explored.add(child.map)
```

➢ Greedy Best First search

- it is used as the main step in the A* star Algorithm
- DS: priority queue, set

```
def search(state, goal_state, heuristic, yield_after):
   cur_node = Node(state)
   frontier = [(heuristic(cur_node), 0, cur_node)]

   explored_set = set()
   forntier_set = set()
   forntier_set.add(cur_node.map)
   while frontier:

       # get the highest priority
       cur_node = heapq.heappop(frontier)[2]
       depth = max(depth, cur_node.depth)

       # if already visited, then continue
       if cur_node.map in explored_set:
           continue

       # add the state to explored_set and remove it from forntier_set
       explored_set.add(cur_node.map)
       forntier_set.remove(cur_node.map)

       # if goal, then we are done.
       if cur_node.is_goal(goal_state):
           break

       # else add all the childs
       cur_node.expand()
       for child in cur_node.children:
           # don't add if already visited
           if child.map in explored_set:
               continue
```

```
        # add if not visited or has lower cost
        if (child.map in forntier_set and cost_so_far[child.map] > child.cost)
                or child.map not in forntier_set:
            cost_so_far[child.map] = child.cost
            forntier_set.add(child.map)
            item = (heuristic(child), index, child)
            heapq.heappush(frontier, item)
            index += 1
```

- **A* star**
  - **Manhattan Distance**
  - **Euclidean Distance**
  - **Misplaced Tiles**

```
def search(state, goal_state, heuristic_type, yield_after):
   def g(node):
       return node.compute_cost()

   tiles_indices = []
   for i in range(len(goal_state)):
       for j in range(len(goal_state)):
           heapq.heappush(tiles_indices, (goal_state[i][j], (i, j)))

   def h(node):
       cost = 0
       for i in range(len(node.state)):
           for j in range(len(node.state)):
               tile_i, tile_j = tiles_indices[node.state[i][j]][1]
               if i != tile_i or j != tile_j and node.state[i][j]!=0:
                   if heuristic_type == "Manhattan Distance":
                       cost += abs(tile_i - i) + abs(tile_j - j)
                   elif heuristic_type == "Euclidean Distance":
                       cost += math.sqrt((tile_i - i) * (tile_i - i) + (tile_j - j) *(tile_j - j))
                   elif heuristic_type == "Misplaced Tiles":
                       cost += 1
                   else:
                       raise Exception("Not supported heuristic")
       return cost

   def f(node):
       return g(node) + h(node)
   return Greedy_best_first.search(state, goal_state, f, yield_after)
```

➢ DLS

- DS: stack, set

```
def search(initial_state, goal_state, limit, yield_after, counter):
   cur_node = Node(initial_state)
   explored = set()
   stack = list([cur_node])
   while stack:
       cur_node = stack.pop()
       if cur_node.depth >= limit:
           continue
       explored.add(cur_node.map)
       if cur_node.is_goal(goal_state):
           break
       cur_node.expand()
       for child in reversed(cur_node.children):
           if child.map not in explored:
               stack.append(child)
               explored.add(child.map)
   if not cur_node.is_goal(goal_state):
       return None
```

➢ IDS

- DS: those of DLS method

```
def search(initial_state, goal_state, yield_after):
      while not sol:
        sol = DLS.search(initial_state, goal_state, depth, yield_after, realTotalVisitedNodes)
        depth += 1
      return sol
```

➢ UCS

- DS: those of greedy best first search

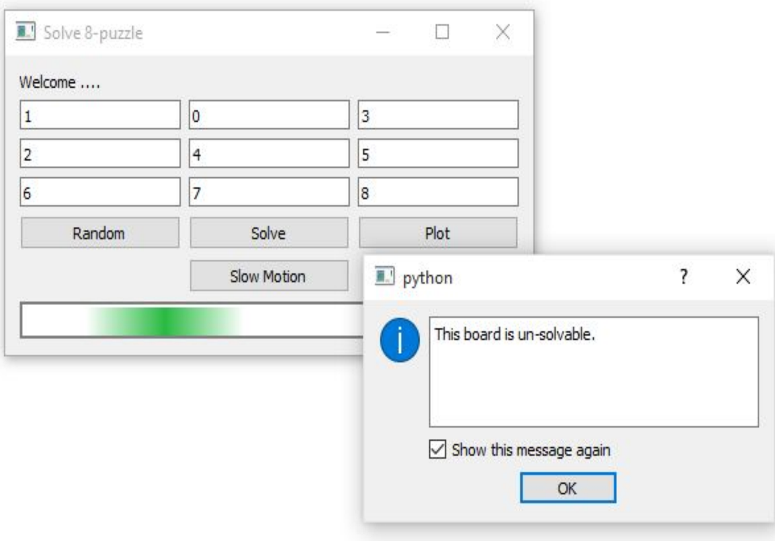
```
def search(initial_state, goal_state, yield_after):
   def g(node):
       return node.compute_cost()
   return Greedy_best_first.search(initial_state, goal_state, g, yield_after)
```

Details

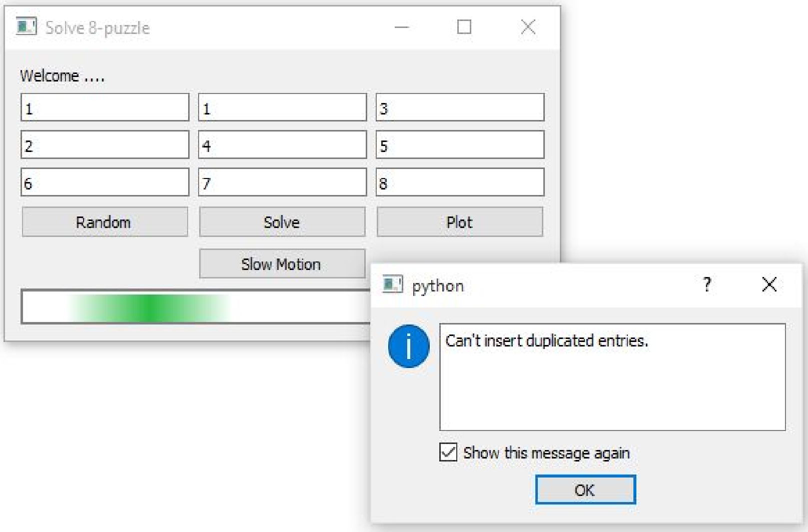
➢ Solvability detection

check the number of inversions of each cell and its subsequent in the state list. If the number of inversions is odd, the puzzle is unsolvable

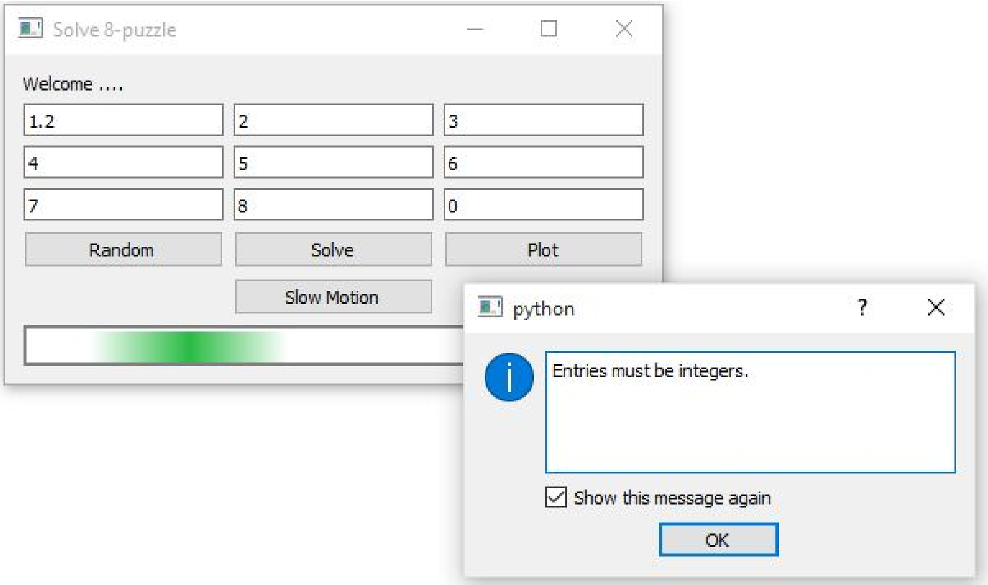
example case : 1, 0, 3, 2, 4, 5, 6, 7, 8, 9



➢ redundant tiles



➢ fraction Handling



➢ path to goal

The path is printed in a traceable format till the goal state is found as shown in figure

➢ cost of path

It is g(n) and is equal to depth in all cases since the move cost is 1 in the puzzle game

cost of a child state is the cost of parent + 1 in the time of expansion

➢ nodes expanded

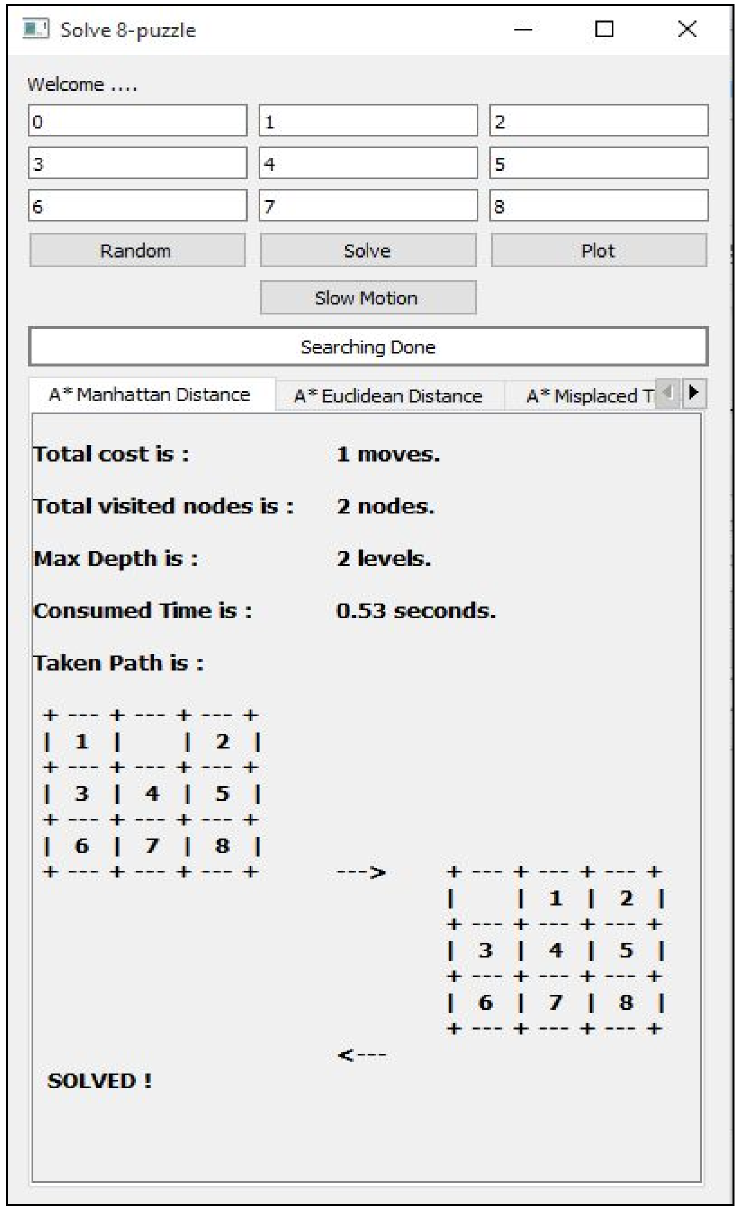
the number of the nodes expanded is printed

➢ search depth

The maximum search depth depth is returned and not the depth of the goal state since it may be larger

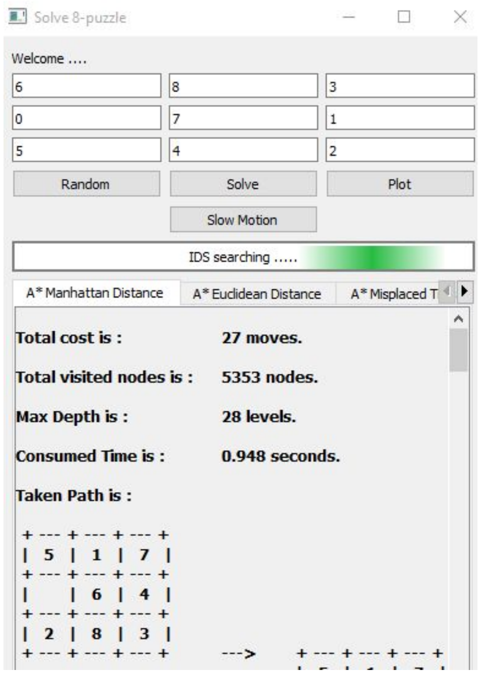
➢ running time

it is printed for each search method as shown in figure

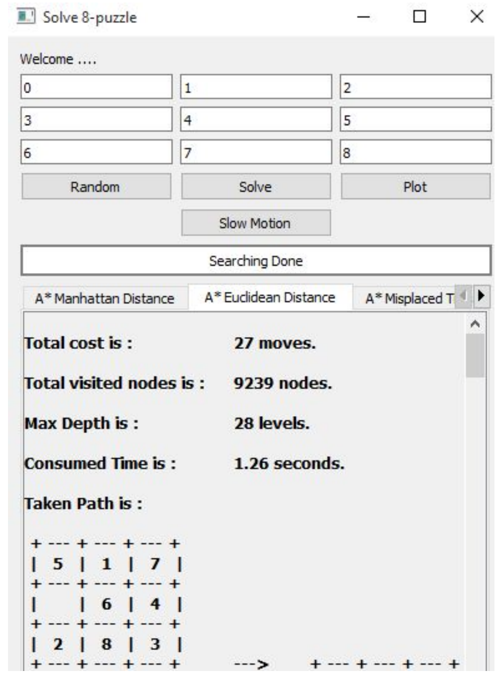

Solve 8-puzzle
Welcome ....
0
1
2
3
4
5
6
7
8
Random
Solve
Plot
Slow Motion
Searching Done
A* Manhattan Distance
A* Euclidean Distance
A* Misplaced T
Total cost is : 1 moves.
Total visited nodes is : 2 nodes.
Max Depth is : 2 levels.
Consumed Time is : 0.53 seconds.
Taken Path is :
1		2
3	4	5
6	7	8
--->
	1	2
3	4	5
6	7	8
<---
SOLVED !

Sample runs

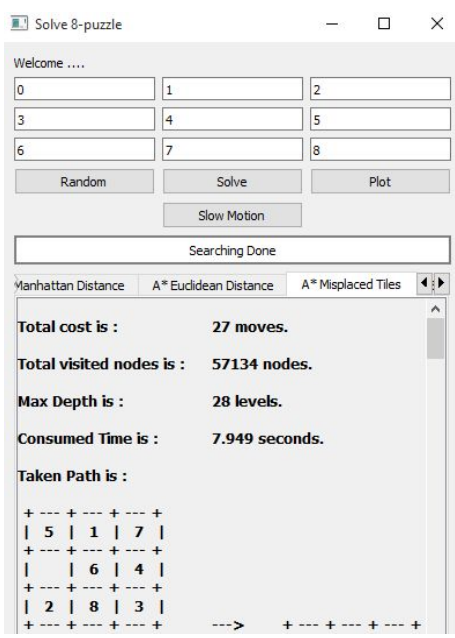
A* Manhattan Distance



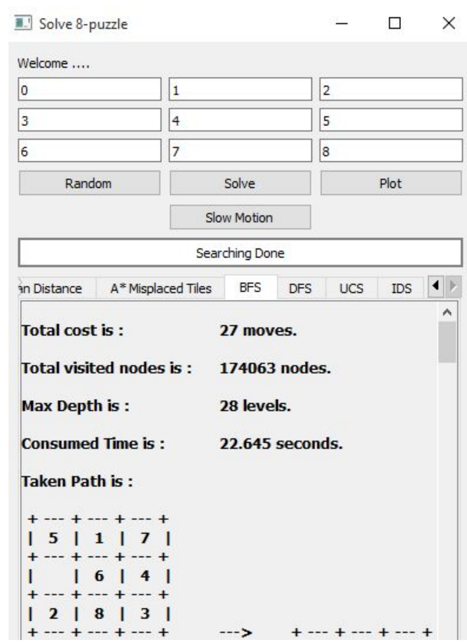
A* Euclidean Distance

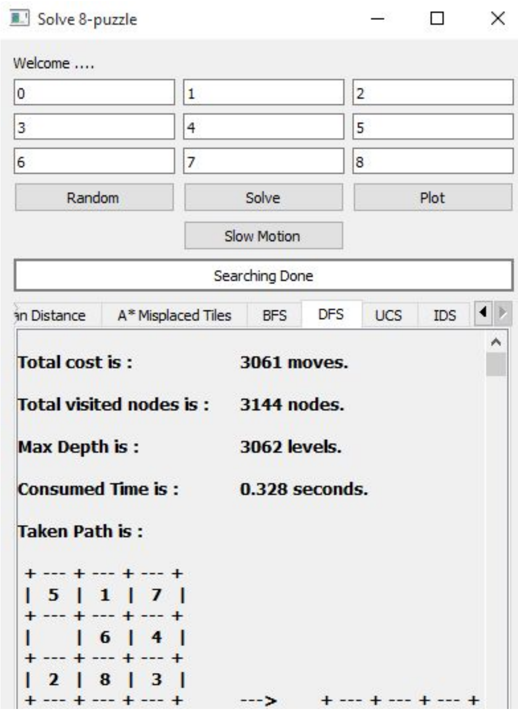
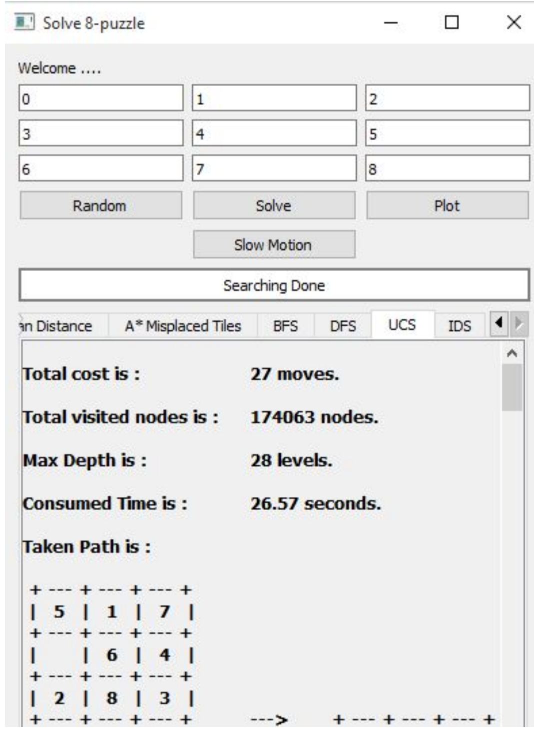
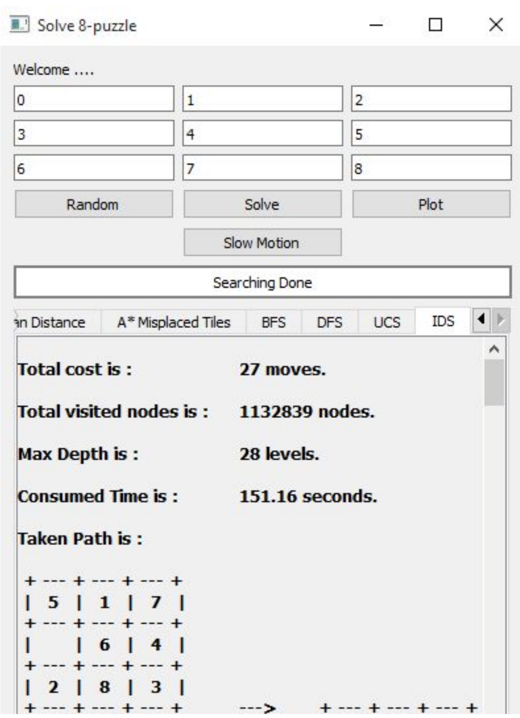


A* Misplaced Tiles



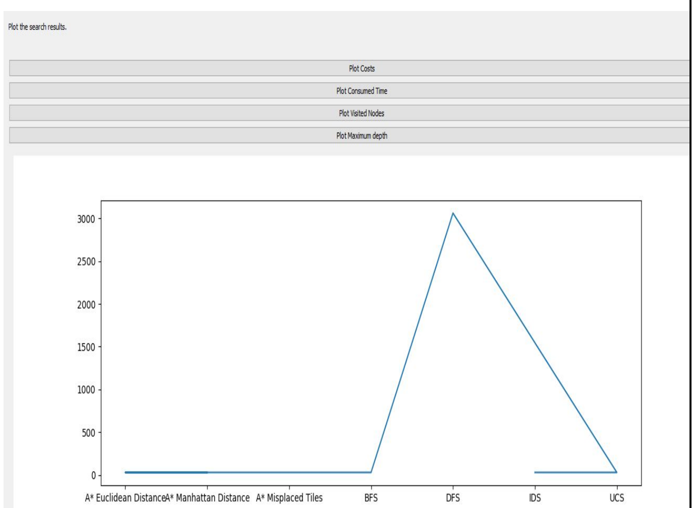
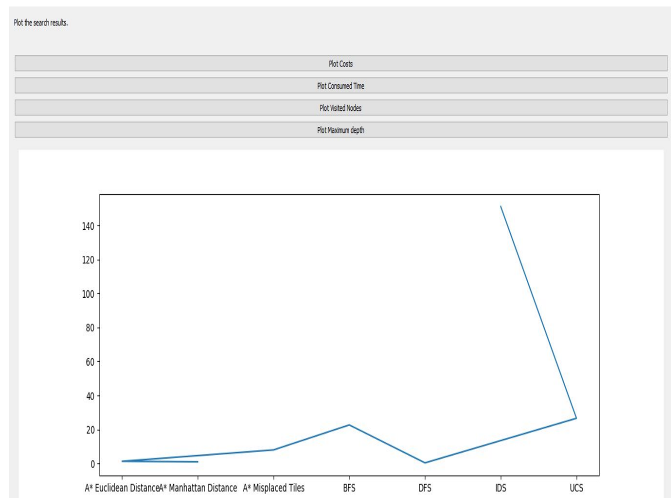
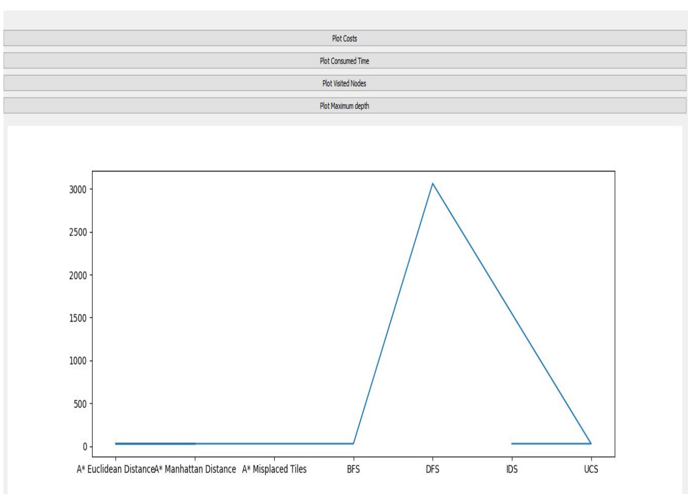
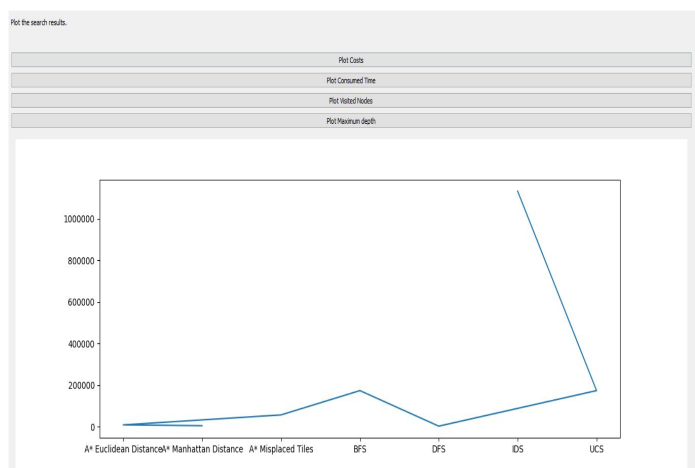
BFS



<table>
<tr><th>DFS</th><th>UCS</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><th>IDS</th><th></th></tr>
<tr><td>
</td><td></td></tr>
</table>

Extra Work

- ➢ **GUI**
- ➢ **Plots**

<table>
<tr><th>cost plot</th><th>running time plot</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><th>max depths plot</th><th>number of expanded nodes plot</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

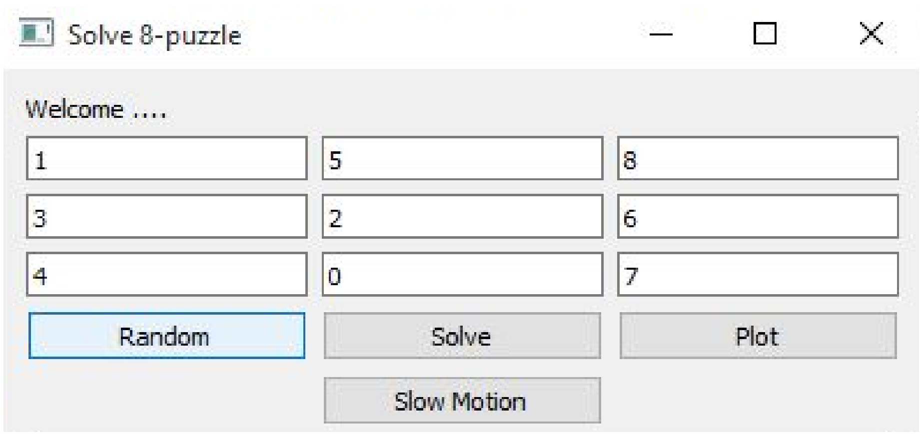
➢ Additional Search techniques

- Greedy best First search
- UCS
- DLS
- IDS,
- A* with misplaced tiles

are implemented as they are described in Section 2

➢ Randomization and slow motion features

- **Random button** initialises the initial board state with random values or the user can enter the required initial state himself



- **slow motion button** allows to keep track of tiles arrangement t the perfect goal if found. Generally , we yield path to the gui after certain amount of times to allow interactivity