AI Assignment 1

Using Informed and Uninformed Search Algorithms to Solve 8-Puzzle

Names

Name	ID
Arsanuos Essa	17
Amr Mohamed Fathy	46
Muhammed Essam Khamis	66

CONTENTS

1) A* algorithm

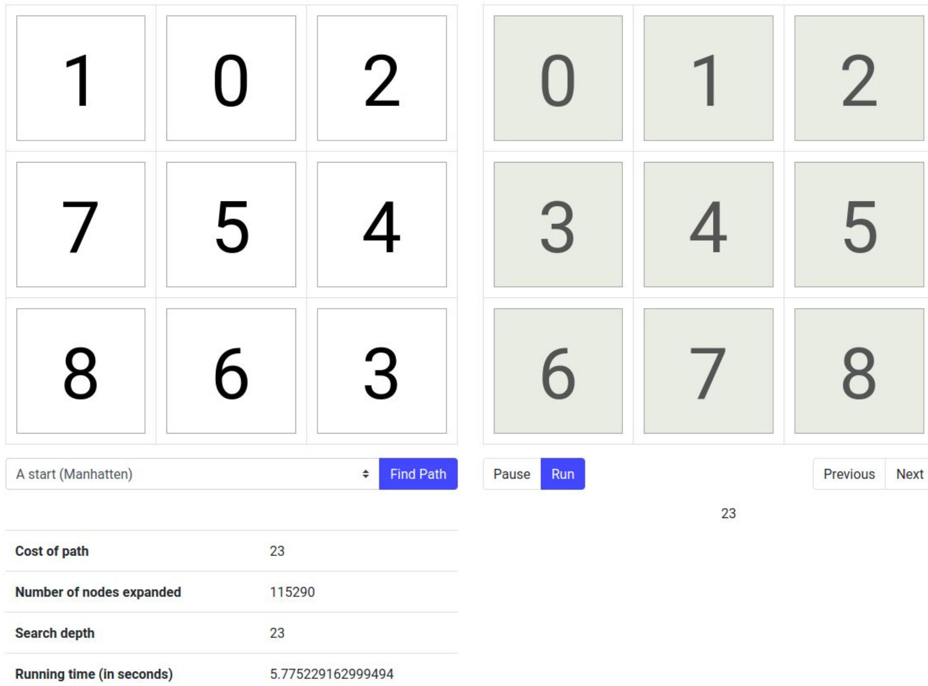
- ## Code

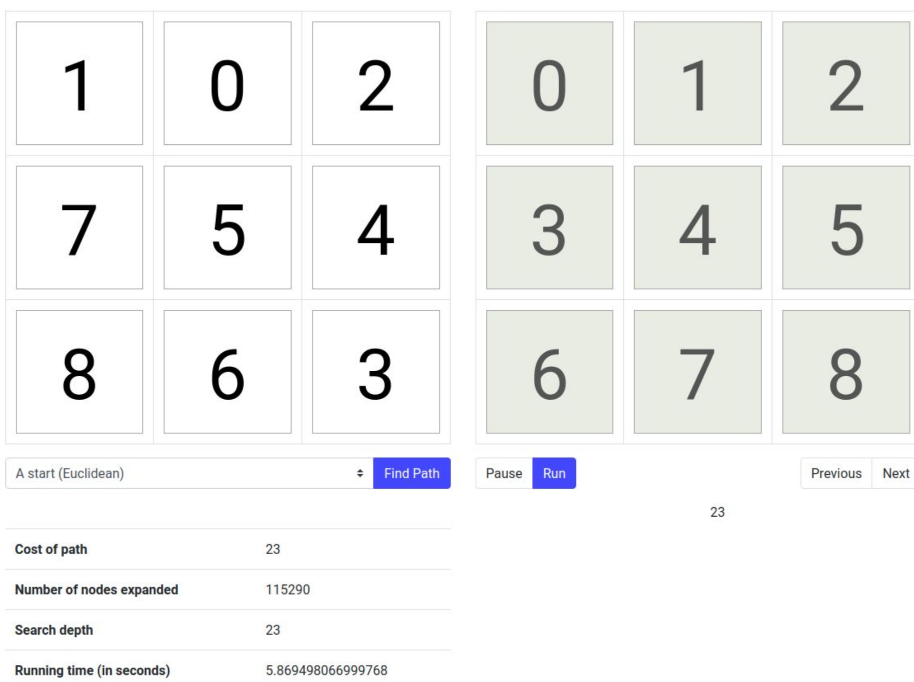
```python
def __init__(self, heuristic):
    self.__heuristic = heuristic
    super().__init__()

def search(self, initial_arr):
    """..."""
    self.__initial_state = State(initial_arr, None, self.__heuristic)
    if not self.check_solvable(self.__initial_state):
        return {'steps': [[-1] * 9], 'cost': -1,
                'search_depth': -1, 'nodes_expanded': -1}

    states_heap = []
    heappush(states_heap, self.__initial_state)
    end = None
    while len(states_heap):
        # break tie by FIFO criteria
        current_explored_state = heappop(states_heap)
        if current_explored_state.is_goal():
            end = current_explored_state
            break
        if current_explored_state not in self.vis:
            self._explored += 1
            self.vis.add(current_explored_state)
            child_states = self.expand(current_explored_state)
            for child in child_states:
                heappush(states_heap, child)

    res = {}
    res['steps'] = self.get_steps(end)
    res['cost'] = end.cost
    res['search_depth'] = end.cost
    res['nodes_expanded'] = self._explored
    return res
```

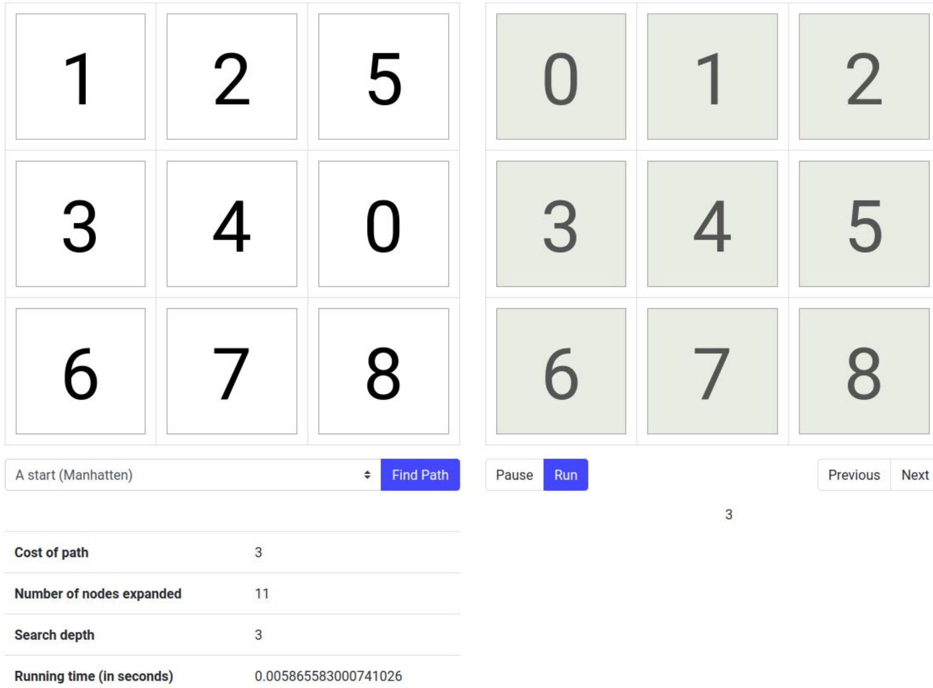
- ## Data Structures
    - Heap that represents a priority queue that returns the smallest element in the queue in terms of total cost.
    - Set to keep the Visited(explored set) states to avoid repeating states using a hashing search in nearly O(1) for a faster search.

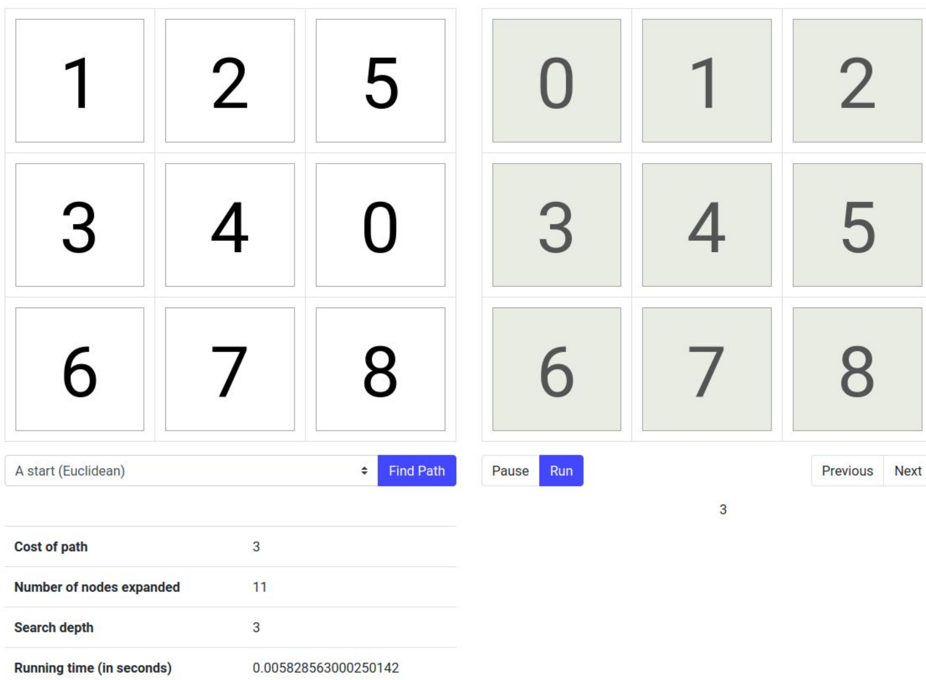
- ## Explanation
    - State object that contains a total cost representing f(n)
    - We found that searching in frontier list (heap) is costly since it will be linear search O(n) So we assumed that we will insert all non visited children and also check if the not not visited the do nothing.
    - At the end returning all required numbers to be displayed in UI.

- **Sample runs**


1 0 2
7 5 4
8 6 3
0 1 2
3 4 5
6 7 8
A start (Manhatten)
Find Path
Pause
Run
Previous
Next
23
Cost of path 23
Number of nodes expanded 115290
Search depth 23
Running time (in seconds) 5.775229162999494


1 0 2
7 5 4
8 6 3
0 1 2
3 4 5
6 7 8
A start (Euclidean)
Find Path
Pause
Run
Previous
Next
23
Cost of path 23
Number of nodes expanded 115290
Search depth 23
Running time (in seconds) 5.869498066999768


1 2 5
3 4 0
6 7 8
0 1 2
3 4 5
6 7 8
A start (Manhatten)
Find Path
Pause
Run
Previous
Next
3
Cost of path 3
Number of nodes expanded 11
Search depth 3
Running time (in seconds) 0.005865583000741026


1 2 5
3 4 0
6 7 8
0 1 2
3 4 5
6 7 8
A start (Euclidean)
Find Path
Pause
Run
Previous
Next
3
Cost of path 3
Number of nodes expanded 11
Search depth 3
Running time (in seconds) 0.005828563000250142

2) DFS

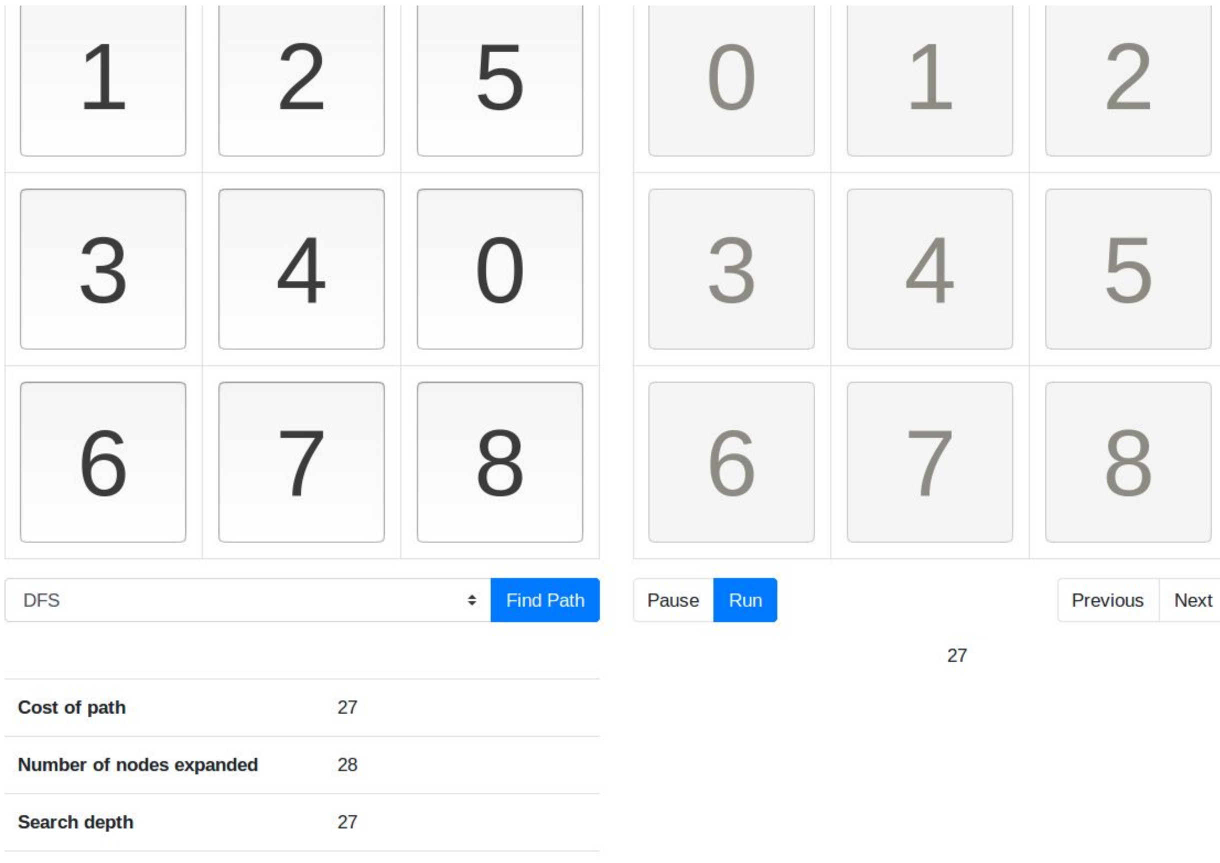
- ## Code

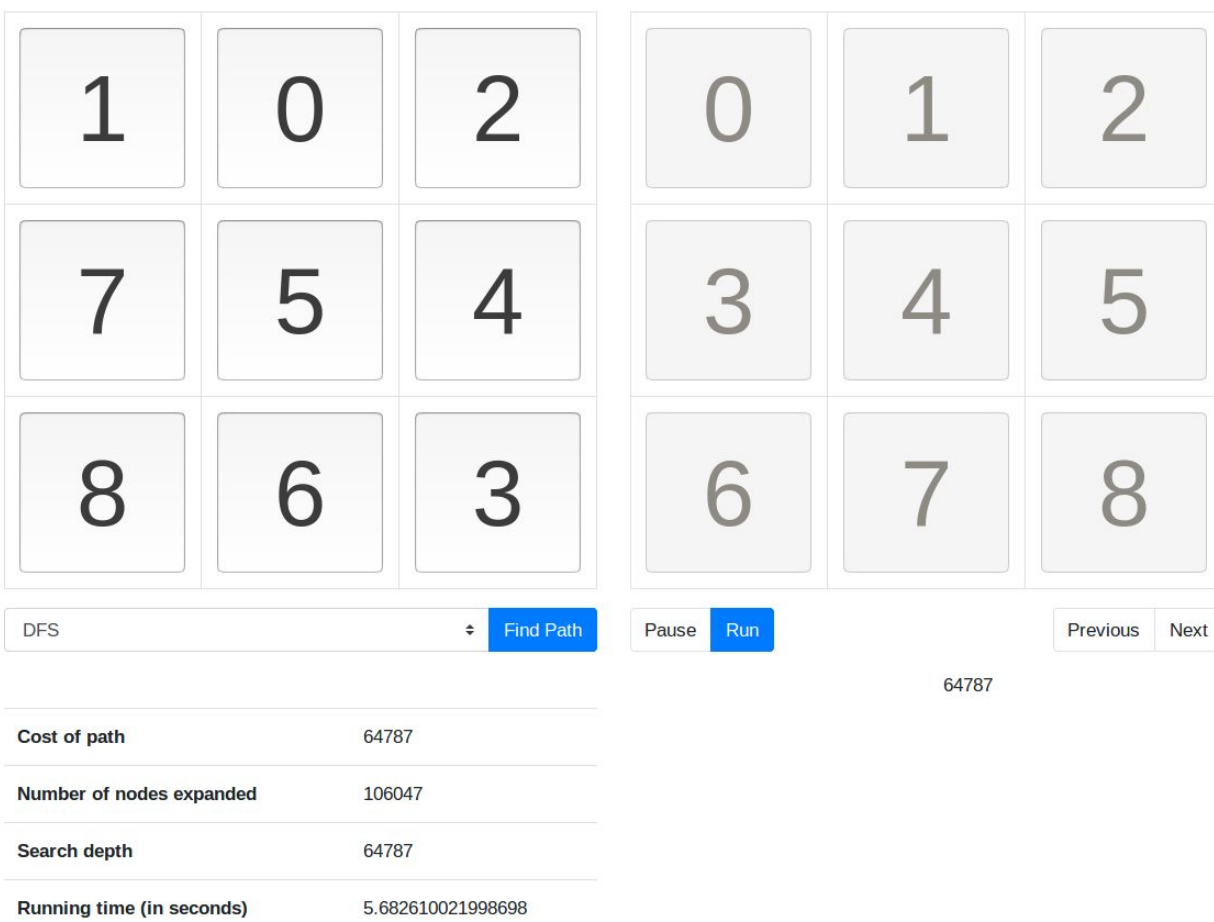
```
def search(self, initial_state):
    res = {}
    start = State(initial_state, None)
    if not self.check_solvable(start):
        return {'steps': [[-1] * 9], 'cost': -1,
                'search_depth': -1, 'nodes_expanded': -1}

    frontier = [start]
    final_state = None
    while frontier:
        state = frontier.pop()
        self.vis.add(state)
        self._explored += 1
        if state.is_goal():
            final_state = state
            break
        neighbours = self.expand(state)
        for neighbour in neighbours:
            #if neighbour not in frontier:
            frontier.append(neighbour)

    steps = self.get_steps(final_state)
    res['steps'] = steps
    res['cost'] = final_state.cost
    res['search_depth'] = final_state.cost
    res['nodes_expanded'] = self._explored
    return res
```

- ## Data Structures
  - Stack.

- ## Explanation
  - Frontier list that represents a stack that returns the first child of the current processed state.
  - Visited(explored set) set that using a hashing search in nearly O(1) for a faster search to check for no duplication of the states visited to avoid loops in the search operation.
  - We found that searching in stack is costly since it will be linear search O(n) So we assumed that we will insert all non visited children and also check if the not not visited the do nothing.
  - At the end returning all required numbers to be displayed in UI.

- **Sample runs**


1 2 5
3 4 0
6 7 8
0 1 2
3 4 5
6 7 8
DFS
Find Path
Pause
Run
Previous
Next
27
Cost of path 27
Number of nodes expanded 28
Search depth 27


1 0 2
7 5 4
8 6 3
0 1 2
3 4 5
6 7 8
DFS
Find Path
Pause
Run
Previous
Next
64787
Cost of path 64787
Number of nodes expanded 106047
Search depth 64787
Running time (in seconds) 5.682610021998698

3) BFS

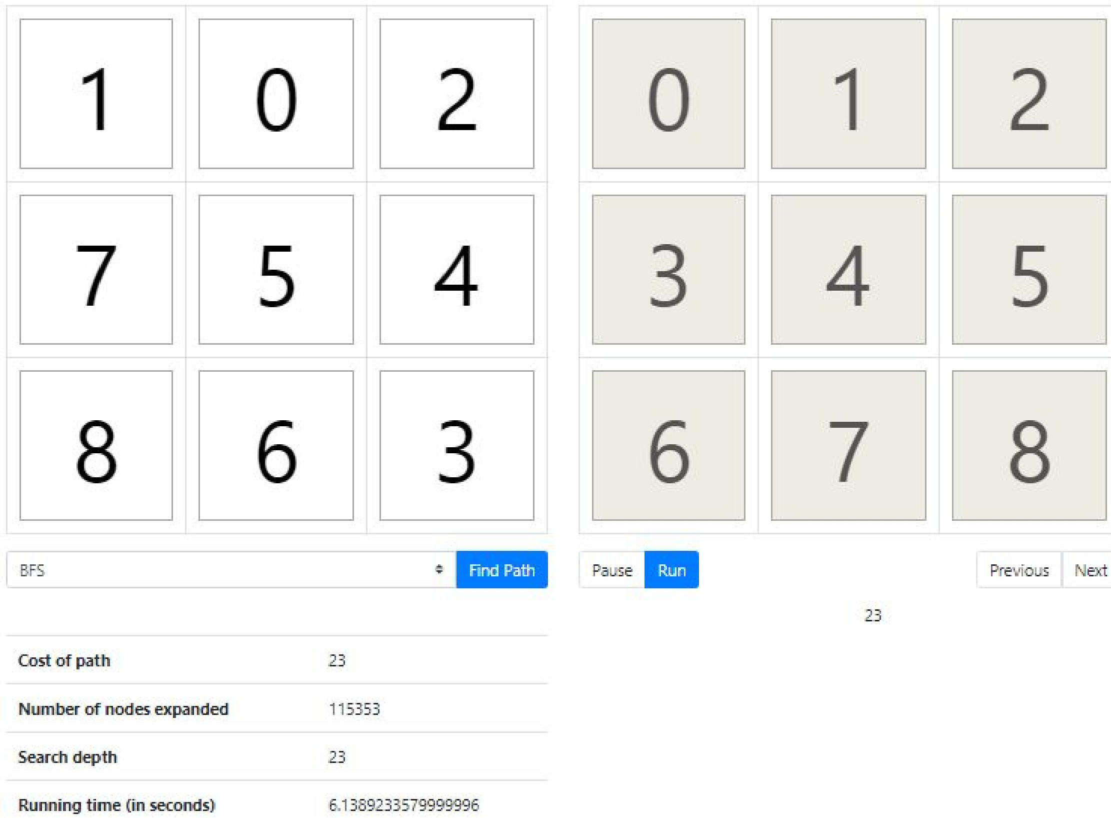
- ## Code

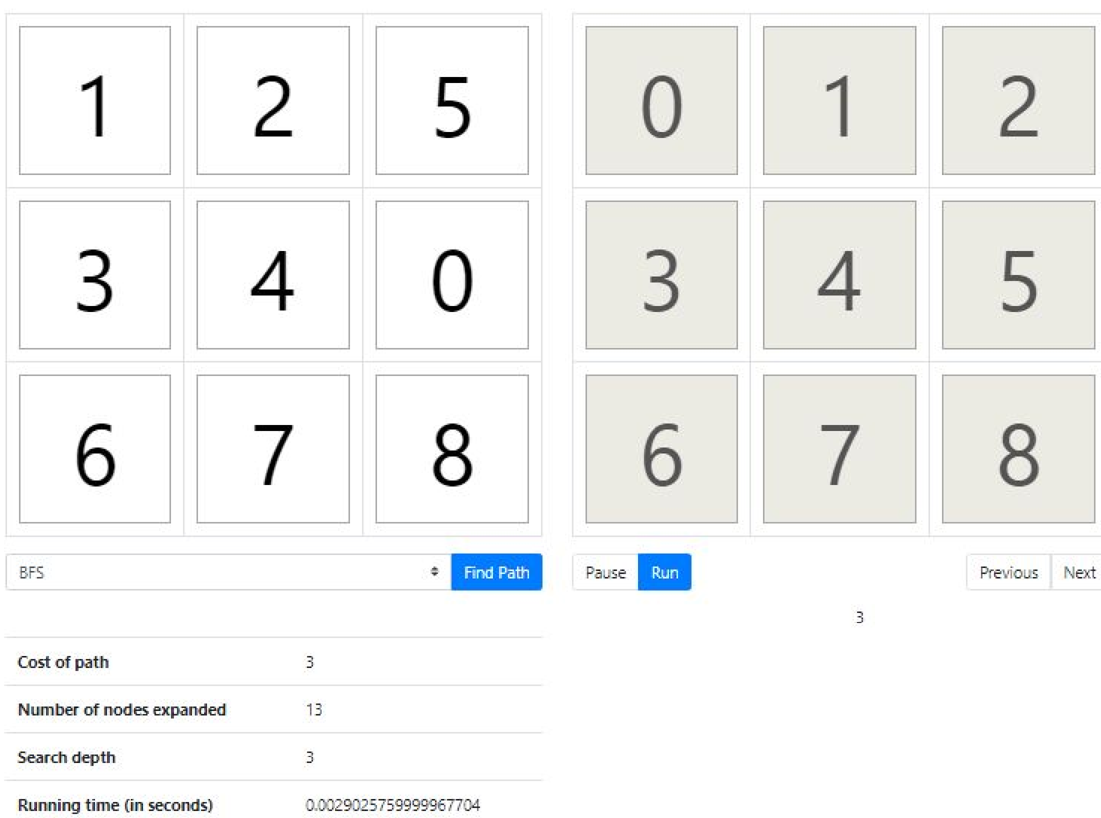
```
class BFS(Agent):

    def __init__(self):
        super().__init__()
        self._optimize_flag = True

    def search(self, initial_state):
        curr_state = State(initial_state, None)
        if not self.check_solvable(curr_state):
            return { 'steps' : [[-1] * 9], 'cost' : -1,
                     'searc_depth' : -1,'nodes_expanded' : -1}

        frontier = [curr_state]
        while frontier:
            curr_state = frontier.pop(0)
            self.vis.add(curr_state)
            self._explored += 1
            if curr_state.is_goal():
                break
            children = self.expand(curr_state)
            for child in children:
                #if child not in frontier:
                frontier.append(child)

        steps = self.get_steps(curr_state)
        res = {}
        res['steps'] = steps
        res['cost'] = curr_state.cost
        res['search_depth'] = curr_state.cost
        res['nodes_expanded'] = self._explored
        return res
```

- ## Data Structures
  - Queue to represent the frontier list.
  - Set to keep the Visited(explored set) states to avoid repeating states using a hashing search in nearly O(1) for a faster search.

- ## Explanation
  - State object that contains a total cost representing f(n)
  - Visited(explored set) set that using a hashing search in nearly O(1) for a faster search.
  - We pop the first inserted states (FIFO) then expand it to find all the child states and insert them in the queue to be explored later.
  - At the end after we reached the goal state we return all required outputs to be displayed in UI.

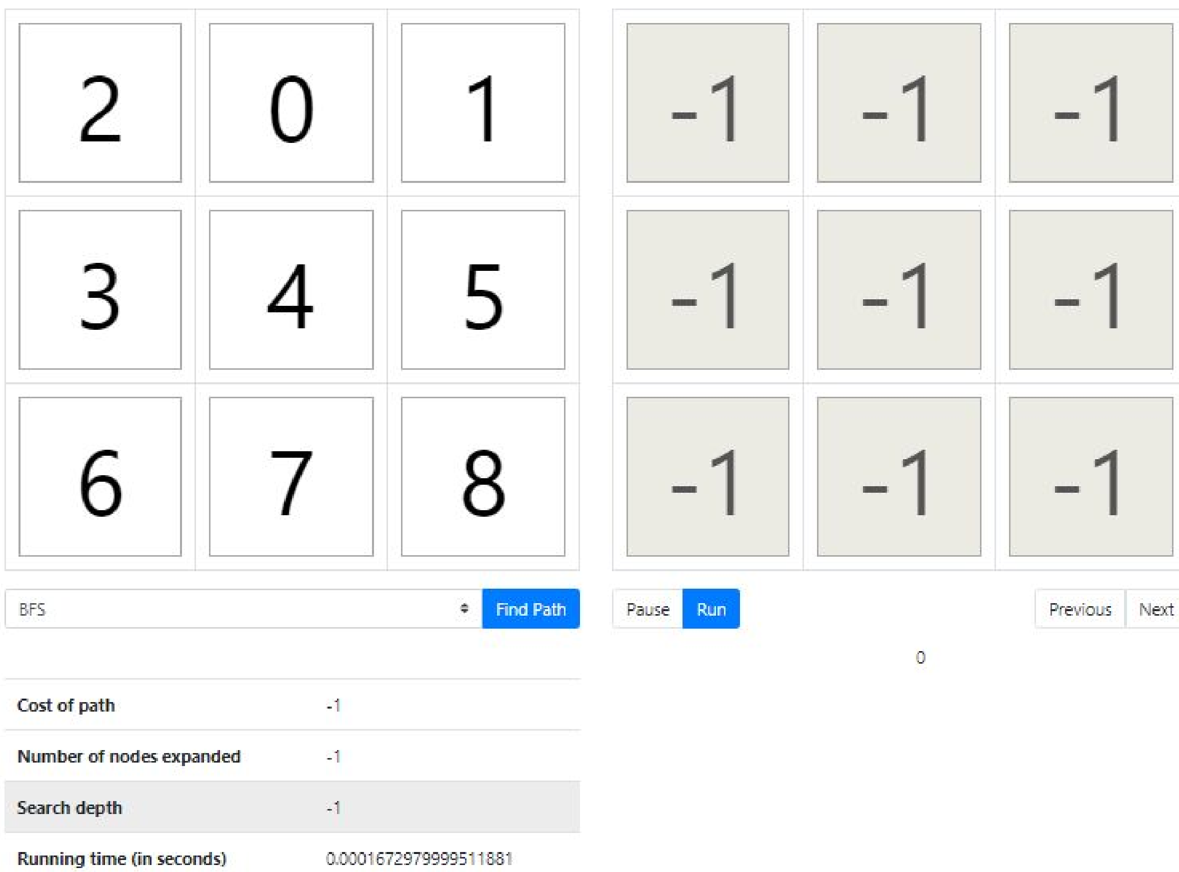
- **Sample runs**


1 0 2
7 5 4
8 6 3
0 1 2
3 4 5
6 7 8
BFS
Find Path
Pause
Run
Previous
Next
23
Cost of path 23
Number of nodes expanded 115353
Search depth 23
Running time (in seconds) 6.1389233579999996


1 2 5
3 4 0
6 7 8
0 1 2
3 4 5
6 7 8
BFS
Find Path
Pause
Run
Previous
Next
3
Cost of path 3
Number of nodes expanded 13
Search depth 3
Running time (in seconds) 0.0029025759999967704

Assumptions

- We check If the puzzle is unsolvable by counting number of inversions in the initial grid, if it is unsolvable we will not run the algorithm as it will be waste of time then we will make alert with an error message in UI and fill the UI output with -1 to indicate an error occured. Otherwise the grid is solvable and we will find the solution.



- We set a timeout with 50 seconds, So if the algorithm can't solve the problem within 50 seconds then we will ignore the request.