Artificial Intelligence

Lab1 Report

Using Informed and Uninformed Search Algorithms to Solve 8-Puzzle

Names & Ids :

Nour Mohamed Mahmoud 7591

Pola Qulta 7685

Peter Mina 7357

1-Data Structure and algorithms

- We used several data structures and algorithms in our agent project:
    - Firstly, we check if the game is solvable or not by the count of the inversion .
    - In the BFS algorithm we used the queue to implement this algorithm with some modifications in the queue , like the function isInQueue() to check if the new state has been in the frontier before or not. The item being pushed and popped is an object of state(the game board in certain state), its parent state , the actual cost and the heuristic.

    - In the DFS algorithm we used the stack also with same modifications as the BFS.

    - In the A* algorithm we used MinHeap , we added some functions like decreaseKey() to update the state path with less cost and other functions.

    - We assumed that max depth on the DFS algorithm is 31 , so we can then backtrack up again if the doesn't exist the max depth.

2-How every Algorithm operate

1. First, we remove a node from the frontier set.

2. Second, we check the state against the goal state to determine if a solution has been found.

3. Finally, if the result of the check is negative, we then expand the node. To expand a given node, we generate successor nodes adjacent to the current node, and add them to the frontier set. Note that if these successor nodes are already in the frontier, or have already been visited, then they should not be added to the frontier again (little deferent in A*).

BFS search

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

DFS search

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

-In the BFS algorithm we expand the shallowest node, but in the DFS we expand the deepest node.

A* search

Taken from the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI)

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

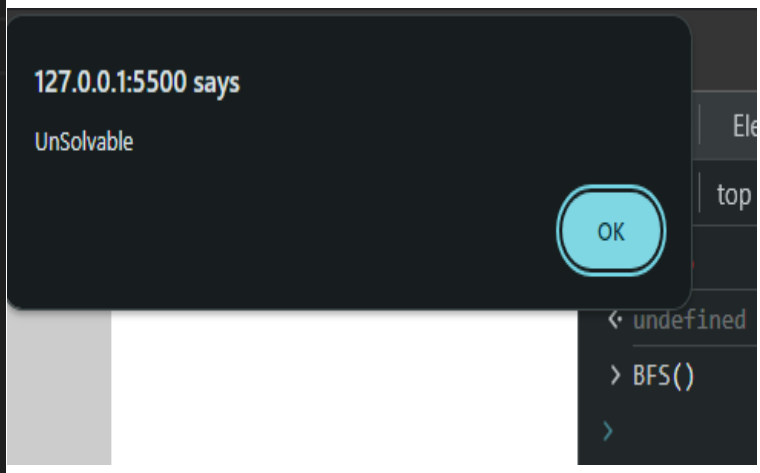
- First, insert the initial state (the initial board, 0 moves, and a null previous state) into a priority queue. Then, delete from the priority queue the state with the minimum priority, and insert onto the priority queue all neighboring states (those that can be reached in one move). Repeat this procedure until the state dequeued is the goal state. The success of this approach hinges on the choice of priority function for a state. We consider two priority functions:
- **Euclidian Distance priority function** , It is the distance between the current cell and the goal cell using the distance formula
- h= sqrt((currentcell:x - goal:x)2+ (currentcell.y - goal:y)2)
- **Manhattan priority function.** The sum of the distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the state using the formula
- h = abs(currentcell:x - goal:x) + abs(currentcell:y - goal:y)

3-Running some test cases

BFS:

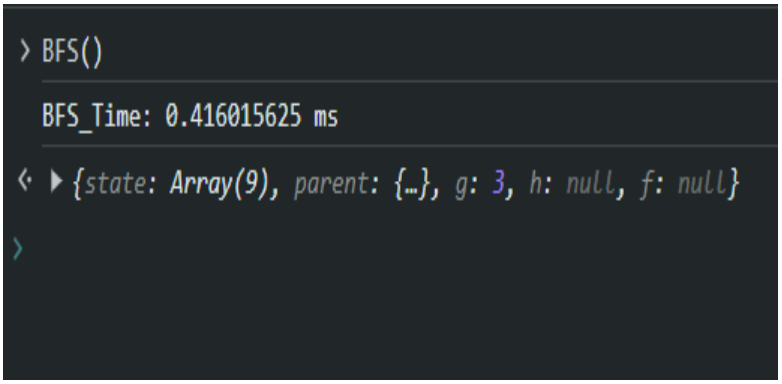
1- Unsolvable Example:

```
var initialState = {
    state: [
        1, 0, 3,
        2, 4, 5,
        6, 7, 8
    ], parent: null,
    f: null,
    g:0,
    h:null,
};
```

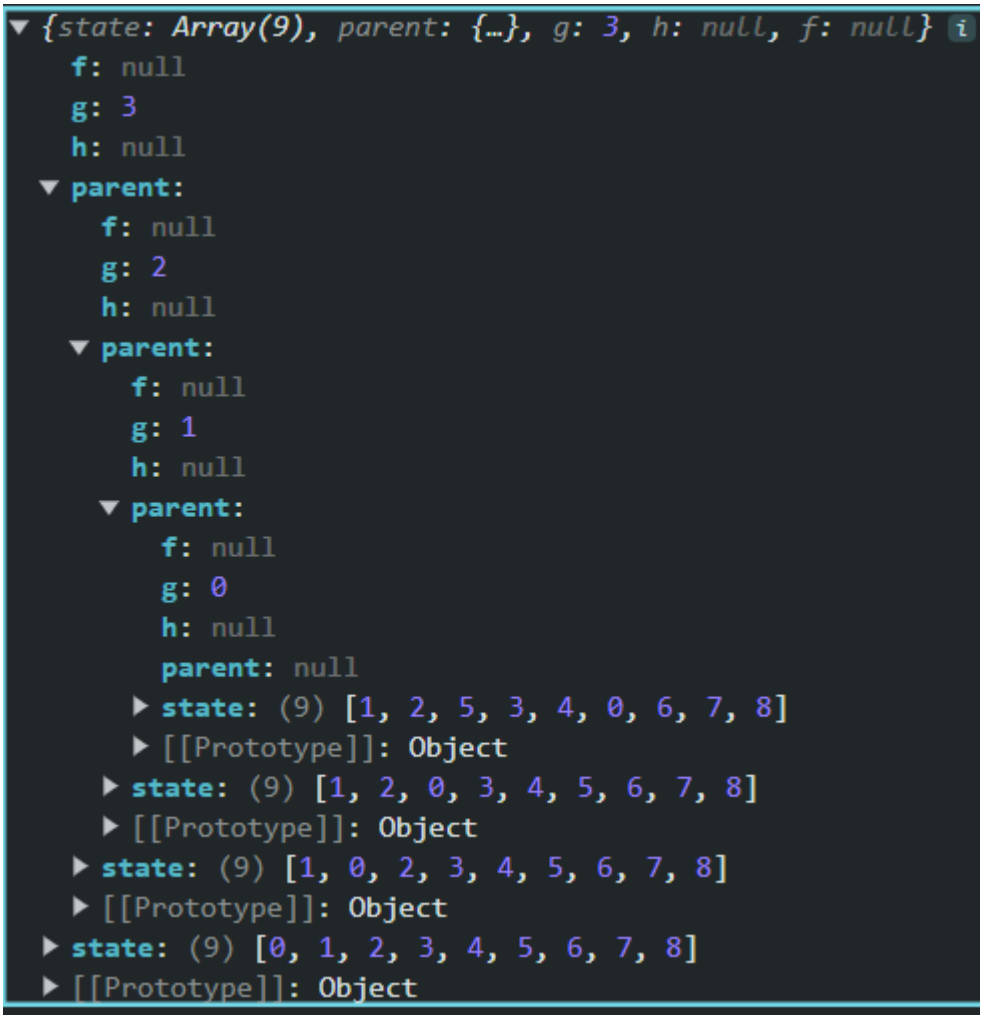


2- Solvable Example:

```
224  var initialState = {
225      state: [
226          1, 2, 5,
227          3, 4, 0,
228          6, 7, 8
229      ], parent: null,
230      f: null,
231      g:0,
232      h:null,
233  };
```



Path to goal, nodes expanded:



Cost of the path (g) = 3

Search Depth = 3

Running time = 0.416

DFS:

```
224  var initialState = {
225      state: [
226          1, 2, 5,
227          3, 4, 0,
228          6, 7, 8
229      ], parent: null,
230      f: null,
231      g:0,
232      h:null,
233  };
```

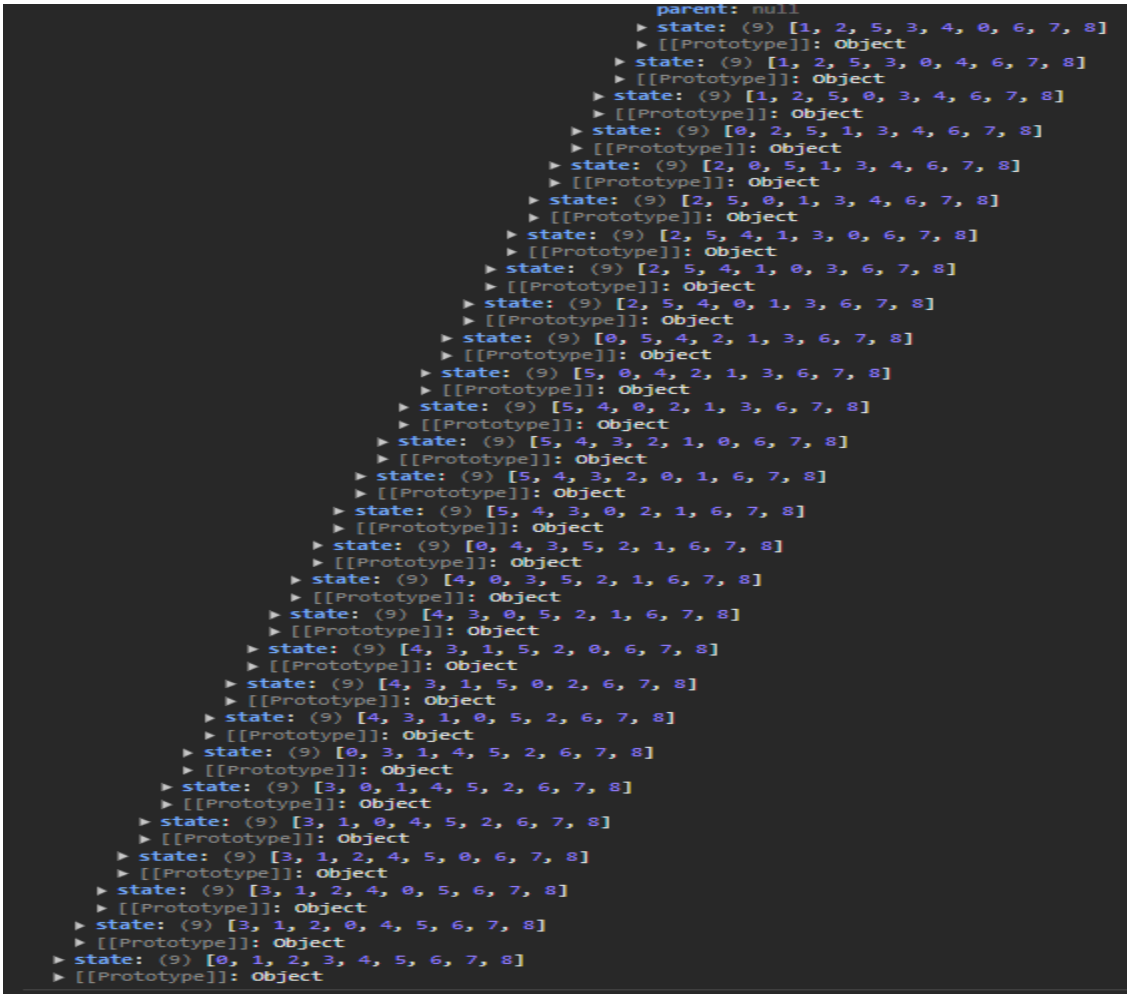
```
> DFS()

  DFS_Time: 0.659912109375 ms

< ▶ {state: Array(9), parent: {…}, g: 27, h: null, f: null}

>
```

Path to goal, nodes expanded:



Cost of the path (g) = 27

Search Depth = 27

Running time = 0.655

A*(Manhattan Distance):

```
224  var initialState = {
225      state: [
226          1, 2, 5,
227          3, 4, 0,
228          6, 7, 8
229      ], parent: null,
230      f: null,
231      g:0,
232      h:null,
233  };
```

```
> A_Star_Search(ManhattanDistance)
  A*_Time: 0.152099609375 ms
< ▶ {state: Array(9), parent: {…}, g: 3, h: 0, f: 3}
```

Path to goal, nodes expanded:

```
> A_Star_Search(ManhattanDistance)
  A*_Time: 0.152099609375 ms
< ▼ {state: Array(9), parent: {…}, g: 3, h: 0, f: 3}
    f: 3
    g: 3
    h: 0
  ▼ parent:
      f: 4
      g: 2
      h: 2
    ▼ parent:
        f: 5
        g: 1
        h: 4
      ▼ parent:
          f: null
          g: 0
          h: null
          parent: null
        ▶ state: (9) [1, 2, 5, 3, 4, 0, 6, 7, 8]
        ▶ [[Prototype]]: Object
      ▶ state: (9) [1, 2, 0, 3, 4, 5, 6, 7, 8]
      ▶ [[Prototype]]: Object
    ▶ state: (9) [1, 0, 2, 3, 4, 5, 6, 7, 8]
    ▶ [[Prototype]]: Object
  ▶ state: (9) [0, 1, 2, 3, 4, 5, 6, 7, 8]
  ▶ [[Prototype]]: Object
```

Cost of the path (g) = 3

Search Depth = 3

Running time = 0.152

A*(Euclidian Distance):

```
224  var initialState = {
225      state: [
226          1, 2, 5,
227          3, 4, 0,
228          6, 7, 8
229      ], parent: null,
230      f: null,
231      g:0,
232      h:null,
233  };
```

```
> A_Star_Search(EuclideanDistance)
  A*_Time: 0.256103515625 ms
< ▶ {state: Array(9), parent: {…}, g: 3, h: 0, f: 3}
```

Path to goal, nodes expanded:

Cost of the path (g) = 3

Search Depth = 3

Running time = 0.256

```
> A_Star_Search(EuclideanDistance)
  A*_Time: 0.256103515625 ms
< ▼ {state: Array(9), parent: {…}, g: 3, h: 0, f: 3} i
    f: 3
    g: 3
    h: 0
  ▼ parent:
      f: 4
      g: 2
      h: 2
    ▼ parent:
        f: 5
        g: 1
        h: 4
      ▼ parent:
          f: null
          g: 0
          h: null
          parent: null
        ▶ state: (9) [1, 2, 5, 3, 4, 0, 6, 7, 8]
        ▶ [[Prototype]]: Object
      ▶ state: (9) [1, 2, 0, 3, 4, 5, 6, 7, 8]
      ▶ [[Prototype]]: Object
    ▶ state: (9) [1, 0, 2, 3, 4, 5, 6, 7, 8]
    ▶ [[Prototype]]: Object
  ▶ state: (9) [0, 1, 2, 3, 4, 5, 6, 7, 8]
  ▶ [[Prototype]]: Object
```

- **A* Notes :**
- **The Manhattan is less time search that the Euclidian distance due to the squaring for the distance calculations .**