Artificial Intelligence

Islam Mustafa Fekry 4595

Abdelrahman Khaled 5013

Abdelrahman Mohammed 5027

8-Puzzle

A board with 8 distinct moveable tiles and an empty space makes up an example of the 8-puzzle game. Any tile horizontally or vertically next to the empty spot on such a board may be legally exchanged with it. The number 0 will be used to represent the blank area in this assignment.

The search space is the set of all possible states reachable from the initial state. The blank space may be swapped with a component in one of the four directions 'Up', 'Down', 'Left', 'Right', one move at a time. The cost of moving from one configuration of the board to another is the same and equal to one. Thus, the total cost of path is equal to the number of moves made from the initial state to the goal state.

Data structure:

Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order is last in first out.

A stack shall be implemented for the DFS. As python does not have an object stack therefore we used the 'list' object so when the search method called is DFS, the function push (in the Frontier class.) uses the normal append which can work

as a typical push function for a stack while the pop (in the frontier class.) calls the normal function python pop that acts as a normal stack pop and pops out the last in object.

Heap

A heap is a specialized tree which is essentially an almost complete tree that satisfies the heap property: in a max heap, for any given node C, if P is a parent node of C, then the key (the value) of P is greater than or equal to the key of C and less than/equal for min heap. The node at the "top" of the heap is called the root node. The import 'heapq' is used so the tree based ds can be used for the A* search. A heap shall be implemented for this search so when the search method called is A*, the function push (in the Frontier class.) calls the heappush function which is specified for that import while the pop (in the frontier class.) calls the function heappop that is customized for heaps as well.

Depth-first search (DFS):

As its name implies, DFS performs the search by traversing the graph branch by branch, that is, it chooses a child of the root's and goes as deep as possible before moving to the next child. The algorithm works as follows:

```
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
```

Sample runs:

Case 1: 1,2,5,3,4,0,6,7,8

```
START
[[1 2 5]
 [3 4 0]
 [6 7 8]]
----------
UP
[[1 2 0]
 [3 4 5]
 [6 7 8]]
----------
LEFT
[[1 0 2]
 [3 4 5]
 [6 7 8]]
----------
LEFT
[[0 1 2]
 [3 4 5]
 [6 7 8]]

Path cost: 3
Path:
UP, LEFT, LEFT

Nodes expanded: 181439
Search depth: 66125
Time taken: 10.073448241000001s
```

Case 2: 3,1,2,0,4,5,6,7,8

```
START
[[3 1 2]
 [0 4 5]
 [6 7 8]]
----------
UP
[[0 1 2]
 [3 4 5]
 [6 7 8]]

Path cost: 1
Path:
UP

Nodes expanded: 3
Search depth: 1
Time taken: 0.00078078300000000071s
```

Case 3: 0,8,7,6,5,4,3,2,1

File dfs3.txt is attached with case 3 output included all details.

Path cost: 41910

Nodes expanded: 144635

Search depth: 65982

Time taken: 8.142810453s

A* search:

A* search uses a cost function called a "heuristic" to decide the next move. Thus, it chooses the least costly move among all possible moves on each iteration. The used heuristics in this assignment are:

Manhattan Distance:

$$cost = |currentCell.x - goal.x| + |currentCell.y - goal.y|$$

Euclidean Distance:

$$cost = \sqrt{(currentCell.x - goal.x)^2 + (currentCell.y - goal.y)^2}$$

The algorithm works as follows:

```
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
```

Sample runs:

Case 1: 1,2,5,3,4,0,6,7,8

Manhattan:

```
START
[[1 2 5]
 [3 4 0]
 [6 7 8]]
Heuristic: 6
Actual cost: 3 (NOT ADMISSABLE)
----------
UP
[[1 2 0]
 [3 4 5]
 [6 7 8]]
Heuristic: 4
Actual cost: 2 (NOT ADMISSABLE)
----------
LEFT
[[1 0 2]
 [3 4 5]
 [6 7 8]]
Heuristic: 2
Actual cost: 1 (NOT ADMISSABLE)
----------
LEFT
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Heuristic: 0
Actual cost: 0 (ADMISSABLE)

Admissable rate: 0.25
Path cost: 3
Path:
UP, LEFT, LEFT

Nodes expanded: 5
Search depth: 3
Time taken: 0.0014355999999999813s
```

Euclidean:

```
START
[[1 2 5]
 [3 4 0]
 [6 7 8]]
Heuristic: 5.23606797749979
Actual cost: 3 (NOT ADMISSABLE)
----------
UP
[[1 2 0]
 [3 4 5]
 [6 7 8]]
Heuristic: 4.0
Actual cost: 2 (NOT ADMISSABLE)
----------
LEFT
[[1 0 2]
 [3 4 5]
 [6 7 8]]
Heuristic: 2.0
Actual cost: 1 (NOT ADMISSABLE)
----------
LEFT
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Heuristic: 0.0
Actual cost: 0 (ADMISSABLE)

Admissable rate: 0.25
Path cost: 3
Path:
UP, LEFT, LEFT

Nodes expanded: 5
Search depth: 3
Time taken: 0.00433191599999991s
```

Case 2: 3,1,2,0,4,5,6,7,8

Manhattan:

```
START
[[3 1 2]
 [0 4 5]
 [6 7 8]]
Heuristic: 2
Actual cost: 1 (NOT ADMISSABLE)
----------
UP
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Heuristic: 0
Actual cost: 0 (ADMISSABLE)

Admissable rate: 0.5
Path cost: 1
Path:
UP

Nodes expanded: 3
Search depth: 1
Time taken: 0.0023492569999999935s
```

Euclidean:

```
START
[[3 1 2]
 [0 4 5]
 [6 7 8]]
Heuristic: 2.0
Actual cost: 1 (NOT ADMISSABLE)
----------
UP
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Heuristic: 0.0
Actual cost: 0 (ADMISSABLE)

Admissable rate: 0.5
Path cost: 1
Path:
UP

Nodes expanded: 3
Search depth: 1
Time taken: 0.00253718699999999963s
```

Case 3: 0,8,7,6,5,4,3,2,1

Manhattan: file .txt attached with tables and ADMISSABLE or NOT.

```
Admissable rate: 0.8064516129032258
Path cost: 30
Path:
RIGHT, DOWN, LEFT, UP, RIGHT, RIGHT, DOWN, DOWN, LEFT, UP, LEFT, DOWN,
RIGHT, UP, UP, RIGHT, DOWN, DOWN, LEFT, UP, UP, LEFT, DOWN, RIGHT, UP,
RIGHT, DOWN, LEFT, LEFT, UP

Nodes expanded: 11900
Search depth: 30
Time taken: 1.991688821s
```

Euclidean: file .txt attached with tables and ADMISSABLE or NOT.

```
Admissable rate: 0.8064516129032258
Path cost: 30
Path:
RIGHT, DOWN, LEFT, UP, RIGHT, RIGHT, DOWN, DOWN, LEFT, UP, LEFT, DOWN,
RIGHT, UP, UP, RIGHT, DOWN, DOWN, LEFT, UP, UP, LEFT, DOWN, RIGHT, UP,
RIGHT, DOWN, LEFT, LEFT, UP

Nodes expanded: 33200
Search depth: 30
Time taken: 5.67259587s
```