RA1911003020480

Nikith Kumar Seemakurthi

Experiment No 2: Developing agent programs for real world problems (8-puzzle)

AIM:

To develop agent programs for real world problems (8-puzzle) using python

ALGORITHM:

1. It is solved using A* algorithm with plug-in heuristics.

2. First move the empty space in all the possible directions in the start state and calculate the f-score for each state.

3. It is pushed into the closed list and the newly generated states are pushed into the open list. A state with the least f-score is selected and expanded again.

4. This process continues until the goal state occurs as the current state.

5. The algorithm chooses the best possible action and proceeds in that path.

SOURCE CODE:

```
import random
import math

_goal_state = [[1,2,3],
               [4,5,6],
               [7,8,0]]

def index(item, seq):
    if item in seq:
        return seq.index(item)
    else:
        return -1

class EightPuzzle:

    def __init__(self):
```

```
        # heuristic value
        self._hval = 0
        # search depth of current instance
        self._depth = 0
        # parent node in search path
        self._parent = None
        self.adj_matrix = []
        for i in range(3):
            self.adj_matrix.append(_goal_state[i][:])

    def __eq__(self, other):
        if self.__class__ != other.__class__:
            return False
        else:
            return self.adj_matrix == other.adj_matrix

    def __str__(self):
        res = ''
        for row in range(3):
            res += ' '.join(map(str, self.adj_matrix[row]))
            res += '\r\n'
        return res

    def _clone(self):
        p = EightPuzzle()
        for i in range(3):
            p.adj_matrix[i] = self.adj_matrix[i][:]
        return p

    def _get_legal_moves(self):
        """Returns list of tuples with which the free space may
        be swapped"""
        # get row and column of the empty piece
        row, col = self.find(0)
        free = []
        if row > 0:
            free.append((row - 1, col))
        if col > 0:
            free.append((row, col - 1))
        if row < 2:
```

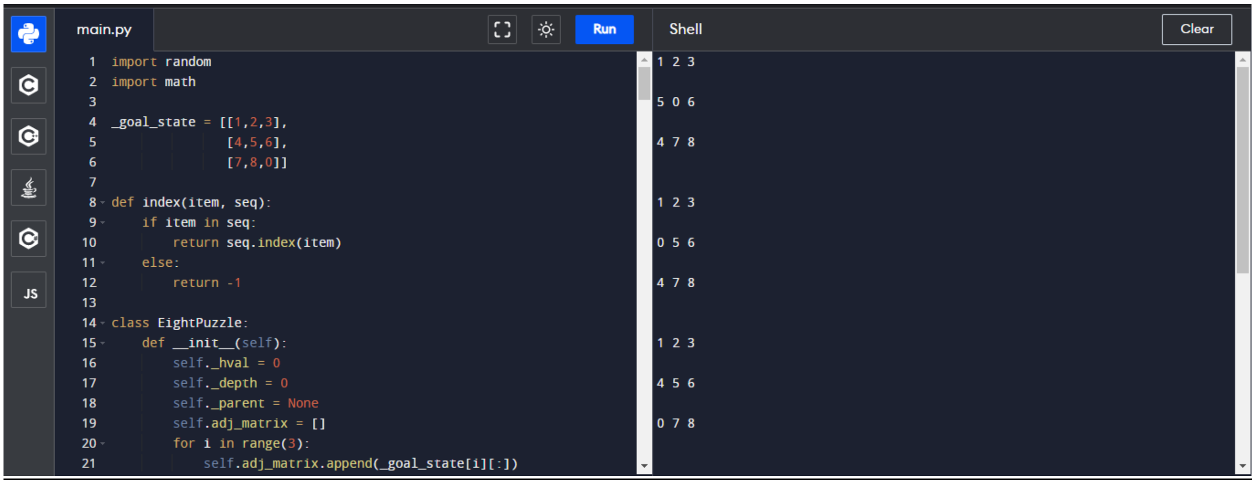
```
            free.append((row + 1, col))
        if col < 2:
            free.append((row, col + 1))
        return free

    def _generate_moves(self):
        free = self._get_legal_moves()
        zero = self.find(0)

        def swap_and_clone(a, b):
            p = self._clone()
            p.swap(a,b)
            p._depth = self._depth + 1
            p._parent = self
            return p
        return map(lambda pair: swap_and_clone(zero, pair), free)

    def _generate_solution_path(self, path):
        if self._parent == None:
            return path
        else:
            path.append(self)
            return self._parent._generate_solution_path(path)

    def solve(self, h):
        def is_solved(puzzle):
            return puzzle.adj_matrix == _goal_state

        openl = [self]
        closedl = []
        move_count = 0
        while len(openl) > 0:
            x = openl.pop(0)
            move_count += 1
            if (is_solved(x)):
                if len(closedl) > 0:
                    return x._generate_solution_path([]), move_count
                else:
                    return [x]
```

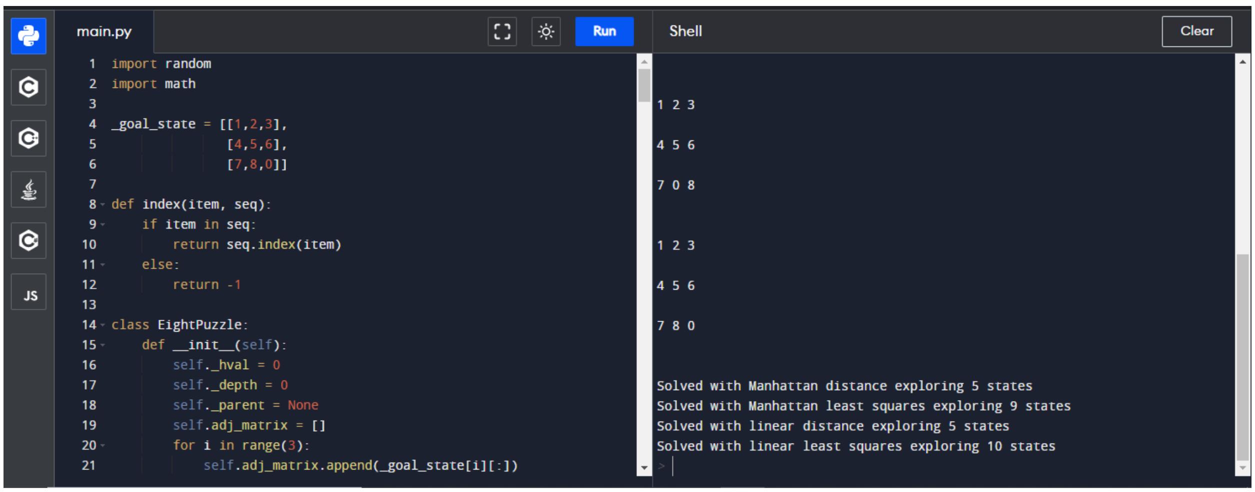
```
        succ = x._generate_moves()
        idx_open = idx_closed = -1
        for move in succ:
            # have we already seen this node?
            idx_open = index(move, openl)
            idx_closed = index(move, closedl)
            hval = h(move)
            fval = hval + move._depth

            if idx_closed == -1 and idx_open == -1:
                move._hval = hval
                openl.append(move)
            elif idx_open > -1:
                copy = openl[idx_open]
                if fval < copy._hval + copy._depth:
                    # copy move's values over existing
                    copy._hval = hval
                    copy._parent = move._parent
                    copy._depth = move._depth
            elif idx_closed > -1:
                copy = closedl[idx_closed]
                if fval < copy._hval + copy._depth:
                    move._hval = hval
                    closedl.remove(copy)
                    openl.append(move)

        closedl.append(x)
        openl = sorted(openl, key=lambda p: p._hval + p._depth)
    return [], 0

def shuffle(self, step_count):
    for i in range(step_count):
        row, col = self.find(0)
        free = self._get_legal_moves()
        target = random.choice(free)
        self.swap((row, col), target)
        row, col = target

def find(self, value):
    if value < 0 or value > 8:
```

```
            raise Exception("value out of range")

        for row in range(3):
            for col in range(3):
                if self.adj_matrix[row][col] == value:
                    return row, col

    def peek(self, row, col):
        return self.adj_matrix[row][col]

    def poke(self, row, col, value):
        self.adj_matrix[row][col] = value

    def swap(self, pos_a, pos_b):
        temp = self.peek(*pos_a)
        self.poke(pos_a[0], pos_a[1], self.peek(*pos_b))
        self.poke(pos_b[0], pos_b[1], temp)

def heur(puzzle, item_total_calc, total_calc):
    t = 0
    for row in range(3):
        for col in range(3):
            val = puzzle.peek(row, col) - 1
            target_col = val % 3
            target_row = val / 3

            # account for 0 as blank
            if target_row < 0:
                target_row = 2

            t += item_total_calc(row, target_row, col, target_col)

    return total_calc(t)

def h_manhattan(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: abs(tr - r) + abs(tc - c),
                lambda t : t)

def h_manhattan_lsq(puzzle):
```

```
    return heur(puzzle,
                lambda r, tr, c, tc: (abs(tr - r) + abs(tc - c))**2,
                lambda t: math.sqrt(t))

def h_linear(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: math.sqrt(math.sqrt((tr - r)**2 + (tc - c)**2)),
                lambda t: t)

def h_linear_lsq(puzzle):
    return heur(puzzle,
                lambda r, tr, c, tc: (tr - r)**2 + (tc - c)**2,
                lambda t: math.sqrt(t))

def h_default(puzzle):
    return 0

def main():
    p = EightPuzzle()
    p.shuffle(20)
    print (p)
    path, count = p.solve(h_manhattan)
    path.reverse()
    for i in path:
        print (i)
    print ("Solved with Manhattan distance exploring", count, "states")
    path, count = p.solve(h_manhattan_lsq)
    print ("Solved with Manhattan least squares exploring", count, "states")
    path, count = p.solve(h_linear)
    print ("Solved with linear distance exploring", count, "states")
    path, count = p.solve(h_linear_lsq)
    print ("Solved with linear least squares exploring", count, "states")
#    path, count = p.solve(heur_default)
#    print ("Solved with BFS-equivalent in", count, "moves")

if __name__ == "__main__":
    main()
```

OUTPUT:





RESULT:

Hence agent programs for real world problems (8-puzzle) using python is developed.