PROGRAMMING PROJECT 1

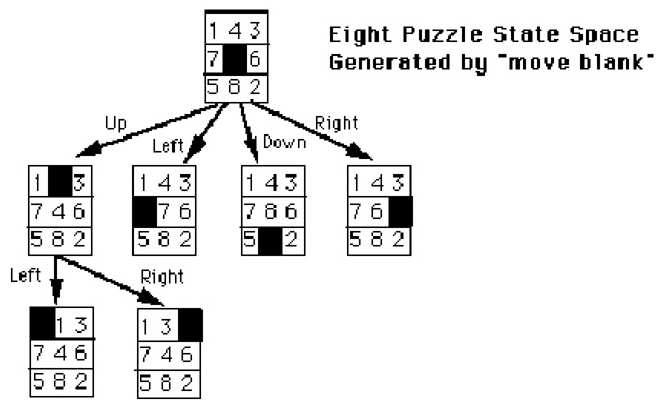
Solving the 8-puzzle using A* Algorithm

Anushree Srivastava
Shilpa Goel

8-Puzzle Problem: The eight puzzle consists of a 3 x 3 grid with 8 consecutively numbered tiles arranged on it. Any tile adjacent to the space can be moved on it. A number of different goal states are used.

State space representation: A state space essentially consists of a set of nodes representing each state of the problem, arcs between nodes representing the legal moves from one state to another, an initial state and a goal state. The state space is searched to find a solution to the problem. Here 0 represents the blank position (space) on the board.

- In the state space representation of the problem:
  - **Nodes** of a graph correspond to partial problem solution **states.**
  - **Arcs** correspond to **steps (application of operators)** in a problem solving process **-** The operators can be thought of in terms of the direction that the blank space effectively moves. i.e**. up, down, left, right**
  - **The root** of the graph corresponds to the **initial state** of the problem.
  - **The goal** node is a leaf node which corresponds to a **goal state**



A* Search Algorithm: It is an informed search algorithm or a best first search meaning that it is formulated in terms of weighted graphs : starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n)=g(n)+h(n)$$

where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal.

Two different examples of admissible heuristics :
Hamming distance:The number of Misplaced Tiles
Manhattan distance: The distance between two points measured along axes at right angles.

Problem Implementation Details:

The 8 puzzle is one of the heuristic search problem. The objective of the puzzle is to slide the tiles horizontally or vertically into the spaces until the configuration matches the goal configuration using A* algorithm.

We have used Python Programming Language to implement it.

Global Variables:

- _initial_state : Starting state of the problem
- _goal_state : Goal state of the problem
- _generatedNodes : No of nodes generated by each of the heuristic

Functions:

- def _getIndex(item, queue)
- def __init__(self)
- def __eq__(self, other)
- def __str__(self)
- def _get_possible_moves(self)
- def _generate_moves(self)
- def swap_and_clone(a, b)
- def _generate_solution_path(self, allNodes)
- def solve(self, h)
- def is_solved(puzzle)
- def findCoord(self, value)
- def getValue(self, row, col)
- def setValue(self, row, col, value)
- def swap(self, pos_a, pos_b)
- def misplaced_Tiles(puzzle)
- def manhattan(puzzle)
- def main()

The main() function defined above is called at the end of the program after defining all the functions, then it first solves the problem using Manhattan distance heuristic and then misplaced tiles heuristic.

Test Cases:

S.No	Initial State	Goal State	Manhattan Distance Heuristic		Misplaced Tiles Heuristic	
			Nodes Generated	Nodes Expanded	Nodes Generated	Nodes Expanded

1	1 2 3 4 8 0 7 6 5	1 2 3 4 5 6 7 8 0	11	5	17	8
2	1 2 3 7 4 5 6 8 0	1 2 3 8 6 4 7 5 0	18	9	41	21
3	2 8 1 3 4 6 7 5 0	3 2 1 8 0 4 7 5 6	12	6	14	7
4	1 2 6 7 4 3 0 8 5	1 2 3 4 5 6 7 8 0	857	534	2629	1628

Solution Paths:

Manhattan Heuristic Path				**Misplaced Tile Heuristic Path**			
1	**2**	**3**	**4**	**1**	**2**	**3**	**4**
1 2 3 4 8 5 7 6 0	1 2 3 7 4 0 6 8 5	2 8 1 3 4 0 7 5 6	1 2 6 0 4 3 7 8 5	1 2 3 4 8 5 7 6 0	1 2 3 7 4 0 6 8 5	2 8 1 3 4 0 7 5 6	1 2 6 0 4 3 7 8 5
1 2 3 4 8 5 7 0 6	1 2 3 7 0 4 6 8 5	2 8 1 3 0 4 7 5 6	1 2 6 4 0 3 7 8 5	1 2 3 4 8 5 7 0 6	1 2 3 7 0 4 6 8 5	2 8 1 3 0 4 7 5 6	1 2 6 4 0 3 7 8 5
1 2 3 4 0 5 7 8 6	1 2 3 7 8 4 6 0 5	2 0 1 3 8 4 7 5 6	1 2 6 4 3 0 7 8 5	1 2 3 4 0 5 7 8 6	1 2 3 7 8 4 6 0 5	2 0 1 3 8 4 7 5 6	1 2 6 4 3 0 7 8 5
1 2 3 4 5 0 7 8 6	1 2 3 7 8 4 0 6 5	0 2 1 3 8 4 7 5 6	1 2 6 4 3 5 7 8 0	1 2 3 4 5 0 7 8 6	1 2 3 7 8 4 0 6 5	0 2 1 3 8 4 7 5 6	1 2 6 4 3 5 7 8 0
1 2 3 4 5 6 7 8 0	1 2 3 0 8 4 7 6 5	3 2 1 0 8 4 7 5 6	1 2 6 4 3 5 7 0 8	1 2 3 4 5 6 7 8 0	1 2 3 0 8 4 7 6 5	3 2 1 0 8 4 7 5 6	1 2 6 4 3 5 7 0 8
	1 2 3 8 0 4 7 6 5	3 2 1 8 0 4 7 5 6	1 2 6 4 0 5 7 3 8		1 2 3 8 0 4 7 6 5	3 2 1 8 0 4 7 5 6	1 2 6 4 0 5 7 3 8
	1 2 3 8 6 4 7 0 5		1 2 6 4 5 0 7 3 8		1 2 3 8 6 4 7 0 5		1 2 6 4 5 0 7 3 8
	1 2 3 8 6 4		1 2 0 4 5 6		1 2 3 8 6 4		1 2 0 4 5 6

	7 5 0		7 3 8 1 0 2 4 5 6 7 3 8 1 5 2 4 0 6 7 3 8 1 5 2 4 3 6 7 0 8 1 5 2 4 3 6 7 8 0 1 5 2 4 3 0 7 8 6 1 5 2 4 0 3 7 8 6 1 0 2 4 5 3 7 8 6 1 2 0 4 5 3 7 8 6 1 2 3 4 5 0 7 8 6 1 2 3 4 5 6 7 8 0		7 5 0		7 3 8 1 0 2 4 5 6 7 3 8 1 5 2 4 0 6 7 3 8 1 5 2 4 3 6 7 0 8 1 5 2 4 3 6 7 8 0 1 5 2 4 3 0 7 8 6 1 5 2 4 0 3 7 8 6 1 0 2 4 5 3 7 8 6 1 2 0 4 5 3 7 8 6 1 2 3 4 5 0 7 8 6 1 2 3 4 5 6 7 8 0

Source Code:

```
# Anushree Srivastava and Shilpa Goel

# Global Variables
_initial_state = [[2, 8, 1],
            [3, 4, 6],
            [7, 5, 0]]
```

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

_generatedNodes = 0

def _getIndex(item, queue):
    """Helper function that returns -1 for non-found index value of a seq"""
    if item in queue:
        return queue.index(item)
    else:
        return -1

class AStar8Puzzle:

    def __init__(self):
        # heuristic value
        self._hn = 0
        # search g cost of current instance
        self._gn = 0
        # parent node in search path
        self._parent = None
        self.genState = []
        # initialize generated state as initial state
        for i in range(3):
            self.genState.append(_initial_state[i][:])

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

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

    """Returns list of tuples with which the free space may
        be swapped"""
    def _get_possible_moves(self):
```

```
        # get row and column of the empty piece
        row, col = self.findCoord(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_possible_moves()
        zero = self.findCoord(0)

        def swap_and_clone(a, b):
            p = AStar8Puzzle()
            for i in range(3):
                p.genState[i] = self.genState[i][:]  # Make Copy
            p.swap(a, b)
            p._gn = self._gn + 1
            p._parent = self
            return p

        return map(lambda pair: swap_and_clone(zero, pair), free)

    def _generate_solution_path(self, allNodes):
        if self._parent is None:
            return allNodes
        else:
            allNodes.append(self)
            return self._parent._generate_solution_path(allNodes)

    """Performs A* search for goal state.
         h(puzzle) - heuristic function, returns an integer
     """
    def solve(self, h):
        def is_solved(puzzle):
            if puzzle.genState == _goal_state:
                return True

        frontier = [self]
```

```
    explored = []
    move_count = 0
    while len(frontier) > 0:
        x = frontier.pop(0)
        move_count += 1
        if (is_solved(x)):
            if len(explored) > 0:
                return x._generate_solution_path([]), move_count, _generatedNodes
            else:
                return x, 0, 0

        successors = x._generate_moves()
        idx_open = idx_closed = -1
        for move in successors:
            # Checks if node is present in frontier or explored queues
            idx_open = _getIndex(move, frontier)
            idx_closed = _getIndex(move, explored)
            _hn = h(move)
            fn = _hn + move._gn

            if idx_closed == -1 and idx_open == -1:
                move._hn = _hn
                frontier.append(move)
            elif idx_open > -1:
                copy = frontier[idx_open]
                if fn < copy._hn + copy._gn:
                    # copy move's values over existing
                    copy._hn = _hn
                    copy._parent = move._parent
                    copy._gn = move._gn
            elif idx_closed > -1:
                copy = explored[idx_closed]
                if fn < copy._hn + copy._gn:
                    move._hn = _hn
                    explored.remove(copy)
                    frontier.append(move)

        explored.append(x)
        _generatedNodes = len(frontier)+len(explored) - 1
        frontier = sorted(frontier, key=lambda p: p._hn + p._gn)

    # if finished state not found, return failure
    return [], 0, 0

"""--------------------- UTILITY FUNCTIONS --------------------------"""
```

```
    # Find coordinates of specified value
    def findCoord(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.genState[row][col] == value:
                    return row, col

    # Returns value at specified coordinates
    def getValue(self, row, col):
        return self.genState[row][col]

    # Sets given value to the specified coordinates
    def setValue(self, row, col, value):
        self.genState[row][col] = value

    # Swaps values at the specified coordinates
    def swap(self, pos_a, pos_b):
        temp = self.getValue(*pos_a)
        self.setValue(pos_a[0], pos_a[1], self.getValue(*pos_b))
        self.setValue(pos_b[0], pos_b[1], temp)
    """-----------------------------------------------------------------------"""


# Calculating Misplaced Tiles heuristics
def misplaced_Tiles(puzzle):
    t = 0
    for row in range(3):
        for col in range(3):
            val = puzzle.getValue(row, col)
            if val != _goal_state[row][col] and val > 0:
                t += 1
    return t


# Calculating Manhattan Distance heuristic
def manhattan(puzzle):
    t = 0
    for row in range(3):
        for col in range(3):
            val = puzzle.getValue(row, col)
            for row1 in range(3):
                for col1 in range(3):
```

```
                if _goal_state[row1][col1] == val:
                    goal_col = col1
                    goal_row = row1
            # if value is 0, skip adding to heuristic value
            if val > 0:
                t += abs(goal_row - row) + abs(goal_col - col)

    return t


# Main execution
# Prints Nodes
def main():
    p = AStar8Puzzle()
    print("\nInitial State : ")
    print(p)

    all_nodes, expanded, generated = p.solve(manhattan)
    if isinstance(all_nodes, (list, tuple)):
        all_nodes.reverse()
        print("Manhattan heuristic Path : ")
        for i in all_nodes:
            print(i)

        print("Number of Nodes Generated: ", generated)
        print("Number of Nodes Expanded: ", expanded-1)

    else:
         print("Start state is goal state!")

    all_nodes, expanded, generated = p.solve(misplaced_Tiles)
    if isinstance(all_nodes, (list, tuple)):
        all_nodes.reverse()
        print("--------------------------------------------------------------")
        print("Misplaced Tiles heuristic Path : ")
        for i in all_nodes:
            print(i)

        print("Number of Nodes Generated: ", generated)
        print("Number of Nodes Expanded: ", expanded - 1)

# Execution Starts here
# main function call
main()
```