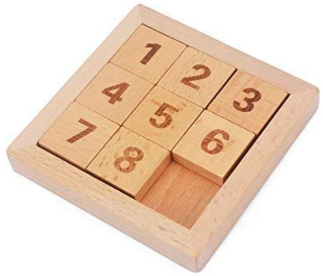
A* Algorithm-8 Puzzle Problem



DOCUMENTATION REPORT

PROGRAMMING PROJECT 1

ITCS 6150 - Intelligent Systems

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

DEVRAJ PATEL

801076509

UNC CHARLOTTE
College of Computing and Informatics

Table of Contents

PROBLEM FORMULATION

1.1 INTRODUCTION

8 Puzzle Brief :

Given a 3*3 grid contains 8 tiles, with one empty square. Each tile has a number from 1 to 8. We can move the adjacent tiles horizontally or vertically into the empty square, our goal is to rearrange the tiles so the number on each of the tile match the final order. There are more general puzzle possible like 16 Puzzle, 32 Puzzle,…., N Puzzle.

The 8 puzzle problem is described below by an example.

Example 1:

```
1 3 _       1 _ 3      1 2 3      1 2 3       1 2 3
4 2 5  =>   4 2 5  =>  4 _ 5  =>  _ 4 5  =>   4 5 6
7 8 6       7 8 6      7 8 6      7 8 6       7 8 _
(initial)                                     (goal)
```

In our current project, we have computed A * algorithm which solves 8 puzzle problem.

A* search strategy :

A* search strategy is a Recursive Informed search strategy, Current node sequentially determine a next node from either Misplaced Tiles strategy or Manhattan Search. Here, a priority queue is maintained for each node generated. Node with minimum heuristic value among all the neighboring nodes will be dequeued and gets expanded further. Until the goal state is found, this algorithm keeps finding lowest cost path to the goal node itself.

PROGRAM STRUCTURE

2.1 Functions and Procedures

Our code implements A* search using Misplaced Tiles and Manhattan Distance approaches. It takes Initial and Goal states as an input from the user.

Functions/Procedures in the code :

try_move_tile_down()
:
try_move_tile_up()

Try_move_tile_right()

try_move_tile_left()

It checks the validity of moving any a tile down/up/right/left. It returns a new state if it is valid.

get_heuristic_cost(): It return the heuristic cost corresponding to Misplaced Tiles or Manhattan by using another functions heuristic_misplaced() and heuristic_manhattan() 3unctions.

Print_path() : It starts printing all the sequential path from the root node to goal node in stepwise manner as soon as the goal state is found. It prints states and depth of each node on way to goal state.

A_star_search() : It's a main feature of our code. It maintains a priority queue. Nodes get added to this queue, and nodes which have minimum heuristic value among all the neighbour states will be dequeued and added to visited nodes. If the node finally matches the goal state, It will tell the printer to print the path and depth.

2.2 Global and Local variables

Global Variables in the program:

Np:numpy object

Initial_State: start state in matrix

Goal_State : goal state in matrix

Local variables in a program:

empty_tile_index- Index of '0'

state – current state

immed_parent – *parent node*

movement – *move up, left, down, right*

depth : *depth of the node in the tree*

step_cost :*the cost to take the step*

path_cost *: overall g(n), the cost to reach the current node*

heuristic_cost: *h(n), cost to reach goal state from the current node*

new_state- new generated state

i,j – loop variables for manhatten distance calculation

a,b – position pool for calculating Manhattan distance

sum_manhatten- manhatten distance

queue_num_nodes_popped – counter to calculate number of nodes dequeued

visited – stores all the visited nodes

queue – Priority queue storing nodes and returning best heuristic value

Program Code:

```
import numpy as np
import time

class Apply_A_Star():
    def __init__(self, state, immed_parent, movement, depth, step_cost, path_cost, heuristic_cost):
        self.state = state
        self.immed_parent = immed_parent  # parent node
        self.movement = movement  # tiles can move up, left, down, right
        self.depth = depth  # depth of the node
        self.step_cost = step_cost  # g(n), step-cost, In our case : 1
        self.path_cost = path_cost  # Overall g(n)
        self.heuristic_cost = heuristic_cost  # h(n), cost to reach goal state from the current node

        ''' For deriving child nodes'''
        self.move_tile_up = None
        self.move_tile_left = None
        self.move_tile_down = None
        self.move_tile_right = None

    # check if moving down is valid
    def try_move_tile_down(self):
        # index of the empty tile
        empty_tile_index = [i[0] for i in np.where(self.state == 0)]
        if empty_tile_index[0] == 0:
            return False
        else:
            value_above = self.state[empty_tile_index[0] - 1, empty_tile_index[1]]  # value of the
upper tile
            new_state = self.state.copy()
            new_state[empty_tile_index[0], empty_tile_index[1]] = value_above
            new_state[empty_tile_index[0] - 1, empty_tile_index[1]] = 0
            return new_state, value_above

    # check if moving right is valid
    def try_move_tile_right(self):
        empty_tile_index = [i[0] for i in np.where(self.state == 0)]
        if empty_tile_index[1] == 0:
            return False
        else:
            value_left = self.state[empty_tile_index[0], empty_tile_index[1] - 1]  # value of the left
tile
            new_state = self.state.copy()
            new_state[empty_tile_index[0], empty_tile_index[1]] = value_left
            new_state[empty_tile_index[0], empty_tile_index[1] - 1] = 0
```

```
            return new_state, value_left

    # check if moving up is valid
    def try_move_tile_up(self):
        empty_tile_index = [i[0] for i in np.where(self.state == 0)]
        if empty_tile_index[0] == 2:
            return False
        else:
            value_below = self.state[empty_tile_index[0] + 1, empty_tile_index[1]]  # value of the
lower tile
            new_state = self.state.copy()
            new_state[empty_tile_index[0], empty_tile_index[1]] = value_below
            new_state[empty_tile_index[0] + 1, empty_tile_index[1]] = 0
            return new_state, value_below

    # check if moving left is valid
    def try_move_tile_left(self):
        empty_tile_index = [i[0] for i in np.where(self.state == 0)]
        if empty_tile_index[1] == 2:
            return False
        else:
            value_right = self.state[empty_tile_index[0], empty_tile_index[1] + 1]  # value of the
right tile
            new_state = self.state.copy()
            new_state[empty_tile_index[0], empty_tile_index[1]] = value_right
            new_state[empty_tile_index[0], empty_tile_index[1] + 1] = 0
            return new_state, value_right

    # return user specified heuristic cost
    def get_heuristic_cost(self, new_state, goal_state, heuristic_function, path_cost, depth):
        if heuristic_function == 'num_misplaced':
            return self.heuristic_misplaced(new_state, goal_state)
        elif heuristic_function == 'manhattan':
            return self.heuristic_manhattan(new_state, goal_state) - path_cost + depth

    # return heuristic cost: number of misplaced tiles
    def heuristic_misplaced(self, new_state, goal_state):
        cost = np.sum(new_state != goal_state) - 1  # minus 1 to exclude the empty tile
        if cost > 0:
            return cost
        else:
            return 0  # when all tiles matches

    # return heuristic cost: sum of Manhattan distance to reach the goal state
    def heuristic_manhattan(self, new_state, goal_state):
        current = new_state
```

```
        # digit and coordinates they are supposed to be
        goal_position_dic = {1: (0, 0), 2: (0, 1), 3: (0, 2), 8: (1, 0), 0: (1, 1), 4: (1, 2), 7: (2, 0), 6: (2,
1),
                             5: (2, 2)}
        sum_manhattan = 0
        for i in range(3):
            for j in range(3):
                if current[i, j] != 0:
                    sum_manhattan += sum(abs(a - b) for a, b in zip((i, j), goal_position_dic[current[i,
j]]))
        return sum_manhattan

    # once the goal node is met, rewind back to the root node and print out the path
    def print_path(self):
        print "\nGoal has been found!!!!"
        # Establish stacks to generate output hierarchy
        state_Anc = [self.state]
        movement_Anc = [self.movement]
        depth_Anc = [self.depth]
        # add node information as rewinding back up to root
        while self.immed_parent:
            self = self.immed_parent
            state_Anc.append(self.state)
            movement_Anc.append(self.movement)
            depth_Anc.append(self.depth)
        # print the path
        step_counter = 0
        while state_Anc:
            print 'step', step_counter
            print state_Anc.pop()
            print 'movement=', movement_Anc.pop(), ', depth=', str(depth_Anc.pop()), '\n'
            step_counter += 1

    # search based on path cost + heuristic cost
    def a_star_search(self, goal_state, heuristic_function):
        start = time.time()

        queue = [
            (self, 0)]  # queue of (found but unvisited nodes)
        queue_num_nodes_popped = 0  # number of nodes popped off the queue
        queue_exp = 1  # checking expanding nodes

        depth_queue = [(0, 0)]  # queue of node depth, (depth, path_cost+heuristic cost)
        path_cost_queue = [(0, 0)]  # queue for path cost, (path_cost, path_cost+heuristic cost)
        visited = set([])  # record visited states
```

```
        while queue:
            # sort queue in ascending order
            queue = sorted(queue, key=lambda x: x[1])
            depth_queue = sorted(depth_queue, key=lambda x: x[1])
            path_cost_queue = sorted(path_cost_queue, key=lambda x: x[1])

            # update maximum length of the queue
            if len(queue) > queue_exp:
                queue_exp = len(queue)

            current_node = queue.pop(0)[0]  # select and remove the first node in the queue
            queue_num_nodes_popped += 1
            current_depth = depth_queue.pop(0)[0]  # select and remove the depth for current node
            current_path_cost = path_cost_queue.pop(0)[0]  # # select and remove the path cost for
reaching current node
            visited.add(
                tuple(current_node.state.reshape(1, 9)[0]))  # avoid repeated state, which is
represented as a tuple
            #print visited
            # when the goal state is found, rewind back to the root node and print out the path
            if np.array_equal(current_node.state, goal_state):
                current_node.print_path()
                print 'Generated', str(queue_num_nodes_popped)
                print 'Expanded:', str(queue_exp)
                print 'Time spent: %0.2fs' % (time.time() - start)
                return True

            else:
                # check if moving upper tile down is a valid move
                if current_node.try_move_tile_down():
                    new_state, value_above = current_node.try_move_tile_down()
                    # check if  already visited
                    if tuple(new_state.reshape(1, 9)[0]) not in visited:
                        path_cost = current_path_cost + value_above
                        depth = current_depth + 1
                        # get heuristic cost
                        h_cost = self.get_heuristic_cost(new_state, goal_state, heuristic_function,
path_cost, depth)
                        # Establish a new child node
                        total_cost = path_cost + h_cost
                        current_node.move_tile_down = Apply_A_Star(state=new_state,
immed_parent=current_node, movement='down', depth=depth,step_cost=1,
path_cost=path_cost, heuristic_cost=h_cost)
                        queue.append((current_node.move_tile_down, total_cost))
                        depth_queue.append((depth, total_cost))
                        path_cost_queue.append((path_cost, total_cost))
```

```
        # check if moving left tile to the right is a valid move
        if current_node.try_move_tile_right():
            new_state, value_left = current_node.try_move_tile_right()
            # check if  already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                path_cost = current_path_cost + value_left
                depth = current_depth + 1
                # get heuristic cost
                h_cost = self.get_heuristic_cost(new_state, goal_state, heuristic_function,
path_cost, depth)
                # Establish a new child node
                total_cost = path_cost + h_cost
                current_node.move_tile_right = Apply_A_Star(state=new_state,
immed_parent=current_node, movement='right',
                                            depth=depth, \
                                            step_cost=1, path_cost=path_cost, heuristic_cost=h_cost)
                queue.append((current_node.move_tile_right, total_cost))
                depth_queue.append((depth, total_cost))
                path_cost_queue.append((path_cost, total_cost))

        # check if moving lower tile up is a valid move
        if current_node.try_move_tile_up():
            new_state, value_below = current_node.try_move_tile_up()
            # check if  already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                path_cost = current_path_cost + value_below
                depth = current_depth + 1
                # get heuristic cost
                h_cost = self.get_heuristic_cost(new_state, goal_state, heuristic_function,
path_cost, depth)
                # Establish a new child node
                total_cost = path_cost + h_cost
                current_node.move_tile_up = Apply_A_Star(state=new_state,
immed_parent=current_node, movement='up', depth=depth,step_cost=1, path_cost=path_cost,
heuristic_cost=h_cost)
                queue.append((current_node.move_tile_up, total_cost))
                depth_queue.append((depth, total_cost))
                path_cost_queue.append((path_cost, total_cost))

        # check if moving right tile to the left is a valid move
        if current_node.try_move_tile_left():
            new_state, value_right = current_node.try_move_tile_left()
            # check if  already visited
            if tuple(new_state.reshape(1, 9)[0]) not in visited:
                path_cost = current_path_cost + value_right
```

```
                    depth = current_depth + 1
                    # get heuristic cost
                    h_cost = self.get_heuristic_cost(new_state, goal_state, heuristic_function,
path_cost, depth)
                    # Establish a new child node
                    total_cost = path_cost + h_cost
                    current_node.move_tile_left = Apply_A_Star(state=new_state,
immed_parent=current_node, movement='left', depth=depth, step_cost=1, path_cost=path_cost,
heuristic_cost=h_cost)
                    queue.append((current_node.move_tile_left, total_cost))
                    depth_queue.append((depth, total_cost))
                    path_cost_queue.append((path_cost, total_cost))

initial_ar = [int(x) for x in raw_input("Enter Start State, numbers split by space(ex- 1 2 3 4 5 7 6
8 0):").split()]
#initial_state=np.array([1,2,3,7,4,5,6,8,0]).reshape(3,3)
initial_state=np.array([initial_ar]).reshape(3,3)
print initial_state
goal_ar = [int(x) for x in raw_input("Enter goal State, numbers split by space(ex- 1 2 3 4 5 6 7 8
0):").split()]
#goal_state = np.array([1,2,3,8,6,4,7,5,0]).reshape(3,3)
goal_state=np.array([goal_ar]).reshape(3,3)
print goal_state
print "Misplaced Tiles :"
print 'Wait. Calculating........................\n'
root =
Apply_A_Star(state=initial_state,immed_parent=None,movement=None,depth=0,step_cost=0,
path_cost=0,heuristic_cost=0)
root.a_star_search(goal_state,heuristic_function = 'num_misplaced')
print "\nManhattan Heuristic:"
root.a_star_search(goal_state,heuristic_function = 'manhattan')
```

Sample Outputs :

1)

```
Enter Start State, numbers split by space(ex- 1 2 3 4 5 7 6 8 0):2 8 1 3 4 6 7 5 0
[[2 8 1]
 [3 4 6]
 [7 5 0]]
Enter goal State, numbers split by space(ex- 1 2 3 4 5 6 7 8 0):3 2 1 8 0 4 7 5 6
[[3 2 1]
 [8 0 4]
 [7 5 6]]
Misplaced Tiles :
Wait. Calculating.......................
Goal has been found!!!!
step 0
[[2 8 1]
 [3 4 6]
 [7 5 0]]
movement= None , depth= 0

step 1
[[2 8 1]
 [3 4 0]
 [7 5 6]]
movement= down , depth= 1

step 2
[[2 8 1]
 [3 0 4]
 [7 5 6]]
movement= right , depth= 2

step 3
[[2 0 1]
 [3 8 4]
 [7 5 6]]
movement= down , depth= 3

step 4
[[0 2 1]
 [3 8 4]
 [7 5 6]]
movement= right , depth= 4

step 5
[[3 2 1]
 [0 8 4]
 [7 5 6]]
```

movement= up , depth= 5

step 6
[[3 2 1]
 [8 0 4]
 [7 5 6]]
movement= left , depth= 6

Generated 55
Expanded: 42
Time spent: 0.01s

Manhattan Heuristic:

Goal has been found!!!!
step 0
[[2 8 1]
 [3 4 6]
 [7 5 0]]
movement= None , depth= 0

step 1
[[2 8 1]
 [3 4 0]
 [7 5 6]]
movement= down , depth= 1

step 2
[[2 8 1]
 [3 0 4]
 [7 5 6]]
movement= right , depth= 2

step 3
[[2 0 1]
 [3 8 4]
 [7 5 6]]
movement= down , depth= 3

step 4
[[0 2 1]
 [3 8 4]
 [7 5 6]]
movement= right , depth= 4

step 5
[[3 2 1]
 [0 8 4]
 [7 5 6]]
movement= up , depth= 5

step 6
[[3 2 1]
 [8 0 4]
 [7 5 6]]
movement= left , depth= 6

Generated 10
Expanded: 9
Time spent: 0.00s

Sample 2:

Enter Start State, numbers split by space(ex- 1 2 3 4 5 7 6 8 0):1 2 3 7 4 5 6 8 0
[[1 2 3]
 [7 4 5]
 [6 8 0]]
Enter goal State, numbers split by space(ex- 1 2 3 4 5 6 7 8 0):1 2 3 8 6 4 7 5 0
[[1 2 3]
 [8 6 4]
 [7 5 0]]
Misplaced Tiles :
Wait. Calculating.......................

Goal has been found!!!!
step 0
[[1 2 3]
 [7 4 5]
 [6 8 0]]
movement= None , depth= 0

step 1
[[1 2 3]
 [7 4 0]
 [6 8 5]]
movement= down , depth= 1

step 2
[[1 2 3]
 [7 0 4]
 [6 8 5]]
movement= right , depth= 2

step 3
[[1 2 3]
 [7 8 4]

```
 [6 0 5]]
movement= up , depth= 3

step 4
[[1 2 3]
 [7 8 4]
 [0 6 5]]
movement= right , depth= 4

step 5
[[1 2 3]
 [0 8 4]
 [7 6 5]]
movement= down , depth= 5

step 6
[[1 2 3]
 [8 0 4]
 [7 6 5]]
movement= left , depth= 6

step 7
[[1 2 3]
 [8 6 4]
 [7 0 5]]
movement= up , depth= 7

step 8
[[1 2 3]
 [8 6 4]
 [7 5 0]]
movement= left , depth= 8

Generated 489
Expanded: 319
Time spent: 0.12s

Manhattan Heuristic:

Goal has been found!!!!
step 0
[[1 2 3]
 [7 4 5]
 [6 8 0]]
movement= None , depth= 0

step 1
[[1 2 3]
 [7 4 0]
 [6 8 5]]
movement= down , depth= 1
```

```
step 2
[[1 2 3]
 [7 0 4]
 [6 8 5]]
movement= right , depth= 2

step 3
[[1 2 3]
 [7 8 4]
 [6 0 5]]
movement= up , depth= 3

step 4
[[1 2 3]
 [7 8 4]
 [0 6 5]]
movement= right , depth= 4

step 5
[[1 2 3]
 [0 8 4]
 [7 6 5]]
movement= down , depth= 5

step 6
[[1 2 3]
 [8 0 4]
 [7 6 5]]
movement= left , depth= 6

step 7
[[1 2 3]
 [8 6 4]
 [7 0 5]]
movement= up , depth= 7

step 8
[[1 2 3]
 [8 6 4]
 [7 5 0]]
movement= left , depth= 8

Generated 30
Expanded: 21
Time spent: 0.01s
```

Sample 3:

```
Enter Start State, numbers split by space(ex- 1 2 3 4 5 7 6 8 0):2 8 3 4 1 6 5 7 0
[[2 8 3]
 [4 1 6]
 [5 7 0]]
Enter goal State, numbers split by space(ex- 1 2 3 4 5 6 7 8 0):1 2 3 4 5 6 7 8 0
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Misplaced Tiles :
Wait. Calculating.......................


Goal has been found!!!!
step 0
[[2 8 3]
 [4 1 6]
 [5 7 0]]
movement= None , depth= 0

step 1
[[2 8 3]
 [4 1 0]
 [5 7 6]]
movement= down , depth= 1

step 2
[[2 8 3]
 [4 0 1]
 [5 7 6]]
movement= right , depth= 2

step 3
[[2 0 3]
 [4 8 1]
 [5 7 6]]
movement= down , depth= 3

step 4
[[0 2 3]
 [4 8 1]
 [5 7 6]]
movement= right , depth= 4
```

```
step 5
[[4 2 3]
 [0 8 1]
 [5 7 6]]
movement= up , depth= 5

step 6
[[4 2 3]
 [5 8 1]
 [0 7 6]]
movement= up , depth= 6

step 7
[[4 2 3]
 [5 8 1]
 [7 0 6]]
movement= left , depth= 7

step 8
[[4 2 3]
 [5 0 1]
 [7 8 6]]
movement= down , depth= 8

step 9
[[4 2 3]
 [5 1 0]
 [7 8 6]]
movement= left , depth= 9

step 10
[[4 2 0]
 [5 1 3]
 [7 8 6]]
movement= down , depth= 10

step 11
[[4 0 2]
 [5 1 3]
 [7 8 6]]
movement= right , depth= 11

step 12
[[4 1 2]
 [5 0 3]
```

```
 [7 8 6]]
movement= up , depth= 12

step 13
[[4 1 2]
 [0 5 3]
 [7 8 6]]
movement= right , depth= 13

step 14
[[0 1 2]
 [4 5 3]
 [7 8 6]]
movement= down , depth= 14

step 15
[[1 0 2]
 [4 5 3]
 [7 8 6]]
movement= left , depth= 15

step 16
[[1 2 0]
 [4 5 3]
 [7 8 6]]
movement= left , depth= 16

step 17
[[1 2 3]
 [4 5 0]
 [7 8 6]]
movement= up , depth= 17

step 18
[[1 2 3]
 [4 5 6]
 [7 8 0]]
movement= up , depth= 18

Generated 4675
Expanded: 2829
Time spent: 5.02s
```

Manhattan Heuristic:

Goal has been found!!!!

```
step 0
[[2 8 3]
 [4 1 6]
 [5 7 0]]
movement= None , depth= 0

step 1
[[2 8 3]
 [4 1 6]
 [5 0 7]]
movement= right , depth= 1

step 2
[[2 8 3]
 [4 1 6]
 [0 5 7]]
movement= right , depth= 2

step 3
[[2 8 3]
 [0 1 6]
 [4 5 7]]
movement= down , depth= 3

step 4
[[2 8 3]
 [1 0 6]
 [4 5 7]]
movement= left , depth= 4

step 5
[[2 8 3]
 [1 5 6]
 [4 0 7]]
movement= up , depth= 5

step 6
[[2 8 3]
 [1 5 6]
 [4 7 0]]
movement= left , depth= 6

step 7
[[2 8 3]
 [1 5 0]
 [4 7 6]]
```

movement= down , depth= 7

step 8
[[2 8 3]
 [1 0 5]
 [4 7 6]]
movement= right , depth= 8

step 9
[[2 0 3]
 [1 8 5]
 [4 7 6]]
movement= down , depth= 9

step 10
[[0 2 3]
 [1 8 5]
 [4 7 6]]
movement= right , depth= 10

step 11
[[1 2 3]
 [0 8 5]
 [4 7 6]]
movement= up , depth= 11

step 12
[[1 2 3]
 [4 8 5]
 [0 7 6]]
movement= up , depth= 12

step 13
[[1 2 3]
 [4 8 5]
 [7 0 6]]
movement= left , depth= 13

step 14
[[1 2 3]
 [4 0 5]
 [7 8 6]]
movement= down , depth= 14

step 15
[[1 2 3]

[4 5 0]
[7 8 6]]
movement= left , depth= 15

step 16
[[1 2 3]
[4 5 6]
[7 8 0]]
movement= up , depth= 16

Generated 1164
Expanded: 745
Time spent: 0.51s

3.1 References:

1) https://www.cs.princeton.edu/courses/archive/spr08/cos226/assignments/8puzzle.html
2) www.stackoverflow.com
3) www.geeksforgeeks.com