8 – Puzzle using A* Algorithm

PROJECT DOCUMENTATION REPORT

PROJECT 1

ITCS 6150 - Intelligent Systems

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO

Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Anoosh Guddehithlu Prathap Kumar	**801200789**
Sharat Sindoor	**801203539**

I. Problem Formulation

1.1 Introduction

8-puzzle

It is a 3x3 matrix with 8 square blocks containing 1 to 8 and a blank square. The main idea of 8 puzzle is to reorder these squares into a numerical order of 1 to 8 and last square as blank. Each of the squares adjacent to blank block can move up, down, left or right depending on the edges of the matrix.

```
   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
```

A* Algorithm

The project is about solving an 8-puzzle using A* search algorithm. A* algorithm is a recursive algorithm that calls itself until a solution is found. In this algorithm we consider two heuristic functions, misplaced tiles heuristic and Manhattan distance heuristic. Misplaced tiles heuristic calculates the misplaced number of tiles of the current state as compared to the goal state. Manhattan distance heuristic function measures the least steps needed for each of the tiles in the 8-puzzle initial or current state to arrive to the goal state position. In this project we have implemented the A* algorithm using both the heuristics.

We are calculating value of g(n) which is a measure of step cost for each move made from current state to next state which is initially set to zero. For each of the heuristic we have implemented f(n)= g(n)+h(n), where g(n) is the step cost and h(n) is the heuristic function used. Each of the states is explored using priority queues, which stores the position and f(n) value as key value pair. Then using merge sort technique priority queue is sorted and the next state to be explored is selected based on the least f(n) value.

In this project, we have used the optimality of A* by not allowing any state generation of previously traversed nodes. We have used two sets they are unexplored set and explored set to implement this function of A* algorithm. Unexplored set stores all the previously expanded states and explored set (priority queue in source code) which stores all of the non-duplicate states and sorts them according to f(n) value.

The reason for this project to be unique is the functionality it offers to user to enter Start state and Goal state, thus offering the generalized solution for any Start state against any Goal state. The program comes to an end if the A* algorithm has not found an optimal path within a runtime limit of 120 seconds.

II. Program structure

2.1 Global variables

Variable name	Variable type
puz	Integer list (user input for initial state from [0-8] split by spaces)
end	Integer list (user input for goal state from [0-8] split by spaces)
best_path	Integer list of state space after every optimal move
total_moves_count	Integer value that holds the total number of moves made to reach the solution
visited_count	Integer value containing total number of explored nodes

2.2 Functions and Procedures

Function/procedure	Description
calculate(puz, goal_state)	This function uses Manhattan heuristic to solve the 8-puzzle problem. In this function g_value, h_value and f_value is calculated and a priority queue is used to store the state f_value and position value as key value pair. The merge sort technique is used to find the least cost state and explore that state first if it hasn't already been explored. We call all() to check for repeats. We call the m_distance() to compute the Manhattan distance. We identify the blank tile and then check for the next available steps. Then generate the new states and check if this new state is the goal state or not.
fastst_solution(curr_state)	This function takes all the states as input and traverses through each of the states to find the fastest solution and returns the best optimal solution.

get_coordinates (puz)	This function assigns integer value as coordinates to any input that has been passed as the parameter and returns the single integer value of the coordinates.
m_distance (puz, goal)	This function calculates the Mahanttan distance of each element in the initial matrix by calculating the optimal moves each element should do to reach the position as mentioned in the goal state.
all()	This function uses set method in python to check if the current state is a duplicate of any other state that has been traversed in past or not and returns 1 if it is true and 0 if it is not.
missed_places ()	This function takes current and goal state as input and returns the count of number of misplaced tiles. If there are no misplaced tiles then zero is returned.
evaluate_missed()	This function solves the 8-puzzle problem using Misplaced Tiles heuristic. In this function g_value, h_value and f_value is calculated and a priority queue is used to store the node fn and position value as key value pair. We use the merge sort technique to find the least cost node and explore that node first if it hasn’t already been explored. We call all() to check for repeated states. We call the missed_places() to compute the misplaced tile cost . We identify the blank tile and then check for the next available steps and generate the new nodes and check if this new node is the goal state or not.

2.3 SOURCE CODE

# Import Packages and Library

```
import numpy

import time

from copy import deepcopy as dc
```

# This function takes input of start/initial state and calculates the fast/best path to goal/final state

```
def fastest_solution(curr_state):

    x=1

    y=0

    z = len(curr_state) - x

    fast_sol = numpy.array([], int).reshape(-x, 9)

    while z != -x:

        fast_sol = numpy.insert(fast_sol, y, curr_state[z]['puzzle'], y)

        z = (curr_state[z]['parent'])

    return fast_sol.reshape(-x, 3, 3)
```

# The all() function evaluates if intermediate state is unique or not and if it was explored before.

```
def all(array_checker):

    sets=[]

    for x in sets:

        for array_checker in x:

            return 1
        else:

            return 0


# This function is used to Calculate Manhattan distance between each tile of the goal/final state and start
state

def m_distance(puz, goal):

    u = abs(puz % 3 - goal % 3)

    v = abs(puz // 3 - goal // 3)



    mncost = v + u

    return sum(mncost[1:])


# The missed_places function checks the count of wrongly placed tiles in the intermediate state

def missed_places(puz,go_state):

    miss_cost = numpy.sum(puz != go_state) - 1

    if miss_cost < 0:

        return 0

    else:

        return miss_cost


# This function will indentify the coords of each goal or initial states values

def get_coordinates(puz):

    k=9

    p = numpy.array(range(k))

    for x, y in enumerate(puz):

        p[y] = x

    return p
```

```
# This function uses Manhattan heuristics to start evaluating the puzzle
def calculate(puz, goal_state):
    counts = numpy.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],
                dtype =  [('move',  str, 1),('pos', list),('head', int)])
    dt_state = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]
    l=-1
    m=0
# Initializing the parent_id, h_value and g_value where h_value is m_distance function
    g_cost = get_coordinates(goal_state)
    parent_id = l
    g_value = m
    h_value = m_distance(get_coordinates(puz), g_cost)
    curr_state = numpy.array([(puz, parent_id, g_value, h_value)], dt_state)

# Priority queues is used with pos as keys and f_value.
    dt_priority_q = [('pos', int),('fn', int)]
    priority_q = numpy.array( [(0, h_value)], dt_priority_q)

   while True:
        priority_q = numpy.sort(priority_q, kind='mergesort', order=['fn', 'pos'])
        pos, f_value = priority_q[0]
        priority_q = numpy.delete(priority_q, 0, 0)
        # The initial element is picked by using a merge sort to sort the priority queue
        puz, parent_id, g_value, h_value = curr_state[pos]
        puz = numpy.array(puz)
        # Identify the zero square in input

        zero = int(numpy.where(puz == 0)[0])

        g_value = g_value + 1

        o = 1

        starting_time = time.time()

        for s in counts:

            o = o + 1

            if zero not in s['pos']:

                # A copy of current state is used to Generate a new state

                explored_states = dc(puz)

                explored_states[zero], explored_states[zero + s['head']] = explored_states[zero + s['head']],
explored_states[zero]

                # The all fucntion is called to check is a state is opened or not

                if ~(numpy.all(list(curr_state['puzzle']) == explored_states, 1)).any():

                    ending_time = time.time()

                    if (( ending_time - starting_time ) > 2):

                        print(" The 8 puzzle is unsolvable ! \n")

                        exit

                    # Call the function manhattan to calcuate the costs involved

                    h_value = m_distance(get_coordinates(explored_states), g_cost)

                     # Generate and append the list with new state

                    q = numpy.array([(explored_states, pos, g_value, h_value)], dt_state)

                    curr_state = numpy.append(curr_state, q, 0)

                    # The sum of cost to reach front and the cost to reach to the goal state is f(n)

                    f_value = g_value + h_value


                    q = numpy.array([(len(curr_state) - 1, f_value)], dt_priority_q)
                priority_q = numpy.append(priority_q, q, 0)
                 # Checking if the state in explored_states are matching the goal states.
                if numpy.array_equal(explored_states, goal_state):
                    return curr_state, len(priority_q)



    return curr_state, len(priority_q)



# This function is the starting point of the heuristics misplaced tiles
def evaluvate_misplaced(puz, goal_state):
    counts = numpy.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],
                dtype =  [('move',  str, 1),('pos', list),('head', int)])

    dt_state = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

    g_cost = get_coordinates(goal_state)
# In this part we initialize the parent_id, h_value and g_value where h_value is missed_places()
    x=-1
    z=0
    parent_id = x
    g_value = z
    h_value = missed_places(get_coordinates(puz), g_cost)
    curr_state = numpy.array([(puz, parent_id, g_value, h_value)], dt_state)

# Priority queues are used with pos as keys and f_value
```

```
    dt_priority_q = [('pos', int),('fn', int)]

    priority_q = numpy.array([(0, h_value)], dt_priority_q)

    while 1:
        priority_q = numpy.sort(priority_q, kind='mergesort', order=['fn', 'pos'])
        pos, f_value = priority_q[0]
        # The first element for exploring is picked by using Merge sort to sort the priority queue
        priority_q = numpy.delete(priority_q, 0, 0)
        puz, parent_id, g_value, h_value = curr_state[pos]
        puz = numpy.array(puz)
         # Identify the zero square in input
        zero = int(numpy.where(puz == 0)[0])
        # Cost of g_value is increased by 1
        o = 1
        g_value = g_value + o
        starting_time = time.time()
        for s in counts:
            o = o + 1
            if zero not in s['pos']:
                 # Create a new_state as a copy of current state
                explored_states = dc(puz)
                explored_states[zero], explored_states[zero + s['head']] = explored_states[zero + s['head']],
explored_states[zero]
                # The explored_states function is called to check if the state has been previously opened or not.
                if ~(numpy.all(list(curr_state['puzzle']) == explored_states, 1)).any():
                    ending_time = time.time()
                    if (( ending_time - starting_time ) > 2):
                        print(" This puzzle cannot be solved using A*  Algorithm \n")
                        break
                    # This parts Calls the missed_places function to chech the costs
                    h_value = missed_places(get_coordinates(explored_states), g_cost)
                    # Generate and create state in the set of explored_states
                    q = numpy.array([(explored_states, pos, g_value, h_value)], dt_state)
                    curr_state = numpy.append(curr_state, q, 0)
                    f_value = g_value + h_value

                    q = numpy.array([(len(curr_state) - 1, f_value)], dt_priority_q)
                    priority_q = numpy.append(priority_q, q, 0)
                    # Check if the state in explored_states is matching the end state.
                    if numpy.array_equal(explored_states, goal_state):
                        return curr_state, len(priority_q)

    return curr_state, len(priority_q)

 # Accept User input for initial/start state
puz =[int(x) for x in input("Enter the Start/Initial State for example [1 2 3 4 5 6 7 8 0] :~  \n").split()]
# Accept User input of end state
end=[int(x) for x in input("Enter the Goal/End State for example [1 2 3 4 5 6 7 8 0] :~ \n ").split()]
choice = int(input("1. Misplaced tiles  \n2. Manhattan distance\n"))

if(choice == 1 ):
    curr_state, visited = evaluvate_misplaced(puz, end)
    best_path = fastest_solution(curr_state)

    print(str(best_path).replace('[', ' ').replace(']', ''))

    e=1

    total_moves_count = len(best_path) - e

    print('Steps taken to reach goal:',total_moves_count)

    visited_count = len(curr_state) - visited

    print('Nodes visited: ',visited_count, "\n")

    print('Nodes generated:', len(curr_state))


if(choice == 2 ):

    curr_state, visited = calculate(puz, end)

    best_path = fastest_solution(curr_state)

    print(str(best_path).replace('[', ' ').replace(']', ''))

    e=1

    total_moves_count = len(best_path) - e

    print('Steps taken to reach Goal state:',total_moves_count)

    visited_count = len(curr_state) - visited

    print('Nodes Visited: ',visited_count, "\n")

    print('Nodes Generated:', len(curr_state))
```

2.4 SAMPLE OUTPUT

Start State	**Misplaced Tiles**		**Manhattan Distance**		**Goal State**
	Nodes Visited	**Nodes Generated**	**Nodes Visited**	**Nodes Generated**	
1 2 3 7 4 5 6 8 0	23	44	9	19	1 2 3 8 6 4 7 5 0
2 8 1 3 4 6 7 5 0	7	15	6	13	3 2 1 8 0 4 7 5 6
7 2 4 5 0 6 8 3 1	3812	5886	282	452	1 2 3 4 5 6 7 8 0
3 5 1 4 2 6 7 8 0	645	1035	228	371	1 3 5 4 2 6 7 8 0
1 2 3 7 4 5 6 8 0	23	44	9	19	1 2 3 8 6 4 7 5 0
1 3 2 4 5 6 0 8 7	3945	6151	1042	1645	1 2 3 4 5 6 7 8 0

REFERENCES

- https://stackoverflow.com
- https://github.com
- https://www.cs.priceton.edu/courses/
- https://www.d.umn.edu/~jrichar4/8puz.html
- https://www.geeksforgeeks.org