



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:

Harshitha Keshavaraju Vijayalakshmi

801084151

Tannu D. Singh

801085297

Table of Contents

1. PROBLEM FORMULATION	2
1.1 Introduction.....	2
2. PROGRAM STRUCTURE	4
2.1 Global/ Local Variables	4
2.2 Functions/Procedures	4
2.3 Source code	5
2.4 Sample output	13
3. CONCLUSIONS	14
4. REFERENCES	14

I. Problem Formulation

1.1 Introduction

8-puzzle

It is 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*

The project is about solving 8-puzzle using A* search algorithm. A* 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 state space generation using both the heuristics.

We are also calculating $g(n)$ which is a measure of step cost for each move made from current state to next state, initially it is set to zero. For each of the heuristic we have implemented $f(n) = g(n) + h(n)$, where $g(n)$ is step cost and $h(n)$ is the heuristic function used. Each of the states is explored using priority queue, which stores the position and $f(n)$ value as key value pair. Then using merge sort technique priority queue is sorted and the next node 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 node generation of previously traversed nodes. We have used two sets they are closed set and open set to implement this functionality of A* algorithm. Closed set stores all the previously expanded nodes and open set (priority queue in source code) which stores all of the non-duplicate nodes 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 initial state and goal state, thus offering the generalized solution for any initial 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 two minutes.

II. Program structure

2.1 Global variables

Variable name	Variable type
puzzle	Integer list (user input for initial state from [0-8])
goal	Integer list (user input for goal state from [0-8])
bestpath	Integer list of state space after every optimal move
visit	Integer value containing total number of explored nodes
totalmoves	Integer value that holds the total number of moves made to reach the solution

2.2 Functions and Procedures

Function/procedure	Description
evaluate(puzzle, goal)	<p>This function solves the 8-puzzle problem using manhattan heuristic.</p> <p>In this function $g(n)$, $h(n)$ and $f(n)$ 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 repeats. We call the $manhattan()$ to compute the manhattan distance. We identify the blank tile and then check for the next available steps. Then generate the new nodes and check if this new node is the goal state or not.</p> <p>Here, $f(n) = g(n) + h(n)$ $h(n)$ calculates the sum of number of steps each tile in the current state needs to take to reach to goal state. $g(n)$ is the step cost of each step made.</p>
bestsolution(state)	This function takes all the states as input and traverses through each of the states to find the best solution and returns the best optimal solution.

<code>coordinates(<value>)</code>	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.
<code>manhattan(initial, goal)</code>	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.
<code>all()</code>	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.
<code>misplaced_tiles()</code>	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.
<code>evaluate_misplaced()</code>	<p>This function solves the 8-puzzle problem using Misplaced Tiles heuristic.</p> <p>In this function $g(n)$, $h(n)$ and $f(n)$ is calculated and a priority queue is used to store the node fn and position value as key value pair.</p> <p>We use the merge sort technique to find the least cost node and explore that node first if it hasn't already been explored.</p> <p>We call <code>all()</code> to check for repeats.</p> <p>We call the <code>misplaced_tiles()</code> to compute the misplaced tile cost .</p> <p>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.</p> <p>Here, $f(n) = g(n) + h(n)$</p> <p>$h(n)$ calculates the number of misplaced tiles in current state as compared to goal state.</p> <p>$g(n)$ is the path cost for each step made.</p>

2.3 SOURCE CODE

```
from copy import deepcopy
import numpy as np
import time
```

takes the input of current states and evaluvates the best path to goal state

def bestsolution(state):

```
bestsol = np.array([], int).reshape(-1, 9)

count = len(state) - 1

while count != -1:

    bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)

    count = (state[count]['parent'])

return bestsol.reshape(-1, 3, 3)
```

this function checks for the uniqueness of the iteration(it) state, weather it has been previously traversed or not.

def all(checkarray):

```
set=[]

for it in set:

    for checkarray in it:

        return 1

    else:

        return 0
```

calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state

def manhattan(puzzle, goal):

```
a = abs(puzzle // 3 - goal // 3)

b = abs(puzzle % 3 - goal % 3)

mhcost = a + b

return sum(mhcost[1:])
```

will calculate the number of misplaced tiles in the current state as compared to the goal state

def misplaced_tiles(puzzle,goal):

```
mscost = np.sum(puzzle != goal) - 1

return mscost if mscost > 0 else
```

will indentify the coordinates of each of goal or initial state values

def coordinates(puzzle):

```
pos = np.array(range(9))

for p, q in enumerate(puzzle):

    pos[q] = p

return pos
```

start of 8 puzzle evaluvation, using Manhattan heuristics

def evaluvate(puzzle, goal):

```
steps = np.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), ('position', list), ('head', int)])

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

initializing the parent, gn and hn, where hn is manhattan distance function call

```
costg = coordinates(goal)

parent = -1

gn = 0

hn = manhattan(coordinates(puzzle), costg)

state = np.array([(puzzle, parent, gn, hn)], dtstate)
```

We make use of priority queues with position as keys and fn as value.

```
dtpriority = [('position', int), ('fn', int)]

priority = np.array( [(0, hn)], dtpriority)

while 1:

    priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

    position, fn = priority[0]

    priority = np.delete(priority, 0, 0)
```


sort priority queue using merge sort, the first element is picked for exploring remove from queue what we are exploring

```
puzzle, parent, gn, hn = state[position]
```

```
puzzle = np.array(puzzle)
```

```
# Identify the blank square in input
```

```
blank = int(np.where(puzzle == 0)[0])
```

```
gn = gn + 1
```

```
c = 1
```

```
start_time = time.time()
```

```
for s in steps:
```

```
    c = c + 1
```

```
    if blank not in s['position']:
```

```
        # generate new state as copy of current
```

```
        openstates = deepcopy(puzzle)
```

```
        openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],  
openstates[blank]
```

```
        # The all function is called, if the node has been previously explored or not
```

```
        if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
```

```
            end_time = time.time()
```

```
            if (( end_time - start_time ) > 2):
```

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

```
                exit
```

```
        # calls the manhattan function to calculate the cost
```

```
        hn = manhattan(coordinates(openstates), costg)
```

```
        # generate and add new state in the list
```

```
        q = np.array([(openstates, position, gn, hn)], dtype=state)
```

```
        state = np.append(state, q, 0)
```

```
        # f(n) is the sum of cost to reach node and the cost to reach from the node to the goal  
state
```

```
        fn = gn + hn
```

```

q = np.array([(len(state) - 1, fn)], dtpriority)
priority = np.append(priority, q, 0)

# Checking if the node in openstates are matching the goal state.
if np.array_equal(openstates, goal):
    print(' The 8 puzzle is solvable ! \n')
    return state, len(priority)

return state, len(priority)

```

start of 8 puzzle evaluation, using Misplaced tiles heuristics

```

def evaluate_misplaced(puzzle, goal):

    steps = np.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), ('position', list), ('head', int)])

    dtstate = [('puzzle', list), ('parent', int), ('gn', int), ('hn', int)]
    costg = coordinates(goal)

    # initializing the parent, gn and hn, where hn is misplaced_tiles function call
    parent = -1
    gn = 0
    hn = misplaced_tiles(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

```

We make use of priority queues with position as keys and fn as value.

```

dtpriority = [('position', int), ('fn', int)]
priority = np.array([(0, hn)], dtpriority)

```

```

while 1:

    priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
    position, fn = priority[0]

    # sort priority queue using merge sort,the first element is picked for exploring.
    priority = np.delete(priority, 0, 0)

    puzzle, parent, gn, hn = state[position]

    puzzle = np.array(puzzle)

    # Identify the blank square in input
    blank = int(np.where(puzzle == 0)[0])

    # Increase cost g(n) by 1

    gn = gn + 1

    c = 1

    start_time = time.time()

    for s in steps:

        c = c + 1

        if blank not in s['position']:

            # generate new state as copy of current

            openstates = deepcopy(puzzle)

            openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],
openstates[blank]

            # The check function is called, if the node has been previously explored or not.

            if ~(np.all(list(state['puzzle']) == openstates, 1)).any():

                end_time = time.time()

                if (( end_time - start_time ) > 2):

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

                    break

            # calls the Misplaced_tiles function to calculate the cost

            hn = misplaced_tiles(coordinates(openstates), costg)

```

```

    # generate and add new state in the list

    q = np.array([(openstates, position, gn, hn)], dtype=dtstate)
    state = np.append(state, q, 0)

    # f(n) is the sum of cost to reach node and the cost to reach from the node to the goal
state
    fn = gn + hn

    q = np.array([(len(state) - 1, fn)], dtype=dtpriority)
    priority = np.append(priority, q, 0)

    # Checking if the node in openstates are matching the goal state.
    if np.array_equal(openstates, goal):
        print('The 8 puzzle is solvable \n')
        return state, len(priority)

    return state, len(priority)

# ----- Program start -----

# User input for initial state
puzzle = []
print("Input vals from 0-8 for start state ")
for i in range(0,9):
    x = int(input("enter vals :"))
    puzzle.append(x)

# User input of goal state
goal = []
print("Input vals from 0-8 for goal state ")
for i in range(0,9):
    x = int(input("Enter vals :"))
    goal.append(x)

```

```
n = int(input("1. Manhattan distance \n2. Misplaced tiles"))
```

```
if(n == 1 ):
```

```
    state, visited = evaluate(puzzle, goal)
```

```
    bestpath = bestsolution(state)
```

```
    print(str(bestpath).replace('[', ' ').replace(']', ''))
```

```
    totalmoves = len(bestpath) - 1
```

```
    print('Steps to reach goal:',totalmoves)
```

```
    visit = len(state) - visited
```

```
    print("Total nodes visited: ",visit, "\n")
```

```
    print("Total generated:", len(state))
```

```
if(n == 2):
```

```
    state, visited = evaluate_misplaced(puzzle, goal)
```

```
    bestpath = bestsolution(state)
```

```
    print(str(bestpath).replace('[', ' ').replace(']', ''))
```

```
    totalmoves = len(bestpath) - 1
```

```
    print('Steps to reach goal:',totalmoves)
```

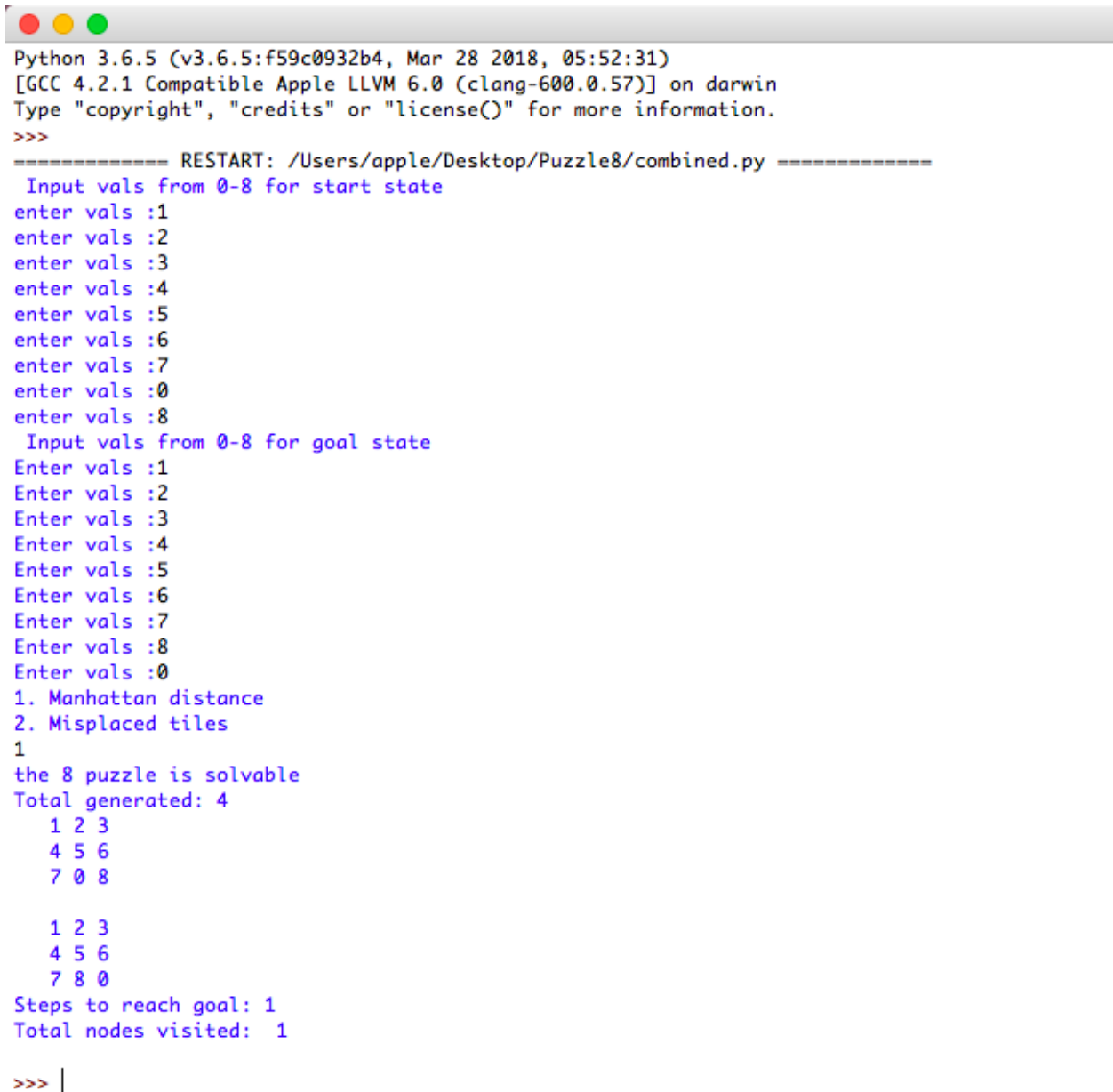
```
    visit = len(state) - visited
```

```
    print("Total nodes visited: ",visit, "\n")
```

```
    print("Total generated:", len(state))
```

2.3 SAMPLE OUTPUT

The output is shown in Figure 1 for input [1,2,3,4,5,6,7,0,8] for goal as [1,2,3,4,5,6,7,8,0]

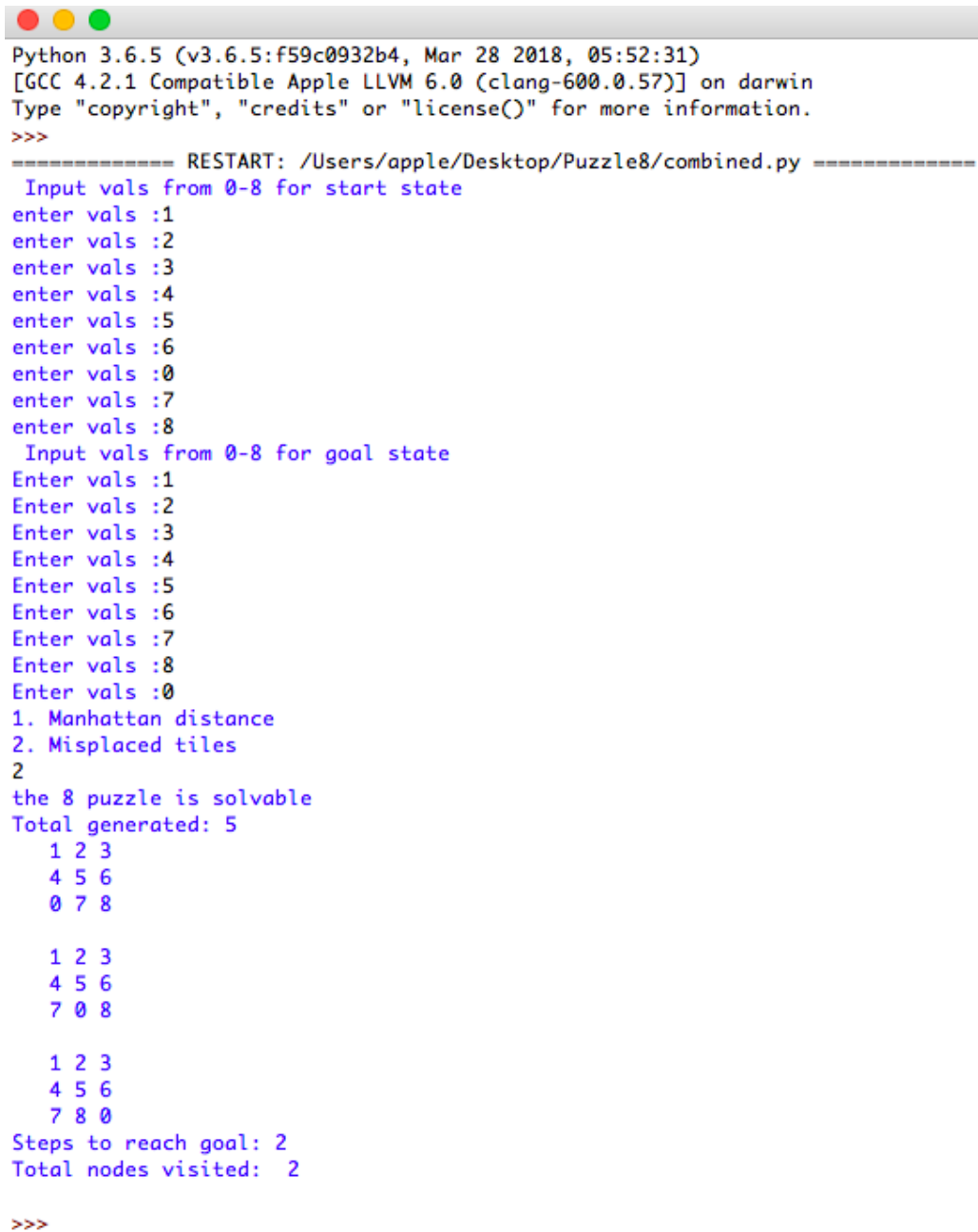


```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/apple/Desktop/Puzzle8/combined.py =====
  Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :3
enter vals :4
enter vals :5
enter vals :6
enter vals :7
enter vals :0
enter vals :8
  Input vals from 0-8 for goal state
Enter vals :1
Enter vals :2
Enter vals :3
Enter vals :4
Enter vals :5
Enter vals :6
Enter vals :7
Enter vals :8
Enter vals :0
1. Manhattan distance
2. Misplaced tiles
1
the 8 puzzle is solvable
Total generated: 4
  1 2 3
  4 5 6
  7 0 8

  1 2 3
  4 5 6
  7 8 0
Steps to reach goal: 1
Total nodes visited: 1
>>> |
```

Figure 1

The output is shown in Figure 2 for input [1,2,3,4,5,6,0,7,8] for goal as [1,2,3,4,5,6,7,8,0]



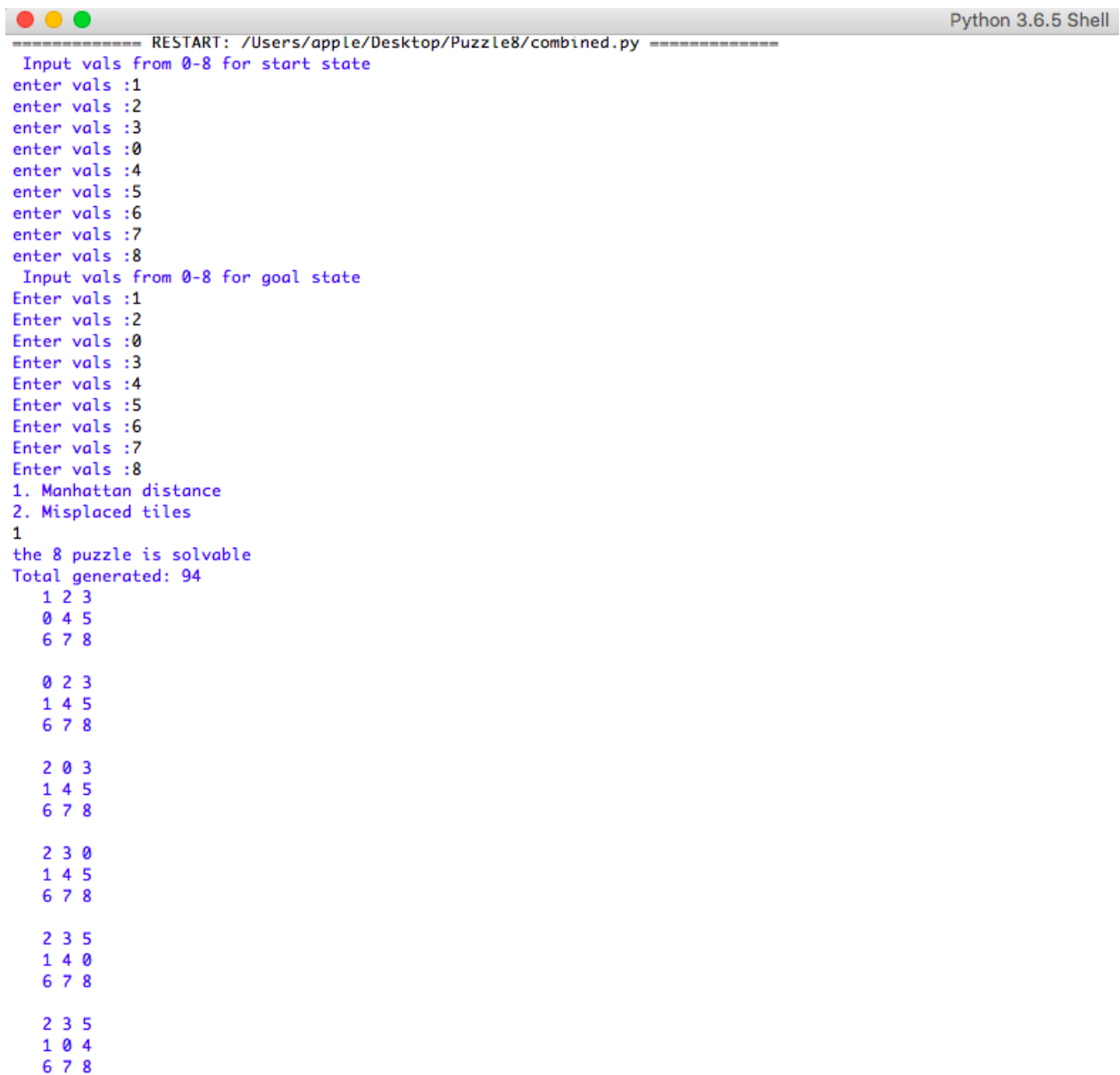
```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: /Users/apple/Desktop/Puzzle8/combined.py =====
  Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :3
enter vals :4
enter vals :5
enter vals :6
enter vals :0
enter vals :7
enter vals :8
  Input vals from 0-8 for goal state
Enter vals :1
Enter vals :2
Enter vals :3
Enter vals :4
Enter vals :5
Enter vals :6
Enter vals :7
Enter vals :8
Enter vals :0
1. Manhattan distance
2. Misplaced tiles
2
the 8 puzzle is solvable
Total generated: 5
  1 2 3
  4 5 6
  0 7 8

  1 2 3
  4 5 6
  7 0 8

  1 2 3
  4 5 6
  7 8 0
Steps to reach goal: 2
Total nodes visited: 2
>>>
```

Figure 2

The output is shown in Figure 2 for input [1,2,3,0,4,5,6,7,8] for goal as [1,2,0,3,4,5,6,7,8]



```
Python 3.6.5 Shell
===== RESTART: /Users/apple/Desktop/Puzzle8/combined.py =====
Input vals from 0-8 for start state
enter vals :1
enter vals :2
enter vals :3
enter vals :0
enter vals :4
enter vals :5
enter vals :6
enter vals :7
enter vals :8
Input vals from 0-8 for goal state
Enter vals :1
Enter vals :2
Enter vals :0
Enter vals :3
Enter vals :4
Enter vals :5
Enter vals :6
Enter vals :7
Enter vals :8
1. Manhattan distance
2. Misplaced tiles
1
the 8 puzzle is solvable
Total generated: 94
 1 2 3
 0 4 5
 6 7 8

 0 2 3
 1 4 5
 6 7 8

 2 0 3
 1 4 5
 6 7 8

 2 3 0
 1 4 5
 6 7 8

 2 3 5
 1 4 0
 6 7 8

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



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

2 0 5
1 3 4
6 7 8

0 2 5
1 3 4
6 7 8

1 2 5
0 3 4
6 7 8

1 2 5
3 0 4
6 7 8

1 2 5
3 4 0
6 7 8

1 2 0
3 4 5
6 7 8
Steps to reach goal: 11
Total nodes visited: 53
```

Figure 3

CONCLUSION

The 8 puzzle algorithm is solved using A* algorithm which uses heuristics such as manhattan distance and misplaced tile.

REFERENCES

1. <http://www.d.umn.edu/~jrichar4/8puz.html>
2. <https://www.cs.princeton.edu/courses/>
3. <http://stackoverflow.com>
4. <http://www.github.com>

