How to run the program

1. The folder should contain following files:

- *14_puzzle.py*
- *header.py*
- *Input1.txt*
- *Input2.txt*
- *Input3.txt*
- *Output1.txt*
- *Output2.txt*
- *Output3.txt*

2. Run the following command in terminal:

> python3 14_puzzle.py Input1.txt

An output file ***Output1.txt*** will be created in the same folder.

> python3 14_puzzle.py Input2.txt

An output file ***Output2.txt*** will be created in the same folder.

> python3 14_puzzle.py Input3.txt

An output file ***Output3.txt*** will be created in the same folder.

3. Source code and Outputs:

14_puzzle.py

```
from header import ATree
import os
import sys


# find the initial state and goal state
def read_input(file_path):
    initial_state = []
    goal_state = []
    fd = open(file_path, 'r')

    # find the initial state
    for i in range(4):
        line = fd.readline().strip('\n').split(' ')
        initial_state.append([])
        for j in line:
            initial_state[i].append(int(j))
```

```
    # skip empty line
    fd.readline()

    # find goal state
    for i in range(4):
        line = fd.readline().strip('\n').split(' ')
        goal_state.append([])
        for j in line:
            goal_state[i].append(int(j))

    return initial_state, goal_state


def find_goal_state(init_state, goal_state, output_file):
    atree = ATree(init_state, goal_state)
    atree.search_goal()
    print('Solving...')

    # create a new output file
    if os.path.exists(output_file):
        os.remove(output_file)
    fd = open(output_file, 'a')

    # write initial state
    for i in range(4):
        st = str(init_state[i][0]) + ' ' + str(init_state[i][1]) + ' ' +
str(init_state[i][2]) + ' ' + str(
            init_state[i][3]) + '\n'
        fd.write(st)
    fd.write('\n')

    # write goal state
    for i in range(4):
        st = str(goal_state[i][0]) + ' ' + str(goal_state[i][1]) + ' ' +
str(goal_state[i][2]) + ' ' + str(
            goal_state[i][3]) + '\n'
        fd.write(st)
    fd.write('\n')

    # write depth
    fd.write(str(atree.depth))
    fd.write('\n')

    # write number of nodes
    fd.write(str(atree.nodes_num))
    fd.write('\n')

    # write actions
    st = ''
    for action in atree.actions_to_goal:
        st += action + ' '
    st += '\n'
    fd.write(st)
```

```
    # write costs
    st = ''
    for cost in atree.cost_of_node:
        st += str(cost) + ' '
    st += '\n'
    fd.write(st)
    print('Finished.')


def main():
    # get input file
    input_file = sys.argv[1]

    # name output file
    output_file = 'Output'+input_file[5]+'.txt'

    initial_stat, goal_state = read_input(input_file)
    find_goal_state(initial_stat, goal_state, output_file)


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

header.py

```
import copy


# Priority queue, which can replace the same state stored in the queue
when `put`
class MyPriorityQueue:
    def __init__(self):
        self.queue = []

    def empty(self):
        return len(self.queue) == 0

    def put(self, priority_node):
        flag = 0  # check if there is an existing state in the queue
        for i in range(len(self.queue)):
            if self.queue[i][1].state == priority_node[1].state:
                flag = 1
                if self.queue[i][0] > priority_node[0]:
                    self.queue[i] = priority_node
                    break
        if flag == 0:
            self.queue.append(priority_node)
            return 1  # nodes number changed
        if flag == 1:
            return 0  # nodes number not changed
```

```
    def get(self):
        index = 0
        for i in range(len(self.queue)):
            if self.queue[i][0] < self.queue[index][0]:
                index = i
        item = self.queue[index]
        self.queue.remove(item)
        return item


class Node:
    def __init__(self, state, goal_state, parent_node, act_to_this,
blank1, blank2):
        self.state = state
# the state of the node
        self.goal_state = goal_state
# the goal state
        self.actions = ['L1', 'L2', 'R1', 'R2', 'U1', 'U2', 'D1', 'D2']
# actions that the node can take
        self.act_to_this = act_to_this
# which action the parent took to this state
        self.parent = parent_node
# the parent of the node
        self.path_cost = 0
# the g(n) of the node, same as level
        self.level = 0
# the level of the node
        self.h_cost = 0
# h(n), estimated cost
        self.total_cost = 0
# total cost, e.g. evaluation function f(n)
        self.blank1 = blank1
        self.blank2 = blank2
        self.changed_blank1 = self.blank1
        self.changed_blank2 = self.blank2

        self._remove_impossible_action()
        self._cal_h_cost()
        self._cal_path_cost_and_level()
        self._cal_total_cost()

    # remove impossible actions
    def _remove_impossible_action(self):
        if self.blank1[0] == 0:
            self.actions.remove('U1')
        if self.blank1[0] == 3:
            self.actions.remove('D1')
        if self.blank1[1] == 0:
            self.actions.remove('L1')
        if self.blank1[1] == 3:
            self.actions.remove('R1')
        if self.blank2[0] == 0:
            self.actions.remove('U2')
        if self.blank2[0] == 3:
```

```
            self.actions.remove('D2')
        if self.blank2[1] == 0:
            self.actions.remove('L2')
        if self.blank2[1] == 3:
            self.actions.remove('R2')

    # move the blank tile
    def take_action(self, action):
        state = copy.deepcopy(self.state)
        if action == 'U1':
            state[self.blank1[0] - 1][self.blank1[1]],
state[self.blank1[0]][self.blank1[1]] = state[self.blank1[0]]
[self.blank1[1]],state[elf.blank1[0] - 1][self.blank1[1]]
            self.changed_blank1 = [self.blank1[0] - 1, self.blank1[1]]
            self.changed_blank2 = self.blank2
        if action == 'D1':
            state[self.blank1[0] + 1][self.blank1[1]],
state[self.blank1[0]][self.blank1[1]] = state[self.blank1[0]]
[self.blank1[1]],state[self.blank1[0] + 1][self.blank1[1]]
            self.changed_blank1 = [self.blank1[0] + 1, self.blank1[1]]
            self.changed_blank2 = self.blank2
        if action == 'L1':
            state[self.blank1[0]][self.blank1[1] - 1],
state[self.blank1[0]][self.blank1[1]] = state[self.blank1[0]]
[self.blank1[1]],state[self.blank1[0]][self.blank1[1] - 1]
            self.changed_blank1 = [self.blank1[0], self.blank1[1] - 1]
            self.changed_blank2 = self.blank2
        if action == 'R1':
            state[self.blank1[0]][self.blank1[1] + 1],
state[self.blank1[0]][self.blank1[1]] = state[self.blank1[0]]
[self.blank1[1]],state[self.blank1[0]][self.blank1[1] + 1]
            self.changed_blank1 = [self.blank1[0], self.blank1[1] + 1]
            self.changed_blank2 = self.blank2

        if action == 'U2':
            state[self.blank2[0] - 1][self.blank2[1]],
state[self.blank2[0]][self.blank2[1]] = state[self.blank2[0]]
[self.blank2[1]],state[self.blank2[0] - 1][self.blank2[1]]
            self.changed_blank2 = [self.blank2[0] - 1, self.blank2[1]]
            self.changed_blank1 = self.blank1
        if action == 'D2':
            state[self.blank2[0] + 1][self.blank2[1]],
state[self.blank2[0]][self.blank2[1]] = state[self.blank2[0]]
[self.blank2[1]],state[self.blank2[0] + 1][self.blank2[1]]
            self.changed_blank2 = [self.blank2[0] + 1, self.blank2[1]]
            self.changed_blank1 = self.blank1
        if action == 'L2':
            state[self.blank2[0]][self.blank2[1] - 1],
state[self.blank2[0]][self.blank2[1]] = state[self.blank2[0]]
[self.blank2[1]],state[self.blank2[0]][self.blank2[1] - 1]
            self.changed_blank2 = [self.blank2[0], self.blank2[1] - 1]
            self.changed_blank1 = self.blank1
        if action == 'R2':
            state[self.blank2[0]][self.blank2[1] + 1],
```

```
state[self.blank2[0]][self.blank2[1]] = state[self.blank2[0]]
[self.blank2[1]],state[self.blank2[0]][self.blank2[1] + 1]
            self.changed_blank2 = [self.blank2[0], self.blank2[1] + 1]
            self.changed_blank1 = self.blank1

        return state

    # calculate the path cost and level, g(n)
    def _cal_path_cost_and_level(self):
        if self.parent is None:
            self.path_cost = 0
            self.level = 0
        else:
            self.path_cost = self.parent.path_cost + 1
            self.level = self.path_cost

    # calculate the estimated cost, h(n)
    def _cal_h_cost(self):
        for i in range(4):
            for j in range(4):
                if self.state[i][j] != 0:
                    for m in range(4):
                        for n in range(4):
                            if self.state[i][j] == self.goal_state[m][n]:
                                self.h_cost += abs(m - i) + abs(n - j)  #
manhattan distance

    # calculate the total cost, f(n)
    def _cal_total_cost(self):
        self.total_cost = self.h_cost + self.path_cost


class ATree:
    def __init__(self, init_state, goal_state):
        self.init_state = init_state
# initial state
        self.blank = []
# blanks in the state
        for i in range(4):
            for j in range(4):
                if self.init_state[i][j] == 0:
                    self.blank.append([i, j])
        self.blank1 = self.blank[0]
        self.blank2 = self.blank[1]
        self.goal_state = goal_state
# goal state
        self.depth = 0
# depth of the tree
        self.root = Node(init_state, goal_state, None, None, self.blank1,
self.blank2)  # create the root node
        self.frontier = MyPriorityQueue()
# the MyPriorityQueue of frontier nodes
        self.actions_to_goal = []
# store the actions to goal
```

```
        self.traversed = []
# store the explored states
        self.nodes_num = 1
# how many nodes in the tree
        self.cost_of_node = []
# the cost of every node picked

        self.frontier.put((self.root.total_cost, self.root))
        self.cost_of_node.append(self.root.total_cost)

    # create children for a given node
    def _create_child(self, node):
        children = []
        for action in node.actions:
            child_state = node.take_action(action)
# find state after an action

            # avoid the possibility that two blanks swap position
            if child_state not in self.traversed and child_state !=
node.state:
                child_node = Node(child_state, self.goal_state, node,
action, node.changed_blank1, node.changed_blank2)
                children.append(child_node)
        return children

    # find the goal state
    def _found(self, state_node):
        node = copy.deepcopy(state_node)
        self.depth = state_node.level
        while node is not None:
            self.actions_to_goal.insert(0, node.act_to_this)
# insert every step to goal node
            self.cost_of_node.insert(0, node.total_cost)
# insert every cost to goal node
            node = node.parent
        self.actions_to_goal.remove(None)
# remove the action of the root

    # cannot find the goal state
    def _not_found(self):
        raise Exception('Cannot find a possible solution.')

    # search for the goal state
    def search_goal(self):
        while not self.frontier.empty():  # search the frontier
            state_node = self.frontier.get()[1]  # get the lowest cost
node
            # self.actions_to_goal.append(state_node.act_to_this)  # store
action
            # self.cost_of_node.append(state_node.total_cost)  # store the
total cost
            if state_node.state == self.goal_state:  # find the goal
                self._found(state_node)
                return
```

```
            else:
                self.traversed.append(state_node.state)  # put state into
traversed
                children = self._create_child(state_node)  # expand the
node
                for child_node in children:
                    # add children to frontier, in MyPriorityQueue,
                    # it will automatically detect same state node and
decide which can stay in the queue
                    self.nodes_num +=
self.frontier.put((child_node.total_cost, child_node))
        self._not_found()  # if the loop ends then no goal found
```

Output1:

```
1 2 3 4
5 0 6 7
8 9 0 10
11 12 13 14

1 2 4 0
8 5 3 7
11 9 6 10
0 12 13 14

6
64
L1 U2 U2 R2 D1 D1
6 6 6 6 6 6 6
```

Output2:

```
1 5 3 13
8 0 6 4
0 10 7 9
11 14 2 12

1 3 4 13
8 5 7 9
10 0 6 12
11 14 0 2

12
439
R2 R1 R1 U2 U2 R2 D1 D1 L1 D2 D2 L2
10 12 12 12 12 12 12 12 12 12 12 12 10
```

Output3:

```
9 13 7 4
12 3 0 1
2 0 5 6
14 10 11 8

9 3 13 4
2 7 1 0
10 12 0 5
14 11 8 6

14
303
U1 L1 D1 L1 D1 D2 R1 R2 R2 U1 R1 R1 U2 L2
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
```