When running the program, it will ask you about which weight for the A* search you would like to choose. After entering the corresponding value, the output file will be generated. It contains the weight chosen, the nodes generated, the moves performed, and the f(n) value with the weighted calculation. To pass in a different input file, line 123 would have to be altered and to change the output file name, line 132 would have to be altered. Below are the sample outputs when passing in the input files and the source code.

```
What weight would you like to choose? (1.0, 1.2, 1.4)
```

Source Code:

```
#Tahmidur Rabb
#11 puzzle problem implementing weighted A* algorithm



import copy
from queue import PriorityQueue  # Using priority queue during iterations of puzzle
objects

print("What weight would you like to choose? (1.0, 1.2, 1.4) ") #Prompts for weighted
A star chosen
weight_choice = input() #Takes in input

class Astar(object):  #A star search implementing 11 puzzle

   def __init__(self, puzzle, depth): #Initial attributes and values
       self.puzzle = puzzle
       #Storing next move and current move in list and will append later during each
iteration
       self.next_move = []  #Recording every next move
       self.curr_moves= []  #Recording current move
       self.depth = depth  # current level g(n)
       self.astar_val = 0 # f(n)

   def __gt__(self, other):  #Checking for other efficient algorithms to compare with
self
       if self.astar_val > other.astar_val:  #If self is greater than output overload
           return True
```

```
        else:
            return False

    def copy(self, puzzle):  # Copies array over and provides instance
        temp = [] #Used for copying over values
        for item in puzzle: #Iterates through given puzzle
            arr = []
            for values in item:
                arr.append(values) #Appends values to array
            temp.append(arr) #Appends lst values of array to temp
        return temp

    def checks_move(self):  #Sets rules for 11 puzzle and checks next move
        temp_next_move = []  # a next move array local to this function
        if self.puzzle[0][0] != '0' and self.puzzle[1][0] != '0' and self.puzzle[2][0]
!= '0': #Performs left
            temp_next_move.append('L')

        if self.puzzle[0][3] != '0' and self.puzzle[1][3] != '0' and self.puzzle[2][3]
!= '0': #Performs right
            temp_next_move.append('R')

        if self.puzzle[0][0] != '0' and self.puzzle[0][1] != '0' and self.puzzle[0][2]
!= '0' and self.puzzle[0][3] != '0': #Performs up
            temp_next_move.append('U')

        if self.puzzle[2][0] != '0' and self.puzzle[2][1] != '0' and self.puzzle[2][2]
!= '0' and self.puzzle[2][3] != '0': #Performs down
            temp_next_move.append('D')

        self.next_move = temp_next_move  #self set to temp value

    def aStar(self, goal_board, weight): #Sets up initial and goal states
        mh_distance = 0  #For sum of manhattan distances
        for i in range(1, 12): #Iterates through 12 possible places
            start_board = check_blank(self.puzzle, str(i))
            end_board = check_blank(goal_board, str(i))
            mh_distance = mh_distance + abs(start_board[0] - end_board[0]) +
abs(start_board[1] - end_board[1]) #performs calculation for heuristic

        self.astar_val = self.depth + (float(weight) * mh_distance)  #f(n) = g(n) +
h(n)
```

```
def check_blank(puzzle, target):  #Function to check for blank state in puzzle
   # this tedious loop over and over again
   for i in range(3):
       for j in range(4):
           if puzzle[i][j] == target:
               return [i, j]


def copy2(puzzle):  #Copies and makes instance of values in puzzle
   temp = []
   for item in puzzle:
       arr = []
       for values in item:
           arr.append(values)
       temp.append(arr)
   return temp

def read_file(fname):  # open file function
   f = open(fname, "r")
   file_info = f.read().splitlines()
   inp_board = [line.split() for line in file_info] #Splits line and reads file
   return inp_board[0:3], inp_board[4:7] #First board is initial and reads initial
state, second board is the goal state


def next_move(curr_board, move):  # switching positions when blank tile moves left,
right, up, or down
   blank_tile = check_blank(curr_board, '0')  #Checks for 0 since 0 represents blank
tile in puzzle
   replace_board = copy2(curr_board) #Copies board and will modify it
   if move == 'U': #If move is up
       replace_board[blank_tile[0]][blank_tile[1]] = replace_board[blank_tile[0] -
1][blank_tile[1]] #Performs swap movement
       replace_board[blank_tile[0] - 1][blank_tile[1]] = '0'  #Sets it to blank tile

   elif move == 'D': #If move is down
       replace_board[blank_tile[0]][blank_tile[1]] = replace_board[blank_tile[0] +
1][blank_tile[1]]  #Performs swap movement
       replace_board[blank_tile[0] + 1][blank_tile[1]] = '0' #Sets it to blank tile
```

```
    elif move == 'L': #If move is left
        replace_board[blank_tile[0]][blank_tile[1]] =
replace_board[blank_tile[0]][blank_tile[1] - 1] #Performs swap movement
        replace_board[blank_tile[0]][blank_tile[1] - 1] = '0' #Sets it to blank tile

    elif move == 'R': #If move is right
        try: #Setting try and except in case list indices are out of range
            replace_board[blank_tile[0]][blank_tile[1]] =
replace_board[blank_tile[0]][blank_tile[1] + 1] #Performs swap movement
            replace_board[blank_tile[0]][blank_tile[1] + 1] = '0' #Sets it to blank
tile
        except:
            replace_board[blank_tile[0]][blank_tile[1]] =
replace_board[blank_tile[0]][blank_tile[1]]

    return replace_board

def new_board(curr_puzz, move, goal_board):  # create new board based on move, and
updates board, will run after every move
    updated_board = next_move(curr_puzz.puzzle, move) #Board that replaces current
board
    updated_puzzle = Astar(updated_board, curr_puzz.depth + 1) #New Astar objec board
with changed board layout and depth
    curr_moves = updated_puzzle.copy(curr_puzz.curr_moves) #Copying current and past
moves
    curr_moves.append(move)
    updated_puzzle.curr_moves = curr_moves
    updated_puzzle.aStar(goal_board,weight_choice) #Keeps goal board state
    updated_puzzle.checks_move()
    return updated_puzzle

def main():

    node_count = 1  #Keeps track of nodes generated
    input_board, goal_board = read_file("Input3.txt")



    input_puzzle = Astar(input_board, 0) #Creates board based on input board
information
    input_puzzle.checks_move() #Checks for blank and starts performing next move
    input_puzzle.aStar(goal_board, weight_choice) #Tracks heuristic
    repeated_move = [input_board] #Checks for repeated move
```

```
    priorityq = [input_puzzle] #Sets priority queue to puzzle board

    f = open("output3b.txt", "w") #Creates new file and will write
    i = 0
    j = 0
    f.write("Initial Board: ")
    f.write("\n")
    while i < 3:  #Designing 4 * 3 layout board  #Input Board
        f.write(str(input_board[i]))
        f.write("\n")
        i += 1

    f.write("\n")
    f.write("Goal Board: ")
    f.write("\n")

    while j < 3: #Goal Board
        f.write(str(goal_board[j])) #Writes to file
        f.write("\n")
        j +=1

    f.write("\n")

    f.write("W: " + str(weight_choice)) #Keeps weighted A* search for reference
    f.write("\n")

    heuristic = []  # holds f values until goal is reached
    while priorityq: #While priority queue has elem
        priorityq.sort(reverse=True)  #Sorts based on f(n)
        top_puzzle = priorityq.pop() #Instance of puzzle currently being looked at
        heuristic.append(top_puzzle.astar_val) # appends f(n) values
        if top_puzzle.puzzle == goal_board: #Checks if current state is equal to goal
state
            f.write("Depth: ")  #Depth
            f.write(str(top_puzzle.depth))
            f.write("\n")

            f.write("Nodes created: ") #Total nodes created
            f.write(str(node_count))
            f.write("\n")

            f.write("Moves: ") #Records moves that were taken to reach goal state
```

```
            for i in top_puzzle.curr_moves:
                f.write(str(i).strip("[]''"))  # List of moves to reach goal state
                f.write(' ')
            f.write("\n")

            break #Terminates loop since goal state has been found

        for moves in top_puzzle.next_move:
            new_puzzle = new_board(top_puzzle, moves, goal_board)
            if new_puzzle.puzzle in repeated_move: #Checks if in repeated move already
disregards it, then continues towards another move
                continue        # if we come across something we've seen already, go
back to start of for loop
            else:
                node_count += 1 #Only tracks unique moves
                repeated_move.append(new_puzzle.puzzle) #Appends unique move
                priorityq.append(new_puzzle)

    heuristic.pop(0)
    f.write("A* star values: ")  #Write astar files along with weighted
    for elem in heuristic:
        elem = round(elem, 1) #Rounding float to avoid any complications
        f.write(str(elem)) #Writing to file
        f.write(" ")
    #Checking if f(n) values exceed depth
    if len(heuristic) != top_puzzle.depth:
        heuristic.pop()


main()
```

Input_1 with W = 1.0:

```
Initial Board:
['2', '0', '6', '4']
['3', '10', '7', '9']
['11', '5', '8', '1']

Goal Board:
['2', '10', '6', '4']
['11', '3', '8', '9']
['0', '7', '5', '1']

W: 1.0
Depth: 7
Nodes created: 19
Moves: D R D L U L D
A* star values: 7.0 7.0 7.0 7.0 7.0 7.0 7.0
```

Input_1 with W = 1.2:

```
Initial Board:
['2', '0', '6', '4']
['3', '10', '7', '9']
['11', '5', '8', '1']

Goal Board:
['2', '10', '6', '4']
['11', '3', '8', '9']
['0', '7', '5', '1']

W: 1.2
Depth: 7
Nodes created: 19
Moves: D R D L U L D
A* star values: 8.2 8.0 7.8 7.6 7.4 7.2 7.0
```

Input_1 with W = 1.4:

```
Initial Board:
['2', '0', '6', '4']
['3', '10', '7', '9']
['11', '5', '8', '1']

Goal Board:
['2', '10', '6', '4']
['11', '3', '8', '9']
['0', '7', '5', '1']

W: 1.4
Depth: 7
Nodes created: 19
Moves: D R D L U L D
A* star values: 9.4 9.0 8.6 8.2 7.8 7.4 7.0
```

Input_2 with W = 1.0:

```
Initial Board:
['2', '0', '6', '4']
['3', '10', '7', '9']
['11', '5', '8', '1']

Goal Board:
['2', '7', '8', '4']
['10', '6', '9', '1']
['3', '11', '0', '5']

W: 1.0
Depth: 13
Nodes created: 29
Moves: R D D L L U R U R D R D L
A* star values: 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0
```

Input_2 with W = 1.2:

```
Initial Board:
['2', '0', '6', '4']
['3', '10', '7', '9']
['11', '5', '8', '1']

Goal Board:
['2', '7', '8', '4']
['10', '6', '9', '1']
['3', '11', '0', '5']

W: 1.2
Depth: 13
Nodes created: 29
Moves: R D D L L U R U R D R D L
A* star values: 15.4 15.2 15.0 14.8 14.6 14.4 14.2 14.0 13.8 13.6 13.4 13.2 13.0
```

Input_2 with W = 1.4:

```
Initial Board:
['2', '0', '6', '4']
['3', '10', '7', '9']
['11', '5', '8', '1']

Goal Board:
['2', '7', '8', '4']
['10', '6', '9', '1']
['3', '11', '0', '5']

W: 1.4
Depth: 13
Nodes created: 29
Moves: R D D L L U R U R D R D L
A* star values: 17.8 17.4 17.0 16.6 16.2 15.8 15.4 15.0 14.6 14.2 13.8 13.4 13.0
```

Input_3 with W = 1.0:

```
Initial Board:
['8', '7', '2', '4']
['10', '6', '9', '1']
['0', '11', '5', '3']

Goal Board:
['10', '6', '8', '4']
['9', '7', '0', '2']
['11', '5', '3', '1']

W: 1.0
Depth: 17
Nodes created: 86
Moves: R U R U L L D R D R R U L U L D R
A* star values: 13.0 13.0 13.0 13.0 15.0 15.0 15.0 15.0 15.0 15.0 15.0 15.0 15.0 15.0 15.0 15.0 17.0 17.0 17.0 17.0 17.0
```

Input_3 with W = 1.2:

```
Initial Board:
['8', '7', '2', '4']
['10', '6', '9', '1']
['0', '11', '5', '3']

Goal Board:
['10', '6', '8', '4']
['9', '7', '0', '2']
['11', '5', '3', '1']

W: 1.2
Depth: 17
Nodes created: 117
Moves: R U R U L L D R D R R U L U L D R
A* star values: 15.4 15.2 15.0 14.8 17.0 16.8 17.0 16.8 17.4 17.2 17.6 17.4 17.2 17.0 17.4 17.2 17.8 19.0 19.0 18.8 18.8 18.6 19.0
```

Input_3 with W = 1.4:

```
Initial Board:
['8', '7', '2', '4']
['10', '6', '9', '1']
['0', '11', '5', '3']

Goal Board:
['10', '6', '8', '4']
['9', '7', '0', '2']
['11', '5', '3', '1']

W: 1.4
Depth: 17
Nodes created: 117
Moves: R U R U L L D R D R R U L U L D R
A* star values: 17.8 17.4 17.0 16.6 19.0 18.6 19.0 18.6 19.8 19.4 20.2 19.8 19.4 19.0 19.8 19.4 20.6 21.0 21.0 20.6 20.6 20.2 21.0
```