CS441

Junyi Feng

1/29/2022

This program is implemented in Python by using the NumPy, math, and operator libraries. Blank is replaced by 0. I implemented three different heuristics for A* search and Best-first search, two described in the textbook(h1 = the number of misplaced tiles, and h2 = the sum of the Manhattan distance of the tiles from their goal positions)and one of my own(h3 = the sum of the Euclidean distance of the tiles from their goal positions).

To change the initial state and goal state, just modify the two lists named **initState** and **goalState**.The program assumes that all inputs are correct. Then, there would be a goal state index dictionary auto-generated by the NumPy array names **dic**.

```
initState = [[3, 7, 2], [6, 8, 0], [4, 5, 1]]
goalState = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

```
dic = np.zeros((9, 2))
k = 0
for i in range(3):  # goal
    for j in range(3):
        dic[k][0] = i
        dic[k][1] = j
        k += 1
```

Also, the program has two node classes named **aStarNode** and **gbfdNode**, one for Best-first search and one for A* search. A node contents the current state, parent's node and f(n) value(and **aStarNode** has g(n) value). To choose different heuristic functions, just common out the other two you don't need. The f(n) is automatically generated when initializing a node.

```
class aStarNode:
    def __init__(self, state, gn, parent):
        self.state = state
        self.gn = gn
        self.parent = parent
        # self.fn = h1(state) + gn
        # self.fn = h2(state) + gn
        self.fn = h3(state) + gn


class gbfsNode:
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent
        # self.fn = h1(state)
        # self.fn = h2(state)
        self.fn = h3(state)
```

After that, the program will begin to run from **main**. The first thing **main** does is check if the solution exists by calculating the inversions. Print “No solution” and exit the program if solutions do not exist. Then do A* search and Best-first search respectively in **aStar** and **gbfs**.

```
def aStar():
    list1 = [aStarNode(initState, 0, None)]
    visit = []
    count = 0
    while list1:
        list1.sort(key=operator.attrgetter('fn'))
        last = list1[0]  # index
        visit.append(last.state)
        # print(last.state)
        count += 1
        if count == 20000:
            return last, count
        if last.state == goalState:
            return last, count
        del list1[0]
        acts = getAvailActs(last.state)
        for i in acts:
            availAct = aStarNode(i, last.gn + 1, last)
            if availAct.state not in visit:
                # print(availAct.state)
                list1.append(availAct)
                visit.append(availAct.state)

    return None, count

def gbfs():
    list2 = [gbfsNode(initState, None)]
    visit = []
    count = 0
    while list2:
        list2.sort(key=operator.attrgetter('fn'))
        last = list2[0]
        visit.append(last.state)
        count += 1
        if last.state == goalState:
            return last, count
        if count == 20000:
            return last, count
        del list2[0]
        acts = getAvailActs(last.state)
        for i in acts:
            availAct = gbfsNode(i, last)
            if availAct.state not in visit:
                list2.append(availAct)
                visit.append(availAct.state)

    return None, count
```

Those two search methods are almost the same. They have two lists one called **list** and the other one called **visit**. First, append the initial node to the **list** set a counter to record the number of iterations. In the while loop, sort the list so let the smallest value of f(n)s' index be 0. Get that node from the list[0], delete it from the list and append its state to **visit**. Compare that node's state to the goal state, return the node and counter if they are the same. In addition, if the counter match 20000 which means exceeds the maximum iterations, return the node and counter as well. Last use the current node's state to get its next available movement by **getAvailActs**. For those available movements, append to **list** and **visit** if they do not exist in **visit**.

Last, the h1 is to calculate the number of misplaced tiles. It is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once. The h2 is to calculate the sum of the distances of the tiles from their goal positions as known as Manhattan distance. the distance will count is the sum of the horizontal and vertical distances because tiles cannot move along diagonals. It is admissible because all any move can do is move one tile one step closer to the goal. The h3 is to calculate the sum of the distances for the line segment between tiles and their goal positions as known as Euclidean distance. It is also admissible because it never overestimates the distance to the goal state.

```
def h1(state):
    hn = 0
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] != goalState[i][j]:
                hn += 1
    return hn


def h2(state):
    hn = 0
    for i in range(len(state)):
        for j in range(len(state[i])):
            loc = dic[state[i][j]]
            y, x = loc[0], loc[1]
            hn += abs(y - i) + abs(x - j)
    return hn


def h3(state):
    hn = 0
    for i in range(len(state)):
        for j in range(len(state[i])):
            loc = dic[state[i][j]]
            y, x = loc[0], loc[1]
            hn += math.sqrt(abs(y - i) ** 2 + abs(x - j) ** 2)
    return hn
```

From the result of solution paths, we can see that the speed and steps are almost the same for A* and BFS when the test cases are not complicated. If the test case is very complex, A* will be much faster than BFS and required fewer steps. Meanwhile, h1, h2 and h3 have the same or similar speed and steps under the A* search in most cases. But in BFS, it shows that the numbers of steps are h2 > h3 > h1.

```
import numpy as np
import math
import operator

"""
initState = [[1, 2, 3], [0, 4, 6], [8, 5, 7]]
goalState = [[0, 1, 3], [4, 2, 5], [7, 8, 6]]   #no solution
"""

initState = [[3, 7, 2], [6, 8, 0], [4, 5, 1]]
goalState = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]

"""
initState = [[2, 8, 3], [1, 0, 4], [7, 6, 5]]
goalState = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
"""
"""
initState = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
goalState = [[1, 0, 2], [3, 4, 5], [6, 7, 8]]
"""

"""
initState = [[2, 8, 3], [6, 0, 4], [1, 7, 5]]
goalState = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
"""
"""
initState = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
goalState = [[3, 1, 2], [4, 0, 6], [7, 5, 8]]
"""

dic = np.zeros((9, 2))
k = 0
for i in range(3):  # goal index dictionary auto generater
```

```
    for j in range(3):
        dic[k][0] = i
        dic[k][1] = j
        k += 1

step = 0


def checkInversions(no0):
    count = 0
    for i in range(len(no0)):
        for j in range(i + 1, len(no0)):
            if no0[i] > no0[j]:
                # print(no0[i], " - ", no0[j])
                count += 1
    return count


def checkSolution():
    initNo0 = ""
    goalNo0 = ""
    # print(type(initState))
    for i in range(len(initState)):
        for j in range(len(initState[i])):
            if initState[i][j] != 0:
                initNo0 += str(initState[i][j])

    for i in range(len(goalState)):
        for j in range(len(goalState[i])):
            if goalState[i][j] != 0:
                goalNo0 += str(goalState[i][j])

    inv1 = checkInversions(initNo0)
    print("")
    inv2 = checkInversions(goalNo0)
    invResult = inv1 - inv2
    # print(invResult)
    if invResult % 2 != 0:
        return 0
    return 1
```

```
def copyState(state):  # copy state without change
    new = []
    for i in state: new.append(i[:])
    return new


def h1(state):  # number of misplaced tiles
    hn = 0
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] != goalState[i][j]:
                hn += 1
    return hn


def h2(state):  # Manhattan distance
    hn = 0
    for i in range(len(state)):
        for j in range(len(state[i])):
            loc = dic[state[i][j]]
            y, x = loc[0], loc[1]
            hn += abs(y - i) + abs(x - j)
    return hn


def h3(state):  # Euclidean distance
    hn = 0
    for i in range(len(state)):
        for j in range(len(state[i])):
            loc = dic[state[i][j]]
            y, x = loc[0], loc[1]
            hn += math.sqrt(abs(y - i) ** 2 + abs(x - j) ** 2)
    return hn


def getAvailActs(state):
    acts = []
    for a in range(len(state)):  # find the index for 0
```

```
        for b in range(len(state[a])):
            if state[a][b] == 0:
                break
        if state[a][b] == 0:
            break

    for i in range(len(state)):  #swap
        for j in range(len(state[i])):
            if ((i - a) ** 2 + (j - b) ** 2) == 1:
                cp = copyState(state)
                cp[a][b] = cp[i][j]
                cp[i][j] = 0
                acts.append(cp)
    # print(acts[0])
    # print(acts)
    return acts


class aStarNode:
    def __init__(self, state, gn, parent):
        self.state = state
        self.gn = gn
        self.parent = parent
        self.fn = h1(state) + gn
        # self.fn = h2(state) + gn
        #self.fn = h3(state) + gn


class gbfsNode:
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent
        self.fn = h1(state)
        # self.fn = h2(state)
        #self.fn = h3(state)


def printPath(root):
    if root is None:
        return
```

```
    global step
    step += 1
    printPath(root.parent)
    # print(root.state)
    for i in root.state:
        print(i)
    print("")


def aStar():
    list1 = [aStarNode(initState, 0, None)]
    visit = []
    count = 0
    while list1:
        list1.sort(key=operator.attrgetter('fn'))
        last = list1[0]
        visit.append(last.state)
        # print(last.state)
        count += 1
        if count == 20000:
            return last, count
        if last.state == goalState:
            return last, count
        del list1[0]
        acts = getAvailActs(last.state)
        for i in acts:
            availAct = aStarNode(i, last.gn + 1, last)
            if availAct.state not in visit:
                # print(availAct.state)
                list1.append(availAct)
                visit.append(availAct.state)

    return None, count


def gbfs():
    list2 = [gbfsNode(initState, None)]
    visit = []
    count = 0
    while list2:
```

```
        list2.sort(key=operator.attrgetter('fn'))
        last = list2[0]
        visit.append(last.state)
        count += 1
        if last.state == goalState:
            return last, count
        if count == 20000:
            return last, count
        del list2[0]
        acts = getAvailActs(last.state)
        for i in acts:
            availAct = gbfsNode(i, last)
            if availAct.state not in visit:
                list2.append(availAct)
                visit.append(availAct.state)

    return None, count


if __name__ == '__main__':

    if checkSolution() == 0:
        print("No solution")
    else:
        root1, count1 = aStar()

        print("A*: ")
        printPath(root1)
        if count1 == 10000:
            print("exceeds limit, solution not found")
        print("Steps: ", step)
        print("")
        root2, count2 = gbfs()
        step = 0
        print("gbfs: ")
        printPath(root2)
        if count2 == 10000:
            print("exceeds limit, solution not found")
        print("Steps: ", step)
```