BTTH - NM TTNT - TUẦN 5

21280099 - Nguyễn Công Hoài Nam

Ngày 4 tháng 12 năm 2023

I. Bài toán

TSP (Traveling Salesperson Problem): Cho trước n thành phố và các khoảng cách d_{ij} giữa mỗi cặp thành phố, tìm tour ngắn nhất sao cho mỗi thành phố được viếng thăm chỉ một lần.
Sử dụng thuật toán A^* để giải bài toán TSP và heuristic được sử dụng là cây khung nhỏ nhất.

II. Cài đặt

1. Kiểm tra lỗi

Chương trình hoạt động đúng và không có lỗi

2. Hiểu biết về bài thực hành

Một chương trình viết bằng ngôn ngữ Python giải quyết bài toán TSP bằng thuật toán A*

a. Các Class

TreeNode

```
# Structure to represent tree nodes in the A* expansion
class TreeNode(object):
    def __init__(self,c_no,c_id,f_value,h_value,parent_id):
        self.c_no = c_no
        self.c_id = c_id
        self.f_value = f_value
        self.h_value = h_value
        self.parent_id = parent_id
```

Lớp dùng để biểu diễn một nút (thành phố) trong cây tìm kiếm A^*, gồm các thuộc tính

c_no: số thứ tự của thành phố (nút)

c_id: ID của nút

f_value: giá trị hàm f của nút

h_value: giá trị hàm heuristic của nút

parent_id: ID của nút cha

FringeNode

```
# Structure to represent fringe nodes in the A* fringe list
class FringeNode(object):
    def __init__(self,c_no,f_value):
        self.f_value = f_value
        self.c_no = c_no
```

Đại diện cho một nút trong danh sách FringeNode gồm thứ tự và giá trị f_value, danh sách này chứa các nút mà nút hiện tại có thể đi được, thuật toán A^* lựa chọn nút kế tiếp dựa vào mức độ ưu tiên thông qua thuộc tính f_value (giá trị f nhỏ nhất)
Graph

```
class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
```

Dùng để biểu diễn đồ thị bao gồm số đỉnh của đồ thị (vertices), và một ma trận kề ban đầu khởi tạo mặc định bằng 0.
Bao gồm các phương thức:

- printMST(self, parent, d_temp, t)
- minKey(self, key, mstSet)
- primMST(self, d_temp, t)

printMST dùng để in ra MST (Minimum Spanning Tree) và tính trọng số của cây khung đó để đánh giá

```
# A utility function to print the constructed MST stored in parent[]
def printMST(self, parent, d_temp, t):
    #print("Edge \tWeight")
    sum_weight = 0
    min1 = 10000
    min2 = 10000
    r_temp = {} #Reverse dictionary
    for k in d_temp:
        r_temp[d_temp[k]] = k

    for i in range(1, self.V):
        # print(parent[i]. "-", i, "\t", self.graph[i][parent[i]])
        sum_weight = sum_weight + self.graph[i][parent[i]]
        if (graph[0][r_temp[i]] < min1):
            min1 = graph[0][r_temp[i]]
        if (graph[0][r_temp[parent[i]]] < min1):
            min1 = graph[0][r_temp[parent[i]]]
        if (graph[t][r_temp[i]] < min2):
            min2 = graph[t][r_temp[i]]
        if (graph[t][r_temp[parent[i]]] < min2):
            min2 = graph[t][r_temp[parent[i]]]

    return (sum_weight + min1 + min2)%10000
```

minKey dùng để tìm nút có trọng số thấp nhất của các đỉnh chưa được duyệt, để thêm vào cây khung

```
# A utility function to find the vertex with
# minimum distance value, from the set of vertices
# not yet included in shortest path tree
def minKey(self, key, mstSet):

    # Initilaize min value
    min = sys.maxsize
    for v in range(self.V):
        if key[v] < min and mstSet[v] == False:
            min = key[v]
            min_index = v

    return min_index
```

primMST sử dụng thuật toán Prim để xác định cây khung nhỏ nhất, hàm trả về tổng trọng số của cây khung đó

```
def primMST(self, d_temp, t):

    # Key values used to pick minium weight edge in cut
    key = [sys.maxsize] * self.V
    parent = [None] * self.V # Array to store constructed MST
    # Make key 0 so that this vertex is picked as fisrt vertex
    key[0] = 0
    mstSet = [False] * self.V
    sum_weight = 10000
    parent[0] = -1 # First node is always the root of

    for c in range(self.V):

        # Pick the minimum distance vertex from the set of vertices not yet processed
        # u is always equal to src in first iteration
        u = self.minKey(key, mstSet)

        # Put the minimum distance vertex in the shortest path tree
        mstSet[u] = True
```

```
        # Update dist value of the adjacent vertices of the picked vertex only if the
        # current distance is greater than new distance and
        # the vertex in not in the shortest path tree
        for v in range(self.V):
            # graph[u][v] is non zero only for adjacent vertices of m
            # mstSet[v] is false for vertices not yet included in MST
            # update the key only if graph[u][v] is smaller than key[v]
            if self.graph[u][v] > 0 and mstSet[v] == False and key[v] > self.graph[u][v]:
                key[v] = self.graph[u][v]
                parent[v] = u
    return self.printMST(parent, d_temp, t)
```

b. Functtion

heuristic(tree, p_id, t, V, graph) dựa vào cây khung nhỏ nhất, ước lượng heuristic từ nút gốc đến nút đích

```
def heuristic(tree, p_id, t, V, graph):
    visited = set()  # Set to store visited nodes
    visited.add(0)
    visited.add(t)
    if (p_id != -1):
        tnode = tree.get_node(str(p_id))
        # Find all visited nodes and add them to the set
        while (tnode.data.c_id != 1):
            visited.add(tnode.data.c_no)
            tnode = tree.get_node(str(tnode.data.parent_id))
    l = len(visited)
    num = V - l  # No of unvisited nodes
    if (num != 0):
        g = Graph(num)
        d_temp = {}
        key = 0
        # d_temp dictionary stores mappings of original city no as (key) and new sequential no as
    value for MST to work
        for i in range(V):
            if (i not in visited):
                d_temp[i] = key
                key = key + 1
        i = 0
        for i in range(V):
            for j in range(V):
                if ((i not in visited) and (j not in visited)):
                    g.graph[d_temp[i]][d_temp[j]] = graph[i][j]
        # print(g.graph)
        mst_weight = g.primMST(d_temp, t)
        return mst_weight
    else:
        return graph[t][0]
```

1. Khởi tạo tập hợp `visited`:
   - Thêm các nút 0 và `t` vào tập hợp `visited`.
2. Kiểm tra `p_id` khác -1 (không là nút gốc):
   - Đi từ `p_id` lên đến nút gốc để tìm tất cả các nút đã thăm và thêm chúng vào tập hợp `visited`.
3. Tính độ dài và số nút chưa thăm:
   - Độ dài của tập hợp `visited` được lưu trong biến `l`.
   - Tính số lượng nút chưa thăm (`num`) bằng cách trừ số nút đã thăm từ tổng số nút (`V`).
4. Kiểm tra số nút chưa thăm:
   - Nếu `num` khác 0:
     - Khởi tạo một đồ thị mới `g` với `num` đỉnh.
     - Tạo một từ điển `d_temp` để ánh xạ số hiệu của các nút chưa thăm sang số hiệu mới từ 0 đến `num` - 1.
     - Xây dựng đồ thị mới `g` chỉ chứa các nút chưa thăm và trọng số cạnh giữa chúng từ đồ thị ban đầu `graph`.
     - Tính trọng số của cây khung nhỏ nhất (`mst_weight`) bằng cách thực hiện thuật toán Prim (`primMST`) trên `g` với `d_temp` và `t` làm điểm đích.

- Trả về `mst_weight`.

- Ngược lại:
  - Nếu không có nút nào chưa thăm (`num` == 0), trả về trọng số của cạnh từ thành phố `t` đến thành phố 0 trong đồ thị ban đầu `graph`.

checkPath(tree, toExpand, V) kiểm tra xem đường đi tìm được có đi qua tất cả các đỉnh của đồ thị hay không (gọi là hoàn chỉnh)

```
def checkPath(tree, toExpand, V):
    tnode = tree.get_node(str(toExpand.c_id))  # Get the node to expand from the tree
    list1 = list()  # List to store the path
    # For 1st node
    if (tnode.data.c_id == 1):
        # print("In If")
        return 0
    else:
        # print("In else")
        depth = tree.depth(tnode)  # Check depth of the tree
        s = set() # set to store nodes in the path
        # nodes in the way to the set and list
        # do up in the tr  using the parent pointer and add all
        while (tnode.data.c_id != 1):
            s.add(tnode.data.c_no)
            list1.append(tnode.data.c_no)
            tnode = tree.get_node(str(tnode.data.parent_id))
        list1.append(0)
        if (depth == V and len(s) == V and list1[0] == 0):
            print("Path complete")
            list1.reverse()
            print(list1)
            return 1
        else:
            return 0
```

startTSP(graph, tree, V) tổng hợp giải bài toán TSP

```
def startTSP(graph, tree, V):
    goalState = 0
    times = 0
    toExpand = TreeNode(0, 0, 0, 0, 0)  # Node to expand
    key = 1  # Unique Identifier for a node in the tree
    heu = heuristic(tree, -1, 0, V, graph)  # Heurisitic for node 0 in the tree
    tree.create_node("1", "1", data=TreeNode(0, 1, heu, heu, -1))  # Create 1st node in the tree i.e.
     0th city
    fringe_list = {}  # Fringe List(Dictionary)(FL)
    fringe_list[key] = FringeNode(0, heu)  # Adding 1st node in FL
    key = key + 1
    while (goalState == 0):
        minf = sys.maxsize
        # Pick node having min f_value from the fringe list
        for i in fringe_list.keys():
            if (fringe_list[i].f_value < minf):
                toExpand.f_value = fringe_list[i].f_value
                toExpand.c_no = fringe_list[i].c_no
                toExpand.c_id = i
                minf = fringe_list[i].f_value

        h = tree.get_node(str(toExpand.c_id)).data.h_value # heuristic value of selected node
        val = toExpand.f_value - h # g value of selected node
        path = checkPath(tree, toExpand, V) # check path of tselected node if it is complete or not
        #  If node to expand is 0 and path is complete, we are done
        # We check node at the time of expansion and not at the time of generation
        if (toExpand.c_no == 0 and path == 1):
            goalState = 1
            cost = toExpand.f_value #total actual cost incurred
        else:
            del fringe_list[toExpand.c_id] # remove node from FL
            j = 0
            # Evaluate f_values and h_values of adjacent nodes of the node to expand
            while (j < V):
                if (j != toExpand.c_no):
                    h = heuristic(tree, toExpand.c_id, j, V, graph)  # Heuristic calc
                    f_val = val + graph[j][toExpand.c_no] + h  # g(parent) + g(parent->child) + h(
    child)
                    fringe_list[key] = FringeNode(j, f_val)
                    tree.create_node(str(toExpand.c_no), str(key), parent=str(toExpand.c_id),
                                     data=TreeNode(j, key, f_val, h, toExpand.c_id))
                    key = key + 1
                j = j + 1
    return cost
```

Các bước:

1. Khởi tạo
   - `goalState = 0`: Nút gốc = 0
   - `times = 0`: Biến này đếm số lần lặp trong quá trình tìm kiếm.
   - `toExpand`: Đây là một đối tượng `TreeNode` đại diện cho nút cần mở rộng trong quá trình tìm kiếm.
   - `key = 1`: Đây là một giá trị định danh duy nhất cho mỗi nút trong cây tìm kiếm.
   - `heu`: Đây là giá trị heuristic cho nút 0 trong cây tìm kiếm.
   - `tree`: Đây là cây tìm kiếm, được đại diện bằng một đối tượng cây.
   - `fringe_list`: Đây là một từ điển (dictionary) đại diện cho danh sách fringe (FL). Mỗi khóa trong từ điển là một giá trị key và mỗi giá trị tương ứng là một đối tượng `FringeNode`.
   - `cost`: Đây là biến lưu trữ giá trị chi phí tối ưu tìm thấy cho TSP.
2. Tạo nút đầu tiên trong cây tìm kiếm:
   - Sử dụng giá trị heuristic của nút 0 (được tính bằng hàm `heuristic`) để tạo nút đầu tiên trong cây tìm kiếm. Nút này đại diện cho thành phố 0.
   - Gán giá trị `heu` cho `f_value` của nút đầu tiên.
   - Thêm nút đầu tiên này vào cây tìm kiếm và danh sách fringe.
3. Bắt đầu vòng lặp tìm kiếm:
   - Trong khi chưa đạt được trạng thái mục tiêu (`goalState = 0`), tiếp tục thực hiện các bước sau:
   - Tìm nút có giá trị `f_value` nhỏ nhất trong danh sách fringe (`fringe_list`) để mở rộng.
   - Lấy giá trị `h_value` của nút được chọn từ cây tìm kiếm.
   - Tính giá trị `val` bằng cách trừ `h_value` từ `f_value` của nút được chọn.
   - Kiểm tra xem đường đi từ nút được chọn đã hoàn thành hay chưa bằng cách sử dụng hàm `checkPath`. Nếu đã hoàn thành, gán `goalState = 1` và lưu trữ giá trị `f_value` vào biến `cost`.
   - Nếu đường đi chưa hoàn thành, loại bỏ nút được chọn khỏi danh sách fringe.
   - Duyệt qua tất cả các thành phố khác (từ 0 đến V) để tính toán giá trị `f_value` và `h_value` cho các nút con của nút được chọn.
   - Thêm các nút con này vào danh sách fringe và cây tìm kiếm.
4. Trả về giá trị `cost`:
   - Khi đã đạt được trạng thái mục tiêu, trả về giá trị `cost` là giá trị tối ưu của TSP.

Hàm **main** khởi tạo số đỉnh V và đồ thị, thực thi thuật toán

```
if __name__ == '__main__':
    V = 4
    graph = [[0, 5, 2, 3], [5, 0, 6, 3], [2, 6, 0, 4], [3, 3, 4, 0]]
    tree = Tree()
    ans = startTSP(graph, tree, V)
    print("Ans is " + str(ans))
```

III. Kết quả

```
/Data/Data\ chung/Course\ HK1\ 23-24/Introduce\ to\ AI/Tuan\ 5/Souce
_code_Tuan5.py
Path complete
[0, 2, 3, 1, 0]
Ans is 14
```

Hình 1: Console chạy thuật toán

Cây khung: [0,2,3,1,0] có trọng số 14