Report AI Lab5

Nguyễn Quốc Bảo

18110053

Bài toán nhân viên bán hàng đi du lịch (TSP) có thể được giải quyết thông qua kinh nghiệm cây bao trùm tối thiểu (MST), được sử dụng để ước tính chi phí hoàn thành một chuyến tham quan, với điều kiện là một chuyến tham quan đã được xây dựng. Chi phí MST của một tập hợp các thành phố là tổng nhỏ nhất của chi phí liên kết của bất kỳ cây nào kết nối tất cả các thành phố.

1. Bài toán TSP là tìm một con đường tối thiểu (tổng chiều dài) qua các thành phố tạo thành một vòng khép kín. MST là một phiên bản phù hợp cho điều đó bởi vì nó yêu cầu một đồ thị (tổng độ dài) tối thiểu mà không cần phải là một vòng khép kín, nó có thể là bất kỳ đồ thị nào được kết nối đầy đủ. Theo phương pháp heuristic, MST có thể chấp nhận được, nó luôn ngắn hơn hoặc bằng một vòng lặp kín.

2. Khoảng cách đường thẳng quay trở lại thành phố xuất phát là một phương pháp kinh nghiệm khá yếu, nó ước tính rất lớn khi có nhiều thành phố. Trong giai đoạn sau của cuộc tìm kiếm khi chỉ còn lại một vài thành phố, điều đó không quá tệ. Để nói rằng MST thống trị khoảng cách đường thẳng có nghĩa là MST luôn cho giá trị cao hơn. Điều này rõ ràng là đúng bởi vì một MST bao gồm nút mục tiêu và nút hiện tại hoặc phải là đường thẳng, hoặc nó phải bao gồm hai hoặc nhiều đường cộng lại với nhau. (Tất cả điều này giả định bất bình đẳng tam giác.)

3. A simple generator can be:

   Generator(Integer Nums){

   . POINTS = nil

   . N = 0

   . **while**(N < Nums){

   . (X,Y) = (Random(0,1),Random(0,1))

   . **if**((X,Y) $\notin POINTS$){

   . $POINTS \leftarrow (X, Y)$

   . $N = N + 1$

   . }

   . }

   . **return** $POINTS$

   }

4. Thuật toán dùng để xây dựng MST có thể dùng Kruskal và Prim. Ở đây tôi đang dùng Kruskal để xây dưng MST Và thuật toán tìm kiếm A* để giải quyết các trường hợp của TSP..

   Thuật Toán Kruskal

```python
def Kruskal_algo(self):
    result = [] # This will store the resultant MST
    Sorted = self.sort_weight()
    parent = []
    rank =[]
    # An index variable, used for sorted edges
    i = 0
    # An index variable, used for result[]
    e = 0
    for node in range(self.vertices):
        parent.append(node)
        rank.append(0)
    while e < self.vertices - 1:

        # Step 2: Pick the smallest edge and increment
        # the index for next iteration
        node, visited, cost = Sorted[i]
        i = i + 1
        x = self.find(parent, node)
        y = self.find(parent, visited)

        # If including this edge does't
        #  cause cycle, include it in result
        #  and increment the indexof result
        # for next edge
        if x != y:
            e = e + 1
            result.append([node, visited, cost])
            self.union(parent, rank, x, y)
        # Else discard the edge

    minimumCost = 0
    print("Edges in the constructed MST")
    for node, visited, cost in result:
        minimumCost += cost
        print("{} -- {} = {}".format(self.name_ct[node],self.name_ct[visited], cost))
    print("Minimum Spanning Tree" , minimumCost)
```

1.Sắp xếp các cạnh theo trọng số khoảng cách.

2.Chọn những cạnh có trọng số nhỏ nhất. Nếu cạnh không phải chu trình thì giữ lại và xét tiếp các cạnh tiếp theo.Lặp lại bước 2 cho tới khi đã duyết hết các cạnh.

```python
# A utility function to find set of an element i
# (uses path compression technique)
def find(self, parent, i):
    if parent[i] == i:
        return i
    return self.find(parent, parent[i])

# A function that does union of two sets of x and y
# (uses union by rank)
def union(self, parent, rank, x, y):
    xroot = self.find(parent, x)
    yroot = self.find(parent, y)

    # Attach smaller rank tree under root of
    # high rank tree (Union by Rank)
    if rank[xroot] < rank[yroot]:
        parent[xroot] = yroot
    elif rank[xroot] > rank[yroot]:
        parent[yroot] = xroot

    # If ranks are same, then make one as root
    # and increment its rank by one
    else:
        parent[yroot] = xroot
        rank[xroot] += 1
```

Hàm tìm những cạnh có liên kết và xét hạng của các đỉnh.

Thuật Toán A*

```python
# A* search
def aStart(self,start,goal):
    Open = PriorityQueue()
    f = 0; g = 0; h = 0
    root = (f,g,h,start)
    Open.put(root)
    Close = []
    father = {}
    path = [goal]
    while True:
        if Open.empty():
            raise Exception("No way Exception")
        current_f,current_h,current_g,current_node = Open.get()
        Close.append(current_node)
        if current_node == goal:
            key = goal
            while key in father.keys():
                value = father.pop(key)
                path.append(value)
                key = value
                if key == start:
                    break
            path.reverse()
            return path
        if current_node not in self.graph:
            continue
        for Node in self.graph[current_node]:
            node,grid,heuris = Node[0],Node[-2],Node[-1]
            grid = current_g + min_cost(current_g,grid)
            f = grid + heuris
            if node not in Close:
                Open.put((f, grid, heuris, node))
                if node not in father.keys():
                    father[node] = current_node
```

Kết quả

```
Edges in the constructed MST
Lugoj -- Mehadia = 70
Oradea -- Zerind = 71
Arad -- Zerind = 75
Drobeta -- Mehadia = 75
Rimnicu Vilcea -- Sibiu = 80
Bucharest -- Urziceni = 85
Eforie -- Hirsova = 86
Iasi -- Neamt = 87
Bucharest -- Giurgiu = 90
Iasi -- Vaslui = 92
Pitesti -- Rimnicu Vilcea = 97
Hirsova -- Urziceni = 98
Fagaras -- Sibiu = 99
Bucharest -- Pitesti = 101
Lugoj -- Timisoara = 111
Arad -- Timisoara = 118
Craiova -- Drobeta = 120
Craiova -- Pitesti = 138
Urziceni -- Vaslui = 142
Minimum Spanning Tree 1835


path A*:
Lugoj --> Mehadia --> Drobeta --> Craiova --> Pitesti --> Bucharest
```

Tập các cạnh xây dựng theo MST.