BÁO CÁO THỰC HÀNH NHẬP MÔN TRÍ TUỆ NHÂN TẠO

Tuần 5

Bài toán. The traveling saleperson problem (TSP) can be solved via the minimum spanning tree (MST) heuristic, which is used to estimate the cost of completing a tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link costs of any tree that connects all the cities.

(a) Show how this heuristic can be derived from a relaxed version of the TSP.

(b) Show how the MST heuristic dominates straight-line distance.

(c) Write a problem generator for instances of the TSP where cities are represented by random points in the unit square.

(d) Find an efficient algorithm in the liteature for constructing the MST, and use it with an admissible search algorithm to solve instances of the TSP.

Lời giải

(a) Bài toán người bán hàng (TSP) đi tìm đường đi ngắn nhất qua các thành phố sao cho đường đi tạo thành một chu trình. Cây khung tối thiểu (MST) là một dạng *relaxed* của TSP vì nó cũng đi tìm đường đi ngắn nhất qua các đỉnh của một đồ thị mà không cần tạo thành chu trình. Điều đó có nghĩa là mỗi điểm có thể được duyệt nhiều lần và chi phí cho mỗi lần lặp lại đường đi không được tính vào tổng chi phí.

(b) MST heuristic bao gồm các khoảng cách dạng đường thẳng vì khoảng cách ngắn nhất từ hai thành phố bất kỳ A, B là khoảng cách theo dạng đường thẳng bao gồm một hoặc nhiều đường thẳng gộp lại nối 2 thành phố đó với nhau. Điều trên đúng do kết quả của bất đẳng thức tam giác.

(c) Một hàm tạo đơn giản có thể tạo là

A problem generator

```
function GENERATOR(integer numbers) return danh sách các điểm
    inputs: numbers, số lượng điểm.
    POINTS ← NULL
    N ← 0
    while(N < numbers) do
        (X, Y) ← (Random(0, 1), Random(0, 1))
```

```
            if ((X, Y) ∉ POINTS) then do
                POINTS ← (X, Y)
                N ← N + 1
        return POINTS
```

(d) Sử dụng thuật toán Kruskal để xây dựng cây khung tối thiểu (MST).

1 Dữ liệu đầu vào

```
10
0   8   13  0   0   14  0   8   0   0
8   0   9   12  0   0   0   0   0   11
13  9   0   0   13  15  0   0   0   0
0   12  0   0   19  0   0   0   0   0
0   0   13  19  0   15  0   0   0   0
14  0   15  0   15  0   22  18  0   0
0   0   0   0   0   22  0   0   21  0
8   0   0   0   0   18  0   0   10  8
0   0   0   0   0   0   21  10  0   12
0  11   0   0   0   0   0   8   12  0
```

Trong đó

- Dòng 1: N là số đỉnh.
- N dòng tiếp theo là ma trận kề M của đồ thị với quy ước
  - $M[i][j] = w$: có đường nối trực tiếp từ i đến j với chi phí là w ($w > 0$).
  - $M[i][j] = 0$: không có đường nối trực tiếp từ i đến j.

2 Xử lý dữ liệu đầu vào

Đọc dữ liệu từ file ở trên và trả về

- **vertices**: số đỉnh.
- **adj_matrix**: ma trận kề.

```python
def handle_input(name_file):
    with open(name_file, 'r') as f:
        vertices = int(f.readline())
        adj_matrix = [[int(num) for num in line.split('\t')] for line in f]
        f.close()
    return vertices, adj_matrix
```

3 Khởi tạo class Graph

```python
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)
```

Trong đó

- **V**: biến lưu số đỉnh.
- **graph**: danh sách các đường đi của đồ thị.

Kết quả hình thành graph

```
0       -->     [(1, 8), (2, 13), (5, 14), (7, 8)]
1       -->     [(0, 8), (2, 9), (3, 12), (9, 11)]
2       -->     [(0, 13), (1, 9), (4, 13), (5, 15)]
3       -->     [(1, 12), (4, 19)]
4       -->     [(2, 13), (3, 19), (5, 15)]
5       -->     [(0, 14), (2, 15), (4, 15), (6, 22), (7, 18)]
6       -->     [(5, 22), (8, 21)]
7       -->     [(0, 8), (5, 18), (8, 10), (9, 8)]
8       -->     [(6, 21), (7, 10), (9, 12)]
9       -->     [(1, 11), (7, 8), (8, 12)]
```

4 Các hàm và phương thức hỗ trợ thực thi bài toán

- Hàm **add_edge** là hàm kết nối 2 điểm lại với nhau cùng với trọng số khoảng cách giữa chúng.

```
def add_edge(self, src, dest, weight):
    self.graph[src].append((dest, weight))
```

- Hàm **display_graph** là hàm mô tả các đường đi có trong đồ thị.

```
def display_graph(self):
    for node in self.graph:
        print(node, "\t-->\t", self.graph[node])
```

- Hàm **find** là hàm tìm ra tập hợp con chứa phần tử **i**.

```
# 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])
```

- Hàm **union** là hàm kết hợp hai tập hợp **x** và **y** lại thành một.

```
# 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
```

5 Thuật toán Kruskal

5.1 Ý tưởng thuật toán

Một cây khung tối tiểu sẽ có **(V - 1)** cạnh trong đó **V** là số đỉnh có trong đồ thị.
Các bước để tìm MST bằng cách dùng thuật toán Kruskal

Thuật toán Kruskal

(1) Sắp xếp các cạnh theo thứ tự tăng dần trọng số của các cạnh.

(2) Lấy ra cạnh có trọng số nhỏ nhất. Kiểm tra xem nó có tạo thành một chu trình hay không với các cạnh đang có trong cây khung được hình thành. Nếu không tạo thành chu trình thì thêm nó vào cây khung, ngược lại, bỏ qua cạnh đó.

(3) Lặp lại bước (2) cho đến khi có **V - 1** cạnh trong cây khung.

5.2 Hàm MST_Kruskal

```
def MST_Kruskal(self):
        temp_graph = []
        for cur_node in self.graph:
            current_adj_nodes = self.graph[cur_node]
            for i in range(len(current_adj_nodes)):
                node, weight = current_adj_nodes[i]
```

```
                temp_graph.append((cur_node, node, weight))

        result = []  # This will store the resultant MST
        # An index variable, used for sorted edges
        i = 0
        # An index variable, used for result[]
        e = 0

        # Step 1:  Sort all the edges in
        # non-decreasing order of their
        # weight.  If we are not allowed to change the
        # given graph, we can create a copy of graph
        sorted_graph = sorted(temp_graph, key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        while e < self.V - 1:

            # Step 2: Pick the smallest edge and increment
            # the index for next iteration
            u, v, w = sorted_graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # 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([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        minimumCost = 0
        print("Edges in the constructed MST using Kruskal")
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d (%d)" % (u, v, weight))
        print("Minimum Spanning Tree - minimum total cost: " , minimumCost)
```

5.3 Kết quả thực thi

```
Edges in the constructed MST using Kruskal
0 -- 1 (8)
0 -- 7 (8)
7 -- 9 (8)
1 -- 2 (9)
7 -- 8 (10)
1 -- 3 (12)
2 -- 4 (13)
0 -- 5 (14)
6 -- 8 (21)
Minimum Spanning Tree - minimum total cost:  103
```