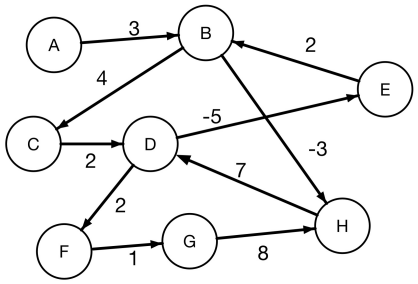
Problem Set 6 — Module 6

Shenyu Li

October 18th 2024

1.Bellman-Ford algorithm

Apply the Bellman-Ford algorithm to the following graph with source vertex A:



1 Problem Principle

The Bellman-Ford algorithm computes the shortest path from a source vertex to all other vertices in a weighted directed graph. The algorithm can handle graphs with negative weights and detects negative weight cycles.

2 Graph and Weights

Given a directed graph with the following edges and weights:

- A → B, weight 3
- B → C, weight 4
- D → E, weight -5
- E → B, weight 2
- C → D, weight 2
- D → F, weight 2

- H → D, weight 7
- B → H, weight -3
- F → G, weight 1
- G → H, weight 8

3 Solution Method

The Bellman-Ford algorithm relaxes the edges $V - 1$ times (where V is the number of vertices), performs 7 iterations, relaxing all edges in the graph, and the final shortest distances are computed.

4 Code Implementation

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices  # Number of vertices
        self.edges = []    # List of edges

    def add_edge(self, u, v, weight):
        self.edges.append((u, v, weight))

    def bellman_ford(self, source):
        # Initialize distances
        dist = [float("inf")] * self.V
        dist[source] = 0

        # Perform V-1 iterations of edge relaxation
        for i in range(self.V - 1):
            print(f"Iteration {i + 1}:")
            for u, v, weight in self.edges:
                if dist[u] != float("inf") and dist[u] + weight < dist[v]:
                    dist[v] = dist[u] + weight
            print(f"Current distance array: {dist}\n")

        # Check for negative-weight cycles
        for u, v, weight in self.edges:
            if dist[u] != float("inf") and dist[u] + weight < dist[v]:
                print("Graph contains a negative-weight cycle.")
                return None

        return dist
```

```
# Create the graph
g = Graph(8)  # 8 vertices: A, B, C, D, E, F, G, H

# Add edges and weights
g.add_edge(0, 1, 3)  # A -> B, weight 3
g.add_edge(1, 2, 4)  # B -> C, weight 4
g.add_edge(3, 4, -5) # D -> E, weight -5
g.add_edge(4, 1, 2)  # E -> B, weight 2
g.add_edge(2, 3, 2)  # C -> D, weight 2
g.add_edge(3, 5, 2)  # D -> F, weight 2
g.add_edge(7, 3, 7)  # H -> D, weight 7
g.add_edge(1, 6, -3) # B -> H, weight -3
g.add_edge(5, 6, 1)  # F -> G, weight 1
g.add_edge(6, 7, 8)  # G -> H, weight 8

# Run Bellman-Ford algorithm
source = 0  # Source vertex is A
distances = g.bellman_ford(source)

# Print final results
if distances is not None:
    print("Final shortest distances from A:")
    for i, distance in enumerate(distances):
        print(f"Vertex {i}: {distance}")
```

5 Step-by-step Execution

Initialization:

$$\text{dist} = [0, \infty, \infty, \infty, \infty, \infty, \infty, \infty]$$

Relaxation Steps:

- **1st Relaxation:**
  Current distance array: [0, 3, 7, 9, ∞, 11, 0, 8]
- **2nd Relaxation:**
  Current distance array: [0, 3, 7, 9, 4, 11, 0, 8]
- **3rd Relaxation:**
  Current distance array: [0, 3, 7, 9, 4, 11, 0, 8]
- **4th Relaxation:**
  Current distance array: [0, 3, 7, 9, 4, 11, 0, 8]
- **5th Relaxation:**
  Current distance array: [0, 3, 7, 9, 4, 11, 0, 8]

- **6th Relaxation:**
  Current distance array: [0, 3, 7, 9, 4, 11, 0, 8]
- **7th Relaxation:**
  Current distance array: [0, 3, 7, 9, 4, 11, 0, 8]

Final Results

The shortest distances from source vertex A to each vertex are:

To vertex 0:	0
To vertex 1:	3
To vertex 2:	7
To vertex 3:	9
To vertex 4:	4
To vertex 5:	11
To vertex 6:	0
To vertex 7:	8

2.Graphs with Negative Weights

1. Does the idea of distance still make sense if there are negative weights?

Yes, it does, Even if some roads (or edges) have negative weights, Distance means how far apart two points are. Negative weights can make some paths shorter, but they can also make things tricky. If have a loop with negative weights, keep going around and making the distance smaller and smaller forever.

2. Do the algorithms (Floyd-Warshall, Dijkstra, Bellman-Ford) still work with negative weights?

- **Floyd-Warshall Algorithm:** Yes, this one works with negative weights. It can find the shortest paths between all points and even checks if there are any loops that make the distance go down forever.
- **Dijkstra's Algorithm:** No, this one doesn't work well with negative weights. It thinks that once it finds the shortest path to a point, it can't get any better. But negative weights can mess this up and give wrong answers.
- **Bellman-Ford Algorithm:** Yes, this one is made for graphs with negative weights. It can find the shortest path from one starting point to all other points and can also check for those tricky loops.

3. Where might you find graphs with negative weights?

Find graphs with negative weights in things like network routing. In computer networks, some rules might give negative weights to certain paths to make sure that data goes through those paths instead. This helps to make things work better and makes sure no one path gets too busy. It's like telling cars to take a quieter road so that all roads are used fairly and traffic moves smoothly.

5.2 Minimum Spanning Tree: Prim's and Kruskal's Algorithms

This question demonstrates the execution of Prim's and Kruskal's algorithms on a given undirected graph to find the Minimum Spanning Tree (MST), detail the intermediate steps, including the cost array for Prim's algorithm and the disjoint-set structure for Kruskal's algorithm.

Graph Representation

The graph is represented by the following edges and weights:

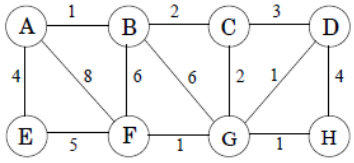
- A - B: 1
- B - C: 2
- C - D: 3
- D - H: 4
- C - G: 2
- G - D: 1
- E - F: 5
- G - H: 1
- F - G: 1
- A - F: 8
- A - E: 4
- B - F: 6
- B - G: 6

Prim's Algorithm

Start Prim's algorithm from node A and choose edges based on alphabetic ordering when there is a tie.

Graph Representation

Assume the following graph:



Code Implementation

```
import heapq

def prim(graph, start):
    mst = []
    visited = set()
    min_heap = [(0, start)]  # (cost, node)

    while min_heap:
        cost, u = heapq.heappop(min_heap)
        if u not in visited:
            visited.add(u)
            mst.append((cost, u))

            for v, weight in graph[u]:
                if v not in visited:
                    heapq.heappush(min_heap, (weight, v))

    return mst
```

Cost Array Table

Vertex in Graph	Edge in Graph	Cost
A	-	0
B	(A,B)	0+1=1
C	(B,C)	1+2=3
D	(C,G)	3+2=5
E	(G,D)	5+1=6
F	(G,F)	6+1=7
G	(G,H)	7+1=8
H	(E,A)	8+4=12

Relaxation Steps

During each step, we relax the edges as follows:

- Step 1: Relax edges from A: update costs for B.
- Step 2: Relax edges from B: update costs for C.
- Step 3: Relax edges from C: update costs for D.
- Step 4: Relax edges from D: update costs for E.
- Step 5: Relax edges from E: update costs for F.
- Step 6: Relax edges from G: update costs for H.

Kruskal's Algorithm Steps

The minimum spanning tree (MST) will be constructed using Kruskal's algorithm to the same graph, using a disjoint-set data structure with path compression.The following steps outline the process:

1. Sort all edges in non-decreasing order of their weights:
    - A - B: 1
    - G - D: 1
    - G - H: 1
    - F - G: 1
    - B - C: 2
    - C - G: 2
    - C - D: 3
    - D - H: 4
    - A - E: 4
    - E - F: 5
    - B - F: 6
    - B - G: 6
    - A - F: 8
2. Initialize the disjoint-set (union-find) structure:
    - A, B, C, D, E, F, G, H are all in separate sets.
3. Process each edge:

Node	Edge Added	Cost
A	-	0
B	(A,B)	$0 + 1 = 1$
C	(B,C)	$1 + 2 = 3$
D	(C,G)	$3 + 2 = 5$
E	(G,D)	$5 + 1 = 6$
F	(G,F)	$6 + 1 = 7$
G	(G,H)	$7 + 1 = 8$
H	(E,A)	$8 + 4 = 12$

Final Disjoint-Sets Structure

The disjoint sets after processing each edge are as follows:

1. After adding (A, B): A, B, C, D, E, F, G, H
2. After adding (B, C): A, B, C, D, E, F, G, H
3. After adding (C, G): A, B, C, D, E, F, G
4. After adding (G, D): A, B, C, D, G, E, F, H
5. After adding (G, F): A, B, C, D, G, F, E, H
6. After adding (G, H): A, B, C, D, G, F, H, E
7. After adding (E, A): A, B, C, D, G, F, H, E

Code Implementation

```
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])  # Path compression
        return self.parent[u]

    def union(self, u, v):
        root_u = self.find(u)
        root_v = self.find(v)
        if root_u != root_v:
            if self.rank[root_u] > self.rank[root_v]:
                self.parent[root_v] = root_u
            elif self.rank[root_u] < self.rank[root_v]:
```

```
                self.parent[root_u] = root_v
            else:
                self.parent[root_v] = root_u
                self.rank[root_u] += 1

def kruskal(edges, num_nodes):
    ds = DisjointSet(num_nodes)
    mst = []
    for u, v, weight in sorted(edges, key=lambda x: x[2]):
        if ds.find(u) != ds.find(v):
            ds.union(u, v)
            mst.append((u, v, weight))
    return mst
```

Conclusion

The total cost of the minimum spanning tree using Kruskal's algorithm is 12.

5.13 Huffman encoding

To solve this problem, use the following steps to construct the Huffman tree and determine the encoding for each character.

List the Characters and Frequencies

given the following characters and their frequencies:

- A: 31%
- C: 20%
- G: 9%
- T: 40%

Build the Huffman Tree

First, arrange the characters in ascending order of frequency:

$$\text{G (9\%)},\ \text{C (20\%)},\ \text{A (31\%)},\ \text{T (40\%)}$$

Combine the two lowest frequencies (G and C):

$$G + C = 9\% + 20\% = 29\%$$

Next, combine the new node (29%) with the next smallest frequency (A):

$$(G + C) + A = 29\% + 31\% = 60\%$$

Finally, combine the new node (60%) with the remaining character (T):

$$((G + C) + A) + T = 60\% + 40\% = 100\%$$

Assign Codes to the Characters

The Huffman tree structure allows us to assign binary codes:

- T: 1 (since it was combined last)
- A: 01
- C: 001
- G: 000

Final Encoding

Thus, the Huffman encoding for the characters is:

- A: 01
- C: 001
- G: 000
- T: 1

Using the Huffman encoding algorithm, the characters A, C, G, and T are assigned the binary codes 01, 001, 000, and 1, respectively.