Lecture 28

Doyoun Kim

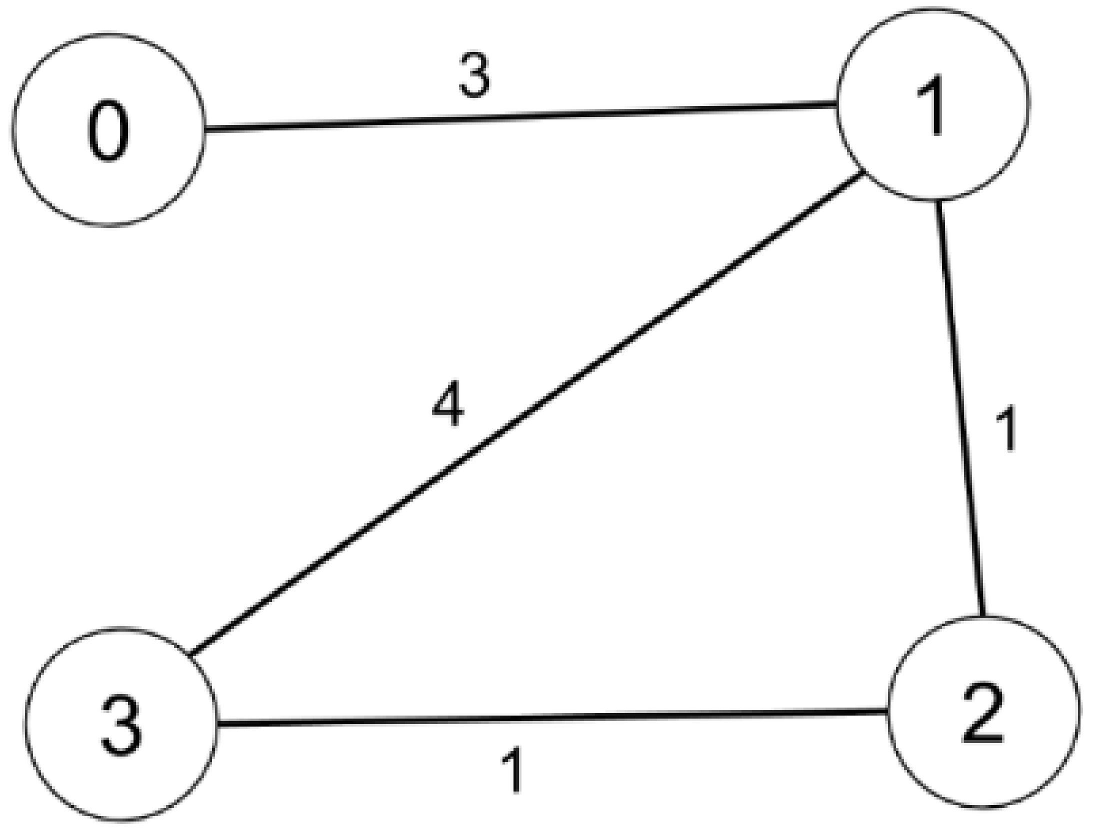
Overview

A goal of this exercise is to practice implementing specific functions in C++ to solve algorithmic problems.

In this exercise, you will implement three functions using SSSP and APSP.

- **Find the hub**: Find the hub by implementing a function that identifies the city with the maximum number of reachable cities within a given transportation cost threshold, using single-source and all-pairs shortest path algorithms.

Find the hub




```
Input: n = 4, edges = [[0,1,3],[1,2,1],[1,3,4],[2,3,1]], distanceThreshold = 4
Output: 3
Explanation: The figure above describes the graph. The neighboring cities at a
distanceThreshold = 4 for each city are:
City 0 -> [City 1, City 2]
City 1 -> [City 0, City 2, City 3]
City 2 -> [City 0, City 1, City 3]
City 3 -> [City 1, City 2]
Cities 1 and 2 have 3 neighboring cities at a distanceThreshold = 4, but we
have to return city 2 since it has the greatest number.
```

Find the hub

- **Goal**:
  - Implement two different algorithms to solve the same problem: **Floyd-Warshall** and **Bellman-Ford.**

- **Input**
  - **n**: The number of cities in the network, numbered from 0 to n-1.
  - **edges**: A list of arrays, where each array represents a bidirectional transportation route: edges[i] = [from_i, to_i, weight_i]

    - from_i: The starting city of the route.

    - to_i: The destination city of the route.

    - weight_i: The transportation cost for this route.
  - **distanceThreshold**: The maximum transportation cost allowed when determining reachable cities.

Find the hub

- **Output**
  - Return the numerical index of the city that satisfies the conditions:

1. Maximizes the number of reachable cities within the distance threshold.
2. In the event of a tie, choose the city with the largest numerical index.

Run exercise.cpp

- Find the TODO sections in exercise.cpp files and implement them correctly based on the instructions.
- There are 2 TODOs.
- $ g++ exercise.cpp -o exercise -std=c++11
- $ ./exercise

```
Test Case 1 (Floyd Warshall) Output: 2
Test Case 1 (Bellman Ford) Output: 2
Test Case 2 (Floyd Warshall) Output: 4
Test Case 2 (Bellman Ford) Output: 4
Test Case 3 (Floyd Warshall) Output: 2
Test Case 3 (Bellman Ford) Output: 2
Test Case 4 (Floyd Warshall) Output: 3
Test Case 4 (Bellman Ford) Output: 3
Test Case 5 (Floyd Warshall) Output: 0
Test Case 5 (Bellman Ford) Output: 0
Test Case 6 (Floyd Warshall) Output: 5
Test Case 6 (Bellman Ford) Output: 5
Test Case 7 (Floyd Warshall) Output: 12
Test Case 7 (Bellman Ford) Output: 12
```

Pseudo code – TODO 1

```
FUNCTION floydWarshall(n, edges, distanceThreshold):
    # Step 1: Initialize distance matrix with large values (∞)
    CREATE floydAdj[n][n]
    FOR i FROM 0 TO n-1:
        FOR j FROM 0 TO n-1:
            IF i == j:
                floydAdj[i][j] = 0   # Distance to self is 0
            ELSE:
                floydAdj[i][j] = ∞  # Default to infinity

    # Step 2: Fill matrix with edge distances
    FOR edge IN edges:
        u, v, weight = edge
        floydAdj[u][v] = weight
        floydAdj[v][u] = weight   # Since it's undirected
```

Pseudo code – TODO 1

```
# Step 3: Update distances using Floyd-Warshall
FOR k FROM 0 TO n-1:
    FOR i FROM 0 TO n-1:
        FOR j FROM 0 TO n-1:
            floydAdj[i][j] = MIN(floydAdj[i][j], floydAdj[i][k] + floydAdj[k][j])
```

Pseudo code – TODO 1

```
# Step 4: Find the city with the most reachable cities
  max_count = 0
  ans_city = -1

  FOR i FROM n-1 DOWNTO 0:
    reachable_count = 0
    FOR distance IN floydAdj[i]:
      IF distance <= distanceThreshold:
        reachable_count += 1

    # Update if this city reaches more cities
    IF reachable_count > max_count:
      max_count = reachable_count
      ans_city = i
```

```
# Step 5: Return the result
  RETURN ans_city
```

Solutions – TODO 1

```
int floydWarshall (int n, std::vector<std::vector<int>>& edges, int distanceThreshold) {
    std::vector<std::vector<int>> floydAdj(n, std::vector<int>(n, 1000001));

    for(int i = 0; i < n; i++) {
        floydAdj[i][i] = 0;
    }

    for(const auto& edge: edges) {
        floydAdj[edge[0]][edge[1]] = edge[2];
        floydAdj[edge[1]][edge[0]] = edge[2];
    }

    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                floydAdj[i][j] = std::min(floydAdj[i][j], floydAdj[i][k] + floydAdj[k][j]);
            }
        }
    }
```

Distance to self is 0

Step 1: Initialize distance matrix with large values (∞)

Solutions – TODO 1

```
int floydWarshall (int n, std::vector<std::vector<int>>& edges, int distanceThreshold) {
    std::vector<std::vector<int>> floydAdj(n, std::vector<int>(n, 1000001));

    for(int i = 0; i < n; i++) {
        floydAdj[i][i] = 0;
    }
```

Step 2: Fill matrix with edge distances

```
    for(const auto& edge: edges) {
        floydAdj[edge[0]][edge[1]] = edge[2];
        floydAdj[edge[1]][edge[0]] = edge[2];   # Since it's undirected
    }

    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                floydAdj[i][j] = std::min(floydAdj[i][j], floydAdj[i][k] + floydAdj[k][j]);
            }
        }
    }
```

Solutions – TODO 1

```
int floydWarshall (int n, std::vector<std::vector<int>>& edges, int distanceThreshold) {
    std::vector<std::vector<int>> floydAdj(n, std::vector<int>(n, 1000001));

    for(int i = 0; i < n; i++) {
        floydAdj[i][i] = 0;
    }

    for(const auto& edge: edges) {
        floydAdj[edge[0]][edge[1]] = edge[2];
        floydAdj[edge[1]][edge[0]] = edge[2];
    }
```

Step 3: Update distances using Floyd-Warshall

```
    for(int k = 0; k < n; k++) {
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                floydAdj[i][j] = std::min(floydAdj[i][j], floydAdj[i][k] + floydAdj[k][j]);
            }
        }
    }
```

Solutions – TODO 1

Step 4: Find the city with the most reachable cities

```
    int max_count = 0, ans_city;
    for(int i = n - 1; i >= 0; i--) {
        int count = 0;
        for(int d: floydAdj[i])
            if(d <= distanceThreshold) {
                count++;
            }

        if(count > max_count) {
            max_count = count;
            ans_city = i;
        }
    }
    return ans_city;
}
```

Pseudo code – TODO 2

```
FUNCTION bellmanFord(n, edges, distanceThreshold):
    max_count = 0
    ans_city = -1
    FOR src FROM 0 TO n-1:
        dist = [∞] * n      # the dist array is reset for each source
        dist[src] = 0
```

Pseudo code – TODO 2

```
FUNCTION bellmanFord(n, edges, distanceThreshold):
    max_count = 0
    ans_city = -1
    FOR src FROM 0 TO n-1:
        dist = [∞] * n   # the dist array is reset for each source
        dist[src] = 0

        # Step 1: Relax all edges (n-1 times)
        FOR i FROM 0 TO n-2:
            FOR edge IN edges:
                u, v, weight = edge
                dist[v] = MIN(dist[v], dist[u] + weight)
                dist[u] = MIN(dist[u], dist[v] + weight)    # Since it's undirected
```

Pseudo code – TODO 2

```
FOR src FROM 0 TO n-1:
    dist = [∞] * n
    dist[src] = 0
    # Step 1: Relax all edges (n-1 times)

    # Step 2: Count reachable cities
    reachable_count = 0
    FOR d IN dist:
        IF d <= distanceThreshold:
            reachable_count += 1

    # Step 3: Update the best city
        IF reachable_count > max_count OR (reachable_count == max_count AND src > ans_city):
            max_count = reachable_count
            ans_city = src

    Return ans_city
```

Solutions – TODO 2

```cpp
int bellmanFord(int n, std::vector<std::vector<int>>& edges, int distanceThreshold) {
    int max_count = 0, ans_city = -1;

    for (int src = 0; src < n; src++) {
        std::vector<int> dist(n, 1000001);
        dist[src] = 0;

        for (int i = 0; i < n - 1; ++i) {
            for (const auto& edge : edges) {
                int u = edge[0], v = edge[1], weight = edge[2];
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                }
                if (dist[v] + weight < dist[u]) {
                    dist[u] = dist[v] + weight;
                }
            }
        }
```

Set each city as the starting point.

Solutions – TODO 2

```
int bellmanFord(int n, std::vector<std::vector<int>>& edges, int distanceThreshold) {
    int max_count = 0, ans_city = -1;

    for (int src = 0; src < n; src++) {
        std::vector<int> dist(n, 1000001);
        dist[src] = 0;

        for (int i = 0; i < n - 1; ++i) {
            for (const auto& edge : edges) {
                int u = edge[0], v = edge[1], weight = edge[2];
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                }
                if (dist[v] + weight < dist[u]) {
                    dist[u] = dist[v] + weight;
                }
            }
        }
```

Step 1: Relax all edges (n-1 times)

Solutions – TODO 2

Step 2: Count reachable cities

```
            int count = 0;
            for (int d : dist) {
                if (d <= distanceThreshold) {
                    count++;
                }
            }

            if (count > max_count || (count == max_count && src > ans_city)) {
                max_count = count;
                ans_city = src;
            }
        }

        return ans_city;
    }
```

Solutions – TODO 2

```
        int count = 0;
        for (int d : dist) {
            if (d <= distanceThreshold) {
                count++;
            }
        }
```

Step 3: Update the best city

```
        if (count > max_count || (count == max_count && src > ans_city)) {
            max_count = count;
            ans_city = src;
        }
    }

    return ans_city;
}
```

Thank you