CS 630, Fall 2024, Homework 8

Yifei Bao

December 7, 2024

Problem 1 ***Graph centrality***

1.

Algorithm 1: *ComputeDistance*$(G(V, E))$

```
for s ∈ V do
    /* Initialize the count of shortest paths from s to every vertex */
    for v ∈ V do
        σ_s[v] ← 0;
    σ_s[s] ← 1;
    /* Initialize distances and a queue for BFS */
    dist[s] ← 0;
    for v ∈ V, v ≠ s do
        dist[v] ← ∞;
    Q ← a new empty queue;
    Q.enqueue(s);
    while Q is not empty do
        u ← Q.dequeue();
        for w ∈ Adj[u] do
            if dist[w] = ∞ then
                dist[w] ← dist[u] + 1;
                Q.enqueue(w);
                σ_s[w] ← σ_s[u] /* First time reaching w, inherit count from u */
            else if dist[w] = dist[u] + 1 then
                σ_s[w] ← σ_s[w] + σ_s[u] /* Another shortest path found */
    for v ∈ V do
        P[s, v] ← σ_s[v];
return P, dist;
```

Time complexity. For a single source s, the time complexity is $O(|V| + |E|)$, because we perform a BFS traversal of the graph $G(V, E)$ starting from s. Since we have $|V|$ sources, the total time complexity is $O(|V|^2 + |V||E|)$.

Correctness. BFS ensures that the level structure represents the shortest path distances By counting all valid paths to the next level during BFS, we accurately compute the number of shortest paths $p(s, v)$.

2. Since *path* is a shortest path from s to t with total length k , if *path* $s \to v$ is not the shortest path from s to v , we could replace it with a shorter path, forming a path from s to t shorter than path . This contradicts the assumption that *path* is the shortest path.

3.

Algorithm 2: $ComputeBetweenness(G(V, E))$

```
1 P, dist ← ComputeDistance(s, G);
2 for v ∈ V do
3     B[v] ← 0;
  /* Use dist and P to compute B[v]                                    */
4 for v ∈ V do
5     for s ∈ V do
6         for t ∈ V, t ≠ s do
7             if dist[s, t] = dist[s, v] + dist[v, t] then
8                 B[v] ← B[v] + (P[s,v]·P[v,t]) / P[s,t];
9 return B;
```

Time complexity. The time complexity is $O(|V|^3)$.

Correctness. Based on the proof of 2, we know that If a vertex v is on path then path $s - v$ and path $v - t$ are shortest path between the respective vertices. So we only need to check if $dist(s, t) = dist(s, v) + dist(v, t)$ to determine if v is on the shortest path between s and t.

Problem 2 ***Centrality using an adjacency matrix***

1. The basic idea is to use the adjacency matrix to compute the number of shortest paths of length p between every pair of vertices. We can use the matrix multiplication to compute the number of paths of length p between every pair of vertices. We can then use the matrix to update the shortest path distances and the number of shortest paths between every pair of vertices.

Time complexity. The time complexity is $O(kn^3)$.

Correctness. The definition of the power of a matrix A^p is that the (i, j) entry of A^p is the number of paths of length p from vertex i to vertex j. We can use the matrix A^p to update the shortest path distances and the number of shortest paths between every pair of vertices. When $p = 1$, the matrix A is the adjacency matrix of the graph, and the (i, j) entry of A is 1 if there is an edge between vertex i and vertex j. When increasing p, if we did not find a shorter path, we can update the number of shortest paths between every pair of vertices. If it already exists a same length path, we can add the number of paths.

Algorithm 3: $ComputeShortestPathsByMatrix(A, n, k)$

```
1  Initialize dist[n][n], P[n][n];
2  for i = 1...n do
3      for j = 1...n do
4          if i == j then
5              dist[i][j] ← 0;
6              P[i][j] ← 1;
7          else if A[i][j] == 1 then
8              dist[i][j] ← 1;
9              P[i][j] ← 1;
10         else
11             dist[i][j] ← ∞;
12             P[i][j] ← 0;
13 for p = 2...k do
14     for i = 1...n do
15         B ← A^p/* Compute the matrix to get the number of paths of length p */
16         for i = 1...n do
17             for j = 1...n do
18                 if B[i][j] ≠ 0 then
19                     if dist[i][j] > p then
20                         dist[i][j] ← p;
21                         P[i][j] ← B[i][j];
22                     else if dist[i][j] == p then
23                         P[i][j] ← P[i][j] + B[i][j];
24 return dist, P;
```

2.

Algorithm 4: $ComputeBetweenessByMatrix(A, n, k)$

```
1  dist, count ← ComputeShortestPathsByMatrix(A, n, k);
2  for v ∈ V do
3      B[v] ← 0;
4  for v = 1 ... n do
5      for s = 1 ... n do
6          for t = 1 ... n do
7              if s! = t then
                   /* Check if v is on the shortest path between s and t          */
8                  if dist[s][t] == dist[s][v] + dist[v][t] then
9                      B[v] ← B[v] + (P[s,v]·P[v,t]) / P[s,t];
10 return B;
```

Time complexity. The time complexity is $O(n^3)$.

Correctness. Due to the correctness of the previous algorithm, we can get correct *dist* and P matrices. We can then use the matrices to compute the betweenness centrality of every vertex. We can use the matrices to check if a vertex v is on the shortest path between s and t, and the method is proved in problem 1.