Homework 8

Name: Colby Frison
OUID: 1135568816
Date: 4/7/2025
Class: CS-4413

Questions

1. Problem 22.1-7 ~ Incidence Matrix and BB^T interpretation

Incidence matrix

In analyzing the directed graph from Figure 22.2(a), we will examine the properties of its incidence matrix B and the resulting matrix BB^T.

To begin our analysis, we must first construct the incidence matrix B. The graph contains 6 vertices and 8 edges in total, relationships are shown below:

- $e_1 : 1 \to 2$
- $e_2 : 1 \to 4$
- $e_3 : 2 \to 5$
- $e_4 : 3 \to 5$
- $e_5 : 3 \to 6$
- $e_6 : 4 \to 2$
- $e_7 : 5 \to 4$

Total edges: 7

Each row(1-7) corresponds to a vertex, and each column(e1-e7) corresponds to an edge. The 6x7 matrix uses the following incidence rule:

Using the incidence rule:

- $b_{ij} = -1$ if edge j leaves vertex i
- $b_{ij} = 1$ if edge j enters vertex i
- $b_{ij} = 0$ otherwise

	e1	e2	e3	e4	e5	e6	e7
1	-1	-1	0	0	0	0	0
2	1	0	-1	0	0	1	0
3	0	0	0	-1	-1	0	0
4	0	1	0	0	0	-1	1
5	0	0	1	1	0	0	-1
6	0	0	0	0	1	0	0

What does BB^T represent

Let $A = BB^T$ where B is our incidence matrix. The resulting matrix A provides crucial information about the graph's structure. For any entry A_{ij} in A:

For diagonal entries $(i = j)$:

- A_{ii} equals the total degree of vertex i (both in-degree and out-degree)
- This is because when we multiply row i of B with column i of B^T, each edge incident to vertex i contributes exactly 1 to the sum:
  - For an outgoing edge: $(-1)^2 = 1$
  - For an incoming edge: $(1)^2 = 1$

For off-diagonal entries $(i \neq j)$:

- A_{ij} represents the relationship between vertices i and j
- When multiplying row i by column j:
  - If there's an edge from i to j: $(-1)(1) = -1$
  - If there's an edge from j to i: $(1)(-1) = -1$
  - If no edge exists between i and j: 0

The fundamental insight is that BB^T **creates a vertex-vertex matrix that reveals how vertices are connected through their shared edges, while preserving directional information.** While diagonal entries A_{ii} give us vertex degrees, the off-diagonal entries A_{ij} are the key feature, showing us the

relationships between pairs of vertices based on their edge connections and directions.

2. Problem 22.2-8 ~ Diameter of a Graph

The diameter of a tree $T = (V, E)$ is defined as the maximum distance between any two vertices in the tree, where distance is measured by the number of edges in the shortest path. We can find this using an application of BFS. The key insight is that we don't need to check every possible pair of vertices - we can find the diameter with just two BFS traversals.

Analysis

The solution works by using these steps:

1. Run first BFS:
    - Pick any starting vertex u
    - Run BFS from u to find the furthest vertex v
    - The vertex v will be one end of the diameter
2. Run second BFS:
    - Start from vertex v that we found
    - Run BFS again to find the furthest vertex w
    - The path from v to w gives us the diameter

This works because in a tree, the diameter must be a path between two leaf nodes. When we run the first BFS, we're guaranteed to find one end of the diameter path. The second BFS then finds the other end.

Algorithm Description

The algorithm efficiently finds the diameter using two BFS traversals:

1. **First BFS**: Start BFS from an arbitrary node u. Find the node v that is farthest away from u. This node v is guaranteed to be one endpoint of *some* diameter of the tree.
2. **Second BFS**: Start another BFS, this time from the node v identified in the first step. Find the node w that is farthest away from v. The distance between v and w is the diameter of the tree.

Pseudocode

```
function BFS(graph, start_node):
    // Initialize distances and parent pointers
    distances = map of node -> infinity
    parents = map of node -> null
    distances[start_node] = 0

    // Initialize queue for BFS
    queue = new Queue()
    queue.enqueue(start_node)

    // Keep track of the farthest node found so far
    farthest_node = start_node
    max_distance = 0

    while queue is not empty:
        current_node = queue.dequeue()

        // Update farthest node if current node is farther
        if distances[current_node] > max_distance:
            max_distance = distances[current_node]
            farthest_node = current_node

        // Explore neighbors
        for neighbor in graph.neighbors(current_node):
            if distances[neighbor] == infinity: // If neighbor not visited
                distances[neighbor] = distances[current_node] + 1
                parents[neighbor] = current_node
                queue.enqueue(neighbor)

    return farthest_node, max_distance

function findTreeDiameter(tree):
    // 1. Pick an arbitrary starting node
    arbitrary_node = tree.getAnyNode()

    // 2. Run first BFS to find one endpoint of the diameter
    endpoint1, _ = BFS(tree, arbitrary_node)

    // 3. Run second BFS from the first endpoint to find the other endpoint and the diameter
    endpoint2, diameter = BFS(tree, endpoint1)

    return diameter
```

Time Complexity

The algorithm is efficient because:

- Each BFS only needs to visit each vertex once
- In a tree, $|E| = |V| - 1$ (number of edges is one less than vertices)
- We do two BFS traversals, each taking $O(V)$ time
- Total runtime: $O(V)$

This is optimal since we need to at least look at each vertex once to find the diameter.

3. Shortest Path from a to p

Using BFS

Breadth-First Search (BFS) is ideal for finding the shortest path in an unweighted graph. It explores vertices layer by layer, ensuring the shortest path is found first.

BFS Steps:

1. Start at vertex a.
2. Use a queue to explore vertices by distance from a.
3. Track parents to reconstruct the path.
4. Stop when vertex p is reached.

Execution Summary:

- Begin with a, marking it as visited.
- Explore neighbors, updating distances and parents.
- Continue until p is reached.
- Reconstruct the path using parent pointers.

Result:

- Shortest path: $a \to b \to d \to g \to k \to n \to p$
- Path length: 6 edges

Using Dijkstra's Algorithm

Dijkstra's algorithm is used for graphs with non-negative weights. Here, it behaves like BFS since all weights are 1.

Dijkstra's Steps:

1. Start at vertex a.
2. Use a priority queue to explore vertices by increasing distance.
3. Track parents for path reconstruction.
4. Stop when vertex p is reached.

Execution Summary:

- Initialize distances and priority queue with a.
- Explore neighbors, updating distances and parents.
- Continue until p is reached.
- Reconstruct the path using parent pointers.

Result:

- Shortest path: $a \to b \to d \to g \to k \to n \to p$
- Path length: 6 edges

Conclusion

Both BFS and Dijkstra's algorithm yield the same shortest path from a to p with a total distance of 6 edges. This is expected since all edge weights are 1. Other paths of the same length exist, but the one provided is valid.