CS 344 Problem Set 3: Graphs - Structure and Connectivity

Kanghwi Lee

Question 1:

(a) G is a bi-partite graph.

A graph G = (V,E) is a bi-partite graph if its vertex set V can be split into two disjoint groups L and R such that all edges have precisely one endpoint in each of these two sets (i.e., $E \subseteq L \times R$).

(b) The vertices of G can be colored using only two colors such that every edge is between vertices of opposite colors.

If a graph G can be colored using only two colors, say black and white, so that every edge connects vertices of opposite colors, then G is a bi-partite graph. The two colors can correspond to the two disjoint groups, L and R mentioned in (a).

(c) G has no odd-length cycles.

A graph G has no odd-length cycles if and only if it can be divided into two disjoint sets of vertices L and R, such that all edges in the graph connect vertices between these two sets. This is because if there were an odd-length cycle, it would not be possible to divide the vertices into two sets such that all edges connect vertices of opposite sets, violating the definition of a bi-partite graph.

Now, let's prove the equivalence:

(a) ⇔ (b):

If G is a bi-partite graph, we can easily color the vertices using two colors. Start by selecting an arbitrary vertex as the starting point and assign it one color, say black. Then, for each neighbor of the black vertices, assign the other color, white. Continue this process recursively until all vertices are colored, ensuring that every edge connects vertices of opposite colors. Hence, (a) implies (b).

Conversely, we can color the vertices of G using two colors such that every edge connects vertices of opposite colors. In that case, we can define the sets L and R as follows: L contains all the black-colored vertices, and R contains all the white-colored vertices. Since every edge connects vertices of opposite colors, all edges will have one endpoint in L and the other in R, satisfying the definition of a bi-partite graph. Hence, (b) implies (a).

(a) ⇔ (c):

As discussed earlier, if G is a bi-partite graph, it has no odd-length cycles. This is because odd-length cycles would require edges to connect vertices within the same set, which contradicts the definition of a bi-partite graph. Therefore, (a) implies (c).

On the other hand, if G has no odd-length cycles, we can divide its vertices into two sets L and R, such that all edges connect vertices between these two sets. This arrangement satisfies the definition of a bi-partite graph. So, (c) implies (a).

Linear Time Algorithm to Determine Whether a Graph is Bi-partite:

We can use a depth-first search (DFS) based algorithm to determine if a given graph is bi-partite. The algorithm maintains a color assignment for each vertex and tries to color the vertices such that no adjacent vertices have the same color. If it successfully colors all vertices, the graph is bi-partite; otherwise, it is not.

Step-by-step outline of the algorithm:

1. Initialize an empty dictionary (or array) to store the color assignment of each vertex. Let's call it "color."
2. Start DFS from an arbitrary vertex in the graph.
3. During the DFS traversal, color the current vertex alternately with black and white and assign the color to the "color" dictionary for that vertex.
4. For each neighbor of the current vertex, check if it already has a color assigned. If yes, make sure the color differs from the current vertex's color. If not, assign the opposite color to the neighbor and continue DFS from that neighbor.
5. If, during the DFS traversal, you encounter a vertex whose neighbor already has the same color, it means the graph is not bi-partite. Return "False."
6. If the DFS traversal completes without conflicts, i.e., all vertices are successfully colored, the graph is bi-partite. Return "True."

The DFS-based algorithm runs in linear time, O(|V| + |E|), where |V| is the number of vertices, and |E| is the number of edges in the graph. This is because each vertex and each edge is visited only once during the traversal.

Question 2:

(a) Ancestor Queries:

We can use a linear-time preprocessing algorithm called Euler tour and LCA (Lowest Common Ancestor) to answer ancestor queries efficiently. The Euler tour is a tree traversal that visits each vertex twice, once when entering the subtree and once when leaving the subtree.

Step 1: Euler Tour Traversal and Depth Recording Perform an Euler tour traversal of the binary tree T starting from the root r. During the traversal, keep track of the depth of each visited vertex. Record the first occurrence of each vertex during the traversal as its entry time, and record the second occurrence as its exit time.

Step 2: Setup Sparse Table (RMQ) for LCA Using the depth and entry time information obtained in Step 1, construct a sparse table (also known as RMQ - Range Minimum Query) to efficiently find the Lowest Common Ancestor (LCA) of any two vertices in O(1) time after O(n log n) preprocessing time. The sparse table allows us to find the LCA of two vertices in constant time using O(log n) memory.

Step 3: Answering Ancestor Queries Now that we have the LCA data structure, answering ancestor queries is straightforward. Given a pair (x, y), where x is a potential ancestor of y, we find the LCA of x and y using the sparse table. If the LCA is x, then x is an ancestor of y, and we return "YES". Otherwise, we return "NO".

The preprocessing step (Step 1 and Step 2) takes O(n log n) time due to the construction of the sparse table, but after that, answering each ancestor query (Step 3) can be done in O(1) time, making it an overall linear-time algorithm.

(b) Calculate Array A for Maximum Descendant Value:

We can use a depth-first traversal of the binary tree to calculate the array A where A[v1] = a(v1), considering v1 as the root vertex.

Step 1: Perform Depth-First Traversal Start a depth-first traversal of the binary tree T from the root v1. During the traversal, visit each vertex v, and for each vertex, do the following:

1. Calculate the maximum s-value among its children (if any).
2. Set the value a(v) to be the maximum of its own s-value and the maximum s-value among its children.

Step 2: Recursive Implementation The depth-first traversal can be implemented using recursion. When visiting a vertex v, the function will first visit its left child (if it exists), then visit its right child (if it exists), and finally calculate a(v) based on the information gathered from its children.

Step 3: Calculate A[v1] After completing the depth-first traversal, we will have calculated the value of a(v) for each vertex v in the binary tree. The array A will be fully populated with the maximum descendant values.

The linear-time complexity of this algorithm arises from the fact that each vertex is visited only once during the depth-first traversal, and the calculations for each vertex take constant time.

This algorithm runs in linear time, O(n), where n is the number of vertices in the binary tree.

Question 3:

(a) Time Complexity:

Let's analyze the time complexity of this "fixed" Dijkstra's algorithm with the addition of the largest edge weight.

1. Finding the weight of the largest edge (max): This step involves traversing through all the edges in the graph once to find the maximum edge weight. The time complexity for this step is O(E), where E is the number of edges in the graph.
2. Adding max to all edge weights: After finding the maximum edge weight, we need to add it to all edge weights to make them non-negative. Again, this requires traversing through all the edges, and the time complexity for this step is also O(E).
3. Dijkstra's algorithm with a binary heap: The time complexity of Dijkstra's algorithm using a binary heap for priority queue is O((V + E) log V). Since we already adjusted the edge weights in the previous steps, this is the same as the standard Dijkstra's time complexity.

Combining all the steps, the overall time complexity of this "fixed" Dijkstra's algorithm is O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph.

(b) Validity of the Fix:

The "fix" suggested by adding the maximum edge weight to all edge weights to make them non-negative does not affect the relative distances between vertices in the graph. The fix ensures that all edge weights become positive, and this does not change the shortest paths between vertices.

Proof: Let's consider the original weight of an edge e as w(e) and the maximum edge weight in the graph as max. After applying the "fix," the new weight of edge e (denoted as w'(e)) will be:

$w'(e) = w(e) + \max$

When Dijkstra's algorithm runs on these new weights, all the shortest paths are found based on the modified weights. The shortest path distance d(s, v) from the source vertex s to any other vertex v in the original graph was:

$d(s, v) = \min \{ d(s, u) + w(u, v) \}$ for all edges (u, v) in the original graph

After the fix, the shortest path distance d'(s, v) using the modified weights becomes:

$d'(s, v) = \min \{ d(s, u) + w'(u, v) \}$ for all edges (u, v) in the modified graph

Substituting the value of w'(u, v) into the above equation:

$d'(s, v) = \min \{ d(s, u) + (w(u, v) + \max) \}$ for all edges (u, v) in the original graph

But since max is added to all the weights uniformly, it does not affect the relative order of distances between vertices. So, the shortest path distances remain the same even after applying the "fix."

Therefore, this "fix" results in finding the shortest paths from vertex s to all other vertices. It is important to note that this fix only applies to graphs with negative edge weights and does not work for graphs with negative cycles, as Dijkstra's algorithm does not handle negative cycles.

Question 4:

(a) Algorithm to determine whether the car can travel from city s to city t:

To determine whether the car can travel from city s to city t, we can use a breadth-first search (BFS) algorithm. The BFS algorithm efficiently explores all possible paths from city s to city t while keeping track of the remaining fuel capacity in the tank. The algorithm will terminate as soon as it finds a valid path or exhausts all possible paths without finding a valid one.

1. Create a queue to perform BFS.
2. Initialize a boolean array visited[V] to keep track of visited cities. Initially, set all elements to false.
3. Enqueue the starting city s into the queue and mark it as visited.
4. Start a while loop that continues until the queue is empty.
5. Dequeue a city u from the queue.
6. Check if u is the destination city t. If true, return true (there exists a path from s to t).
7. Iterate through all cities v connected to u by highways.
8. Check if the length of the highway luv is less than or equal to the remaining fuel capacity L. If true and v is not visited, enqueue v into the queue and mark it as visited.
9. Repeat steps 5 to 8 until the queue is empty or the destination city t is found.
10. If the destination city t is not found after the BFS, return false (there is no valid path from s to t).

The time complexity of this algorithm is linear, O(V + E), since it explores each city and each highway exactly once.

(b) Algorithm to compute the minimum tank capacity needed to travel from city s to city t:

To find the minimum tank capacity needed to travel from city s to city t, we can use a binary search combined with a modified version of the BFS algorithm.

1. Find the maximum highway length, L_max, in the given graph. This can be done by iterating through all the highways and finding the maximum value.
2. Initialize the minimum tank capacity, min_capacity, as 0 and the maximum tank capacity, max_capacity, as L_max.
3. Perform a binary search between min_capacity and max_capacity.
4. For each mid_capacity, perform the modified BFS algorithm with a slight modification: Whenever the BFS reaches a city v connected to city u with highway length luv > mid_capacity, skip that highway and do not enqueue v. The rest of the BFS remains the same as in the previous algorithm.
5. If the BFS reaches city t at any step during the binary search, update min_capacity to mid_capacity and set max_capacity to mid_capacity - 1, to search for a smaller tank capacity.
6. If the BFS does not reach city t, set min_capacity to mid_capacity + 1, to search for a larger tank capacity.
7. Repeat steps 3 to 6 until min_capacity is greater than max_capacity.
8. Return the final value of min_capacity as the minimum tank capacity needed to travel from city s to city t.

The time complexity of this algorithm is O((V + E) * log(L_max)), where V is the number of cities and E is the number of highways. The binary search reduces the number of iterations required to efficiently find the minimum tank capacity.

Question 5:

To compute the number of distinct shortest paths from vertex s to vertex t in an undirected graph with all edge weights equal to 1, we can use a variation of Breadth-First Search (BFS) algorithm. The algorithm will have a linear time complexity, O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Let's outline the steps of the algorithm:

1. Initialize variables:

- A queue to store nodes to be processed in BFS traversal.
- A dictionary to store the number of distinct shortest paths for each vertex. Initialize all values to 0, except for the source vertex s, which is set to 1 to indicate that there is one distinct shortest path from s to itself.
- A set to keep track of visited nodes to avoid processing the same node multiple times.

2. Perform BFS traversal starting from vertex s:

- Add vertex s to the queue and mark it as visited.
- While the queue is not empty, do the following:
    - Pop the front vertex from the queue.
    - Explore all adjacent vertices of the current vertex.
    - If an adjacent vertex has not been visited yet, add it to the queue, mark it as visited, and update its number of distinct shortest paths by adding the number of distinct shortest paths to the current vertex.
    - If an adjacent vertex has been visited, update its number of distinct shortest paths by adding the number of distinct shortest paths to the current vertex.

3. The algorithm finishes when the queue becomes empty, and by this point, the number of distinct shortest paths to each vertex is correctly computed.

4. The final result will be the value in the dictionary corresponding to vertex t, which represents the number of distinct shortest paths from s to t.

The algorithm explores the graph breadth-first, starting from the source vertex s. It counts the number of distinct shortest paths to each vertex, propagating the information as it traverses the graph. Since the graph has all edge weights equal to 1, BFS guarantees that the paths found are indeed the shortest. The algorithm's time complexity is O(V + E) because each vertex and edge are visited once during the traversal.