CSCI 2300: Introduction to Algorithms

Homework 1

Lucien Brule
Prof. Bulent Yener
April 27, 2023

1 Problem 1

Problem: An undirected graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. Design a linear time, i.e., O(—V— + —E—), time algorithm to check if a graph is bipartite or not.

Solution:

We can use Breadth-First Search (BFS) to check if a graph is bipartite. The algorithm starts from an arbitrary vertex and assigns it a color (let's say, 0). Then, we visit all its neighbors and assign them the opposite color (1). We continue this process, alternating colors while visiting the graph using BFS. If we ever encounter a situation where a vertex is visited twice with different colors, the graph is not bipartite. Otherwise, the graph is bipartite.

Here is the pseudo code for the algorithm:

```
function is_bipartite(graph):
    n = len(graph)
    colors = [-1] * n

    for vertex in range(n):
        if colors[vertex] == -1:
            if not bfs(graph, vertex, colors):
                return False

    return True

function bfs(graph, start, colors):
    queue = [start]
    colors[start] = 0

    while queue:
        current = queue.pop(0)
        for neighbor in graph[current]:
            if colors[neighbor] == -1:
                colors[neighbor] = 1 - colors[current]
                queue.append(neighbor)
            elif colors[neighbor] == colors[current]:
                return False
```

```
    return True
```

This algorithm has a time complexity of O(—V— + —E—) as BFS traverses each vertex and edge once.

2 Problem 2

Part A:

Problem: Prove that a non-empty DAG (Directed Acyclic Graph) must have at least one source.

Proof:

Assume that there is a non-empty DAG without a source. In this case, every vertex has at least one incoming edge. Start at any vertex, and follow the incoming edges, creating a path. Since there are a finite number of vertices, we must eventually reach a vertex that we have already visited. This would create a cycle in the graph, which contradicts the assumption that the graph is acyclic. Therefore, a non-empty DAG must have at least one source.

Part B:

Problem: What is the time complexity of finding a source in a directed graph or to determine such a source does not exist if the graph is represented by its adjacency matrix? Describe the algorithm.

Solution:

The algorithm for finding a source in a directed graph represented by its adjacency matrix is as follows:

1. Iterate through the columns of the adjacency matrix. 2. For each column, check if all its elements are 0. If so, the corresponding vertex is a source.

Here's the pseudo code for the algorithm:

```
function find_source(matrix):
    for column in range(len(matrix)):
        if all(matrix[row][column] == 0 for row in range(len(matrix))):
            return column

    return None
```

The time complexity of this algorithm is $O(|V|^2)$, as we need to iterate through each element of the adjacency matrix.

Part C:

Problem: What is the time complexity of finding a source in a directed graph or to determine such a source does not exist if the graph is represented by its adjacency list? Describe the algorithm.

Solution:

The algorithm for finding a source in a directed graph represented by its adjacency list is as follows:

1. Create an array with a length equal to the number of vertices, initializing all values to 0. 2. Iterate through the adjacency list and increment the corresponding entry in the array for each incoming edge. 3. Iterate through the array and find the first entry with a value of 0.

Here's the pseudo code for the algorithm:

```
function find_source(adj_list):
    in_degree = [0] * len(adj_list)

    for vertex in adj_list:
        for neighbor in adj_list[vertex]:
            in_degree[neighbor] += 1

    for vertex in range(len(in_degree)):
        if in_degree[vertex] == 0:
            return vertex

    return None
```

The time complexity of this algorithm is $O(|V| + |E|)$, as we need to iterate through the adjacency list and then through the $\text{in}_d egree array$.

3 Problem 3

Problem: Describe a linear time algorithm to compute the neighbor degree for each vertex in an undirected graph. The neighbor degree of a node x is defined as the sum of the degree of all of its neighbors.

Solution:

To compute the neighbor degree for each vertex in an undirected graph, we can follow the algorithm below:

Algorithm: Initialize an array 'degree' of length equal to the number of vertices in the graph to store the degree of each vertex. Initialize an array 'neighbor$_d$egree'$of length equal to the number of vertices in the graph to store the neighbor degree of each ver$

Here's the pseudo code for the algorithm:

```
function compute_neighbor_degree(adj_list):
    n = len(adj_list)
    degree = [0] * n
    neighbor_degree = [0] * n

    for vertex in range(n):
        degree[vertex] = len(adj_list[vertex])

    for vertex in range(n):
        for neighbor in adj_list[vertex]:
            neighbor_degree[vertex] += degree[neighbor]

    return neighbor_degree
```

This algorithm has a time complexity of $O(|V| + |E|)$, as we iterate through the adjacency list twice.

4 Problem 4

Part A:

Problem: Assume the graph is a DAG (Directed Acyclic Graph). Describe a linear time algorithm to compute the reachability weight for all vertices.

Solution:

For a DAG, we can use a topological sorting to find the reachability weight for all vertices. The algorithm is as follows:

1. Perform a topological sort on the DAG. 2. Initialize an array 'reachability$_w$eight'$of length equal to$

Here's the pseudo code for the algorithm:

```
function compute_reachability_weight_DAG(adj_list, weight):
    sorted_vertices = topological_sort(adj_list)
    n = len(adj_list)
    reachability_weight = [0] * n
```

```
    for vertex in reversed(sorted_vertices):
        max_neighbor_weight = 0
        for neighbor in adj_list[vertex]:
            max_neighbor_weight = max(max_neighbor_weight, reachability_weight[neighb
        reachability_weight[vertex] = weight[vertex] + max_neighbor_weight

    return reachability_weight
```

This algorithm has a time complexity of $O(|V| + |E|)$, as the topological sort takes linear time and we iterate through the adjacency list once.

Part B:

Problem: Assume that the graph is a general directed graph (with possible cycles). Describe a linear time algorithm to find the reachability weight for all vertices.

Solution:

For a general directed graph, we can use the strongly connected components (SCC) decomposition to find the reachability weight for all vertices. The algorithm is as follows:

1. Compute the strongly connected components of the graph using Tarjan's algorithm or Kosaraju's algorithm. 2. For each strongly connected component, compute the maximum weight among its vertices and replace the weight of all vertices in the component with the maximum weight. 3. Create a new DAG by collapsing each strongly connected component into a single vertex and removing self-loops and duplicate edges. 4. Apply the algorithm for DAGs (from Part A) to compute the reachability weight for all vertices in the new DAG.

Here's the pseudo code for the algorithm:

```
function compute_reachability_weight_general(adj_list, weight):
    scc = strongly_connected_components(adj_list)
    n = len(adj_list)

    # Replace the weight of each vertex with the maximum weight in its SCC
    for component in scc:
        max_weight = max(weight[vertex] for vertex in component)
        for vertex in component:
            weight[vertex] = max_weight

    # Create a new DAG from the original graph
```

```
    new_adj_list = collapse_scc_to_DAG(adj_list, scc)

    # Compute the reachability weight using the DAG algorithm
    reachability_weight = compute_reachability_weight_DAG(new_adj_list, weight)

    return reachability_weight
```

This algorithm has a time complexity of $O(|V|+|E|)$, as the SCC decomposition, collapsing the SCC into a DAG, and computing the reachability weight using the DAG algorithm all take linear time.