Problem Set 4 — Module 4

shenyu li

October 14th 2024

3.4

Graph (i)

(a) Order of Strongly Connected Components

The strongly connected components of Graph (i) are:

- HIG: These vertices are strongly connected because there are paths between them.
- $CDFJ$: Similarly, these vertices also form a strongly connected component.
- A, B, E: Each of these vertices is isolated in terms of strong connectivity. A is reachable from B, but not vice versa, and E is isolated.

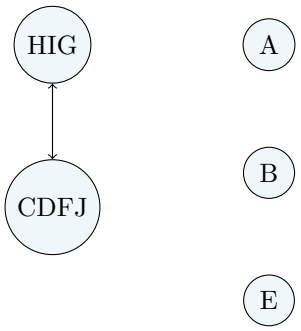
Thus, the order of the strongly connected components in Graph (i) is $E, B, A, HIG, CDFJ$.

(b) Source and Sink SCCs

- **Source SCCs:** HIG and $CDFJ$ are source SCCs, as there are no edges leading to them from other SCCs.
- **Sink SCCs:** E, B, A are sink SCCs, as they do not point to any other SCCs.

(c) Metagraph

The metagraph for Graph (i) is illustrated below:



(d) Minimum Number of Edges to Add

To make the entire graph strongly connected, add at least 2 edges to connect the source SCCs to the sink SCCs.

Graph (ii)

(a) Order of Strongly Connected Components

The strongly connected components of Graph (ii) are:

- ABE: These vertices are strongly connected.
- C: This vertex is isolated.
- $DHGFI$: These vertices are strongly connected as there are paths among them.

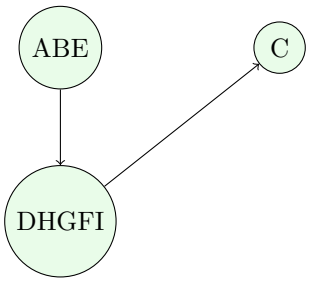
Thus, the order of the strongly connected components in Graph (ii) is $ABE, C, DHGFI$.

(b) Source and Sink SCCs

- **Source SCC:** ABE is the source SCC, as there are no edges leading to it from other SCCs.
- **Sink SCC:** $DHGFI$ is the sink SCC, as it does not point to any other SCCs.

(c) Metagraph

The metagraph for Graph (ii) is illustrated below:



(d) Minimum Number of Edges to Add

To make the graph strongly connected, add at least 1 edge to connect ABE to $DHGFI$.

1. Initialize an empty adjacency list for G^R.
2. For each vertex V_1 in G:
   - For each edge (V_1, V_2):
     - Add an edge (V_2, V_1) to the adjacency list of G^R.
3. The time complexity of this algorithm is $O(V+E)$, where V is the number of vertices and E is the number of edges.

3.5

The approach for reversing a directed graph can be described as follows:

For each vertex V_1 in the graph G, traverse its outgoing edges. If there exists an edge $(V_1 \to V_2)$, then in the transposed graph (or reverse graph) G^R, create an edge $(V_2 \to V_1)$. This process ensures that all edges are reversed appropriately.

the time complexity of this algorithm is $O(V + E)$, where V is the number of vertices and E is the number of edges.

Below is a Python function to compute the reverse graph:

```
def reverse_graph(graph):
    reversed_graph = {v: [] for v in graph.keys()}

    for v1 in graph:
        for v2 in graph[v1]:
            # Add the reverse edge
            reversed_graph[v2].append(v1)

    return reversed_graph

# Example usage
```

```
original_graph = {
    1: [2, 3],
    2: [4],
    3: [4],
    4: []
}

reversed_graph = reverse_graph(original_graph)
print("Reversed Graph:", reversed_graph)
```

3.7

A bipartite graph is defined such that its vertices can be divided into two sets, ensuring that no edges exist between vertices within the same set, analyze this property through the following parts:

(a) Algorithm to Determine Bipartiteness

perform a Depth-First Search (DFS) on the graph and alternate coloring the vertices red and black based on their levels in the DFS tree. The algorithm proceeds as follows:

1. Initialize all vertices as uncolored. 2. Start DFS from any unvisited vertex, coloring it red. 3. For each adjacent vertex: - If the vertex is uncolored, color it with the opposite color. - If the vertex is already colored with the same color as the current vertex, return false (indicating that the graph is not bipartite). 4. For each non-tree edge (u, v): - If u and v have the same color, return false.

The algorithm runs in $O(V + E)$ time complexity, as it visits each vertex and edge exactly once.

Python implementation of the algorithm:

Listing 1: Check if a Graph is Bipartite

```
def is_bipartite(graph):
    color = {}

    def dfs(node, c):
        color[node] = c
        for neighbor in graph[node]:
            if neighbor not in color:
                if not dfs(neighbor, 1 - c):
                    return False
            elif color[neighbor] == color[node]:
                return False
        return True
```

```
    for vertex in graph:
        if vertex not in color:
            if not dfs(vertex, 0):  # Start coloring with red (0)
                return False
    return True

# Example usage
graph_example = {
    0: [1, 3],
    1: [0, 2],
    2: [1, 3],
    3: [0, 2]
}

print("Is the graph bipartite?", is_bipartite(graph_example))
```

(b) Mathematical Proof

From the algorithm above, conclude that a graph is bipartite if and only if for every non-tree edge (u, v), the vertices u and v have different colors. In the context of the DFS tree, this means that vertices u and v must be separated by an odd number of layers.

If an odd-length cycle exists, it implies that the distance between two vertices in that cycle is even (as it returns to the starting vertex), which means the cycle length would be odd. Thus, the presence of an odd-length cycle indicates that the graph cannot be bipartite.

if the graph contains no cycles of odd length, it is indeed bipartite. The proof follows that if color the graph using two colors (red and black), then any cycle formed must be even, confirming that the graph is bipartite.

(c) Maximum Number of Colors Required

In a graph containing exactly one odd-length cycle, at most three colors are needed to color the vertices. The odd-length cycle requires three colors because when attempting to color the cycle, if use two colors, we would find that adjacent vertices must have different colors. When returning to the starting vertex of the cycle,find that it also needs to share a color with one of its adjacent vertices, thus necessitating a third color.

3.13

In this problem, the properties of connectivity in undirected and directed graphs through the following parts:

(a) Proof of Connectivity in Undirected Graphs

In any connected undirected graph $G = (V, E)$, there exists a vertex $v \in V$ such that the removal of v leaves G connected.

Formal Proof: Consider the depth-first search (DFS) tree of the graph G. A DFS tree is constructed by traversing the graph starting from any vertex and exploring as far as possible along each branch before backtracking.

1. Let T be the DFS tree of G. 2. By the nature of a DFS tree, it will contain vertices organized in levels corresponding to their depth in the traversal. 3. Leaf nodes in the DFS tree are defined as vertices with no further children. 4. Removing a leaf node from the DFS tree does not affect the connectivity of the remaining nodes, as all its ancestors and descendants are still connected through alternative paths. 5. Therefore, remove any leaf node v, the remaining vertices still maintain connectivity through the paths established by the DFS traversal.

Thus, it is evident that there exists at least one vertex $v \in V$ such that removing it keeps the graph G connected.

Here is a Python implementation that demonstrates removing a leaf vertex in a graph:

Listing 2: Remove Leaf Vertex

```
def remove_leaf(graph, leaf):
    if leaf in graph:
        del graph[leaf]
        for node in graph:
            if leaf in graph[node]:
                graph[node].remove(leaf)
    return graph

# Example usage
graph_example = {
    1: [2, 3],
    2: [1, 4],
    3: [1],
    4: [2]
}

print("Graph before removing leaf:", graph_example)
graph_after_removal = remove_leaf(graph_example, 3)
print("Graph after removing leaf:", graph_after_removal)
```

(b) Example of a Strongly Connected Directed Graph

Consider the directed graph represented by the following edges:

$$A \to BB \to CC \to AA \to D$$

This graph is strongly connected because reach every vertex from any starting vertex. However, if remove vertex D, left with the directed cycle $A \to B \to C \to A$, which is not strongly connected, as cannot reach D anymore.

(c) Example of a Directed Graph with Two Strongly Connected Components

Let denote two strongly connected components in a directed graph as follows:

- Component 1: $U = \{A, B, C\}$ with edges $A \to B, B \to C, C \to A$.
- Component 2: $V = \{D, E, F\}$ with edges $D \to E, E \to F, F \to D$.

These two components are strongly connected but not connected to each other.

Now, if construct the meta graph for these components, it would look like:

$$U \to V$$

If add an edge from V to U, the entire graph would become strongly connected. Therefore, no single edge addition between the components can maintain the graph's strong connectivity unless we connect them directly.

3.16

The objective is to compute the minimum number of semesters required to complete all courses, given that a student can take any number of courses in a single semester.

utilize a modified topological sorting algorithm that employs multiple queues to achieve this goal.

Steps of the Algorithm

1. Initialize Data Structures: - Create an array to store the in-degree of each vertex (course). - Initialize multiple queues to represent each semester.

2. Populate Initial Queue: - Add all vertices with an in-degree of 0 (courses without prerequisites) to the first queue.

3. Process Each Semester: - For each semester (queue): - Dequeue all vertices from the current semester queue and process them. - For each neighbor (dependent course) of the dequeued vertex: - Decrease the in-degree of the neighbor by 1. - If the in-degree of the neighbor becomes 0, enqueue it to the next semester's queue.

4. Count Semesters: - The total number of queues processed will represent the minimum number of semesters needed to complete the curriculum.

Python Code Example

Listing 3: Minimum Semesters Algorithm

```
from collections import deque, defaultdict

def min_semesters(num_courses, prerequisites):
    in_degree = [0] * num_courses
    graph = defaultdict(list)

    for course, prereq in prerequisites:
        graph[prereq].append(course)
        in_degree[course] += 1

    queues = []
    current_queue = deque()

    for i in range(num_courses):
        if in_degree[i] == 0:
            current_queue.append(i)

    semester_count = 0

    while current_queue:
        queues.append(current_queue)
        semester_count += 1
        next_queue = deque()

        while current_queue:
            course = current_queue.popleft()
            for neighbor in graph[course]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    next_queue.append(neighbor)

        current_queue = next_queue

    return semester_count

# Example usage
num_courses = 4
prerequisites = [(1, 0), (2, 1), (3, 1)]
print("Minimum semesters required:", min_semesters(num_courses, prerequisites))
```

Time Complexity

The running time of this algorithm is $O(V + E)$, where V is the number of courses (vertices) and E is the number of prerequisite relationships (edges). This is considered linear time, which meets the problem's requirements.