Theory Assignment-4: ADA Winter-2024

Aditya Sharma (2022038) Ayan Kumar Singh (2022122)

1 Algorithm Description

We have the following functions in our code:

- `dfs`: Recursive implementation of Depth First Search acts as a helper function for `topologicalSort`.
- `topologicalSort`: We perform topological sort of the given Directed Acyclic Graph (DAG) and store and return it in an array.
- `identifyCutVertices`: Identifies cut vertices by checking whether for a vertex b such that $s < b < t$ there exists an edge (a, c) such that $a < b < c$.

We have used the following data structures in our code:

- `unordered_map<int, int> vertexToIndex`: Used to retrieve the index of a vertex in the topological order in $O(1)$ time. **Note:** As all the vertex numbers are unique, unordered_map (i.e., hashmap) provides $O(1)$ access and insertion time.
- `vector<int> sorted`: Used to store vertices in the topological order.
- **Intuition**: As we are given a Directed Acyclic Graph (DAG), denoted as $G = (V, E)$, a topological sort algorithm returns a sequence of vertices in which the vertices never come before their predecessors on any paths. In other words, if $(u, v) \in E$, v never appears before u in the sequence.
- This linear ordering of the graph can be represented as a horizontal line of ordered vertices, such that all edges point only to the right.

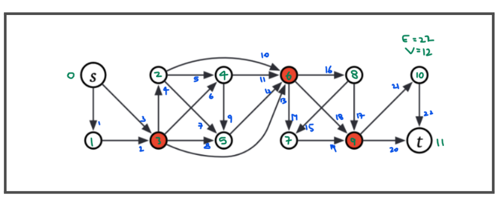

Figure 1: This is an example DAG graph with its vertices marked for reference

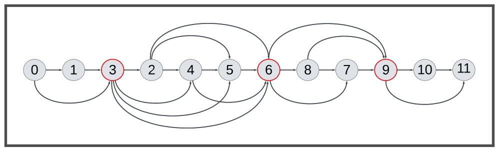

Figure 2: Topological ordering of the graph in Figure 1 with its vertices arranged in linear fashion from left to right

- The vertices with red color boundary are the cut vertices. For these vertices, let them be 'b', there doesn't exist vertices a and c such that there exists an edge (a, b) and (b, c) such that $a < b < c$ in the topological ordering.

2 Algorithm Steps

- Perform a topological sort on the given directed acyclic graph (DAG) to obtain a linear ordering of vertices.
- Iterate through the sorted vertices in order. For each vertex v, find all its adjacent vertices and mark the vertices between currentVertex and edgeEnd as not cut vertices.
- Output the remaining vertices as the (s, t)-cut vertices of the graph.

3 Complexity Analysis

3.1 Space Complexity Analysis:

- **vector<vector<int>>graph(n):** This stores the adjacency list representation of the graph, which requires $O(V + E)$ space, where V is the number of vertices, and E is the number of edges.
- **vector<bool>visited(n, false):** This vector keeps track of visited vertices during the DFS traversal, requiring $O(V)$ space.
- **stack<int>order:** This stack is used to store the topological order of vertices during the DFS traversal. In the worst case, where the graph is a directed acyclic graph, the stack can hold all vertices, requiring $O(V)$ space.
- **vector<int> parent;** This vector stores the parent information for each vertex during the DFS traversal, requiring $O(V)$ space.
- **vector<int>result:** This vector stores the final topological order, requiring $O(V)$ space.

- **unordered_map<int, int>vertexToIndex:** This unordered map stores the mapping between vertices and their indices in the topological order, requiring $O(V)$ space and $O(1)$ retrieval time as vertex numbers are unique.
- **vector<bool>cutVertices(n, true):** This vector stores the information about whether a vertex is a cut vertex or not, requiring $O(V)$ space.
- **NOTE:** `vector<bool> visited(n, false)`, `stack<int> order`, and `vector<int> parent`. These arrays are created during topological sorting via DFS, but then these arrays don't serve any purpose.

Overall, the space complexity of the algorithm is $\mathbf{O(V + E)}$, which is primarily dominated by the space required to store the graph representation.

3.2 Time Complexity Analysis:

- **topologicalSort function:**
  - The DFS traversal takes $O(V + E)$ time in the worst case, where V is the number of vertices, and E is the number of edges.
  - Retrieving the topological order from the stack takes $O(V)$ time.
  - The overall time complexity of the 'topologicalSort' function is $O(V+E)$.
- **identifyCutVertices function:**
  - The function iterates over the topological order, which takes $O(V)$ time.
  - For each vertex, it iterates over its outgoing edges, which takes $O(E)$ time in the worst case.
  - We have a variable named **maxMarkedFalseIndex** that acts as a counter for the third for loop. It prevents re-marking vertices as false in the vector `cutVertices` (initialized with all vertices marked as true). The counter only increments if the edge from the current vertex is greater than **maxMarkedFalseIndex** (in the topological order). Therefore, for the traversal of vertices in the first for loop, this loop runs for all vertices at maximum. Thus, the complexity for overall traversal $O(V) + O(V + E)$.
  - Therefore, the overall time complexity of the `identifyCutVertices` function is $O(V + E)$.

Therefore, the overall time complexity of the algorithm is $\mathbf{O(V + E)}$, which is optimal for graphs since it needs to visit all vertices and edges at least once.

4 PseudoCode

Algorithm 1 Topological Sort and Cut Vertex Identification

```
procedure TopologicalSort(G(V, E))
    n ← |V|
    visited ← vector of size n initialized with false
    order ← empty stack
    parent ← vector of size n initialized with -1
    vertexToIndex ← empty unordered map
    for i ← 0 to n − 1 do
        if ¬visited[i] then
            DFS(i, G, visited, order, parent)
        end if
    end for
    result ← empty vector
    while ¬order.empty() do
        vertex ← order.top()
        order.pop()
        vertexToIndex[vertex] ← result.size()
        result.push_back(vertex)
    end while
    return result
end procedure
procedure DFS(node, G, visited, order, parent)
    visited[node] ← true
    for neighbor ∈ G[node] do
        if ¬visited[neighbor] then
            parent[neighbor] ← node
            DFS(neighbor, G, visited, order, parent)
        end if
    end for
    order.push(node)
end procedure
procedure IdentifyCutVertices(sorted, vertexToIndex, G)
    n ← sorted.size()
    cutVertices ← vector of size n initialized with true
    maxMarkedFalseIndex ← 0
    for i ← 0 to n − 1 do
        currentVertex ← sorted[i]
        for edge ∈ G[sorted[i]] do
            edgeEndIndex ← vertexToIndex[edge]
            if edgeEndIndex > maxMarkedFalseIndex then
                for k ← i + 1 to edgeEndIndex − 1 do
                    cutVertices[sorted[k]] ← false
                end for
                maxMarkedFalseIndex ← edgeEndIndex
            end if
        end for
    end for
    return cutVertices
end procedure
```

5 Main Function

- The adjacency list is used for storing the graph as a 2-D list.
- The returns from the topologicalSort function are stored in an array, and a unordered hashmap VertexToIndex is initialized to get the topological order index for a particular vertex number in O(1) time.
- The identifyCutVertices function is called, and its output is cached in an array.
- Finally, the cut vertices are printed to stdout.

```cpp
int main() {
    int n = 12; // Number of vertices
    vector<vector<int>> graph(n);
    // Adding directed edges to the graph
    graph[0].push_back(1); graph[0].push_back(3);
    graph[1].push_back(3);
    graph[2].push_back(4); graph[2].push_back(5); graph[2].
    push_back(6);
    graph[3].push_back(2); graph[3].push_back(4); graph[3].
    push_back(5); graph[3].push_back(6);
    graph[4].push_back(5); graph[4].push_back(6);
    graph[5].push_back(6);
    graph[6].push_back(7); graph[6].push_back(8); graph[6].
    push_back(9);
    graph[7].push_back(9);
    graph[8].push_back(7); graph[8].push_back(9);
    graph[9].push_back(10); graph[9].push_back(11);
    graph[10].push_back(11);

    // Perform topological sort
    unordered_map<int, int> vertexToIndex;
    vector<int> sorted = topologicalSort(graph, n, parent,
    vertexToIndex);
    // Outputing the sorted vertices
    cout << "Topological Order:";
    for (int vertex : sorted) {
        cout << " " << vertex;
    }
    cout << endl;
    // Identify Cut Vertices
    vector<bool> cutVertices = identifyCutVertices(sorted,
    vertexToIndex, graph);
    // Output cut vertices
    cout << "Cut Vertices:";
    for (int i = 1; i < sorted.size() - 1; ++i) {
        if (cutVertices[sorted[i]]) {
            cout << " " << sorted[i];
        }
    }
    cout << endl;
    return 0;
}
```

Listing 1: C++ code snippet of main function