Theory Assignment-4: ADA Winter-2024

Devansh Grover (2022151)   Garvit Kochar (2022185)

1 Pre-processing

Let $G = (V, E)$ be a **DAG** with two specified vertices s and t. A vertex $v \notin \{s, t\}$ is called an (s, t)-cut vertex if every path from s to t passes through v. We are required to find these cut vertices in the graph G.
To find these cut vertices, we first form a function for running a DFS on a graph G. For our solution, we are using an iterative variant of the DFS the pseudcode for which is given below.

Algorithm 1 Pseudocode for DFS

```
function DFS(node, graph, visited, Stack)
    visited[node] ← true
    for $i$ in graph[node] : do
        if $\neg$visited[$i$] then
            DFS($i$, graph, visited, Stack)
        end if
    end for
    Stack.push(node)
end function
```

Now, let's go through the function's implementation:
1. node: This represents the current node being visited. For simplicity, we are taking it to be an int.
2. graph: This is the graph G passed in the form of a vector that in turn contains vectors containing int values.
3. visited: This parameter is a reference to a boolean vector indicating whether each node has been visited or not.
4. Stack: This parameter is a reference to a stack data structure where we store the nodes in the order they are visited.

The algorithm visits each node in the graph exactly once. Initially, all nodes are unvisited. The DFS function starts at a given node and recursively explores all nodes reachable from that node. For each node i adjacent to the current node node, if i has not been visited ($visited[i] = false$), the DFS function is recursively called for node i. After visiting all adjacent nodes, the current node is pushed onto the stack to maintain the topological order. The DFS algorithm continues until all nodes have been visited. The resulting stack contains the nodes in the order they were visited, forming a reverse topological order.

Since DFS visits every vertex and edge exactly once, the time complexity of DFS is often expressed as $O(V + E)$, where V is the number of vertices and E is the number of edges.

The DFS function we are using is a standard Depth-First Search algorithm. The recurrence relation for DFS is as follows:

$$T(v) = 1 + \sum_{u \in Adj[v]} T(u)$$

where:
$T(v)$ is the total time for the DFS starting from vertex v.
$Adj[v]$ is the set of vertices adjacent to v.
The base case is when a vertex has no unvisited neighbors, in which case the time complexity is O(1).

2 Algorithm Description

1. DFS and Topological Sort: The DFS function is a standard Depth-First Search (DFS) algorithm. It starts at a given node, explores as far as possible along each branch before backtracking. The nodes are then pushed onto a stack in the order they are visited. This stack represents a topological sort of the graph, which is an ordering of the vertices such that for every directed edge (u, v), vertex u comes before v in the ordering.

2. Calculating Paths from s: Initially, the $FindCutVertex$ function calculates the number of paths from the source vertex s to every other vertex in the graph. This is done by initializing a cache table $dp1$ where $dp1[i]$ represents the number of paths from s to vertex i. We iterate over the vertices in the topological order and for each vertex, we update the cache table by adding the number of paths from s to the current vertex to all its adjacent vertices.

3. Calculating Paths to t: The function then calculates the number of paths from every vertex to the target vertex t. This is done similarly to the previous step but on the reversed graph (i.e., the graph with all edges reversed) and with a reversed topological order. We initialize another cache table $dp2$ where $dp2[i]$ represents the number of paths from vertex i to t. We iterate over the vertices in the reversed topological order and for each vertex, we update the cache table by adding the number of paths from the current vertex to t to all its adjacent vertices in the reversed graph.

4. Identifying Cut Vertices: Finally, the algorithm identifies the cut vertices. A vertex v is an (s, t)-cut vertex if and only if every path from s to t passes through v. This is equivalent to saying that the number of paths from s to t is equal to the product of the number of paths from s to v and the number of paths from v to t. Therefore, the algorithm checks for each vertex v if $dp1[v] * dp2[v]$ is equal to $dp1[t]$ (i.e., the total number of paths from s to t). If it is, v is an (s, t)-cut vertex, and the algorithm will print/output this vertex to the console.

To be done in main: The main function is the driver function. It needs to take the number of vertices, edges, the starting vertex, and the target vertex as input. Using this information, we can form a 2D vector to represent the graph G, i.e., vector that in turn contains vectors containing int values. It then passes the starting vertex, target vertex and the G to the FindCutVertex function.

Recurrence of the DP tables:
For $dp1$, which counts the number of paths from s to every other vertex, we can say that for each vertex v, $dp1[v]$ is the sum of $dp1[u]$ for all vertices u that have a directed edge to v. This can be written as:

$$dp1[v] = \begin{cases} 1 & \text{if } v = s \\ \sum_{u \in Adj[v]} dp1[u] & \text{otherwise} \end{cases}$$

Here, $Adj[v]$ represents the adjacency list of vertex v, i.e., the vertices that are directly connected to v by an edge.

Similarly, for $dp2$, which counts the number of paths from every vertex to t, we can say that for each vertex v, $dp2[v]$ is the sum of $dp2[u]$ for all vertices u that v has a directed edge to. This can be written as:

$$dp2[v] = \begin{cases} 1 & \text{if } v = t \\ \sum_{u \in Adj_{rev}[v]} dp2[u] & \text{otherwise} \end{cases}$$

Here, $Adj_{rev}[v]$ represents the adjacency list of vertex v in the reversed graph.

Note: We are using 1-based indexing in our algorithm.

Algorithm 2 Pseudocode for FindCutVertex

```
function FindCutVertex($s, t$, graph)
    $n \leftarrow$ size(graph) $- 1$
    dp1 $\leftarrow$ vector($n + 1, 0$), dp2 $\leftarrow$ vector($n + 1, 0$)
    topo $\leftarrow$ vector(), Stack $\leftarrow$ stack(), visited $\leftarrow$ vector($n + 1$, false)

    for $i \leftarrow 1$ to $n$: do
        if $\neg$visited[$i$]: then
            DFS($i$, graph, visited, Stack)
        end if
    end for

    while $\neg$Stack.empty(): do
        topo.push_back(Stack.top())
        Stack.pop()
    end while

    dp1[$s$] $\leftarrow 1$
    for $i \leftarrow 0$ to topo.size(): do
        node $\leftarrow$ topo[$i$]
        for next_node in graph[node]: do
            dp1[next_node] += dp1[node]
        end for
    end for

    reversed_graph $\leftarrow$ vector($n + 1$)
    for $i \leftarrow 1$ to $n$: do
        for $j$ in graph[$i$]: do
            reversed_graph[$j$].push_back($i$)
        end for
    end for

    reverse(topo) // Reverse the topological order for the reversed graph

    dp2[$t$] $\leftarrow 1$
    for $i \leftarrow 0$ to topo.size(): do
        node $\leftarrow$ topo[$i$]
        for next_node in reversed_graph[node]: do
            dp2[next_node] += dp2[node]
        end for
    end for

    output "Cut vertices: "
    for $i \leftarrow 1$ to $n$: do
        if $i \neq s \land i \neq t \land$ dp1[$i$] $\times$ dp2[$i$] $=$ dp1[$t$]: then
            output $i$
        end if
    end for
    output newline
end function
```

3 Correctness of the Algorithm

First, we compute a topological order of the graph. This is crucial because in a DAG, there exists a directed path from u to v if and only if u comes before v in the topological ordering. It ensures that nodes are processed in an order such that if there is an edge from u to v, then u appears before v in the order. The topological order is stored in the topo vector.
In other words, this property ensures that when calculating the number of paths from s to each vertex and from each vertex to t, each vertex is processed after all its predecessors and before all its successors.

Then, we calculate the number of paths from a source vertex (s) to every other vertex and from every vertex to a target vertex (t). The code uses dynamic programming (DP) to compute these path counts. This ensures that when processing a vertex v, the number of paths to/from all its predecessors/successors have already been calculated.

DP Arrays: $dp1$ and $dp2$ are vectors representing the number of paths.
$dp1[i]$ stores the number of paths from s to vertex i.
$dp2[i]$ stores the number of paths from vertex i to t.

Forward DP (From Source to Other Vertices):
The code iterates through the topological order. For each node node, it updates $dp1[next\_node]$ for all adjacent nodes $next\_node$. This step ensures that we count the paths from s to other vertices.

Reversed Graph and Backward DP (From Other Vertices to Target):
The code constructs a reversed graph (edges reversed). It reverses the topological order. Similar to the forward DP, it iterates through the reversed order. For each node node, it updates $dp2[next\_node]$ for all adjacent nodes $next\_node$. This step counts the paths from other vertices to t.

Cut Vertex Identification: A vertex v is a cut vertex if and only if all paths from s to t pass through v. This is equivalent to saying that the number of paths from s to t is equal to the number of paths from s to v times the number of paths from v to t. This is because every path from s to t that passes through v can be decomposed into a path from s to v and a path from v to t. Therefore, the algorithm correctly identifies the cut vertices by checking if $dp1[v] * dp2[v] == dp1[t]$ for each vertex v.

4 Running Time of the Algorithm

Let the graph be represented by $G = (V, E)$. Then, the running time of our algorithm is $O(V + E)$.
The running time is determined by the following parts:
1. DFS and Topological Sort: As mentioned in the pre-processing part, we already know that the iterative DFS algorithm runs in $O(V + E)$ time. The subsequent topological sort also runs in $O(V + E)$ time, 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 processed exactly once during the topological sort.
2. Computing the number of paths from s to every other vertex: As mentioned before, this part of the algorithm uses $dp1$. Since it iterates over each vertex in the topological order (which requires $O(V)$ time), and for each vertex, it iterates over its outgoing edges (which requires $O(E)$ time in total across all vertices), this part takes $O(V + E)$ time.
3. Reversing the graph: Constructing the reversed graph takes $O(V + E)$ time as it involves iterating over all vertices and edges in the graph.
4. Computing the number of paths from every vertex to t: Similar to the forward pass, this part of the algorithm also runs in $O(V + E)$ time as it iterates over each vertex in the reverse topological order and for each vertex, it iterates over its outgoing edges in the reversed graph. It uses the $dp2$ array/vector.
5. Identifying the cut vertices: This operation involves iterating over all vertices once (therefore it is a normal for loop), so it runs in $O(V)$ time.
Therefore, the running time of the algorithm is the sum of running times from steps 1 to 5, i.e, $4*O(V+E)+O(V)$. Since these parts are executed sequentially, and we consider running time under asymptotic analysis, the running time of our algorithm comes out to be $O(V + E)$.