ADA-HW-4-Rubric

Kuber Budhija

March 2024

1 General Guidelines

Total marking of the problem is 20. Marking has been primarily partitioned into three sections.

- **Algorithm Description / Pseudocode:** Maximum credit for this part is 12. Here we consider either algorithm description or pseudocode. If both are written, choose the one which scored maximum.

  The part marking for section which includes Algorithm Description **or** Pseudocode consists:

  1. Topological sort : 3 marks
  2. Data Structure creation: 2 marks
  3. BFS Traversal: 3 marks
  4. Identification of cut vertices: 4 marks

  Pseudocode with no comments describing will get 10. Moreover, in case of pseudocode, the topological sorting and the data structure might not be mentioned/implemented explicitly, but implicitly the idea should be there to maintain the optimal running time.

- **Running Time:** Maximum credit for this part is 4.
- **Correctness:** Maximum credit for this part is 4.

If the Chegg solution mentioned in alternate solutions is used without any references, mark the submission as plagiarism.

2 Problem Statement

Let $G = (V, E)$ be a directed acyclic graph with two specified vertices s and t. A vertex $v \in \{s, t\}$ is called an (s, t)-cut vertex if every path from s to t passes through v. If G has n vertices and m edges, then design a polynomial-time algorithm that computes all the (s, t)-cut vertices of G.

3 Solution

In this solution, we will provide a $O(m+n)$ algorithm, where $m = |E|$ and $n = |V|$. Here we assume that the source s and sink t are two different vertices, that is, $s \neq t$. Also, we may assume that for every vertex $v \in V \setminus \{t\}$, there exists a path from v to t in G. If this is not the case, that is, if there exists a vertex v which is not reachable to t, then v might create complication for the algorithm, which we are going to describe. However, in the pre-processing step we remove all such vertices from the graph and ensure our assumption to be true.

3.1 Preprocessing

$- - - - - - - - - - ->$ No score for this section

If there is any vertex $v \in V \setminus \{t\}$ which does not have any path to sink t, then we can remove all such vertices in $O(m+n)$ time (by reversing the direction of the edges in the graph and performing DFS from sink to all other vertices in the modified graph vertex). The precise procedure stated as follows.

1. Let $\widehat{G}$ be an auxiliary graph such that $V(\widehat{G}) = V(G)$.
2. For any pair of vertices (u, v), the directed edge uv (with direction from u to v) belongs to edge set $E(\widehat{G})$ if and only if there is a directed edge uv (with direction from v to u) in the input graph G.
3. Do DFS on $\widehat{G}$ starting from t. After completion of DFS, let $\mathcal{X}$ be the set of vertices visited by the DFS traversal.
4. $\forall u \in V(\widehat{G}) - \mathcal{X}$, remove vertex u form graph G.

We claim that, the above processing removes all the unnecessary vertices from G. Why? Think yourself. You need to prove the following corollary.

Corollary 1 *For any vertex $v \in V \setminus \{t\}$, there exists a path from t to v in $\widehat{G}$ if and only if there exists a path form v to t in G.*

Note that we will be using adjacency list representation for the graph. Thus, the preprocessing takes $O(m+n)$ time in each step.

3.2 Algorithm Description

To find every (s, t)-cut vertex in a directed acyclic graph (DAG) G, you can use a depth-first search (BFS)-based algorithm. The basic idea is to perform a BFS traversal of the graph while keeping track of information that helps identify the (s, t)-cut vertices. Here is a detailed explanation of the algorithm:

1. **Perform a Topological Sort:**

   $- - - - - - - - - - ->$ 3 marks

- First, perform a topological sort of the vertices in G. Since G is a DAG, it can be linearly ordered. With abuse of notation, form now on, whenever we refer G, we assume that G is topologically sorted unless otherwise mentioned.

2. **Data Structures:**

   $- - - - - - - - - - - >$ 2 marks

   - There is an array $topo[1 \cdots n]$ of size n, such that:
     - $topo[i]$: represents the vertex at i-th position in topological order.
   - For each vertex v, is a variable far_v, as follows:
     - far_v: represents the position of farthest neighbour of v in the topological order.

3. **BFS Traversal:**

   $- - - - - - - - - - - >$ 3 marks

   We will use BFS traversal for finding the farthest vertex for each vertex v.

   - Initialize the BFS queue and start by adding source s.
   - In the first step we only have source s in the queue, so we will add it's neighbors to the queue and remove source s from the queue and mark it as visited.
     - While adding the neighbours to the queue, we will check their (all of the neighbours) corresponding positions in topologically sorted order and store the maximum position as farthest neighbour of s, that is, set the value of far_v as discussed above.
   - We repeat the previous step for each vertex v until the completion of BFS traversal and update the values of far_v accordingly.

4. **Identify Cut Vertices:**

   $- - - - - - - - - - - >$ 4 marks

   - Initialize two pointers ptr_1 and ptr_2:
     (a) ptr_1 pointing at source s.
     (b) ptr_2 pointing at the farthest neighbour of source s, that is, far_s of source s.
   - Iterate over all vertices in topological order availed by $topo[]$ and update pointers accordingly, as :
     - ptr_1 : moves to next vertex of the order.
     - ptr_2 : gets updated to the farthest neighbour of the vertex located by ptr_1 if and only if position of farthest neighbour is greater than the current position of ptr_2 in order.
   - Whenever both the pointers meet at same position, to be specific, after the update of ptr_1 and before the update of ptr_2, if both of the pointer points to same vertex v, report v as cut vertex.

$- - - - - - - - - - - >$ 12 marks for this section or for following section, do not evaluate both

3.3 Pseudocode

Algorithm 1 Topological Sort

```
function TopologicalSort(graph)
    visited ← set()
    stack ← []
    function DFS(node)
        visited.add(node)
        for neighbor in graph[node] do
            if neighbor $\notin$ visited then
                DFS(neighbor)
            end if
        end for
        stack.append(node)
    end function
    for vertex in graph do
        if vertex $\notin$ visited then
            DFS(vertex)
        end if
    end for
    return stack[::-1]
end function=0
```

Algorithm 2 BFS to Find Farthest Point

```
function BFSFarthestPoint(graph, source, topologically_sorted_order)
    farthest_indices ← $[-1] * \text{len}(graph)$
    queue ← deque([source])
    while queue do
        current ← queue.popleft()
        for neighbor in graph[current] do
            if farthest_indices[neighbor] $= -1$ then
                farthest_indices[neighbor] ← topologically_sorted_order.index(neighbor)
                queue.append(neighbor)
            end if
        end for
    end while
    return farthest_indices
end function=0
```

Algorithm 3 Find Cut Vertices

```
0: function FINDCUTVERTICES(graph, source, sink)
0:   topologically_sorted_order ← TopologicalSort($graph$)
0:   farthest_indices ← BFSFarthestPoint($graph, source, topologically\_sorted\_order$)
0:   cut_vertices ← [0] $*$ len($graph$)
0:   pointer1 ← source
0:   pointer2 ← farthest_indices[source]
0:   while pointer1 $\neq$ sink do
0:     pointer1 ← topologically_sorted_order[topologically_sorted_order.index(pointer1) $+$ 1]
0:     if pointer1 $=$ pointer2 then
0:       cut_vertices[pointer1] ← 1
0:     end if
0:     if pointer2 $<$ farthest_indices[pointer1] then
0:       pointer2 ← farthest_indices[pointer1]
0:     end if
0:   end while
0:   return [vertex for vertex, is_cut in enumerate($cut\_vertices$) if is_cut]
0: end function=0
```

3.4 Time Complexity Analysis

$- - - - - - - - - - - >$ 4 marks

To analyze the time complexity of the algorithm described:

1. **Perform a Topological Sort:**
   - Topological sorting can be done in linear time, $O(m+n)$, where V is the number of vertices and E is the number of edges in the graph.
2. **Initialize Data Structures:**
   - Initializing the arrays requires constant time, $O(1)$.
3. **BFS Traversal:**
   - BFS traversal takes $O(m+n)$ time since it visits each vertex and each edge once.
4. **Identify Cut Vertices:**
   - Takes $O(n)$ time for iterating over all topologically sorted vertices.

Combining these steps, the overall time complexity of the algorithm is $O(m+n)$. This is because the dominant factor in the time complexity is the BFS traversal.

3.5 Proof of Correctness

$- - - - - - - - - - - >$ 4 marks

The correctness of the algorithm can be proven as follows:

1. **Perform a Topological Sort:** $-----------> 1$ mark
   - Since we are dealing with a directed acyclic graph (DAG), topological sorting can be performed, ensuring that vertices are visited in the correct order to respect the direction of edges.
2. **BFS Traversal:** $-----------> 1$ mark
   - Breadth-first search (BFS) traversal ensures that every vertex and edge is visited exactly once. Thus, we can explore all possible paths from the source vertex s to the target vertex t.
3. **Identify Cut Vertices:** $-----------> 2$ marks
   - During the iteration over pointers, if both pointers collide this means, there's no other way to move ahead without crossing that particular vertex.

****

4 Alternate Solutions

4.1 Chegg's solution

This solution performs topological sorting and then performs a DFS Traversal and checks the existence of cut vertex by comparing entry times and $exit[u] < exit[v]$ where u is child of v.

This solution is not exactly correct, it talks about back edges in DAG, maximum score is 6

4.2 Brute Force

calculate all the possible paths from source to sink, iterate over all paths and return all the vertices which occur in all the paths.

This approach is not optimal at all, The maximum marks this can fetch is 2, since time complexity of this approach is $O(N!)$

4.3 optimized Brute force

DFS with Dynamic Programming, won't add much effect since number of paths will go exponentially as the size of graph increases.

The maximum score is 2

4.4 Quadratic Approach

1. Traverse over all nodes.

2. Remove that node, and apply DFS from source, it'll take $O(V+E)$ time, if you can reach from source to sink even after removing that node, then the removed vertex is not a cut vertex, else it is.
3. the overall time complexity is $O(V+E) \times O(V)$

1. The maximum score for Quadratic Approach is 10
2. pseudocode or algo desc: 5 marks
3. Time complexity Analysis: 3 marks
4. Proof of correctness : 2 marks