Theory Assignment-4: ADA Winter-2024

Rachit Arora (2022384) Mahi Mann (2022272)

1 Assumptions

- It is assumed that the input graph is a Directed Acyclic Graph, and no detection for incorrect input is required.
- By definition, s and t are always *s-t cut* vertices, so they are mentioned as *trivial s-t cut vertices* in the implementation, given a path from s to t exists.
- In the sample test graph figures shown below, s is **blue** and t is **green**.

2 Preprocessing

None required.

3 Algorithm Description

Note: The graph is $G = (V, E)$, the start node is s and the target node is t.

1. First, since we are given a Directed Acyclic Graph, it makes sense to topologically sort the nodes.

   The toposort will be done using a variant of DFS, where at the end of every DFS call, the node will be pushed to an array, and the array will be reversed later to get a topologically sorted order from left to right.

2. We will **remove the nodes which were not reachable by node s in the DFS** that we perfomed (because every path from s to t starts from s, and since no path from s to those nodes exists, they cannot occur on a path from s to t and therefore cannot be *s-t cut vertices*. So, it is safe to remove them from the graph.)

   (**Note**: this step removes all nodes before s in the topologically sorted order as well, so we do not need to do that step in post-processing. Proved in the "Proof of Correctness" section.)

3. Similarly, **we remove the nodes such that there is no path from that node to node t**, since for a node to be an *s-t cut vertex*, at least one path from the node to t has to exist. Thus, we can safely remove them from the graph.

   (**Note**: this step removes all nodes after t in the topologically sorted order as well, so we do not need to do that step in post-processing. Proved in the "Proof of Correctness" section.)

   We can do this by performing a **DFS on G^R starting from t**. All nodes not traversed in this DFS will be removed from the graph.

   At this point in the algorithm, every node in the graph is reachable from s, and every node can reach t, and therefore, there is a path going from s to t through every node.

4. Now, we will initialise an empty set S, which will be our final set of *s-t cut vertices* by the time this algorithm ends.

5. Initialise a variable *counter* to 0.

6. **We will iterate through every node q in topologically sorted order and do the following:**
   (a) **Set** *counter* **to** *counter* $- e_i$ where e_i is the **number of incoming edges** at q.
   (b) **If** *counter* **is 0, add** q **to** S**.**
   (c) **Set** *counter* **to** *counter*$+e_o$, where e_o **is the number of edges that initiate from** q **(i.e. outgoing edges).**

   Note: Suppose there were n_f remaining nodes in the graph after step 3. Then, step 6 has $O(n_f + m)$ constant-time operations where m is the number of edges in the graph.
7. Remove s and t from S.
8. Return the final answer, S, the set of all *s-t cut vertices*, excluding s and t.

The idea of why the *counter* step works is as follows:

- **Whenever** *counter* **is 0, an equal number of incoming and outgoing edges was discovered. Thus, every possible path going from** s **can extend to that node.**
- **Whenever** *counter* **is greater than 0 after a node iteration, it means that there are paths to** t **which do not go through that node, and the graph remains connected.**
- *counter* can never be negative in a topologically sorted order, as we will prove in the "Proof of Correctness" section.

4 Complexity Analysis

Note: We will denote n and the number of nodes in the graph, and m as the number of edges in the graph.

Step 1

Step 1 is a topological sort. We can implement this in $O(n+m)$ time because we only need to add a single $O(1)$ operation to the end of every DFS call on n nodes.

This takes $O(m+n)$ **time.**

Steps 2 and 3

These steps remove nodes and edges from the graph. Let $G = (V, E)$ initially.

We will have an arrays which will mark vertices and edges that are to be removed so that we can access their "status of removal" in $O(1)$ time.

So, first we will iterate through all the nodes to see which is included. This will take $O(n)$ time.

Then, we will check all edges, which takes $O(m)$ time.

So, in total, creating the new graph takes $O(m+n)$ **time.**

Steps 4 and 5

2 $O(1)$ **operations, thus** $O(1)$ **time.**

Step 6

Incoming edges for each node can be precomputed in step 1. Thus, the "incoming" part of the *counter* step will take $O(1)$ time for n nodes, therefore $O(n)$ time in total.

Outgoing edges are a little more complicated, because we have to check each edge in a node's adjacecy list manually. This will take $O(m)$ time in total.

Therefore, Step 6 takes $O(m+n)$ **time.**

Step 7

$O(n)$ time, as we simply need to iterate through all nodes in S and check whether they are s or t.

Step 8

$O(1)$ time.

So in total, since each of the 8 steps is at most $O(m+n)$ time, the algorithm runs in $O(m+n)$ time.

5 Proof of Correctness

- First, we prove the properties about the toposort order that we covered in steps 2 and 3.

  Corollary 1: Step 2 removes all nodes before s in the topologically sorted order.

  Proof: Suppose a node k before s was not removed.

  If there is an edge from k to s, then DFS from s reached k at some point (for node k not to be removed). Let this path be P. Then, $P \cup \{k, s\}$ creates a cycle, which is a contradiction because our graph is acyclic.

  If k does not have an edge going to s, then because k was not removed, a path from s to k exists, which contradicts the fact that k is before s in the topologically sorted order.

  Corollary 2: Step 3 removes all nodes after t in the topologically sorted order.

  Proof: Suppose a node k after t is not removed. Then, a path from t to k exists in G^R, which implies that t is before k in the topologically sorted order of G^R, which implies that t is after k in the topologically sorted order of G, which is a contradiction, since we assumed initially that t is before k.

- Now, we will prove the *count* property, that $count = 0$ generates an *s-t cut vertex*.

  Lemma 1: $counter \geq 0$ always holds true

  Proof: Suppose that at some node k, $counter < 0$. Then, the number of incoming edges is more than the number of outgoing edges discovered so far.

  In other words, there exists an edge $\{l, k\}$ for some node l which has not been discovered as an outgoing edge so far.

  But this is impossible, because if an edge $\{l, k\}$ exists, then l is before k in the topologically sorted order, and since we are iterating in topologically sorted order, it should have been discovered as an outgoing edge before it was discovered as an incoming edge.

  But since $counter < 0$ requires l to be topologically after k, we run into a contradiction, and therefore $counter \geq 0$ always holds true.

  Final theorem: Nodes with $counter = 0$ in their iteration are *s-t cut vertices*.

  Proof: Suppose by contradiction that $counter > 0$ at a node k and k is an *s-t cut vertex* ($counter < 0$ is impossible due to Lemma 1). Then, for some node $q \neq l$, there exists an outgoing edge from q to another state $q' \neq q$, which is not incoming at k.

  Since q' has a path to t and s has a path to q', there exists a path from s to t without using k.

  Thus, removing k does not disconnect the final graph G. But we assumed that k was an *s-t cut vertex*, so it *must* disconnect G. This is a contradiction, and therefore $counter = 0$ in node k's iteration.

Addendum 1: C++ implementation

```
#include <bits/stdc++.h>

using namespace std;
```

```cpp
const int N = 2e5 + 1;

vector<vector<int>> adj(N);
vector<vector<int>> adjrev(N);
vector<int> toposorted;

int vis1[N];
int vis2[N];
int incident[N];

void initvis(){
    for(int i = 0; i < N; ++i){
        vis1[i] = 0;
        vis2[i] = 0;
        incident[i] = 0;
    }
}

void dfs1(int s){
    vis1[s] = 1;
    for(const auto &v: adj[s]){
        if(vis1[v] == 0)
            dfs1(v);
        ++incident[v];
    }
    toposorted.push_back(s);
}

void dfs2(int t){
    vis2[t] = 1;
    for(const auto &v: adjrev[t]){
        if(vis2[v] == 0)
            dfs2(v);
    }
}

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);

    initvis();

    int n,m; cin >> n >> m;
    int s,t; cin >> s >> t;

    for(int i = 0; i < m; ++i){
        int fst, snd;
        cin >> fst >> snd;

        adj[fst].push_back(snd);
        adjrev[snd].push_back(fst);
    }

    dfs1(s);
```

```
    dfs2(t);

    if(!vis1[t]){
        cout << "No cut vertices (no path exists)\n";
        return 0;
    }

    reverse(toposorted.begin(), toposorted.end());

    int cnt = 0;
    vector<int> ans;

    for(const auto &node: toposorted){
        if(vis2[node] == 0){
            continue;
        }
        cnt -= incident[node];
        if(cnt == 0){
            ans.push_back(node);
        }
        for(const auto &edge: adj[node]){
            if(vis2[edge])
                ++cnt;
        }
    }

    cout << "Trivial s-t cut vertices: " << ans[0] << ' ' << ans[ans.size() - 1] << '\n';

    cout << "Non-trivial s-t cut vertices: ";

    for(int i = 1; i < ans.size() - 1; ++i){
        cout << ans[i] << ' ';
    }

    if(ans.size() < 3){
        cout << "none";
    }

    cout << '\n';
}
```

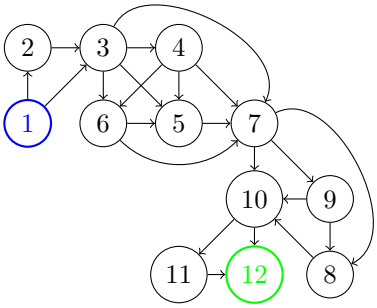
Addendum 2: Sample Tests

5.1 Test 1

Input

```
12 22
1 12
1 2
1 3
2 3
3 4
4 5
3 6
3 5
4 6
6 5
4 7
3 7
5 7
6 7
7 8
7 9
7 10
9 8
9 10
8 10
10 11
10 12
11 12
```

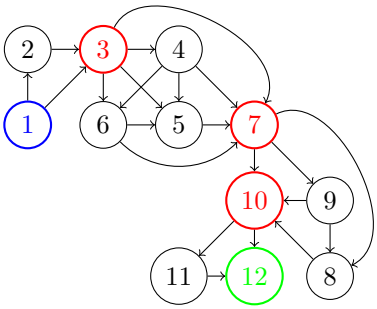
Graph description



Output

```
Trivial s-t cut vertices: 1 12
Non-trivial s-t cut vertices: 3 7 10
```

Output illustration

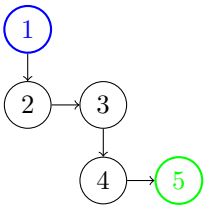


5.2 Test 2

Input

```
5 4
1 5
1 2
2 3
3 4
4 5
```

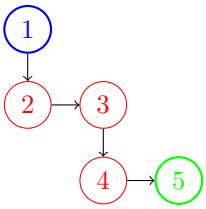
Graph description



Output

```
Trivial s-t cut vertices: 1 5
Non-trivial s-t cut vertices: 2 3 4
```

Output illustration

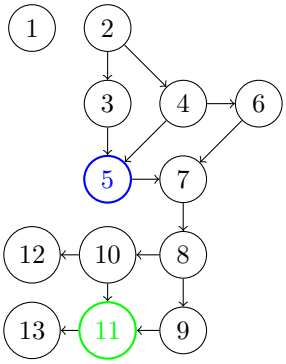


5.3 Test 3

Input

```
13 14
5 11
2 3
2 4
3 5
4 5
4 6
5 7
6 7
7 8
8 9
8 10
9 11
10 11
10 12
11 13
```

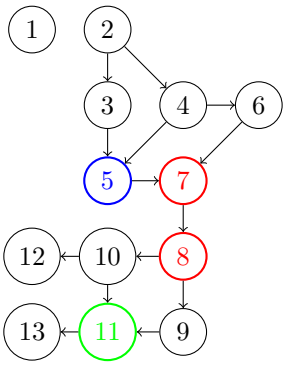
Graph description



Output

```
Trivial s-t cut vertices: 5 11
Non-trivial s-t cut vertices: 7 8
```

Output illustration

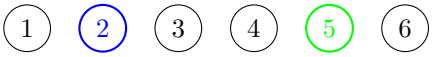


5.4 Test 4

Input

```
6 0
2 5
```

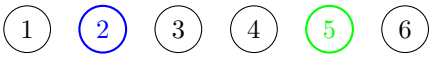
Graph description



Output

```
No cut vertices (no path exists)
```

Output illustration



Addendum 3: People discussed with

Swapnil Panigrahi, Rohak Kansal and Sahil Gupta.

□□□□□
Thank you.