CP implementations

David del Val

December 27, 2020

Contents

Chapter 1

Graphs

1.1 Dijkstra's

Shortest path from `orig` node to `dest` (or to every node) in a graph that does not contain negative edges. It chooses the best path greedily in each iteration and, therefore, it only works on graphs without negative weights.

```cpp
int dijkstra(int orig, int dest, vector<vector<pii>> &graph) {
    vi dists(graph.size(), INT_MAX / 10);
    priority_queue<pii, vector<pii>, greater<pii>> q;

    q.push({0, orig});
    dists[orig] = 0;
    while (!q.empty()) {
        auto a = q.top();
        q.pop();
        if (a.se == dest) { // If we already reached the destination
            return a.fi;
        }
        if (dists[a.se] != a.fi) { // If we have found a better option
            continue;
        }
        for (auto b : graph[a.se]) { // Else iterate through the edges
            if (dists[b.fi] > dists[a.se] + b.se) {
                dists[b.fi] = dists[a.se] + b.se;
                q.push({dists[b.fi], b.fi});
            }
        }
    }
    return dists[dest];
}
```

Running time: $\boldsymbol{\mathcal{O}(V + E\log(E))}$
(V = vertices, E = edges)

Observations

- If we ignore the check in line 30, we can return the distances vector, which will contain the shortest distance from `dist` to every other node.

- If we are doing some kind of pruning it is imperative that we prune as many branches as possible in the main loop. That is to say, we should introduce as many `if` statements in line 36 to make sure that we run the `for` loop as few times as possible.

  An example of this approach is problem `UVA-11635`. In that problem, we add a lot of branches to the queue (we may run the `for` loop twice in some nodes) but we prune them in the main loop. Thus the running time is still acceptable.

1.2 Bellman Ford's

Shortest parth from `orig` to every other node. It is slower than Dijkstra but it works on graphs with negative weights.

This algorithm works by trying to relax every edge $V-1$ times. If there are no negative cycles, after $V-1$ iterations, we must have found the minimum distance to every node. Therefore if after these iterations, we run another iteration and the distance to a node decreases, we must have a negative cycle.

```
20 vi bellmanFord(int orig, vector<pair<pii, int>> edges, int n) {
21     vi dist(n, INT_MAX / 10);
22     dist[orig] = 0;
23     for (int i = 0; i < n - 1; ++i) {
24         for (auto e : edges) {
25             dist[e.fi.se] = min(dist[e.fi.se], dist[e.fi.fi] + e.se);
26         }
27     }
28     // dist[i] contains the shortest path from 0 to i
29 
30     //We can now check for a negative cycle
31     bool negativeCycle = false;
32     for (auto e : edges) {
33         if (dist[e.fi.se] > min(dist[e.fi.se], dist[e.fi.fi] + e.se)) {
34             negativeCycle = true;
35         }
36     }
37     return dist;
38 }
```

Running time: $\mathcal{O}(VE)$
(V = vertices, E = edges)

Observations

- If we keep track of the distance that decrease when we check for a negative cycle, we will get at least one node of every negative cycle present in the graph.

  We can use this, for instance, to check if we can reach a node with a cost smaller than a given bound. If it is connected to a node in a negative cycle, it's distance will be as small as we want it to be (by looping in the cycle).

  This can be seen at play in `UVA-10557`

- If we modify slightly the main loop, iteration i will be the result of considering paths of at most $i + 1$ edges:

```
for (int i = 0; i < n - 1; ++i) {
        for (auto e : edges) {
            dists2[e.fi.se] = min(dists2[e.fi.se], dists[e.fi.fi] + e.se);
        }
    dists = dists2;
}
```

  This can be seen at play in `UVA-11280`

1.3 DFS

1.3.1 Articulation points and bridges

These algorightms can be used in undirected graphs, and we will use the following definitions:

- **Articulation point**. A node whose removal would increase the number of connected components of the graph. That is to say that it "splits" a connected component.
- **Bridge**. An edge whose removal increases the number of connected components in the graph.

We will use a modified version of DFS to solve this problem. We mainly introduce two new properties for every node:

- `num`. Time at which the node was first explored by DFS
- `low`. Earliest node that can be found in the DFS spanning tree that starts from this node

When we visit a node, for every edge, there are two options:

- **Tree edge**. This edge point to a node that has not been discovered yet. As such we explore it (calling `dfs`) and, we update the value of `low` for the current node.

  After the update, we can now process the child since we will not visit it again and it's DFS tree has been fully explored

- **Back edge**. This edge points to a node that has already been visited. Therefore, it will have a relatively low `num` and we use it to update the `low` value of the current node

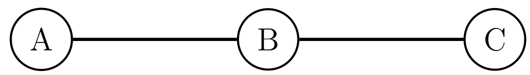
```cpp
vector<vi> adyList; // Graph
vi num, low;        // num and low for DFS
int cnt;            // Counter for DFS
int root, rchild;   // Root and number of (DFS) children
vi artic;           // Contains the articulation points at the end
set<pii> bridges;   // Contains the bridges at the end

void dfs(int nparent, int nnode) {
    num[nnode] = low[nnode] = cnt++;
    rchild += (nparent == root);

    for (auto a : adyList[nnode]) {
        if (num[a] == -1) { // Tree edge
            dfs(nnode, a);
            low[nnode] = min(low[nnode], low[a]);

            if (low[a] >= num[nnode]) {
                artic[nnode] = true;
            }

            if (low[a] > num[nnode]) {
                bridges.insert((nnode < a) ? mp(nnode, a) : mp(a, nnode));
            }
        } else if (a != nparent) { // Back edge
            low[nnode] = min(low[nnode], num[a]);
        }
    }
}
void findArticulations(int n) {
    cnt = 0;
    low = num = vi(n, -1);
    artic = vi(n, 0);
    bridges.clear();

    for (int i = 0; i < n; ++i) {
        if (num[i] != -1) {
            continue;
        }
        root = i;
        rchild = 0;
        dfs(-1, i);
        artic[root] = rchild > 1; //Special case
    }
}
```

Running time: $\mathcal{O}(V + E)$

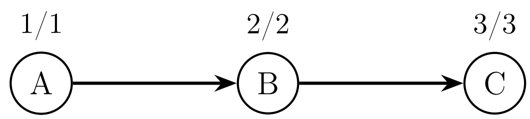
(V = vertices, E = edges)

Explanation

The first graph that we will consider is the following: After applying DFS on A we get the



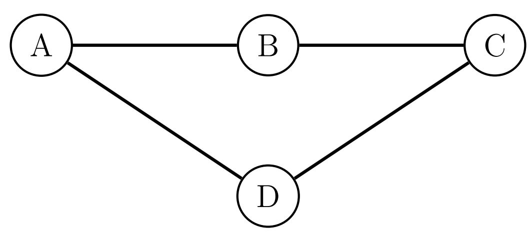
following DFS spanning tree. Above every node, we have included `num / low`.



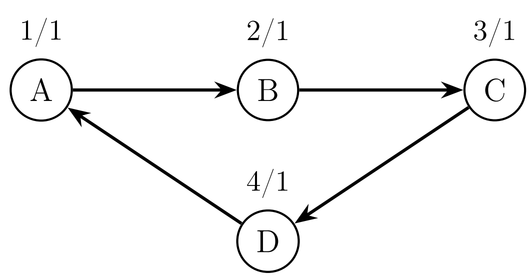
This is a rather simple graph and it's only articulation point is B. This is because it has a child whose `low` value is greater than or equal to B's `num` value. Therefore, there is no connection from that child (namely C) to a node explored before B that does not go through B.

However, we can already see that the root has to be treated as a separate case. The root will be an articulation point iff it has more than one child in its DFS tree. It is important to note that the children of the DFS tree need not be the same as the children of the root in the initial graph.

Let's look at another example



As before, we show the DFS spanning tree:



Now we have no articulation point since there is no node that has a child with a `low` greater than or equal than the parent's `num`. This difference is caused by the fact that now there is an edge that makes C accessible through a path that does not involve traversing B.

In this case, we can see how the root has only one child in the spanning tree but two in the initial graph. This reflects the fact that those numbers need not coincide.

Finally, the condition for a bridge is: `low[child]>num[parent]`. This is equivalent to stating that the child has no other way of reaching either the parent or a node that was explored before the parent

1.3.2 Trajan's algorithm for strongly connected components

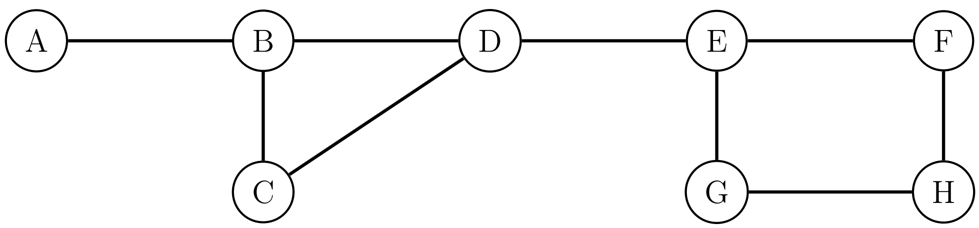
In a directed graph, we say that a subset of vertices comprises a strongly connected component if every vertex is reachable from every other vertex in this subset.

We will now present an algorithm that divides the graph into strongly connected parts that are as large as possible. It will use some of the same concepts as in the previous section.

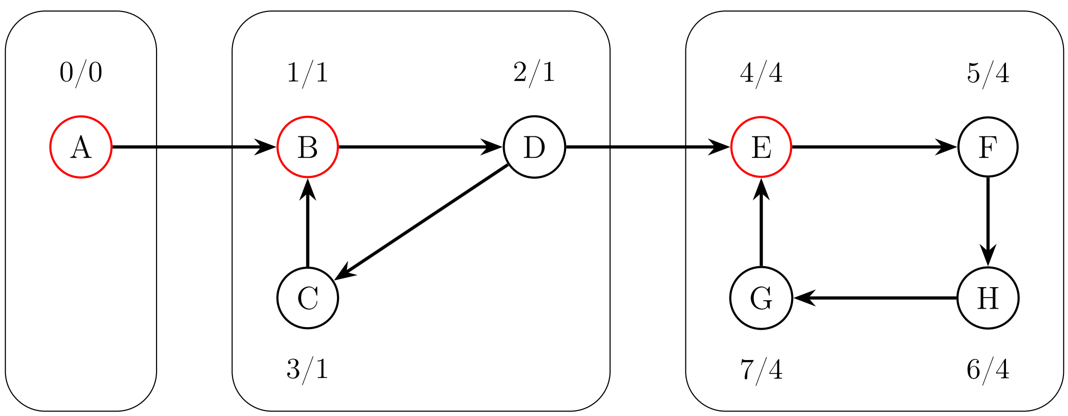
Explanation

The main idea is the following: if when we are done with a node, we have not been able to reach a node that was further back, this one is the root of a SCC.

Let's explain this reasoning with an example:



And, after running DFS, we would get the following spanning tree with three SCCs.



Let's look at the moment in which we close node F. As we can see the `low` value is lower than the `num`. Therefore, there is a connection from F to another node that was visited before and it will be a part of the SCC "generated" by that node.

However, when we close E, we can see that the `low` and the `num` are equal. As a result, there is no way to get to a node that has a number lower than E's through the spanning tree of E. Therefore, there is no way to have a bigger SCC that contains E.

To get the nodes that are part of the SCC, we need to get all the nodes in the stack before the one that "generates" the SCC (g). This is because the stack only contains the nodes that belong to the spanning tree of g and do not belong to any other SCC. Therefore, they must belong to the one generated by g.

Furthermore, all those nodes will have the same `low` value since the `low` any node with a higher value must have been processed before. Therefore, g is accessible from all those nodes and, clearly, all those nodes are accessible from g. Thus, they fulfill the definition of an SCC.

```cpp
vector<vi> adyList; // Graph
vi low, num;        // num and low for DF
int cnt;            // Counter for DFS
stack<int> st;
vi inStack;      // Position in the stack + 1
vector<vi> sccs; // Contains the SCCs at the end

void dfs(int u) {
    low[u] = num[u] = cnt++;
    st.push(u);
    inStack[u] = st.size();
    for (auto a : adyList[u]) {
        if (num[a] == -1) {
            dfs(a);
        }
        if (inStack[a]) {
            low[u] = min(low[u], low[a]);
        }
    }
    if (low[u] == num[u]) { // Root of a SCC
        vi v;
        // Add all the nodes till u (included)
        int lim = inStack[u];
        while (st.size() && st.size() >= lim) {
            v.push_back(st.top());
            inStack[st.top()] = false;
            st.pop();
        }
        sccs.push_back(v);
    }
}
void TarjanSCC(int n) {
    low = num = vi(n, -1);
    inStack = vi(n, 0);
    st = stack<int>();
    cnt = 0;
    sccs = vector<vi>();

    for (int i = 0; i < n; ++i) {
        if (num[i] == -1) {
            dfs(i);
        }
    }
}
```

Running time: $\mathcal{O}(V + E)$

(V = vertices, E = edges)

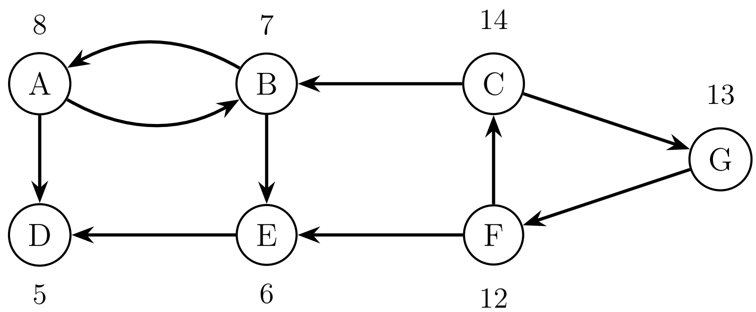
1.4 Kosaraju's

Kosaraju's algorithm is a slightly simpler method for finding the SCC of a graph. On the other hand, it is also somewhat slower than Trajan's approach since it will require running DFS on the graph twice. The algorithm follow this procedure:

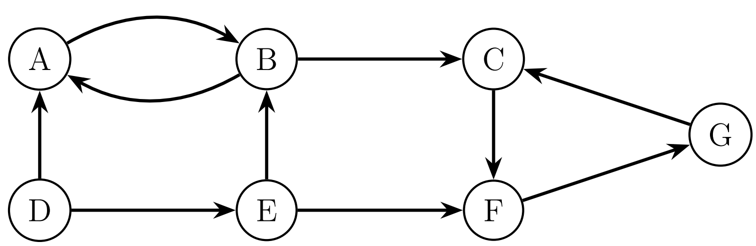
1. Run DFS on the graph and store the vertices in postorder in a list called `postorder`.
2. Reverse the graph
3. Loop through the nodes in `postorder` starting from the one that was closed the last. For each of them check if it has already been added to a SCC (visited). If it hasn't, run DFS from that node and create a SCC with all the nodes that this DFS visits and hadn't been visited before.

Example

Let's look at an example graph where we have run DFS starting first in node A and, since it didn't explore the entire graph, we also run it from node C. We have included the time at which each node was closed:



Now we reverse the graph and we get the following:



Finally, we can loop through the nodes in postorder, starting with the node that was close the last:

$$\text{postorder} = \{C, G, F, A, B, E, D\}$$

1. We run DFS on node C and we get the SCC:

$$s_1 = \{C, F, G\}$$

2. We skip nodes G and F in the list since they are already in a SCC and we run DFS on node A, getting the SCC:

$$s_2 = \{A, B\}$$

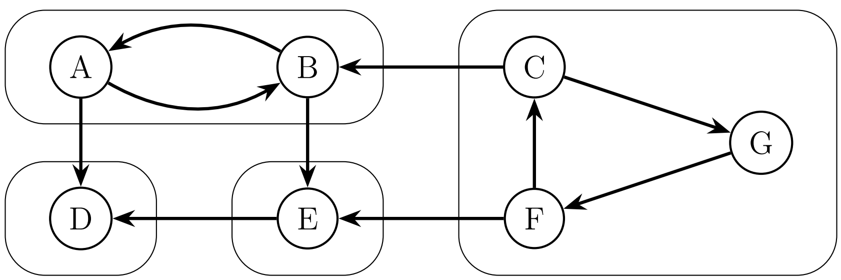
3. We skip node B since it is already in a SCC and we run DFS on node E, getting the SCC:

$$s_3 = \{E\}$$

4. We finally run DFS on node D, and we get the last SCC:

$$s_4 = \{D\}$$

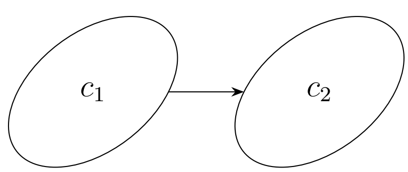
Now, returning to the initial graph, we have found the following SCCs:



1.4.1 Explanation

Let's look at why this algorithm works.

Firstly, we have to take into account that the SCCs of a graph G are preserved when we reverse all the edges and get G^t. The only relevant issue is the order in which we process the SCCs. Let's assume we have a graph that has two SCC's: We have three options:



- *They are not connected.* In this case, it does not matter whether we explore one or the other first since DFS will not "leak" from one of them to the other
- *They are connected only in one direction.* This is the case that the figure shows and it is the most important one. If we explore c_2 before exploring c_1, DFS will first explore the entirety of c_2 and then, when it is exploring c_1 it will not "leak" to c_2 because those nodes are already marked as visited.

- *They are connected in both directions.* This can never happen since, if they were to be connected bidirectionally, they would form a single SCC, not two.

Therefore, the correctness of this algorithm simply depends on exploring the SCCs in the right order.

Let's demonstrate the following claim:

In the previous setting, the maximum closing time of the nodes in c_1 will be greater than the maximum closing time of the nodes in c_2

We just have to distinguish two cases:

- If we started exploring c_2 before c_1, there is no way to get to c_1 from c_2. Therefore, we will start exploring c_1 when we have already closed c_2, which means that all nodes in c_2 will be closed before the nodes in c_1 are even "opened".

- If we started exploring c_1 before c_2, at some point, DFS will leak to c_2 and it will explore it entirely before returning to c_1. Therefore, the closing time of every node in c_2 will be lower than the closing time of the node where DFS was started in c_1.

This completes the proof of that statement. Let's now apply it to check the correctness of this algorithm using induction.

- *Base case:*

  After reversing the graph, we start with the node that was closed the last (a). Let's call it's SCC s_1. Let's assume that there is an edge from s_1 to another SCC (s_α) in G^t, which would make DFS leak to that SCC. However, if that was the case, there would be an edge from s_α to s_1 in G and, therefore, we can apply the previous claim.

  In that scenario, the maximum closing time of s_α would be greater than the maximum closing time of s_1. This is a contradiction because we have stated that $a \in s_1$ is the node that was closed the last.

  Therefore, there cannot be any edge from s_1 to another SCC.

- *Inductive step:*

  Let's now assume that we have already processed n SCCs. We now choose a node b, which is the node with the highest closing time such that $b \notin s_i,\ i = 1, \ldots n$.

  We will call the SCC of this node s_{n+1}. As before, we want to prove that s_{n+1} is not connected to any s_β that hasn't been explored yet in G^t.

  We will assume that there is an edge from s_{n+1} to a $s_\beta \notin \{s_1 \ldots s_n\}$. Therefore, in G, there is an edge from s_β to s_{n+1}, which implies that the maximum closing time of s_β is higher than the maximum closing time of s_{n+1}.

  Let's define $x_\beta :=($ the node with the maximum closing time of s_β $)$. We just stated that the closing time of x_β is greater than the closing time of b (there cannot be a node with a greater closing time in s_{n+1}).

  However, this is a contradiction. Therefore, s_{n+1} cannot be connected to an *SCC* that has not been explored and, thus, DFS will not leak.

$\square$

```cpp
vector<vi> adyList;  // Graph
vector<int> visited; // Visited for DFS
vector<vi> sccs;     // Contains the SCCs at the end

void dfs(int nnode, vector<int> &v, vector<vi> &adyList) {
    if (visited[nnode]) {
        return;
    }
    visited[nnode] = true;
    for (auto a : adyList[nnode]) {
        dfs(a, v, adyList);
    }
    v.push_back(nnode);
}

void Kosaraju(int n) {
    visited = vi(n, 0);
    stack<int> s = stack<int>();
    sccs = vector<vi>();

    vector<int> postorder;
    for (int i = 0; i < n; ++i) {
        dfs(i, postorder, adyList);
    }
    reverse(all(postorder));

    vector<vi> rAdyList = vector<vi>(n, vi());
    for (int i = 0; i < n; ++i) {
        for (auto v : adyList[i]) {
            rAdyList[v].push_back(i);
        }
    }

    visited = vi(n, 0);
    vi data;
    for (auto a : postorder) {
        if (!visited[a]) {
            data = vi();
            dfs(a, data, rAdyList);
            if (!data.empty())
                sccs.pb(data);
        }
    }
}
```

Running time: $\boldsymbol{\mathcal{O}(V+E)}$

(V = vertices, E = edges)