CP implementations

David del Val

December 27, 2020

Contents

Chapter 1

Graphs

1.1 Dijkstra

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

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: $\boldsymbol{\mathcal{O}(VE)}$

(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

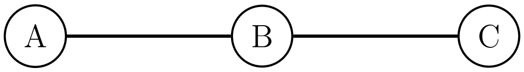
```
vector<vi> adyList; // Graph
vi num, low;        // num and low for DFS
int cnt;            // Counter for DFS
int root, rchild;   // Root and number of (DFS) children
vi artic;           // Contains the articulation points at the end
set<pii> bridges;   // Contains the bridges at the end

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
    }
}
```

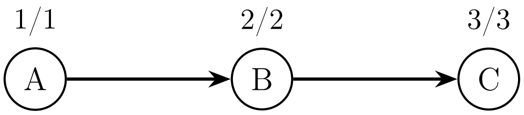
Running time: $\boldsymbol{\mathcal{O}(V + E)}$

(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`.

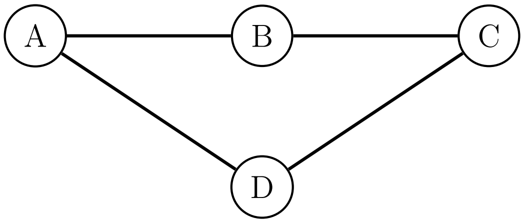




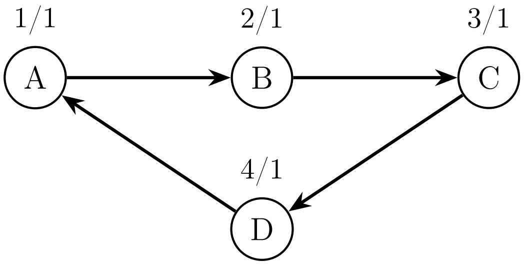
This is a rather simple graph and it's only articulation point is B. This is because it has a child whose `low` value is greater than or equal to B's `num` value. Therefore, there is no connection from that child (namely C) to a node explored before B that does not go through B.

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 algorithm for strongly connected components

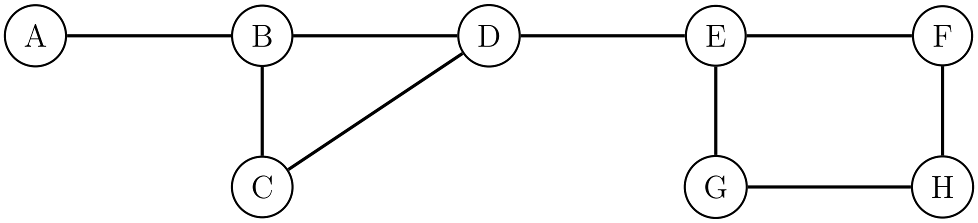
In a directed graph, we say that a subset of vertices comprises a strongly connected component if every vertex is reachable from every other vertex in this subset.

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.

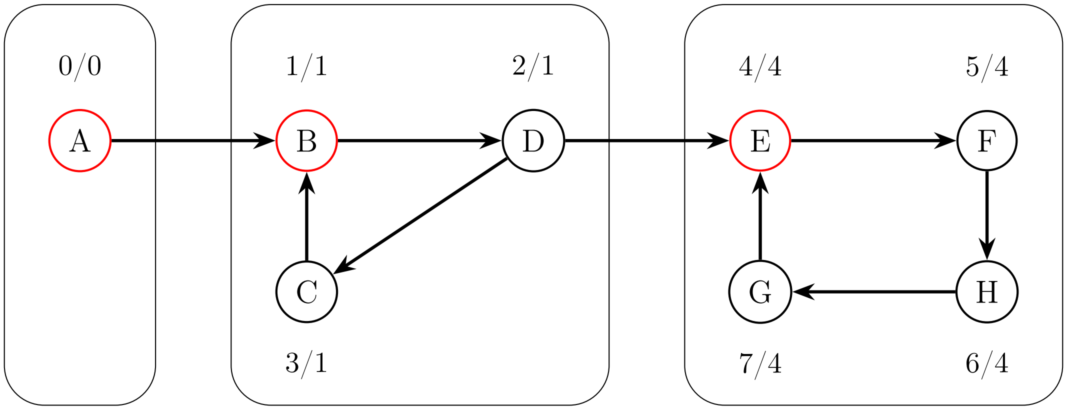
Explanation

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