DFS and application: lecture notes

Moscow International Workshop 2020

November 29, 2020

1 DFS basics

Vertex colors:

- $visited[v] = 0$ (before visiting): White.
- $visited[v] = 1$ (after visiting, but before leaving): Gray.
- $visited[v] = 2$ (after leaving): Black.

Edge types:

- Tree edge $v \to u$: the call to u was made from v, u is a child of v in the DFS tree. u is White when $v \to u$ is processed.
- Back edge $v \to u$: u is an ansector of v in the DFS tree. u is Gray when $v \to u$ is processed.
- Forward edge $v \to u$: u is a descendant of v in the DFS tree. u is Black when $v \to u$ is processed.
- Cross edge $v \to u$: u is neither an ancestor nor a descendant of v in the DFS tree. u is Black when $v \to u$ is processed.

Observation: no cross edges when graph is undirected!

In/out-times $tin[v], tout[v]$: whenever entering/leaving vertex v, store the value of the global counter T in $tin[v]/tout[v]$, increment T.

Observation: v is an ancestor of u in the DFS tree iff $tin[v] \leqslant tin[u] < tout[u] \leqslant tout[v]$.

Observation: if one of v, u is an ancestor of the other, then the one with the smallest tin is closer to the root of the DFS tree.

2 Bridges, articulation points

$up[v]$ = the smallest $tin[u]$ among u reachable by a back-edge from the subtree of v.

dfs(v) pseudocode with $up[v]$ *computation*:

```
v is Gray;
up[v] = ∞;
tin[v] = T, increment T;
for u adjacent to v do
    if reverse of the parent edge then
        continue
    end
    if u is White then
        dfs(u);
        up[v] = min(up[v], up[u]);
    end
    else if u is Gray then
        up[v] = min(up[v], tin[u];
    end
end
v is Black;
```

An edge vu is a *bridge* if its removal disconnects the component containing it.

Only tree edges can be bridges!

A tree edge $v \to u$ is a bridge if $up[u] \geqslant tin[u]$.

A vertex v is an *articulation point* if its removal disconnects the component containing it.
A **non-root** vertex v is an articulation point if it has a child u such that $up[u] \geqslant tin[v]$.
The root vertex v is an articulation point if it has more than one child.

3 Biconnected components

A graph is *edge-biconnected* if it doesn't contain bridges.
A graph is *vertex-biconnected* if it doesn't contain articulation points.

An *edge-biconnected component* consists of **vertices** such that each pair of distinct vertices v, u in the same component lie on a common simple cycle.
A *vertex-biconnected component* consists of **edges** such that each pair of distinct edges e, f in the same component lie on a common simple cycle.

To find edge-biconnected components:

- Use the same dfs as above, and maintain a stack of vertices.
- Each time a White vertex is visited, push it to the stack.
- If the call from v to u is finished, and $up[u] \geqslant tin[u]$, pop all the vertices until (and including) u from the stack. The popped vertices from a new component.
- When the call to the root is finished, the vertices in the stack form a new component.

To find vertex-biconnected components:

- Use the same dfs as above, and maintain a stack of edges.
- Each time a tree edge of a back edge is encountered, push it to the stack.
- If the call from v to u is finished, and $up[u] \geqslant tin[v]$ **or** v is the root, pop all the edges until (and including) $v \to u$ from the stack. The popped edges from a new component.

4 Euler cycle/path

An Euler cycle in a graph visits all its edges exactly once and returns to the starting vertex.
An Euler cycle in an undirected graph exists if the graph is connected and all vertex degrees are even.
An Euler cycle in a directed graph exists if the graph is (strongly) connected and $indeg[v] = outdeg[v]$ for all vertices v.

*visit(v) pseudocode (construct an Euler tour if it exists)***:**

```
while ptr[v] < len(adj[v]) do
    edge e = adj[v][ptr[v]];
    if e was used before then
        increment ptr[v], continue;
    end
    u = the other endpoint of e; mark e as used;
    visit(u);
    prepend e to the Euler cycle;
end
```

Before calling visit(v), initialize all $ptr[v] = 0$.
In undirected case, mark edges as used in **both directions**.

An *Euler path* in a graph visits all its edges exactly once, and possibly does not return to the starting vertex.
An Euler path in an undirected graph exists if the graph is connected, and at most two vertex degrees are odd. If odd-degree vertices exist, they are the endpoints of the path, otherwise the path is also an Euler cycle.
An Euler cycle in a directed graph exists if the graph is (strongly) connected and $indeg[v] = outdeg[v]$ for all vertices v but at most two. If vertices with $indeg[v] \neq outdeg[v]$ exist, then the starting vertex must have $outdeg[v] - indeg[v] = 1$, and the finish vertex must have $outdeg[v] - indeg[v] = -1$. If they do not exist,

then the path is also an Euler cycle.
An Euler path can be found by calling visit for the appropriate starting vertex.

5 Strongly connected components

A directed graph is *strongly connected* if any vertex is reachable from any other.
If vertices v, u are mutually reachable, write $v \sim u$.
$v \sim u$ is an equivalence relation: $v \sim u, u \sim w \implies v \sim w$.
A *strongly connected component (SCC)* of a directed graph is an equivalence class with respect to $v \sim u$.

Kosaraju's algorithm for finding SCCs:

```
for all vertices v do
    if v is White then
        dfs(v);
    end
end
make all vertices White;
for v in decreasing order of tout[v] do
    if v is not White then
        continue;
    end
    dfs(v) using reverse edges of the graph;
    all newly visited vertices form a new SCC;
end
```

6 2-SAT problem

Problem: find an assignment for boolean variables $x_1, \ldots, x_n$ that satisfies a formula $C_1 \wedge \ldots \wedge C_m$, where each C_i is a disjunction of at most two variables or their negations.

$(x_i \vee x_j) \equiv (x_i \Rightarrow \overline{x_j}) \wedge (x_j \Rightarrow \overline{x_i})$.
Create a directed graph with vertices $x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}$, and create a directed edge for each implication.
Note: for each implication $x_i \Rightarrow x_j$ there must be an implication $\overline{x_j} \Rightarrow \overline{x_i}$!
Find SCCs of the graph. If for any i vertices $x_i, \overline{x_i}$ are in the same SCC, then there is no solution.
Otherwise, set $x_i = T$ if $tout[x_i] < tout[\overline{x_i}]$, and $x_i = F$ otherwise.