Algorithms on Graphs

Study Notes by Ruijie Fang

1 DFS

Depth-first-search (DFS) generates a depth-first traversal of a graph $G = (V, E)$, an array $P[\cdot]$ that records the previsit order of the vertices in V, and an array $O[\cdot]$ that records the postvisit order. It runs in $O(V + E)$-time.

Algorithm 1 (Depth-first search). $DFS(G, u)$:

$VIS[u] := 1;$

$previsit(v);$

for $(u, v) \in E$:

  if $\neg VIS[v]$: $DFS(G, v);$

$postvisit(v);$

The subroutines *postvisit* and *previsit* perform constant-time maintenance work; as shown below, we can augment these procedures to perform different tasks.

2 Finding connected components

We can call DFS in a loop through $v \in E$ to count the number of connected components:

Algorithm 2 (Counting connected components in G). $CountCC(G)$:

$t := 0;$

$VIS[0...|V|] := 0;$

for $v \in V$:

if $\neg VIS[v]$:
    $dfs(v);\ t := t + 1;$

The overall time complexity is still $O(|V| + |E|)$ since we only visit each vertex once.

3 Bipartite testing

Theorem 3. *G is bipartite* $\leftrightarrow$ *G is bicolorable.*

We can augment the DFS procedure for bipartite testing. We use an extra array $C[\cdot]$ to denote the color of each vertex $v \in V$. From theorem 3, we denote Black as 1 and White as 0, and a bipartite graph must be correctly colored using 0's and 1's.

Algorithm 4 (Bipartite testing)**. Precondition:** *Initialize* $C[\cdot]$ *to -1 and set* $C[0] := 0$. *Call isBipartite*$(G, 0)$.

Postcondition: *Returns 1 if the graph is bipartite; otherwise returns 0.*

```
isBipartite(G, u) :
    nc := ¬C[u];
    for v ∈ V:
        if ¬(C[v] = −1) ∧ ¬(C[v] = nc): return 0;
        elif C[v] = −1:
            C[v] := nc;
            return isBipartite(G,v);
```

The overall runtime is still $O(|V| + |E|)$.

4 Articulation points

Definition 5 (Articulation points of a graph)**.** An articulation point of G is a vertex $p \in V$ such that the deletion of p from G increases the number of connected components in G.

Lemma 6. *The root of a DFS spanning tree of G is an articulation point if and only if it has more than one children in the spanning tree.*

Lemma 7. *A node v in the DFS spanning tree of G is an articulation point if and only if there exists no back edge from a tree descendant of v to a tree parent of v.*

Lemmas 6 & 7 results in a DFS-based linear-time algorithm for finding articulation points. Let $pre[\cdot]$ denote the order in which DFS traverses the vertices. Let $low[u] = \min\{pre[v]|v$ is an ancestor of u in the DFS spanning tree$\}$. In other words, let $low[u]$ denote the neighbor of u that is nearest to the root of the DFS spanning tree. Then the set of articulation points are $\{v \in V|low[v] \geq pre[v]\}$ (the complement set contains all points whose descendants have back edges) and, if the root node in the DFS spanning tree has more than 1 children, the root node.

Algorithm 8 (Finding articulation points in a graph)**. Precondition:** *v stands for the parent of u in the DFS spanning tree. Call with $ArticulationPoint(G, 0, -1)$.*

```
p := 1; pre[0...|V|] := 0;

ArticulationPoints(G, u, v) :

    pre[u] := p;
    low[u] := pre[u];
    p := p + 1;
    ch := 0; // children count
    for (u, w) ∈ E:
        if ¬pre[v]:
            ch := ch + 1;
            low[u] := min{low[u], dfs(G, w, u)};
            if low[w] ≥ pre[u]:
                report u as articulation point;
        elif pre[w] < pre[u] ∧ w ≠ v :
            low[u] := min{low[u], pre[w]};
    if v < 0 ∧ ch = 1:
        report u as NOT an articulation point;
    return low[u];
```

5 Bridges

Definition 9 (Bridges/cut edges of a graph)**.** A bridge of G is an edge $(u, v) \in E$ such that the deletion of (u, v) from G increases the number of connected components in G.

Continuing our discussion from section 4, we find that if $low[v] > pre[u]$, then edge (u, v) is a bridge. It follows that this characterization suffices for finding bridges, and we only have to modify $ArticulationPoints(\cdot)$ slightly for this case.

Algorithm 10 (Finding bridges in a graph)**. Precondition:** *v stands for the parent of u in the DFS spanning tree. Call with $Bridges(G, 0, -1)$.*

$p := 1;\ pre[0...|V|] := 0;$

$Bridges(G, u, v):$

> $pre[u] := p;$
>
> $low[u] := pre[u];$
>
> $p := p + 1;$
>
> $ch := 0;$ *// children count*
>
> **for** $(u, w) \in E$*:*
>
>   **if** $\neg pre[v]$*:*
>
> $ch := ch + 1;$
>
> $low[u] := \min\{low[u], dfs(G, w, u)\}$**;**
>
>   **if** $low[w] > pre[u]$*:*
>
>   **report** (u, v) *as bridge;*
>
>   **elif** $pre[w] < pre[u] \wedge w \neq v :$
>
> $low[u] := \min\{low[u], pre[w]\};$
>
> **return** $low[u];$

6 Biconnected components

We deal with undirected graphs in this section.

Definition 11. A graph G is biconnected if and only if for all $u, v \in V$, there exists at least two vertex-disjoint paths from u to v.

$\leftrightarrow$ for all $u, v \in V$, u and v are in a simple cycle (there exists no articulation points).

Definition 12. A graph G is edge-biconnected if and only if for all $u, v \in V$, there exists at least two edge-disjoint paths from u to v.

$\leftrightarrow$ for all $e \in E$, e is inside at least a single simple cycle (all edges are not bridges).

Definition 13 (Biconnected component of a graph)**.** A subgraph $G' \subseteq G$ is called a biconnected component of G is a maximum biconnected subgraph of G.

Definition 14 (Edge-biconnected component of a graph)**.** Analogous to Def. 13, but the maximum subgraph is edge-biconnected.

By definition, we can find all edge-biconnected components by a graph by finding and deleting all the bridges inside the graph. The resulting connected components are all edge-biconnected.

By definition, each edge belongs to precisely one biconnected subgraph, but a vertex might belong to two biconnected components.

Algorithm 15 (Finding a biconnected component)**. Preconditions:** $pre[1...|V|] := 0$; $isArticulationPoint[1...|V|] := 0$; $bccno[1...|V|] := 0$; $bcc[1...|V|] := \{\}$; $p := 1$; $bccCnt := 0$;

Initialize $S := Stack()$;

$FindBCC(u, p):$ *// p is the parent of u, initially -1.*

$low[u] := pre[u] := p$;

$p := p + 1$;

$ch := 0$;

for $(u, v) \in E$:

if $pre[v] = 0$:

$S.push(u, v)$;

$ch := ch + 1$;

$dfs(v, u)$;

```
        low[u] := min{low[u], low[v]};
        if low[v] ≥ pre[u]: // u is an articulation point
            isArticulationPoint[u] := 1;
            bccCnt := bccCnt + 1;
            while ¬S.empty():
              (u', v') := S.top(); S.pop();
              if (bccno[u'] ≠ bccCnt):
                add u' to bcc[bccCnt];
                bccno[u'] := bccCnt;
              if (bcc[v'] ≠ bccCnt):
                add v' to bcc[bccCnt];
                bccno[v'] := bccCnt;
              if u' = u ∧ v' = v:
                break;
    elif pre[v] < pre[u] ∧ v ≠ p:
        S.push(u, v);
        low[u] := min{low[u], pre[v]};
if p < 0 ∧ ch > 1) isArticulationPoint[u] := 1;
```

For finding edge-biconnected components, we can just remove all the bridges and count the number of connected components.

7 Strongly connected components of directed graphs

All vertices within the same SCC (Strongly Connected Component) of a directed graph G can reach each other. However, due to the nature of the directed graph, finding SCCs is not as simple as finding connected components.

Tarjan's Algorithm. Tarjan's idea is still DFS-based, but it uses extra information to separate the different SCCs within the same DFS traversal. The resulting algorithm has the same time bound as DFS. For a single SCC $C \subseteq G$, the first vertex encountered during the DFS traversal is the ancestor of all other vertices in C within the DFS spanning tree. If we output C immediately after we visited its first vertex, we can separate different SCCs efficiently. The key to the problem, therefore, is to record the first vertex in C encountered during the DFS traversal of G. This makes this problem highly similar to finding articulation points: if a vertex u is the first vertex encountered, then there must not be a back edge to u's ancestor in the descendants of u.

Algorithm 16 (Tarjan's SCC Algorithm)**. Preconditions:** *Initialize* $pre[1...|V|] := 0$, $lowlink[1...|V|] := 0$, $sccno[1...|V|] := 0$, $p := 1$, $sccCnt := 0$;

Initialize $S := Stack()$;

$TarjanSCC(u)$:

$pre[u] := lowlink[u] := p$;

$p := p + 1$;

$S.push(u)$;

for $(u, v) \in E$:

  if $pre[v] = 0$:

    $dfs(v)$;
    $lowlink[u] := \min\{lowlink[u], lowlink[v]\}$;

  elif $sccno[v] = 0$:

    $lowlink[u] := \min\{lowlink[u], pre[v]\}$;

if $lowlink[u] = pre[u]$:

  $sccCnt := sccCnt + 1$;

  while $\neg S.empty()$:

    $v := S.top()$; $S.pop()$;
    $sccno[v] := sccCnt$;
    if $v = u$:
      break;

8 2SAT

Some day.