DFS Search on Undirected Graphs

Algorithm : Design & Analysis

[13]
In the last class…

- Directed Acyclic Graph
  - Topological Order
  - Critical Path Analysis
- Strongly Connected Component
  - Strong Component and Condensation
  - Leader of Strong Component
  - The Algorithm
DFS Search on Undirected Graph

- Undirected and Symmetric Digraph
- UDF Search Skeleton
- Biconnected Components
  - Articulation Points and Biconnectedness
  - Biconnected Component Algorithm
  - Analysis of the Algorithm
What’s the Different for Undirected

- The issue related to traversals for undirected graph is that **one edge may be traversed for two times in opposite directions**.
- For an undirected graph, the depth-first search provides an orientation for each of its edges; they are oriented in the direction in which they are first encountered.
Nontree edges in symmetric digraph

- Cross edge: not existing.
- Back edge:
  - Back to the direct parent: second encounter
  - Otherwise: first encounter
- Forward edge: always second encounter, and first time as back edge
Modifications to the DFS Skeleton

- All the second encounter are bypassed.

- So, the *only substantial modification* is for the possible back edges leading to an ancestor, but not direct parent.

- We need know the *parent*, that is, the direct ancestor, for the vertex to be processed.
DFS Skeleton for Undirected Graph

- \textbf{int} dfsSweep(IntList[] \textit{adjVertices}, \textbf{int} n, \ldots)
- \textbf{int} ans;
- \texttt{<Allocate color array and initialize to white>}
- For each vertex \(v\) of \(G\), in some order
- \textbf{if} (color\([v]\]==white)
- \hspace{1cm} \textbf{int} vAns=dfs(\textit{adjVertices}, \textit{color}, v, -1, \ldots);
- \hspace{1cm} \texttt{<Process vAns>}
- \hspace{1cm} // Continue loop \hspace{1cm} Recording the parent
- \textbf{return} ans;
DFS Skeleton for Undirected Graph

```c
int dfs(IntList[] adjVertices, int[] color, int v, int p, ...)
{
    int w; IntList remAdj; int ans;
    color[v]=gray;
    <Preorder processing of vertex v>
    remAdj=adjVertices[v];
    while (remAdj≠nil)
        w=first(remAdj);
        if (color[w]==white)
            <Exploratory processing for tree edge vw>
            int wAns=dfs(adjVertices, color, w, v ...);
        < Backtrack processing for tree edge vw , using wAns>
        else if (color[w]==gray && w≠p)
            <Checking for nontree edge vw>
            remAdj=rest(remAdj);
        <Postorder processing of vertex v, including final computation of ans>
        color[v]=black;
    return ans;
}
```

In all other cases, the edges are the second encounter, so, ignored.
Complexity of Undirected DFS

- If each inserted statement for specialized application runs in constant time, the time cost is the same as for directed DFS, that is $\Theta(m+n)$.
- Extra space is in $\Theta(n)$ for array \textit{color}, or activation frames of recursion.
Definition of Biconnected Components

- Biconnected component
  - Biconnected graph
  - Bicomponent: a maximal biconnected subgraph

- Articulation point
  - \( v \) is an articulation point if it is in every path from \( w \) to \( x \) (\( w, x \) are vertices different from \( v \))

- A connected graph is biconnected if and only if it has no articulation points.
Bicomponents

Partitioning the set of edges, not of the vertices
Bicomponent Algorithm: the Idea

Ancestors of $v$

$v$ is an articulation point if and only if no back edges linking any vertex in $w$-rooted subtree and any ancestor of $v$.

If $v$ is the articulation point farthest away from the root on the branch, then one bicomponent is detected.

Subtree rooted at $w$

Back edge
Keeping the Track of Backing

- Tracking data
  - For each vertex \( v \), a local variable \( back \) is used to store the required information, as the value of \( discoverTime \) of some vertex.

- Testing for bicomponent
  - At backtracking from \( w \) to \( v \), the condition implying a bicomponent is:
    \[
    wBack \geq discoverTime(v)
    \]
    (where \( wBack \) is the returned back value for \( w \))
Updating the value of `back`

- `v` first discovered
  
  `back = discoverTime(v)`

- Trying to explore, but a back edge encountered
  
  `back = min(back, discoverTime(w))`

- Backtracking from `w` to `v`
  
  `back = min(back, wback)`

Which means: the back value of `v` is the smallest discover time a back edge “sees” from **any** subtree of `v`.

And, when this value is not larger than the discover time of `v`, we know that there is at least one subtree of `v` connected to other part of the graph only by `v`. 
Bicomponent: an Example

first back edge encountered

second back edge encountered
Bicompontent: an Example

backtracking

third back edge encountered
Bicomponent: an Example

backtracking: gBack=discoverTime(B), so, first bicomponent detected.
Bicomponent: an Example

Backtracking from B to E: 
bBack=discoverTime(E), so, 
the second bicomponent is detect

Backtracking from E to F: 
eBack>discoverTime(F), so, 
the third bicomponent is detect
int bicompDFS(v)
    color[v]=gray; time++; discoverTime[v]=time;
    back=discoverTime[v];
    while (there is an untraversed edge vw)
        <push vw into edgeStack>
        if (vw is a tree edge)
            wBack=bicompDFS(w);
            if (wBack≥discoverTime[v])
                Output a new bicomponent
                by popping edgeStack down through vw ;
                back=min(back, wBack);
            else if (vw is a back edge)
                back=min(discoverTime[v], back);
        time++; finishTime[v]=time; color[v]=black;
    return back;
Correctness of Bicomponent Algorithm

- We have seen that:
  - If $v$ is the articulation point farthest away from the root on the branch, then one bicomponent is detected.

- So, we need only prove that:
  - In a DFS tree, a vertex (not root) $v$ is an articulation point if and only if
    1. $v$ is not a leaf;
    2. some subtree of $v$ has no back edge incident with a proper ancestor of $v$. 
In a DFS tree, a vertex (not root) $v$ is an articulation point if and only if (1) $v$ is not a leaf; (2) some subtree of $v$ has no back edge incident with a proper ancestor of $v$.

$\iff$ Trivial

$\Rightarrow$

- By definition, $v$ is on every path between some $x,y$ (different from $v$).
- At least one of $x,y$ is a proper descendent of $v$ (otherwise, $x \leftrightarrow \text{root} \leftrightarrow y$ not containing $v$).
- By contradiction, suppose that every subtree of $v$ has a back edge to a proper ancestor of $v$, we can find a $xy$-path not containing $v$ for all possible cases (only 2 cases)
Case 1

Case 1.1: another is not an ancestor of \( v \)

Case 1.2: another is an ancestor of \( v \)

suppose that every subtree of \( v \) has a back edge to a proper ancestor of \( v \), and, exactly one of \( x, y \) is a descendant of \( v \).
suppose that every subtree of $v$ has a back edge to a proper ancestor of $v$, and, both $x, y$ are descendants of $v$. 
Home Assignments

- pp.380-
  - 7.28
  - 7.35
  - 7.37
  - 7.38
  - 7.40