Lecture 8

Yonghui Wu
Stony Brook University, Fudan University
yhwu@fudan.edu.cn

- Connectivity of Undirected Graphs
- Maximum Matching in Bipartite Graphs

Connectivity of Undirected Graphs

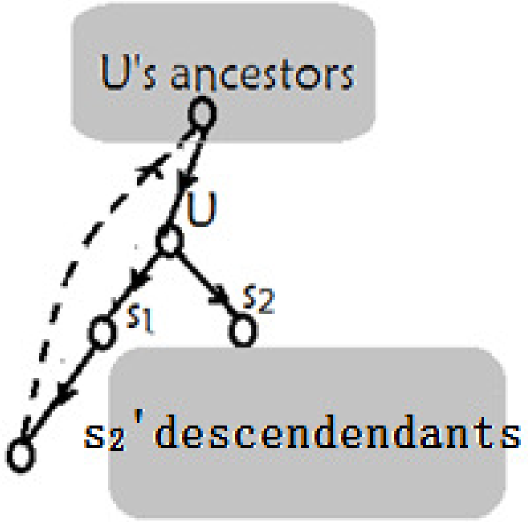
- Connectivity of Undirected Graphs
  - Let $G(V, E)$ be a connected graph.
    - A cut vertex of G is a vertex whose removal disconnects G.
    - A bridge (or a cut edge) of G is an edge whose removal disconnects G.
    - The vertex-connectivity of a graph is the minimum number k of vertices that must be removed to disconnect the graph.
    - The edge-connectivity of a graph is the minimum number k of edges that must be removed to disconnect the graph.
  - The vertex-connectivity and the edge-connectivity of a graph show connectivity of a graph.

- Connected component & Biconnected component
  - A connected component of a graph *G* is a connected subgraph of *G* that is not a proper subgraph of another connected subgraph of *G*.
  - In an unconnected graph, connected components without a cut vertex are called biconnected components. A connected subgraph without a cut vertex is also called a block.

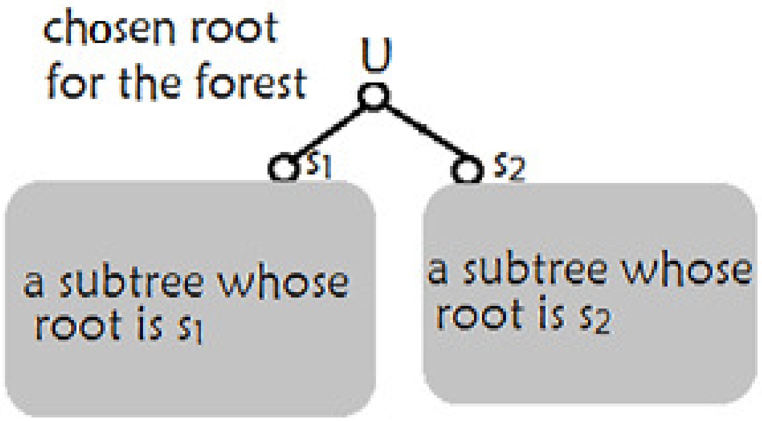
- Function *low* is used to get cut vertices and bridges of a connected graph, and biconnected components of a graph.
- Suppose *pre*[*v*] is the sequence number of vertex *v* in DFS traversal. That is, *pre*[*v*] is the time that vertex *v* is visited. Function *low*[*u*] is the *pre*[*v*] of vertex *v* which is the earliest visited ancestor of *u* and *u*'s descendants.
- $low[u] = \min_{(u,s),(u,w)\in E}\{pre[u], low[s], pre[w]\}$
- where *s* is a child of *u*, and (*u*, *w*) is a back edge.

- In DFS, edges can be classified into four types:
    - Branch edge *T*: Edge (u, v) is a branch edge, if it is the first time that v is visited in DFS.
    - Back edge *B*: Edge (u, v) is a back edge, if u is a descendant of v, and v has been visited, but all descendants of v haven't been visited.
    - Forward edge *F*: Edge (u, v) is a forward edge, if v is a descendant of u, all descendants of v have been visited and $pre[u]<pre[v]$.
    - Cross edge *C*: all other edges (u, v). That is, u and v has no ancestor-descendant relationship in a DFS tree, or u and v are in different DFS trees. All descendants of v have been visited and $pre[u]>pre[v]$.

Function *low* is used to get cut vertices in a connected graph



Property 1
(a)



Property 2
(b)

- Property 1: If vertex *u* isn't a root, *u* is a cut vertex if and only if these exists a child *s* of *u*, $low[s] \geq pre[u]$. That is to say, there is no back edge from *s* and its descendants to *u*'s ancestors.
- In Figure (a), although in the subtree whose root is *s*1 there is a back edge to *u*'s ancestor, there is no back edge to *u*'s ancestor from *s*2 or *s*2's descendants. If *u* is removed, the graph is not connected.

- In an undirected graph, there are only branch edges and back edges. We can calculate *low* and *pre* through DFS, and find whether Property 1 holds or not. The process is as follow.
- If (v, w) is a branch edge *T* (*pre*[*w*]==-1), and if there is no back edge from *w* or *w*'s descendants to *v*'s ancestors (*low*[*w*]≥*pre*[*v*]), then vertex *v* is a cut vertex, and *low*[*v*]=*min*{*low*[*v*], *low*[*w*]}.
- If (v, w) is a back edge *B* (*pre*[*w*]!=-1), then *low*[*v*]=*min*{*low*[*v*], *pre*[*w*]}.

- Property 2: If u is selected as the root, then u is a cut vertex if and only if it has more than one child (Figure (b)).
- In Figure (b), root u has two subtrees whose roots are $s1$ and $s2$ respectively, and there is no cross edge C between the two trees (in an undirected graph, there is no cross edge C). Therefore the graph isn’t connected after vertex u is deleted, and vertex u is a cut vertex.

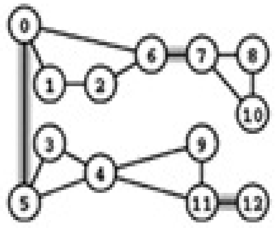
- Based on above two properties, the algorithm calculating cut vertices is as follow.

```
for($i = 0$; $i < n$; $i++$)          //Initialization
    $pre[i] = -1$;
$low[s] = pre[s] = d = 0$;  // vertex $s$: start vertex
$p = 0$;      // the number of children for vertex $s$
for (each $w \in adj[s]$)  $p++$;
if ($p > 1$)
    $s$ is a cut vertex and exit;      //Property 2
$fund\_cut\_point(s)$;          // Property 1
```

Function *low* is used to get the bridge in a connected graph

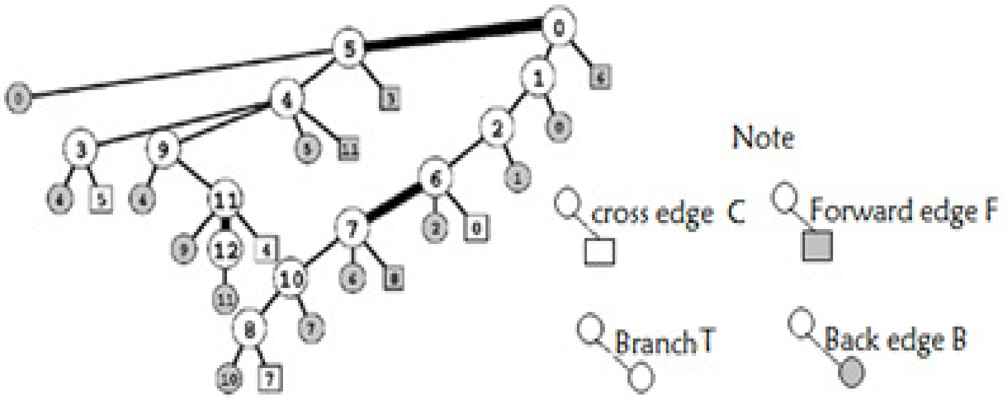
- In an undirected graph, edge (*u*, *v*) is a bridge if and only if (*u*, *v*) is not in any simple circuit.
- The method determining whether an edge is a bridge or not is as follow. Edge (*u*, *v*) is a branch edge discovered by DFS. If there is no back edge connecting *v* and its descendants to *u*'s ancestors; that is, $low[v]>pre[u]$ or $low[v]==pre[v]$; then deleting (*u*, *v*) leads *u* and *v* aren't connected. Therefore edge (*u*, *v*) is a bridge.

- In Figure (a), DFS is used, a DFS tree is gotten as Figure (b), and *pre* and *low* for all vertices are showed in Figure (c). Obviously for v_5, v_7, and v_{12}, $low[v]==pre[v]$, and (v_0, v_5), (v_6, v_7), and (v_{11}, v_{12}) satisfy $low[v]>pre[u]$ for edge (u, v). These edges are bridges in (a).



undirected grapth

(a)



DFS tree

(b)

node number	0	1	2	3	4	5	6	7	8	9	10	11	12
Pre[v]	0	7	8	3	2	1	9	10	12	4	11	5	6
Low[v]	0	0	0	1	1	1	0	10	10	2	10	2	6

The nodes of the pre value and low value

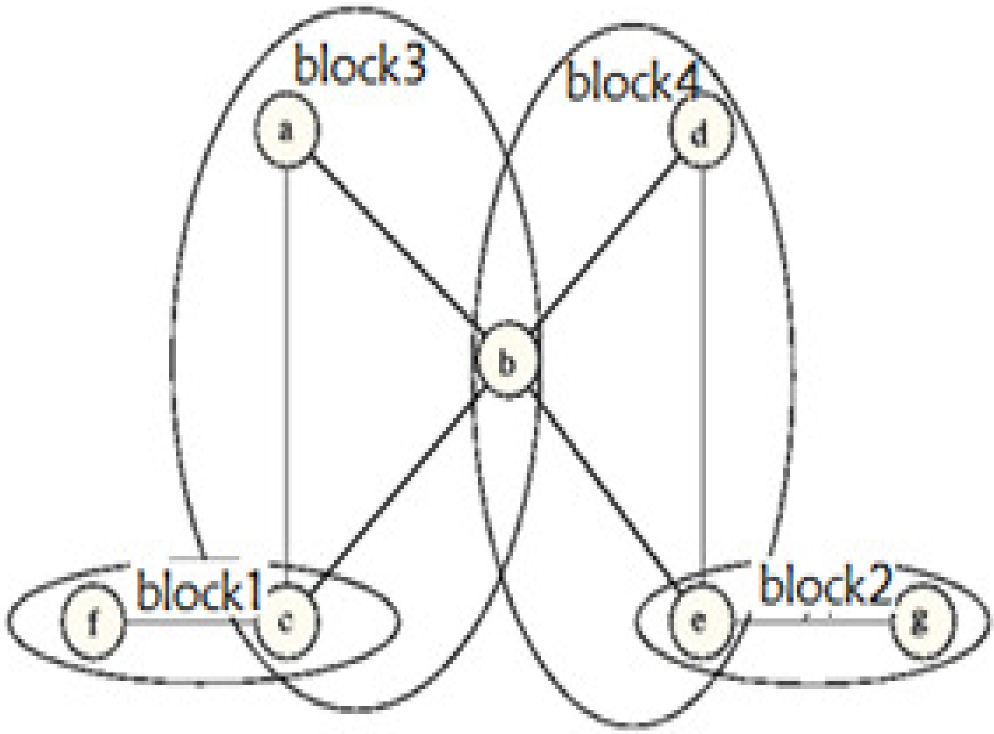
(c)

- In an undirected graph there are only branch edges and back edges. DFS can be used to calculate *low* and *pre* for vertices (initial values for *pre*[ ] are -1), and calculate bridges in the undirected graph. The method is as follow.
- If (v, w) is a branch edge ($pre[w]==-1$), and if there is no back edge from w or w's descendants to u's ancestors, $((low[w]==pre[w])||(low[w]>pre[v]))$, then (v, w) is a bridge, and $low[v]=\min\{low[v], low[w]\}$.
- If (v, w) is a back edge ($pre[w]!=-1$), then $low[v]=\min\{low[v], pre[w]\}$.

```
void fund_bridge (v);              // DFS to find bridges from vertex v
{ int w;
  low[v]= pre[v]=++d;
  for (each w∈ the set of adjacent vertices for v) &(w!=v)  //
Search edge(v, w)
  { if (pre[w]==–1)                  // if (v, w) is a branch edge
     { fund_bridge (w);
          if ((low[w]== pre[w])||( low[w]> pre[v]))
               (v, w) is a branch edge;
          low[v]=min{ low[v], low[w]};
 };
     else low[v]=min{ low[v], pre[w]};  // if (v, w) is a back edge
  }
}
```

Function *low* is used to get biconnected components

- A biconnected component is a connected component without a cut vertex. Biconnected components of a graph are partitions of edges of the graph, that is, every edge must be in a block, and two different blocks don’t contain common edges.
- In Figure 11.6, vertex *b* is a common vertex for block 3 and block 4, vertex *c* is a common vertex for block 3 and block 1, and vertex *e* is a common vertex for block 2 and block 4. The three vertices are cut vertices for the graph. The graph isn’t connected when one of the three vertices is deleted.



cut vertices b,c,e are common vertices for two blocks

- The key to finding a block in an undirected graph is to find a cut vertex. DFS is used to get *low* and *pre* (initial values for *pre*[ ] are -1) and calculate blocks in the undirected graph. The process is as follow.
- For vertex *v*, *u* is the parent for *v*: if *u* is the root, (*u*, *v*) is the first edge for the block; else suppose *f* is *u*'s parent. If *u* is deleted, *v* and *f* aren't connected, then {*f*, *u*, *v*} isn't biconnected, (*u*, *v*) is the first edge for the new block; else (*u*, *v*) and (*f*, *u*) is in a same block. A stack is used to store vertices in the current block.

Knights of the Round Table

- Source: ACM Central Europe 2005
- IDs for Online Judge: POJ 2942, UVA 3523

Maximum Matching in Bipartite Graphs

- A bipartite graph is a graph that its vertex set can be divided into two disjoint subsets such that each edge connects a vertex in one of the two subsets to a vertex in the other subset.
- Given a bipartite graph $G(V, E)$, a matching is a subset of edges $M \subseteq E$, if there is no common vertex for any two edges in M.
- A maximum matching is a matching of maximum cardinality, that is, a matching M is called a maximum matching, if for any other matching M', $|M| \geq |M'|$.

- finding a maximum matching in a bipartite graph

- A perfect matching is a matching which matches all vertices of the graph. That is, every vertex of the graph is incident to exactly one edge of the matching. Every perfect matching must be a maximum matching.

- For a bipartite graph, Hungarian algorithm is used to find a maximum matching or a perfect matching.

Hungarian algorithm used to find a maximum matching

- Hungarian algorithm is the foundation for all algorithms for bipartite matching.
- Given a bipartite graph $G(V, E)$ and a matching M, the set of vertices with which edges in M are incident is called a cover. For matching M, an alternating path is a path which the edges belong alternatively to M and not to M, and an augmenting path is an alternating path that starts from and ends on unmatched vertices. Matching M is the maximum matching in G, if there is no other matching M' in G such that $|M'|>|M|$.

- Hungarian algorithm
  - [1] Initially matching M is empty;
  - [2] Finding an augmenting path p for M, and $M \leftarrow M \oplus p$;
  - [3] Repeat [2] until there is no any augmenting path in G. Matching M is a maximum matching for G.

- DFS algorithm can be used to find an augmenting path. DFS algorithm takes an unmatched vertex as the starting vertex, and it produces an augmenting path p in which the edges belong alternatively to M and not to M.

- DFS algorithm is as follow.

```
bool dfs(int i){     // Determine whether there is an augmenting path starting from vertex i in X
  for (int j=1; j<=m; j++)
if ((!v[j])&&(a[i][j])){        // Search all unvisited vertices which are adjacent to vertex i
      v[j]=1;                          // visit vertex j
      if (pre[j]==0||dfs(pre[j])){        //If the precursor for j is unmatched or there exists an augmenting path starting from the precursor for j, then edge (i, j) is in matching, and return true
         pre[j]=i;
         return 1;
       }
}
   return 0;                                        //return false
}
```

- If *dfs*(*i*) returns true, then vertex *i* is matched. Obviously, for every vertex *i*, *dfs*(*i*) is called, and a maximum matching in a bipartite graph is gotten. Therefore Hungarian algorithm is as follow.

```
  int ans=0;                         //Initialization
  for (int i=1; i<=n; i++){          //Enumeration
      memset(v, 0, sizeof(v));
      if (dfs(i)) ans++;
  }
```

- Suppose there are e edges in a bipartite graph G, vertices in G are divided into two disjoint sets X and Y such that $|X|=|Y|=n$, and M is a matching in G. The time complexity of finding an augmenting path is O(e). In order to get a maximum matching, at most n augmenting paths are required to calculate. Therefore the time complexity of Hungarian algorithm is O(n* e).

Conference

- Source: Bulgarian Online Contest September 2001
- IDs for Online Judge: Ural 1109