Graph Search Algorithms

Exploration of Graphs (Depth First Search and Breadth First Search)

A basic requirement for most graph algorithms is the systematic exploration of a graph starting at some node *s* (or at some set of nodes *S*). The basic idea is simple.

Suppose we have visited some set *S* of nodes already; initially *S = {s}*. Also some of the edges incident to nodes in *S* have been used. In each step the algorithm selects one of the unused edges incident to a node in *S* and explores it, i.e. the edge is marked used and the other end point of the edge is added to *S*. The algorithm terminates when there are no unused edges incident to nodes in *S* left.

Basic Idea of the Algorithm:

```
S = {s};
mark all edges unused;
while (there are unused edges leaving nodes in S) {
   choose any node v in S and an unused edge {v,w} from E;
   mark the edge {v, w} used;
   add w to S
}
```

It is easy to see that, upon termination of the algorithm, the set *S* contains all nodes *v* such that there is a directed path from the start node *s* to node *v*.

In the above algorithm, each time we pick a node that has at least one unused edge directed out of it. So, it is convenient (more efficient) to maintain another set of nodes say S^0 in addition to the set *S*. S^I is a subset of *S*, consisting of all nodes which still have outgoing unused edges. Using *S* and S^I, our basic exploration algorithm can be reformulated as follows:

Exploring a digraph starting from a node s:

```
Explorefrom(s)
{
  S = {s};
  $S^I$ = {s};
  Mark all edges "unused";
  while ( S’ is not empty)
  {
     choose some node v in $S^I$;
     if ( there is no "unused" edge out of v )
        delete v from S’;
     else
     {
        let (v, w) be the next unused edge out of v;
        if ( w is not in S)
           { add w to S; add w to S’; }
     }
  } // end while
} // end Explorefrom
```

Recall from the discussion about adjacency lists, digraphs can be represented as a sequential data structure, such as a list, array, or vector. Each cell in the list is a linked list.

Suppose that we name our sequential container type *vector* and our linked list type *list*. Suppose further that the digraph is represented using an object, say *myDigraph* of type vector<list<number>> (*vector of list of numbers*). Also, we use a vector, say *vecIter*, of iterators of type vector<list<int>::iterator>. The iterator *vecIter*[*i*] will be used to iterate over the *i*th list corresponding to each node *i*. Initially each iterator v*ecIter*[*i*] point to the first item of the *i*th list. (That is, initially, *VecIter*[*i*] = beginning of *myDigraph*[*i*], for each *i*. In the course of the algorithm, the elements of the *i*th list which are to the left of the iterator are used and the element pointed to by the iterator and all other elements to the right of the iterator are unused. Thus, the line

let (v, w) be the next unused edge out of v;

in the above algorithm is equivalent to reading the element pointed by *VecIter*[*v*] and incrementing *VecIter*[*v*] by one position. Thus, the lines

```
if ( there is no "unused" edge out of v )
   delete v from S’;
else
 {
   let (v, w) be the next unused edge out of v;
```

in the above algorithm can be implemented in more refined pseudocode by something such as

```
if ( VecIter[v] == end of myDigraph[i] )
   delete v from S^I;
else
{
   w = dereference VecIter[v];
   move VecIter[v] forward;
```

On the set *S* the operations Insert and Member are executed. On the set S^0 the operations IsEmpty, Insert, Select Some, and Delete. The SelectedOne are executed.

We can use a boolean sequential container to represent the set *S*. Then, the Insert and Member operations can be executed in constant time. Initialization of set *S* to empty (that is setting all cells of the vector to 0) takes O(n) time.

For the set S^0, we can use either a Stack object or a Queue object. Then all operations on S^0 can be executed in constant time.

Depending on the representation of set S^I, stack or queue, we have two versions of code for Explorefrom. They are known under the names **depth first search** (S^I is a stack) and **breadth first search** (S^I is a queue). In depth first search exploration proceeds from the most recent node visited which still has some unused edges. In breadth first search exploration proceeds from the first node visited which still has unused edges.

There are many applications for the depth first search and breadth first search in graph algorithms.

A Closer Look at Depth-First-Search

Input:
Adjacency list of a graph $G = (V,E)$ and a start node $s \in V$.
Output:
For each node, the depth-first number (dfn) of the node.
For each edge, whether the edge is a *tree* edge or a back edge.

```
Dfs(s) {
    (1)     Stack S';
            S'.Push(s);
            Vector dfn;
            For each v in V, set dfn[v] = 0;
            Let i = 0, v = s;
            Mark all edges "unused";

    (2)     i = i+1;
            dfn[v] = i;
            // We will add a line here in the next version

    (3)     if (v has no unused edges), go to (5);

    (4)     choose the next unused edge e = {v, u};
            Mark e "used";
            if (dfn[u] != 0) { // edge e is a back edge
            // We will add a line here in the next version
                go to (3);
            }
            else { // edge e is a tree edge
                S'.Push(u);
                v = u;
                go to (2);
            }

    (5)     if (v == s) {
                // We will add a if statement here in the next version.

                halt;
            }

    (6)     S'.Pop();
            Let p = S'.Top();
            // We will add a if statement here in the next version

            v = p;
            go to (3);

} // end Dfs
```

Recall our earlier algorithm **ExploreFrom(s)** and the set S'. If the set S' is implemented using stack, then the algorithm ExploreFrom(s) is usually referred to as the Depth-First-Search (DFS).

The algorithm **Dfs(s)** is essentially the same as **ExploreFrom(s)** discussed before, except that we number the vertices from 1 to n, the order in which they are discovered. We call this number the depth-first number (dfn). This numbering will be useful in applying the algorithm for more advanced tasks.

A graph is *connected* when there is a path between every pair of vertices. In a *connected graph*, there are no unreachable vertices. A graph that is not connected is *disconnected*.

Suppose that we have applied the Dfs(s) to a finite and connected $G = (V,E)$. Let the set of edges E^0 consisting of all the edges $\{v,u\}$ through which new nodes have been discovered. That is, when we are at node v, we discovered node u such that $dfs[u] == 0$. Also direct each such edge from v to u.

Then it can be shown that, the digraph, say $T = (V,E^0)$, is a directed tree with root s. Moreover, if we ignore the edge directions of T, then T is a spanning tree of G.

We call all the edges of $T = (V,E^0)$, *tree edges* and all the other edges *back edges*. The justification for this name is in the following observation. If an edge $\{x,y\}$ is not a part of $T = (V,E^0)$ then x is either an ancestor or a descendant of y in the directed tree T. Thus, all the non-tree edges connect a node back to one of its ancestors.

As an example, consider the graph of Figure 1.

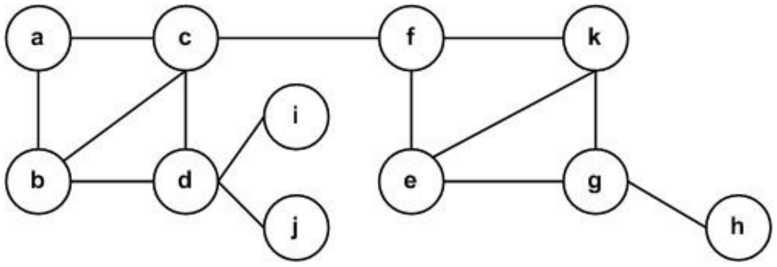

Figure 1: A connected undirected graph

Assume we start the DFS with node *c* (*s* = *c*) and discover *a,b,d,i,j,f,e,g,h,k* in this order. The resulting depth-first numbers, tree edges and back edges are shown in Figure 2, where the tree edges are shown by solid lines and are directed from low to high, and the back edges are shown by dashed lines and are directed from high to low, the direction of the edge indicates the direction in which each edge has been traversed.

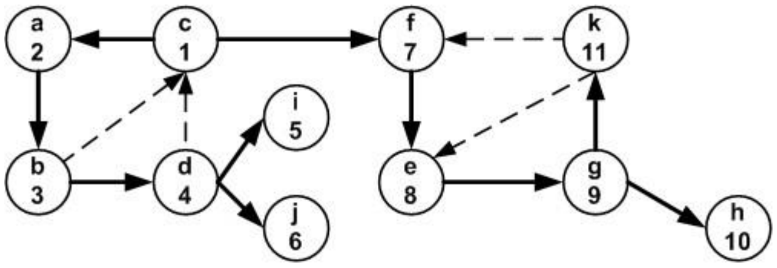

Figure 2: A connected undirected graph

Articulation Points and Biconnectivity:

In a connected graph *G* = (*V,E*), a node $u \in V$, is called an *articulation point* (also known as a *cutpoint*) if the deletion of *u* from *G* yields a graph that is not connected. The nodes *c,d,f,g* in the graph of Figure 1 are the articulation points of *G*.

A connected graph having no articulation points is called a *biconnected* graph. A *biconnected component* is a maximal biconnected subgraph. The graph of Figure 1 has 6 biconnected components as shown in Figure 3.

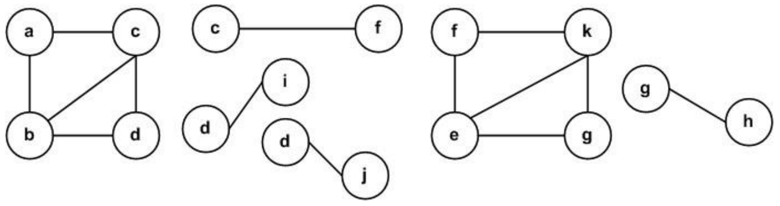

Figure 3: A connected undirected graph

Depth first search provides a linear time algorithm to find all cutpoints in a connected graph. We perform depth first search starting at some node *s*. For each node *v*, we assign the depth first number, *dfn*(*v*), the order in which we discovered the node. Also, for each node *v* in the *dfs tree*, we compute another number called the *lowpoint number* of *v*, denoted by *L*(*v*). The *L*(*v*) for a node *v* is the smallest *dfn*(*u*) of a node *u* that is reachable from *v* using a directed path of zero or more tree edges followed by at most one back edge.

Indeed, a node $v \neq s$ (that is, $dfn(v) \neq 1$) is an articulation point *iff* there is a tree edge directed from *v* to *u* such that $L(u) \geq dfn(v)$. Moreover, the starting node *s* is an articulation point if there are at least two tree edges directed out of it. These two claims can be easily justified.

It is not necessary for us to first complete the DFS and then compute the lowpoint numbers. As we do the DFS, for each node *v*, its *L*(*v*) can be computed by the time we backtrack from *v*. If *v* is a leaf of the DFS tree then *L*(*v*) is the least number in the following set:

{*dfn*(*u*) *such that u = v or there is a back edge from v to u*}.

Initially we can assign $L(v) = dfn(v)$ when we first discover node *v*. Later on, as each back edge from *v* to node *u* is explored, we can update

$L(v) = Min\{L(v), dfn(u)\}$.

Clearly, by the time we backtrack from *v*, all the back edges have been explored, and *L*(*v*) will have the right value.

If *v* is not a leaf of the DFS tree, then *L*(*v*) is the least number in the following set:

{*dfn*(*u*) *such that u = v or there is a back edge from v to u*} ∪
{*L*(*u*) *such that there is a tree edge from v to u*}.

When we backtrack from v, we have backtracked from all its children already, and therefore know their lowpoint. Thus, all we need to do is that when we backtrack from u to v, we update $L(v) = Min\{L(v),L(u)\}$. Now we are ready to present the algorithm for finding the articulation points of a connected graph.

Finding Cutpoints

Input: Adjacency list of a graph $G = (V,E)$ and a start node $s \in V$.
Output: Cutpoints, if any, of G.

```
Dfs_CutPoints(s) {
    (1) Stack S';
        S'.Push(s);
        Vector dfn, low;
        For each v in V, set dfn[v] = 0;
        Let i = 0, v = s;
        Mark all edges "unused";
    (2) i = i+1; dfn[v] = i; low[v] = i;
    (3) if (v has no unused edges), go to (5);
    (4) choose the next unused edge e = {v, u};
        mark e "used";
        if (dfn[u] != 0) { // edge e is a back edge
            low[v] = min(low[v],dfn[u]);
            go to (3); }
        else { // edge e is a tree edge
            S'.Push(u);
            v = u;
            go to (2); }
    (5) if (v == s) {
            if ( s has at least two tree edges directed out of it) {
                Print "the start node s is a cutpoint"; }
            halt;
        }
    (6) S'.Pop();
        let p = S'.Top();
        if (p != s) {
            low[p] = min(low[p], low[v]);
            if (low[v] >= dfn[p]) {
                Print "node p is a cutpoint"; }
        }
        v = p;
        go to (3);
} // end Dfs_CutPoints
```