This note reviews some properties of depth first search (DFS) and describes an efficient algorithm for computing the cut vertices in a graph. We restrict our attention to undirected graphs throughout.

1 Depth first search in undirected graphs

The main idea underlying a DFS traversal is very simple:

i) explore edges that lead to unvisited vertices as soon as we stumble upon them;

ii) if we happen to reach a "dead end", we backtrack.

Below is the pseudo-code for the DFS routines.

Algorithm 1 DFS(G)

```
1. for each vertex u in G do
2.     mark u as unvisited
3.     parent[u] = None
4. time = 0
5. for each vertex u in G do
6.     if haven't visited u yet then
7.         DFS_VISIT(u)
```

Algorithm 2 DFS_VISIT(u)

```
1. time = time + 1
2. d[u] = time
3. mark u as visited
4. for v ∈ N(u) do
5.     if haven't visited v yet then
6.         parent[v] = u
7.         DFS_VISIT(v)
8. time = time + 1
9. f[u] = time
```

DFS traverses the whole input graph, even the graph is not connected. In the process, it generates a DFS forest (defined by the *parent* array) and a *discovery* and *finishing* time for each vertex in the graph. The DFS forest is made up of the following edges

$$\{(u, parent[u]) \mid u \in V \text{ and } parent[u] \neq \text{ None}\} .$$

We will follow the convention that trees in the DFS forest are rooted at the first node discovered in that tree; that is, every time the main DFS routine finds an undiscovered vertex u, a new tree rooted at u is added to the DFS forest T. The forest and the discovery and finishing times have many interesting properties that can be exploited to design algorithms for graph problems.

1.1 Some properties of DFS

Lemma 1 (nesting property)**.** *Let u be a vertex in the graph, and $v_1, \ldots, v_k$ the children of u in the DFS forest. The intervals $[d[v_i], f[v_i]]$ are pair-wise disjoint and span $(d[u], f[v])$.*

Proof. It should be clear from the way the discovery and finishing times are computed: The time is increase just before it is assigned as a discovery time at the beginning of a DFS_VISIT call and just before it is assigned as a finishing time at the end of a DFS_VISIT call. □

Let u and v be two vertices. From the previous lemma, we get that if v is descendant of u then $d[u] < d[v] < f[v] < f[u]$; this follows by applying Lemma 1 repeatedly along the unique u-v path in T. Also, if v is not a descendant of u nor u a descendant of v, the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ must be disjoint.

Lemma 2 (unvisited path property)**.** *Suppose that right before making the call* DFS_VISIT(u) *there was a path $u \rightsquigarrow v$ consisting solely of undiscovered vertices. Then v will become a descendant of u in the DFS forest.*

Proof. Suppose that was not the case. Then let x be the vertex in the path that is closest to u and is not a descendant of u (we know that there is at least one such vertex (v) so x is well defined) and let y be vertex coming before x in the path (notice that y could be u itself and that y is descendant of u). When we executed DFS_VISIT(y), x was undiscovered and we have the edge (y, x) in the graph. Hence x must be a descendant of y, but y itself is a descendant of u, thus contradicting our assumption. □

It follows that if the graph is connected, the DFS forest is made up of a single tree. The following lemma is an easy consequence of the previous lemma.

Lemma 3 (tree and back edges)**.** *Suppose we run DFS on an undirected graph (V, E). Let (u, v) be some edge in E, then u and v must belong to the same tree in the DFS forest. Furthermore, either u is an ancestor of v, or v is an ancestor of u.*

1.2 Finding cut vertices

Let $G = (V, E)$ be a connected undirected graph. We say a vertex $u \in V$ is a cut vertex if its removal from G disconnects the graph. In this section we study the problem of identifying the set of cut vertices in a given connected graph.

The trivial algorithm iterates over the vertices. In each iteration, we test whether the vertex in question is cut by removing the vertex from the graph and testing the connectivity of the resulting graph (we can use DFS or BFS for that). If we assume adjacency list representation, this algorithm runs in $O(nm)$ time[1]. However, this is not the most efficient algorithm for this problem.

Suppose we run DFS on G and obtain a DFS tree T, and discovery and finishing times for each vertex. This provides some additional information about the graph that we can use to quickly identify cut vertices. The next lemmas assume the graph G is connected.

Lemma 4. *Let r be the root of the DFS tree T; then r is a cut vertex if and only if r has two or more children.*

Proof. If r has a single child, then $T - r$ is still a connected graph. On the other hand, if r has two or more children each children subtree becomes a connected component because the only non-tree edges we are allowed to have are back edges. □

[1]DFS takes $O(n + m)$ time, but because we assume our input graph is connected, the number of edges (m) is at least the number of vertices (n) minus one, so $O(m + n)$ becomes $O(m)$

Lemma 5. *Let u be a leaf in the DFS tree T; then u is no a cut vertex.*

Proof. Since u is a leaf, $T - u$ is still connected. □

To test whether an internal node of T is cut, we need to define one new value:

$$up[u] = \min_{v:(u,v)\in E} d[v].$$

It follows from Lemmas 3 and 1 that the neighbor of u attaining the minimum in the above definition is the neighbor closest to the root. The name $up[]$ is supposed to be a reminder of this. Let us define one last value

$$\textit{down-\&-up}[u] = \min_{w\in T_u} up[w],$$

where T_u is the subtree rooted at u in T.

Lemma 6. *Let u be an internal node of T; then u is cut if and only if u has a child v in T such that*

$$\textit{down-\&-up}[v] = d[u].$$

Proof. Let v be a child of u. After removing u the only way T_v can remain connected to the rest of the graph is through a back edge (x, y) where $x \in T_v$ and y is a strict ancestor of u, but then we would have $\textit{down-\&-up}[v] \le up[x] \le d[y] < d[u]$.

On the other hand $\textit{down-\&-up}[v] \le up[v] \le d[u]$, where the last inequality follows from the hace that v is connected to u.

□

Lemmas 4, 5, and 6 give an alternative characterization of cut nodes based on the DFS routine, which forms the basis of a more efficient algorithm.

Algorithm 3 FIND-CUT-VERTICES(G)

```
1. run DFS on G to compute T and d[u] for each u in V
2. answer = ∅
3. if root r of T has two or more children then
4.    add r to answer
5. for v internal node of T do
6.    for each child v of u in T do
7.       if down-&-up[v] = d[u] then
8.          add u to answer
9. return answer
```

The correctness of the algorithm rests on the alternative characterization of cut nodes. The running time is dominated by the last `for` loop. For each internal node u, we need to scan every descendant of u, which takes $O(n)$ time. Thus, the running time becomes $O(n^2)$. This is a big improvement from the trivial $O(nm)$ time algorithm, but the algorithm can be further improved. Using some additional ideas, it is possible to get a linear running time of $O(m)$.