A Review of Depth First Search

You are assumed to be familiar with graph traversal algorithms. In particular, you should know the definition of graph, directed graph,path, directed path, cycle, directed cycle, and be familiar with the adjacency list data structure. This data structure consists of an arrayADJ indexed by the vertices of the graph. The element $ADJ[v]$ is a pointer to the first element of a linked list which contains all the arcs which leave v (for a directed graph this is the arcs whose tail is at v). We say that a node u is *reachable* from a node v in a (directed) graph G if there is a (directed) path from v to u in the graph.

The depth first search algorithm(DFS) is a straightforward recursive algorithm. Pseudcode for this algorithm is given on page 604 of CLRS[1]. DFS takes as input a linked list $G.V$ of the vertices of a (directed) graph G in some order and an adjacency list ADJ for G.

DFS starts the search of the (directed) graph at the first node on the list-the root. It searches as much of the (directed) graph as it can via the first neigbour of this node on the adjacency list before moving on to the second neighbour on the adjacency list. Depth first search discovers which nodes are reachable from the root and builds a (directed) tree containing a path from the root to each such node. This tree is given by a predecessor function which we store in an array Π indexed by the vertices[2], so that for each non-root node of this tree $\Pi[v]v$ is the edge of the tree into v.

If not every node of the graph is reachable from the root, we traverse $G.V$ until we find a node which is not reachable from the root and build a second DFS tree from this new root. This tree contains every node reachable from the second root which is not reachable from the first root. More generally having built i trees, we continue our traversal of $G.V$ to find a node not in any of them, and then build a tree with this node as a root containing all the nodes reachable from this root which were not reachable from any of the earlier roots, and recording its edges in Π.

The depth first search algorithm determines and returns two time stamps for each vertex v in the tree that it builds. The discovery time of v, stored in an array d indexed by V is the first time that the search visited v. The finishing time for v, stored in an array f indexed by V, is the time at which the search algorithm has returned to v after searching from all the vertices

[1]the third edition of *Introduction to Algorithms* by Cormen Lieserson, Rivest, and Stein which can be accessed on skillbooks

[2]CLRS uses $u.\Pi$ for our $\Pi[v]$,u.f for our $f[u]$, and $u.d$ for $d[u]$

on its adjacency list.

We will also make DFS output a linked list, $DECFINISHORDER$ of the vertices of G in decreasing order of finishing time. To do so we modify the CLSR pseudocode for DFS by adding a line before Line 1 which sets $DECFINISHORDER = Nil$ and adding a line after line 10 which appends an element containing v to the front of $DECFINISHORDER$.

Now, we finish with every node in a tree before moving onto the next tree, and we finish the recursive calll handling a node of a tree before finishing the earlier calls for its ancestors. Thus, the time stamps have the following property(see Theorem 22.7 in CLRS):

(P1) For any two distinct vertices u and v exactly one of the following holds:
(i) $d[u] < f[u] < d[v] < d[f]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other in a depth first search tree,
(ii) $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u in a depth first search tree,or
(iii) $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v in a depth first search tree,

They also have the following property (see Theorem 22.9 in CLRS; for CLRS a vertex is white if it has not yet been discovered):

(P2) If at time $d[u]$ there is a (directed) path P from u to v in the (directed) graph G none of whose vertices are discovered before u then v is a descendant of u in the depth first search tree.

The edges of G are divided into *back* edges which go from a vertex to an ancestor in the tree, *forward* edges which go from a vertex to a descendant[3] and *cross* edges which go between two vertices neither of which is a descendant of the other.

(P2) applied to the one edge path uv implies that there are no cross-edges in an undirected graph and in a directed graph for any cross edge uv we must have $v.d < u.d$. By property (P1) we actually have $v.d < v.f < u.d < u.f$ for any such cross edge. Combining this fact with (P1), we see that the following property holds:

(P3) if uv is a forward edge or cross edge in a directed graph then $f[v] < f[u]$, I.e. u comes before v in the order given by $DECFINISHORDER$. On the other hand if uv is a back edge then $f[v] > f[u]$ and v comes before u in this

[3]these include the tree edges which are in the trees the algorithm constructs

order.

Finding A Topological Sort or Determining G has a cycle

A directed graph is *acyclic* or a DAG (directed acyclic graph) if it contains no directed cycle. A *topological sort* of a graph is an ordering of its vertices such that if uv is an edge then u appears before v in the ordering.

Clearly if a directed graph has a topological sort it can have no cycle (as if u is the last vertex of a cycle in the ordering then there is an edge uv of the cycle which contradicts the definition of a topological sort). In fact, using depth first search we can see that a directed graph is a DAG precisely if it has a topological sort.

To see this, we consider running DFS on a directed graph. If we obtain a back edge uv then the graph is not acyclic because the back edge together with the path of the tree from v to u is a cycle. If the graph contains no back edge then (P3) shows that the order given by $DECFINISHORDER$ is a topological sort.

For a more long-winded discussion and an example see Section 22.4 of CLRS. For a visualization package which allows you to see a variety of examples of the sort, see:

https://www.cs.usfca.edu/ galles/visualization/DFS.html

Partioning $V(G)$ Into Strong Components and Finding A Good Order

As discussed in the last class, our alogirthm for solving 2-SAT uses as a subroutine the linear time algorithm FINDING A GOOD ORDER which finds a good ordering of a directed graph[4]. If G is a DAG then every strong component has size 1 and an order is good precisely if it is a topological sort. Thus, in the special case of DAGs, as we just saw we can obtain a good order by running DFS on G using any input ordered list of $V(G)$.

[4]We recall that an ordering of $V(G)$ is good if for some indexing of the strong components of G as $Z_1, ..., Z_k$ so that if $i < j$ then there is no arc from a vertex of Z_j to a vertex of Z_i, for every i and j all the vertices of Z_i appear before all the vertices of Z_j in the order

In the general case, we will again use a DFS. We will see that if we apply DFS to an appropriate chosen input ordered list of the vertices of G then the output order $DECFINISHORDER$ is good.

It turns out that in order to find such an ordered list to be input, we again use DFS. But we apply it to the graph G^T obtained by reversing every arc of G, I.e. $E(G^T) = \{uv | vu \in E(G)\}$.

So FINDING A GOOD ORDER has two phases. In the first phase we apply DFS to G^T to some(any) input ordered list of $V(G)$. In the second phase we apply DFS on G where the input ordered list of $V(G)$ is the ordered list $DECFINISHORDER$ returned in the first phase. Our output is the ordered list $DECFINISHORDER$ obtained in the second phase.

As we shall see the vertex sets of the DFS trees obtained in the second phase are exactly the strong components, so we could also use this algorithm to determine the strong components of a directed graph G, as discussed in Chapter 22 of CLSR.

Our first step in explaining why FINDING A GOOD ORDER works is to show that every directed graph has a good ordering. To do so, we consider the component graph of a directed graph G, defined on page 617 of CLRS, which has a vertex for every strong component of G. There is an edge from the vertex corresponding to a strong component C to the vertex corresponding to a different strong component C' precisely if there is an edge xy of G with $x \in C$ and $y \in C'$. If there is such an edge then there is a path from every vertex of C to every vertex in C' with all of its edges in $C \cup C'$. More generally, we can prove by induction that the following holds:

(P4) if there is a path P from the vertex corresponding to C to the vertex corresponding to C' in the component graph then there is a path from every vertex of C to every vertex of C' in G which lies in the union of the strong components corresponding to the vertices on P.

Hence if there were a directed cycle of the component graph, then for any two vertices u and v in the union of the strong components on the cycle there would be a path from u to v. This contradicts the definition of strong component. Thus, no such cycle exists and the component graph is a DAG.; So, by the results of the last section, there is a topological sort of the component graph. To obtain a good order for G, we can simply replace the vertex corresponding to a strong component in the topological sort of the component graph by the vertices of the strong component in any order. This proves that a good order exists for any directed graph.

To understand why our two phase algorithm works as claimed, we need to consider how DFS treats the strong components of a graph. Since for every two vertices u and v of a strong component C there is a path from u to v through C, (P2) tells us that every vertex of C is a descendant of the first vertex v_C of C discovered So all of C lies in the same tree as v_C, and every strong component lies in one tree.

Now, consider any two distinct strong components C and C' such that there is a path P of the component graph from the vertex corresponding to C to the vertex corresponding to C'. By (P4) this implies that there is a path Q from some vertex c of C to a vertex c' of C'. We consider the first vertex v, which is either on Q or in C', to be discovered. By (P2) every vertex of C' is a descendant of v and hence finishes before v.

If v is in C this means there is a vertex of C finishing after all the vertices of C'. Otherwise since v is not in the strong component C and Q contains a path from c to v there is no path from v to c. So, c is not a descendant of v. Since v was discovered before c, (P1) implies that c finishes after v and hence after all its descendants. So, we have:

(P5) If there is a path of the component graph of G from the vertex corresponding to C to the vertex corresponding to C', then for any DFS on G, there is a vertex of C which finishes after all the vertices of C'.

We want to apply this to G^T. Since its edges are the reverse of those in G, there is a path from y to z in G^T precisely if there is a path from z to y in G. Thus, the strong components of these two graphs are the same and there is a path in the component graph of G^T from the vertex corresponding to C to the vertex corresponding to C' precisely if there is a path of the component graph for G from the vertex corresponding to C' to that corresponding to C. So (P5) yields

(P6) If there is a path of the component graph of G from the vertex corresponding to C to the vertex corresponding to C', then for any DFS on G^T, there is a vertex of C' which finishes after all the vertices of C.

We now consider the second phase of our two phase algorithm. Let v be the root of one of the trees T it constructs. We know T contains all of the vertices of the strong component C containing v. Suppose, for a contradiction, T contains a vertex from some other strong component. Then, there is a path P of T from v to such a vertex and by taking a shortest such path, we can see that all the vertices but its last are in C and its last edge

goes from a vertex of C to a vertex in some other strong component C'.

But, by (P6) there is a vertex of C' which finishes after v in the first phase DFS. Hence it is earlier on the input ordered list of $V(G)$ used in the second phase. So, we considered it as a possible root before we considered v. When doing so we used it as a root unless it had already been added to a tree. In either event we see it is in an earlier tree, which is a contradiction. So, T contains exactly the vertices in the strong component containing v.

Since this holds for every tree, we see that our DFS output does partition the vertex set into its strong components. Furthermore, since the only edges between trees are cross edges, and for every cross-edge uv, we have $f[v] < f[u]$ we see that the order returned by the second phase DFS is a good order.