CS 6901 (Applied Algorithms) – Lecture 4

Antonina Kolokolova

September 22, 2016

1 DFS

A different traversal algorithm DFS (depth first search) explores the recursively. Here, we again use the colour convention of white/gray/black, and also, following CLRS book, keep track of the time a vertex was first discovered ($u.d$) and a time a vertex was coloured black (finishing time, $u.f$). Assume that time is a global variable. Here, we will also use a wrapper to explore all vertices.

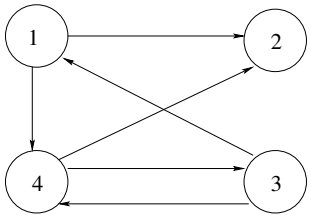
DFS-WRAPPER(G)

```
1  colour all vertices white and put them into a list
2  time ← 0
3  while there is a white vertex u
       do
4          DFS(u)
```

The following DFS algorithm explores the graph starting from a given vertex.

DFS(s)

```
1  time ← time + 1
2  u.d ← time; u.colour ← gray
3  for each v such that (u, v) ∈ E
       do
4          if v is white, DFS(v)
5  u.colour ← black
6  time ← time + 1
7  u.f ← time
```



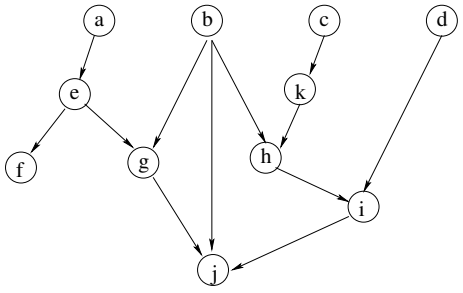
In this example graph, if we start traversing from vertex 1, one of the possible executions of DFS proceeds like this. Set the discovery time for vertex 1 (let us call it v_1 here to avoid confusion) to "1." Then, for example, go to vertex v_4, and set its discovery time $v_4.d = 2$. From there, say we explore v_2 next, with discovery time $v_2.d = 3$. As there is nowhere to go from v_2, we are finished with this vertex; set $v_2.f = 4$ and backtrack to v_4. From there, go to v_3, resulting in $v_3.d = 5$ and $v_3.f = 6$. Back to v_4, which is now done so it gets $v_4.f = 7$. Finally, set $v_1.f = 8$. Sometimes we will use the notation "2/7" to mean the discovery time is 2 and the finish time is 7 for a specific vertex (especially on the pictures). The edges of the DFS tree for this run of DFS are $(1, 4), (4, 2), (4, 3)$.

Note one property of these discovery and finish times of vertices: if a vertex v is a descendant of a vertex u (that is, v is in the tree underneath the u), then $u.d < v.d < v.f < u.f$. This property is quite useful for proving correctness of algorithms based on DFS.

Next, we will see two applications of the Depth First Search (DFS) algorithms: computing a topological order of vertices of a directed acyclic graph (DAG) and finding strongly connected components in a graph. These applications are examples of how a basic algorithm, DFS, can be modified and, in case of strongly connected components, used multiple times to do a more complex tasks with graphs. I will follow CLRS book for this material.

1.1 DAGs and topological sort

Directed Acyclic Graphs, commonly referred to as "DAGs", are a very useful type of graph for modelling a variety of problems where there are dependencies between objects or tasks. Here, the edges indicate the direction of a dependency: for example, if the graph represents a diagram of instructions of assembling furniture, an edge from task A to task B might indicate that task A (say, attaching a table leg) has to be completed before task B (say, securing the table leg with screws). One important advantage of DAGs is that some problems that are NP-hard on general graphs have polynomial-time algorithms on DAGs, for example computing the longest path in the graph. Another example of a DAG is a Boolean circuit: there, the direction of the edges is from the inputs towards the output(s).



In a DAG, vertices with no incoming edges are called *sources*, and vertices with no outgoing edges are *sinks*. Every DAG must have at least one source and at least one sink (which could be the same isolated node).
Vertices in a DAG can be listed in the *topological ordering*, that is, in an ordering $L : V \to \{1, \ldots, n\}$ such that whenever $(u, v) \in E$, $L(u) < L(v)$. There can be more than one topological ordering of a given DAG. But if there is a path in the DAG from u to v, then in any topological ordering u will come before v in the order.

In the example DAG G above, there are four sources a, b, c, d and two sinks f and j. One possible topological ordering of this graph is $a, b, c, d, e, k, f, g, h, i, j$, another $d, a, e, f, b, g, c, k, h, i, j$. As you see, even though the list has to start with a source and end with a sink, they do not have to all be in front (respectively, in the back) of the list, provided the property of topological ordering is satisfied.

It is possible to show that a graph is a DAG if and only if such a topological ordering exists in a graph. Intuitively, consider a cycle in a graph: no matter how you number vertices in this cycle, there will be an

edge from a larger to a smaller vertex (otherwise, the smallest vertex will have no incoming cycle edges and largest no outgoing cycle edges, but then it is not a cycle). For the other direction, suppose there is a topological ordering in a graph. Then it must be acyclic because from any vertex, vertices that came before it are unreachable.

In order to compute a topological ordering of a graph, we will use a topological sorting algorithm based on DFS.

TopoSort(G)
1 Run DFS on G
2 as each vertex v finishes, insert v to the front of a linked list L
3 **return** L

Example 1 *Suppose we run DFS on the graph above. As the initial order of vertices can be arbitrary, let's say that we start with vertex k. The following will be the starting and finishing times in the resulting DFS tree: $k : (1, 8)$, $h : (2, 7)$, $i : (3, 6)$, $j : (4, 5)$. At this point, our list will contain (k, h, i, j), where j was inserted first. Then let us start from b: we will have $b : (9, 12)$ and $g : (10, 11)$ in the next DFS tree; the list now becomes (b, g, k, h, i, j). Continuing in this fashion, and picking a next, then c, then d, we end up with the topological ordering $(d, c, a, e, f, b, g, k, h, i, j)$. That is, $L(d) = 1$, $L(c) = 2$ and so on.*

The running time of this algorithm will be comparable with the running time of DFS, as inserting an element to the front of the list takes constant time. Thus, the whole algorithm runs in time $O(n + m)$.

Now, to convince ourselves that the algorithm indeed does a correct topological sorting of the vertices of its input graph G, we need to show that for any vertices u, v such that (u, v) is an edge, $L(u) < L(v)$, where now $L(u)$ and $L(v)$ we interpret as positions in the list produced by our algorithm. We can show it in two steps.

First, notice that in a DAG, DFS never sees an edge ending in a gray vertex (that is, a vertex for which we have the discovery time, but not yet finishing time). A vertex would be gray if it is on a path from the root of the DFS tree being constructed to the current vertex. But then this part of the path together with the edge to a gray vertex completes a cycle, and we assumed that our graph is a DAG. (Additionally, we can show that if a graph is not a DAG, then DFS will see such an edge).

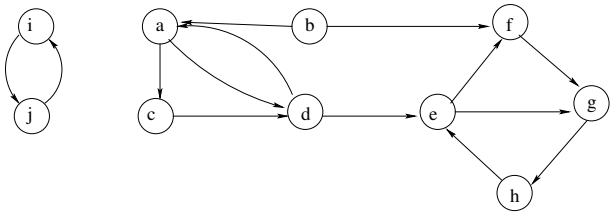
Thus, every edge (u, v) encountered in the run of the DFS on a DAG is either to a white vertex v (v has no discovery time yet), or to a black vertex (v has finishing time). In case when v is white, we know from last time that u is an ancestor of v, so $u.d < v.d < v.f < u.f$. Thus, $v.f < u.f$ and so v will be after u on L. When v is black, then $v.f < u.f$, as $v.f$ is finished while we are still exploring subtree under u. In this case, u also precedes v in L, completing the proof.

1.2 Strongly Connected Components

Another algorithm that uses DFS even more creatively is that for computing strongly connected components in a directed graph (we are not considering just DAGs anymore). A connected component in an undirected graph is any group of vertices where it is possible to get from one vertex in the component to any other; either DFS or BFS will produce connected components of an undirected graph as separate trees, and for each connected component, the corresponding tree will contain all vertices. For a directed graph, the definition is a bit more complicated. A group of vertices in a graph forms a *weakly connected* component if its underlying undirected graph (that is, ignoring the directions of edges) is connected (and maximal). A group of vertices

forms a *strongly connected* component if within this group every vertex is reachable from any other vertex in the component; again, we want every vertex with this property to be part of the component.

Example 2 *This graph has two weakly connected components ($\{i, j\}$ and the rest), but four strongly connected components ($\{i, j\}, \{a, c, d\}, \{b\}, \{e, f, g, h\}$).*



The following algorithm computes strongly connected components of a graph G.

SCC(G)

1 Run DFS on G
2 Compute G^r ▷ where G^r is G with all edge directions reversed
3 Run DFS on G^r, considering vertices in the order of decreasing finishing time computed by DFS(G)
4 **return** DFS forest from the second run; each tree is a separate strongly connected component.

Example 3 *Consider running SCC algorithm on the graph from the previous example. Suppose we obtained the following start/finish times: $a : (3, 4)$, $b : (19, 20)$, $c : (1, 14)$, $d : (2, 13)$, $e : (5, 12)$, $f : (6, 11)$, $g : (7, 10)$, $h : (8, 9)$, $i : (15, 18)$, $j : (16, 17)$ from the first DFS run. Now, running DFS on the reverse graph G^r, we first start with b. As all edges of b are outgoing in G, all edges are incoming in G^r, and so there is nothing else in its connected component. Then run DFS starting with i, obtaining component $\{i, j\}$. Then proceed to c; here, a, b and d are accessible, since edge (e, d) is incoming to this component. As b has been processed already, it does not become part of this tree, leaving only $\{c, a, d\}$ as elements of this component. The final tree is rooted at e, and consists of $\{e, h, g, f\}$.*

First, let us look at the running time of this algorithm. We know that $DFS \in O(n+m)$, and G^r has the same number of vertices and edges as G, so two subsequent runs of DFS will still take time $O(n + m)$ (remember that for a constant number of blocks executed one after another we take the maximum, or, alternatively, that doubling the running time does not change the corresponding O-class). Now, to compute G^r we need to go through G and create a new adjacency list for each vertex, now containing what was formerly incoming edges to this vertex. But this can be done in time $O(n + m)$ as well, by going through all lists of G in order and inserting the corresponding entries into the lists of G^r. Therefore, the total time is $O(n + m)$.

The key fact we need to for the correctness proof is that if u and v are in different connected components C and D, and there is an edge (u, v), then there will be a vertex $w \in C$ with a large finishing time than any vertex $v' \in D$, where the finishing time is with respect to running DFS on G. To see this, take a vertex in w with the largest finishing time of all vertices in C. Then, all vertices in C are in a DFS tree starting from w. Since there is an edge (u, v), DFS will try to follow this edge.

Now, if v is white at that time, then DFS algorithm will explore all of D before backtracking (as D is strongly connected, and there is no path in G from D to C). Then all vertices in D will be descendants of w, so for any $v' \in D$, $w.d < v'.d < v'.f < w.f$. Note that v cannot be gray at that time, as it would imply that it is already on a path from w to v, but such a path would have to leave C, go to D, and come back to C, contradicting the fact that C and D are separate strongly connected components. Thus, if v is not white it would have to be black; in that case, its finishing time for every v' in D was already computed (as they are parts of the same DFS tree), whereas $w.f$ is not assigned yet, so $w.f > v'.f$.

Finally, we need to show that $SCC(G)$ correctly computes strongly connected components. That is, all vertices in the same connected component become parts of the same $DFS(G^r)$ tree (where vertices are

considered in $DFS(G^r)$ in order of decreasing finished time from $DFS(G)$), and any two vertices from different components will be in different $DFS(G^r)$ trees. To see the first fact, note that there is a path from any vertex to any other vertex within a connected component, so once DFS picks a vertex in a component it explores all of it before stopping. This was true even for a single run of $DFS(G)$, so a more interesting part is to show that in the trees of $DFS(G^r)$ each component forms a separate tree.

For the intuition, imagine running DFS on a DAG considering vertices in reverse topological order. The last vertex in this order is a sink: so DFS does not have anywhere to go, and this vertex forms a separate tree. The second to last vertex is either a sink itself (so the same reasoning applies), or has an edge to the last vertex, but since that one is already marked black, the second to last vertex becomes its own tree as well. Now, it can be shown to be true for all vertices of the DAG by induction. Assume that the last i vertices in the topological order (that is, the first i vertices DFS visits) each form a separate tree. Now consider vertex $i+1$ from the end. all of its outgoing edges go to vertices in the tree already processed, so all of them go to black vertices. Therefore, there is nothing to explore and the vertex $i+1$ forms the next DFS tree.

Now, all that is left is to generalize the argument before slightly, to talk about strongly connected components rather than vertices in a DAG. Let us argue by induction again. The vertex with the largest finishing time is in a component C_1 with no edges to other components in G^r, by the fact we discussed at the start of the proof. Thus, the first tree will contain all vertices in that strongly connected component, but no other; and all its vertices will be marked black.

Now, suppose the first i trees outputted by the algorithm form separate strongly connected components $C_1 \dots C_i$. Consider the time when $DFS(G^r)$ starts exploring component C_{i+1}. At that time, all edges from vertices $v \in C_{i+1}$ in G^r are going to vertices $u \in C_1, \dots C_i$, which are already marked black (which follows from the same key fact). Therefore, only vertices in C_{i+1} will become part of the $i+1$st $DFS(G^r)$ tree, completing the proof.