CS 6901 (Applied Algorithms) – Lecture 5

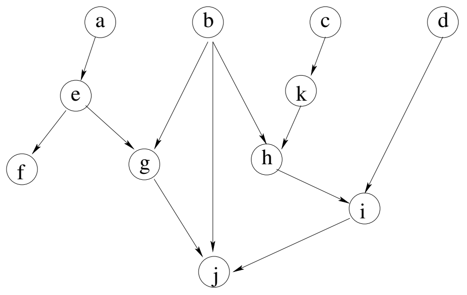
Antonina Kolokolova

September 23, 2014

In this lecture, 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.

0.1 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 $L: V \to \{1, \dots, n\}$ 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 the smaller to a large 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.

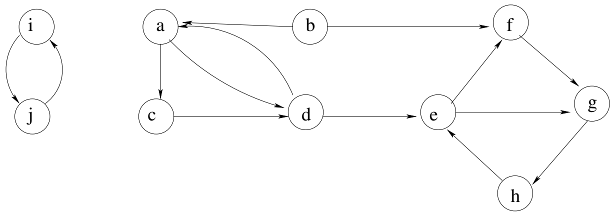
First, notice that in a DAG, DFS never sees an edge ending in a gray vertex (that is, a vertex for which we have the discovery time, but not yet finishing time). A vertex would be gray if it is on a path from the root of the DFS tree being constructed to the current vertex. But then this part of the 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.

0.2 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. We will postpone the running time analysis and correctness proof until next class.

```
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 $\{e, h, g, f\}$.*

For the intuition why this algorithm works, consider a graph where each component is represented by a vertex, its SCC graph. This graph is a DAG, and DFS(G^r) constructs trees in reverse topological order with respect to this DAG.