COS 226 **Precept 6 Notes** November 1, 2021
Fall 2021
Tarjan

In precept we went over the (i) definition and properties of topological orders, and (ii) definitions of various kinds of graph search.

A *topological order* is a total order (we can think of it as a numbering of the vertices from 1 to *n*, or from 0 to *n* – 1 if you prefer) of the vertices of a digraph (directed graph) such that every arc leads from a lower to a higher verex: if *vw* is an arc, $v < w$. The fundamental theorem of topological orders is that a digraph has at least one topological order if and only if it contains no cycles. (It is a DAG, a Directed Acyclic Graph.) This theorem is easy to prove by induction for finite graphs, which is the only kind we study here, but it is also true for infinite graphs. We can prove the theorem by giving an algorithm that finds either a topological order or a cycle. We give two such algorithms, both running in linear time.

The first [D. Knuth, "The Art of Computer Programming," Vol.1, Addison-Wesley, Reading, Mass. (1968)], the more straightforward one, repeatedly finds a vertex with no incoming arcs, gives it the next higher number, and deletes it. If it succeeds in numbering all the vertices, it produces a topological order, since when a vertex is numbered it can only have incoming arcs from previously numbered vertices. If it fails to number all the vertices, then each of the remaining vertices has at least one incoming arc. By starting at any vertex, proceeding backward along any incoming arc, and repeating this, we must eventually return to some previously reached vertex. The part of the traversed path from this vertex to itself is a cycle.

Exercise: Implement this algorithm so that it runs in linear time.

In general a DAG has many different topological orders, not just one. A DAG has a unique topological order if and only if it has a Hamiltonian path, a path containing each vertex exactly once.

The second topological ordering algorithm [R. E. Tarjan, "Depth-first search and linear graph algorithms, S.I.A.M. J. Computing, Vol. 1, No. 2 (1972), pp. 146-160] is to do a depth-first exploration of the graph and number the vertices in the reverse of the postorder generated by the exploration. This is the algorithm that Kevin presented in lecture as an example of the use of depth-first search. It works for the same reason that the first algorithm does, but the proof is subtler. It relies on the fact that the set of vertices so far visited in preorder but not postorder is exactly the set of vertices on the tree path generated by the search from the start vertex to the vertex currently being visited (in pre-or-post-order). Consider the next vertex *v* visited in postorder. This happens after all arcs or edges *vw* are traversed. Either every such *w* has already been visited in postorder, in which case it is safe to give *v* the next-smaller number in the topological order being constructed, or some such *w* has not yet been visited in postorder, in which case *w* is on the path of tree arcs to *v* from *s*, and there is a cycle, which can be found by following parent pointers from *v* back to *w*.

This discussion assumes that we know the properties of depth-first search. Let's back up and talk about graph search. Graph search is one of our most important tools for solving graph problems. Breadth-first search and depth-first search are the most common kinds of graph search, but there are other useful ones, and it behooves us to take a look at graph search in general. I'll present two ways to define graph search, the first as a *vertex-guided* process, and the second, which is more general, as an *edge-guided* process. Breadth-first search is an example of the former; depth-first search is an example of the latter, although as we shall see it can be shoehorned into the vertex-guided framework.

For simplicity I'll discuss directed-graph search, but the ideas carry over to undirected graphs. Reader beware: the discussion to follow is a bit rough around the edges. It needs polishing.

A vertex-guided graph search begins from a starting vertex, say s. It maintains a set S of *labeled* vertices, initially consisting only of s. Thus initialization consists of labeling s, by for example marking s as visited or assigning s a number, and initializing S to contain only s. The search continues until S is empty. The general step of the search consists of removing a vertex v from S and *scanning* it by *traversing* all its outgoing arcs vw. Traversing an edge vw consists of checking whether w has been labeled, and, if not, labeling w (marking it or numbering it), adding it to S, and doing any additional computations required by the specific application. In a vertex-guided search, each vertex is in one of three states: *unlabeled* (not yet labeled); *labeled* (labeled but not scanned); *scanned*.

A single search labels and scans all the vertices reachable from the start vertex s. It labels and scans each of these vertices exactly once, and traverses each of their outgoing arcs exactly once. To visit *all* the vertices and traverse *all* the arcs, we do an *exploration*, which begins with all vertices unlabeled and for *each* vertex s does a search starting from s if s is unlabeled.

A bare-bones vertex-guided search merely marks the edges as they become labeled. The rule for choosing the next vertex to scan determines the specific kind of search. Choosing S to be a queue (always scan the labeled vertex least recently added to S) is the rule that defines *breadth-first* search.

A question I asked in precept is, "Suppose we implement S as a stack. That is, we always scan the vertex *most* recently added to S. Does this give us depth-first search?

The answer (AND THIS IS IMPORTANT) is NO. One way to see this is to suppose that there is an arc from the start vertex s to *every* other vertex in the graph. (If this is not true, think of adding a dummy start vertex s with arcs to all other vertices.). Scanning s adds all other vertices to S. If we number the vertices when they are labeled, then we can get *any* numbering, since the numbering depends on the order in which vertices are added to S, which is the order in which they occur on the outgoing arc list of s. When any other vertex is deleted from S, its outgoing arcs are traversed, but since all vertices have already been labeled, no new vertices go on S. If we number vertices when they are scanned, we still can get any ordering. We have no way to generate a depth-first spanning tree, nor of obtaining depth-first pre-and-post-orders, whether vertices are numbered when labeled or when scanned.

We can make the answer YES, but we need to extend our notion of graph search. (Later we shall circle back to vertex-guided search.) The first step is to allow *S* to be a *bag* (a multiset), rather than just a set. When a vertex *v* is scanned, it adds to *S every* vertex *w* such that *vw* is an arc, whether or not *w* is already in *S*. Since a vertex can now be added *S* multiple times, when removing a vertex from *S* we only scan it if it is not yet scanned.

Rather than viewing *S* as a bag of vertices, it is better to view it as a set of edges. This gives us *edge-guided search*. An edge-guided search maintains the set of *traversable* arcs or edges, rather than the set of scannable vertices. These are the untraversed arcs out of *visited* vertices. Initialization consists of *visiting* the start vertex and adding all its outgoing arcs to *S*. The search continues until *S* is empty. The general step of the search consists of removing an arc *vw* from *S* and traversing it. Traversing *vw* consists of testing whether *w* has been visited; and, if not, visiting it, adding its outgoing arcs or incident edges to *S*, and doing any additional computation needed by the specific application. If *S* is a queue, we get an edge-guided version of breadth-first search, in which a vertex is "visited" when it is scanned rather than when it is labeled. If *S* is a stack, we get a non-recursive version of depth-first search, with the interesting distinction that the arcs out of a vertex *v* are traversed in the reverse of their order on the list of arcs out of *v*, rather than in order, assuming they are added to *S* in their order on the outgoing arc list of *v*. (Of course, if we add them in the reverse of their order on the outgoing list, the traversal order corresponds to order in recursive depth-first search.

Unlike vertex-guided search, in which all arcs out of a given vertex are traversed one after another, in edge-guided search the order of arc traversal is much more flexible: the time between the first traversal of an arc out of a vertex and the last traversal of an arc out of the same vertex may include traversals of arcs out of many other vertices.

In this initial version of edge-guided search, the set *S* can contain many edges entering (or incident to) the same vertex. Given that we generally use incidence lists to implement graph searches and examine arcs in the order they occur on an incidence list, we can modify edge-guided search so that *S* contains at most one outgoing arc per vertex. When we visit a vertex *v*, we add only its first outgoing arc to *S*. When we remove an arc *vw* from *S* and traverse it, we add to *S* the arc after *vw* on the list of arcs out of *v*, unless *vw* is the last arc on this list. Now *S* contains at most one outgoing arc from each vertex.

This does not quite give us a nice non-recursive implementation of depth-first search, though it does reduce the maximum size of *S* from ~*m* to ~*n*. A better idea is to store on stack *S* the arcs along which the search has *advanced* but not yet *retreated*, where these terms are defined in the following recursive implementation of depth-first search:

To do a depth-first search from start vertex *s*, unmark all vertices and call *dfs*(*s*), where *dfs*(*v*) is defined as follows:

```
dfs(v):
   mark v
   previsit(v)
   for each arc vw:
      advance(vw)
      if w is unmarked: [push vw on S]; dfs(w); [pop vw from S]
      retreat(vw)
   postvisit(v)
```

In this pseudocode, *previsit*, *postvisit*, *advance*, and *retreat* are stubs to be replaced by appropriate application-specific computations. Only tree arcs are pushed onto and popped from *S*. Previsit and postvisit are the pre- and post- visits to a vertex. Advance and retreat are the advance and retreat along an arc.

Exercise: Convert the recursive definition of depth-first search into a non-recursive implementation, using stack *S* to eliminate the recursion.

Returning to vertex-guided search, we can obtain at least some of the flexibility of edge-guided search by allowing rearrangement of the vertices in *S* based on traversals of arcs into them. For example, if we always scan the vertex with the most-recently-traversed incoming arc, we get the version of depth-first search that traverses the arcs on each outgoing arc list in reverse order. Implementing this requires the ability to maintain a stack with arbitrary deletion: when an arc into a vertex on the stack is traversed, we move that vertex to the top of the stack. Implementing the stack as a doubly-linked list will do the trick.

Other kinds of vertex-guided search that require rearrangement of *S are shortest-first* search (Dijkstra's algorithm) which computes shortest paths in directed graphs with non-negative arc lengths; *maximum-cardinality* search, in which the next vertex to be scanned is the labeled vertex with the largest number of incoming traversed arcs; and *lexicographic* search, which is a special case of breadth-first search in which level-order ties are broken based on a lexicographic order of the incoming traversed arcs. Shortest-first search is a generalization of breadth-first search and is a fundamental algorithm; maximum-cardinality and lexicographic search have specialized applications.