CSCI 4041: Algorithms and Data Structures
Graph Traversals
Part 3

Last revision April 21, 2020

Here's the agenda.

Partial and total orderings.
Topological sorting.
Example.

In this lecture, we'll look at a possible use for the depth-first graph traversal procedure GRAPH-DEPTH-FIRST that we saw earlier. It involves *topological sorting,* which determines the order that a sequence of actions should be performed to solve a problem. The topological sorting algorithm is described on pages 612–614 of Cormen.

Partial and total orderings. Suppose that a and b are integers, and that $a \neq b$. Then for all choices of a and b, either $a < b$, or $b < a$. We say that '<' is a *total ordering* for the integers, and that the set of integers are *totally ordered* under '<'. (We've seen a similar notion of total ordering earlier in the course, when we discussed sorting algorithms and BST algorithms.)

Suppose instead that a and b are objects, but not necessarily integers. Also suppose that $a \neq b$, and that '$\prec$' is a relation on those objects. (The symbol '$\prec$' is read as *precedes,* or as *banana peel.*) Finally, suppose that for *some* choices of a and b, either $a \prec b$, or $b \prec a$. Then we say that '$\prec$' is a *partial ordering* for those objects, and that the set of those objects is *partially ordered* under '$\prec$'.

(This is an informal definition of partial ordering. The formal definition says that a relation is a partial ordering if and only if it's reflexive, antisymmetric, and transitive. We might need that formal definition if we want to prove theorems about partial orderings. But I don't want to, so the informal definition is good enough.)

Now suppose that a and b are actions to be performed, and that $a \prec b$ means that a is to be performed before b. Then the partial ordering '$\prec$' tells us the sequence in which those actions should be performed. Imagine that we have a set of actions $\{ a_1, a_2 ..., a_n \}$. We'd like to arrange them into an ordered sequence, so that if a_i comes before a_j in the sequence, then either $a_i \prec a_j$, or else a_i and a_j are not compared by '$\prec$'. If we can do that, then the sequence tells us an order in which the actions can be performed.

Here's an example. Tabitha Picasso-Euler is an undergraduate who is majoring in art and mathematics. She's required to take the courses ART 101, ART 102, and ART 103, in that order. She's also required to take MATH 101, MATH 102, MATH 103, and MATH 104, in that order. These requirements can be represented by the following partial ordering.

MATH 101 $\prec$ MATH 102	ART 101 $\prec$ ART 102
MATH 102 $\prec$ MATH 103	ART 102 $\prec$ ART 103
MATH 103 $\prec$ MATH 104	

The relation '$\prec$' doesn't say whether art courses must be taken before mathematics courses, whether mathematics courses must be taken before art courses, or whether courses in both subjects can be taken at the same time.

When Tabitha plans her schedule, she must find a sequence of courses that are consistent with the partial ordering, in the way we've described. Here are a few such sequences. This is not a complete list.

MATH 101 · MATH 102 · MATH 103 · MATH 104 · ART 101 · ART 102 · ART 103

ART 101 · ART 102 · ART 103 · MATH 101 · MATH 102 · MATH 103 · MATH 104

MATH 101 · ART 101 · MATH 102 · ART 102 · MATH 103 · ART 103 · MATH 104

ART 101 · MATH 101 · MATH 102 · ART 102 · MATH 103 · MATH 104 · ART 103

How can Tabitha plan her schedule? An algorithm called *topological sort* can do it. (I don't know why it's called that.) One way to implement this algorithm uses a depth-first graph traversal.

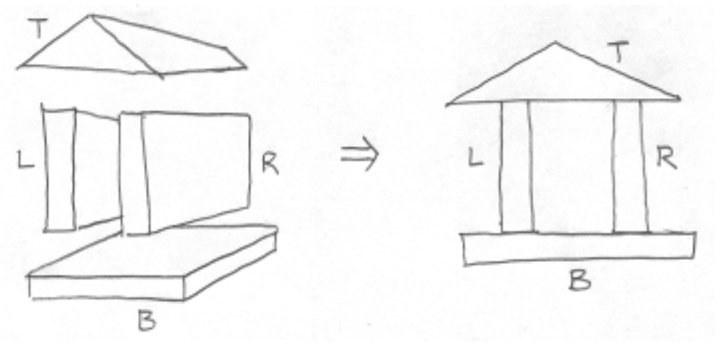
Topological sorting. To make the topological sort algorithm work, we need to represent a partial ordering as a graph.

Suppose *a* and *b* are actions to be performed, and $a \prec b$, so that *a* is performed before *b*. Then the graph has a directed edge from vertex *a* to vertex *b*, like this.

If there are many objects that are compared by the partial ordering relation '≺', then there are many vertexes in the graph that correspond to these objects, and many edges between them that correspond to the comparisons.

The topological sort algorithm works like this. We traverse the graph using the procedure GRAPH-DEPTH-FIRST from a previous lecture. Recall that for each vertex *v* in the graph, the procedure computes a *finish time v.d*. We then visit vertexes in the *reverse* order of their *v.d*'s. If each vertex represents an action, then we visit vertexes in the order that their actions should be performed.

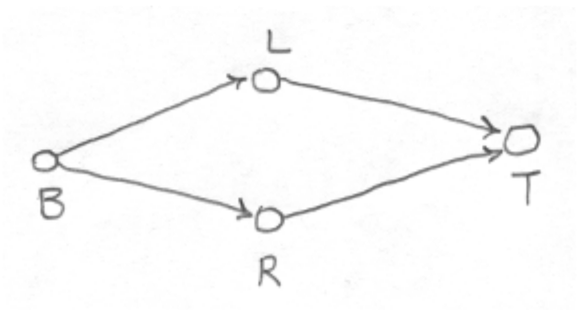
Example. Here's an example. Suppose we want to build a house from pre-fabricated components. (It's a very simple house, but this is a very simple example.) The components include a top *T*, a left wall *L*, a right wall *R*, and a bottom *B*. We can assemble them like this:



The components must be assembled in the correct order. We must start with the bottom *B*, then add the walls *L* and *R*, and finally add the top *T*. We might add *L* and then *R*, or *R* and then *L*; it doesn't matter. Now suppose $a \prec b$ means that component *a* must be added before component *b*. Then the following partial ordering tells us how to add the components.

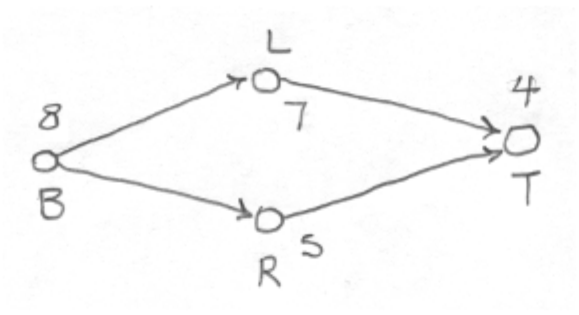
$B \prec L$
$B \prec R$
$L \prec T$
$R \prec T$

A directed graph that represents the partial ordering looks like this.



This is a *directed acyclic graph*, called a *dag* for short. Of course the graph must have no cycles, because if it did, then there would be no way to arrange its vertexes in a correct order. For example, if *a* and *b* are vertexes in a cyclic graph, then we might end up in a situation where $a \prec b$ but also $b \prec a$.

We now call GRAPH-DEPTH-FIRST on this graph. Recall that for each vertex *v*, it computes a discovery time *v.d*, a finish time *v.f*, and a predecessor *v.*π. All we care about are the finish times, so those are the numbers shown here.



We visit vertexes in reverse order of their finish times, so the graph tells us to first add the bottom *B,* then the left wall *L,* then the right wall *R,* and finally the top *T.*

Skeptics may want to run GRAPH-DEPTH-FIRST by hand to make sure this is a right answer. If you do that, note that GRAPH-DEPTH-FIRST uses loops that look at vertexes and edges in unspecified orders. As a result, the procedure may produce answers that are different from this one, but still correct. For example, it might decide to add the right wall *R* before the left wall *L.*

On page 613, Cormen suggests that the topological sort algorithm should report its results using a linear linked list of vertexes. The list is initially empty. Every time a vertex is finished, it adds that vertex to the front of the list. After it finishes running, the list will contain the vertexes in their proper order. Cormen also has a more elaborate example than mine on top of the same page.

The procedure GRAPH-DEPTH-FIRST runs in $\Theta(|V| + |E|)$ time on a graph whose vertexes are in the set *V* and whose edges are in the set *E.* It takes $O(1)$ time to insert a vertex at the front of a list. As a result, topological sort should also run in $\Theta(|V| + |E|)$ time.