Graph Search

(read Section 22 on Elementary Graph Algorithms in Cormen, Leiserson, Rivest, Stein)

We can think of graphs as generalizations of trees, consisting of nodes and of edges connecting nodes. The main difference is that graphs do not, in general, represent hierarchical organizations.

Types of Graphs. Different applications require different types of graphs. The most basic type is the undirected graph consisting of a finite set \( V \) of vertices and a set \( E \) of edges, each an unordered pair of vertices. Writing \( (V_2) \) for the collection of all unordered pairs, we have \( E \subseteq (V_2) \), implying that each edge occurs at most once. Similarly, because each edge is a set (of two vertices), it cannot connect to the same vertex twice. Vertices \( u \) and \( v \) are adjacent and neighbors of each other if \( \{u, v\} \in E \). Other types of graphs are:

- **directed**: \( E \subseteq V \times V \);
- **weighted**: has a weighting function \( w : E \rightarrow \mathbb{R} \);
- **labeled**: has a labeling function \( \ell : V \rightarrow \mathbb{Z} \);
- **non-simple**: there are loops and multi-edges.

Representation. The two most popular data structures for graphs are direct representations of adjacency. Let \( V = \{0, 1, \ldots, n - 1\} \) be the set of vertices. The adjacency matrix is the \( n \)-by-\( n \) matrix \( A = (a_{ij}) \) with

\[
    a_{ij} = \begin{cases} 
    1 & \text{if } \{i, j\} \in E, \\
    0 & \text{if } \{i, j\} \notin E.
\end{cases}
\]

For undirected graphs, we have \( a_{ij} = a_{ji} \), so \( A \) is symmetric. For weighted graphs, we encode more information than just the existence of an edge and define \( a_{ij} \) as the weight of the edge connecting \( i \) and \( j \). The adjacency matrix of the graph in Figure 83 is

\[
    A = \begin{pmatrix}
    0 & 0 & 1 & 0 & 0 \\
    0 & 1 & 1 & 0 & 0 \\
    0 & 1 & 0 & 1 & 0 \\
    0 & 0 & 1 & 0 & 1 \\
    \end{pmatrix},
\]

which is symmetric. Often the number of edges is quite small, maybe not much larger than the number of vertices. In these cases the adjacency matrix wastes memory and a better choice is a sparse matrix representation, referred to as adjacency lists, illustrated in Figure 84. It consists of a linear array \( V \) for the vertices and a list of neighbors for each vertex. For most algorithms we use the following declaration:

```c
typedef struct Edge { int v; Edge *next};
typedef struct Vertex { int d, f, pi; Edge *adj};
```

The \( d, f, \pi \) fields will be used to store auxiliary information used or created by the algorithms.
Depth-First Search. Since graphs are generally not ordered, there are many sequences in which the vertices can be traversed. In fact, it is not entirely straightforward to traverse such that each vertex is visited once and only once. A useful traversal method is depth-first search. It uses a global variable, time, which is incremented during the traversal and used to leave time-stamps behind that avoid repeated visits.

```c
void VISIT(int i)
1  time++; V[i].d = time;
2     forall neighbors j of i do
3      if V[j].d = 0 then
4      V[j].π = i; VISIT(j)
5      endif
6     endfor;
7  time++; V[i].f = time.
```

The test in line 2 checks whether the neighbor j of i has not yet been visited. The assignment in line 3 records that the vertex is visited from vertex i. A vertex is first stamped in line 1 with the time it is encountered. A vertex is second stamped in line 4 with the time its visit has been completed. To prepare the traversal we initialize the global time variable to 0, label all vertices as not yet visited, and call VISIT for all yet unvisited vertices.

```c
time = 0;
forall vertices i do V[i].d = 0 endfor;
forall vertices i do
  if V[i].d = 0 then V[i].π = 0; VISIT(i) endif
endfor.
```

Let n be the number of vertices and m the number of edges in the graph. Depth-first search visits every vertex once and examines every edge twice, once for each endpoint. The running time is therefore O(n + m), which is proportional to the size of the graph and therefore optimal.

Depth-first Forest. Figure 85 illustrates the depth-first traversal of the graph by showing the time-stamps d and f and the pointers π indicating the predecessors in the traversal. We classify an edge \( \{i, j\} \in E \) as a tree edge if \( i = V[j].\pi \) or \( j = V[i].\pi \) and as a back edge otherwise. The tree edges form the depth-first forest of the graph. The forest is a tree if the graph is connected and a collection of trees if it is not connected. Figure 86 shows the depth-first tree of the graph in Figure 85. The time-stamps d are consistent with the preorder traversal of the tree. The time-stamps f are consistent with the postorder traversal of the tree. Each node is stamped twice, once when it is encountered and another time when the visit is complete.

Nesting Lemma. Vertex j is a proper descendent of vertex i in the depth-first forest iff \( V[i].d < V[j].d \) and \( V[j].f < V[i].f \).

This is but a reformulation of the claim made in Problem 3(b) of the second homework assignment.

Directed Graphs and Relations. As mentioned earlier, we have a directed graph if its edges are directed. A directed graph is a way to think and talk about a mathematical relation. A typical problem where relations arise is scheduling. Some tasks are in a definite order while others are unrelated. An example is the scheduling of courses in your studies, as illustrated in Figure 87. Abstractly, a relation is a pair \((V, E)\), where \( V = \{0, 1, \ldots, n - 1\} \) is a finite set of elements and \( E \subseteq V \times V \) is a finite set of ordered pairs. Instead of \((i, j) \in E\) we write \( i \prec j \) and instead of \((V, E)\) we write \((V, \prec)\). If \( i \prec j \) then i is a predecessor of j and j is a successor of i. The terms relation, directed graph, digraph, network are all synonymous.

A cycle in a relation is a sequence \( i_0 \prec i_1 \prec \ldots \prec i_k \prec i_0 \). Even \( i_0 \prec i_0 \) is a cycle. A linear extension of \((V, \prec)\) is an ordering \( j_0, j_1, \ldots, j_{n-1} \) of the elements that is consistent with the relation. Formally, this means that \( j_k \prec j_\ell \) implies \( k < \ell \). A directed graph without cycle is
a directed acyclic graph.

**Extension Lemma.** \((V, \prec)\) has a linear extension iff it contains no cycle.

**Proof.** “⇒” is obvious. We prove “⇐” by induction. A vertex \(s \in V\) is a source if it has no predecessor. Assuming \((V, \prec)\) has no cycle, it has at least one source. Letting \(U = V - \{s\}\), we note that \((U, \prec)\) is a relation that is smaller than \((V, \prec)\). By induction hypothesis, \((U, \prec)\) has a linear extension, \(X\), and adding \(s\) at the front of \(X\) gives a linear extension of \((V, \prec)\).

**Topological Sorting.** The problem of constructing a linear extension is called topological sorting. A natural and fast algorithm follows the idea of the proof: find a source, print it, remove it, and repeat. To expedite the first step of finding a source, each vertex maintains its number of predecessors and a queue stores all sources. First, we initialize this information:

```plaintext
forall vertices do \(V[j].d = 0\) endfor;
forall vertices do
  forall successors do \(V[j].d++\) endfor;
forall vertices do
  if \(V[j].d = 0\) then ENQUEUE(j) endif;
endfor;
```

Next, we compute the linear extension by repeated deletion of a source:

```plaintext
while queue is non-empty do
  \(s =\) DEQUEUE;
  forall successors do \(V[j].d--\);
  if \(V[j].d = 0\) then ENQUEUE(j) endif;
endwhile.
```

The running time is linear in the number of vertices and edges, namely \(O(n + m)\). What happens if there is a cycle in the digraph? We illustrate the above algorithm for the directed acyclic graph in Figure 88. The sequence of vertices added to the queue is also the linear extension computed by the algorithm. If the process starts at vertex \(a\) and if the successors of a vertex are ordered by name then we get \(a, f, d, g, c, h, b, e\), which we can check is indeed a linear extension of the relation.

Another algorithm that can be used for topological sorting is depth-first search. We output a vertex when its visit has been completed, that is, when all its successors and their successors and so on have already been printed. The linear extension is therefore generated from back to front. Figure 89 shows the same digraph as Figure 88 and labels vertices with time stamps. Consider the sequence of vertices in the order of decreasing second time stamp:

\(a(16), f(14), g(13), h(12), d(9), c(8), e(7), b(5)\).

Although this sequence is different from the one computed by the earlier algorithm, it is also a linear extension of the relation.