Illinois Institute of Technology
Department of Computer Science

Lecture 18–19: November 1–8, 2010

CS 430 Introduction to Algorithms
Fall Semester, 2010

1 Graph theory

1.1 Terminology

A graph G is defined as a pair (V, E), where V is a set of *vertices* and E is a set of *edges*. A vertex can be thought of as a point and an edge as a line connecting the two points. (The problem we considered earlier in the lecture could be more precisely considered in graph theoretic terms.) An edge $e \in E$ is typically represented as a pair (u, v) where the edge is from u to v, where $u, v \in V$.

Graphs are simple structures with many useful applications. Depending on the application slight variations on the structure are useful. For instance, graphs can be *directed* or *undirected*. In a directed graph edges are unidirectional: for instance, an edge e might be from v_1 to v_2 but not from v_2 to v_1. A directed graph could model, say, the water pipes in a building or the network of (potentially one-way) roads in a city. In an undirected graph edges are bidirectional: an edge e might connect v_1 with v_2 in both directions. An undirected graph could model, for instance, the network of walkways around campus.[1]

Each edge in a graph may also have an associated *weight*. The weight on an edge could represent the capacity of a pipe or of a road. In rare cases, it is useful to associate weights with vertices rather than (or as well as) edges: for instance, the capacity through a pipe connector or of a road intersection.

Questions about graphs can be of a *structural* or of an *algorithmic* nature. Given a graph, a structural question would be whether it can be drawn with no crossing edges.[2] A related algorithmic question would be how to draw the graph with the fewest crossing edges. In this course we will be most concerned with algorithmic questions.

1.2 Representations

While a graph has a simple mathematical definition, we have yet to examine representations of graphs that are algorithmically useful. Three representations are presented here, adjacency structures (or adjacency lists), adjacency matrices, and incidence matrices.

1.2.1 Adjacency structures

The *adjacency structure* or *adjacency list* representation of a graph $G = (V, E)$ is an array of $|V|$ lists, one for each vertex in V. The list corresponding to the vertex v_i consists of the vertices in V that have edges from v_i. An adjacency structure requires $O(|V| + |E|)$ space.

[1] Note that an undirected graph can be defined in terms of a directed graph: if $(u, v) \in E$, then $(v, u) \in E$ as well.
[2] Such a graph is termed a *planar* graph.

1.2.2 Adjacency matrices

The *adjacency matrix* corresponding to a graph $G = (V, E)$, where $V = v_1, v_2, \ldots, v_{|V|}$, is a $|V| \times |V|$ binary matrix A defined by:

$$A_{ij} = \begin{cases} 1 \text{ if } (v_i, v_j) \in E \\ 0 \text{ otherwise} \end{cases}$$

An adjacency matrix requires $|V|^2$ bits of space.

1.2.3 Incidence matrices

If, for a graph $G = (V, E)$, $V = v_1, v_2, \ldots, v_{|V|}$ and $E = e_1, e_2, \ldots, e_{|E|}$, the *incidence matrix* associated with G is a $|V| \times |E|$ matrix B defined by:

$$B_{ij} = \begin{cases} -1 \text{ if edge } e_j \text{ leaves vertex } v_i \\ 1 \text{ if edge } e_j \text{ enters vertex } v_i \\ 0 \text{ otherwise} \end{cases}$$

An incidence matrix requires $O(|V|\ |E|)$ space.

2 Breadth-first search

2.1 The breadth-first search algorithm

The goal of breadth-first search is to explore a graph. The technique used is to start at an arbitrary vertex[3] and to visit its neighbors. The neighbor's neighbors are then examined in turn, and so on until all vertices have been visited.

In breadth-first search, we color vertices we have not yet visited white, vertices we have visited black, and vertices we are in the process of visiting gray. As well, we keep track of the in-process vertices in a queue. Thus the algorithm must ensure that any vertex colored gray is also on the queue and vice versa. Below is the algorithm of BFS:

```
BFS(G, s)
    for each vertex u ∈ V[G] − {s}
        do color[u] ← WHITE
           d[u] ← ∞
           π[u] ← NIL
    color[s] ← GRAY
    d[s] ← 0
    π[s] ← NIL
    Q ← {s}
    while Q ≠ ∅
        do u ← head[Q]
           for each v ∈ Adj[u]
               do if color[v] = WHITE
```

[3] If the graph is not connected (that is, if the vertices of the graph can be divided into two partitions with no edges between vertices in one partition and vertices in the other), it is actually quite relevant which vertex the algorithm begins with, as only one partition of the graph will be reached by breadth-first search.

```
            then color[v] ← GRAY
                 d[v] ← d[v] + 1
                 π[v] ← u
                 ENQUEUE (Q, v)
        DEQUEUE (Q)
        color[u] ←BLACK
```

This algorithm maintains two additional variables associated with each vertex. One of these is a time stamp, d, that is incremented each time a vertex is visited. The other is π, which identifies the predecessor to the vertex — that is, the vertex from which the current vertex was found. These variables are useful in some of the applications of breadth-first search.

2.2 Time complexity of breadth-first search

What is the time complexity of BFS? Each vertex is added to and removed from the queue at most once, when it is colored gray and when it is colored black, respectively. These queue operations take constant time, so the total work involved is $O(|V|)$.

Each edge is examined no more than twice, once for each vertex. Again, this involves constant time for each vertex, $O(|E|)$ in total.

Thus BFS requires $O(|V| + |E|)$ time.

2.3 An example of Greedy Strategy

Notice that if the graph is disconnected, BFS will only search the component of the graph containing s. Thus BFS is a conveninent algorithm for determining if a graph is connected: select an arbitrary vertex s and run BFS from that vertex. If, after the algorithm is complete, any of the vertices are still colored white, they were not reached by BFS and thus the graph is not connected.

We can use BFS to compute the **shortest path** from one vertex to every other vertex in an unweighted graph.[4] In particular, for any vertex $v \in G$, $d[v]$ is the length of the shortest path from s to v. The path itself can be constructed by following the π pointers "in reverse" from v back to s.

Two observations are crucial to prove the correctness of the BFS algorithm in finding the shortest pathes from one vertex to every other vertex in an unweighted graph:

1. Suppose that during the execution of BFS on a graph $G = \langle V, E\rangle$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r\rangle$, where v_1 is the head of Q and v_r is the tail. Then, $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$ for $i = 1, 2, \ldots, r - 1$.

2. When any new element v is put into the queue, $d[v] = d[u] + 1$, where $d[u]$ is the head of the queue.

These give us some very useful information about the distance values of those vertices in the queue (gray vertices).

Suppose our BFS algorithm is not correct and while traversing the graph, it makes its first mistake in finding the distance from s to v. The distance value of v is set by executing $d[v] \leftarrow d[u] + 1$, where u is the head of the queue. If this is wrong, we must have " distance from s to $v < d[u] + 1$ " and there is another path from s to v which is the shortest path. Since v is a white vertex before $d[v]$ is set, the shortest path must

[4] When weights are introduced, this problem becomes significantly more complex. Typically, for an efficient implementation, priority queues must be used.

contain an edge, say, x–y, where x is a gray vertex and y is a white vertex. $d[x]$ and $d[u]$ are both set and we know the values are correct since our algorithm made its FIRST mistake when computing $d[v]$. x and u are both gray vertices and u is the first element in the queue, therefore by the above observations we have $d[x] \geq d[u]$. Thus

$$\text{distance from } s \text{ to } v \geq d[x] + 1 \geq d[u] + 1,$$

which contradicts with our assumption about "the first mistake".

3 Depth-first search

As we have seen, breadth-first search offers us a method to visit the vertices of a graph. In particular, given a starting vertex s, BFS first visits s, then visits all vertices adjacent to s, then visits all vertices adjacent to those vertices, and so on. An alternative approach, and that used by depth-first search, is, intuitively, to "plunge in" — as each node is visited, visit its children before continuing the search at the same depth, and only back out when no unvisited children remain.

3.1 The depth-first search algorithm

If we modify the breadth-first search algorithm by changing the queue Q to a stack, we have derived depth-first search. Alternatively, we may use recursion to implement the stack; this is the approach typically taken. One small but useful change in the algorithm is a different treatment of time stamps: in DFS, each vertex has two associated time stamps, one, d, holding the discovery time, and the other, f, holding the completion time. The time is incremented at each d or f assignment. Here is the recursive version:

```
DFS(G)
    for each vertex u ∈ V[G]
        do   color[u] ← WHITE
             π[u] ← NIL
    time ← 0
    for each vertex u ∈ V[G]
        do if color[u] = WHITE
             then DFS-visit(u)

DFS-visit(u)
    color[u] ← GRAY
    d[u] ← time ← time + 1
    for each v ∈ Adj[u]
        do if color[v] = WHITE
             then  π[v] ← u
                   DFS-visit(v)
    color[u] ←BLACK
    f[u] ← time ← time + 1
```

3.2 Classification of edges

We can classify the edges in the graph G based on when the DFS algorithm traverses them:

A **tree edge** is an edge from a gray vertex to a white vertex. A tree edge brings the algorithm into deeper territory, as of yet undiscovered.

A **back edge** is an edge from a gray vertex to another gray vertex. Back edges form cycles in the graph, as they indicate that the algorithm has discovered a vertex further back in the path it is exploring.

Edges from gray vertices to black vertices fall into two classes. A **forward edge** connectes the current vertex to a vertex in the subtree rooted at the current vertex (its "descendent"). A **cross edge** connects the current vertex to a vertex in another subtree (for example, it's "cousin", "uncle" or "nephew").

3.3 Time complexity of depth-first search

Depth-first search examines each vertex exactly once. However, it must consider each edge to determine if it leads to an undiscovered (white) vertex. This leads to a running time of $\Theta(|V| + |E|)$.

3.4 The parenthesis theorem

The parenthesis theorem tells us that, for two vertices $u, v \in V$, it cannot be the case that $d[u] < d[v] < f[u] < f[v]$. This is a simple consequence of the depth-first nature of DFS. If the algorithm discovers u and then discovers v, it cannot later back out of u without first backin out of v.

3.5 Applications of depth-first search

3.5.1 Topological sort

One use of a directed acyclic graph (DAG) is for what civil engineers term a PERT[5] chart. A PERT chart indicates dependencies in a large-scale building project. For example, the walls can't be painted before there are walls to be painted. A dependency of v on u would be indicated by an edge from u to v — so, in our example, there would be an edge from an "erect walls" vertex to a "paint walls" vertex.

A **topological sort** of a DAG is a linear ordering of its vertices such that all edges point in the same (planar) direction. A topological sort of a PERT chart would produce one possible valid ordering of the components of the project. Note that for a general DAG, there are many valid topological sorts. Also note that if the graph contains a cycle, topological sort is not possible, as at least one edge would have to point in the wrong direction.

How can we find the topological sort of a DAG? We can simply run the depth-first search algorithm and sort the vertices in order of decreasing finish time. Alternatively, as DFS backs out of a vertex, it adds it to the front of a list. These two approaches produce identical results, although the second is more efficient as it does not need to sort the results of the DFS. The efficient algorithm runs in $\Theta(|V| + |E|)$ time, like DFS itself.

Why does this work? First, notice that a directed graph G has no cycles iff DFS produces no back edges in G. We need only show that for any pair of distinct vertices $u, v \in V$, if $(u, v) \in E$, then $f[v] < f[u]$. In a DFS exploration of G, v cannot be gray, since it would then be a back edge and the graph would have a cycle. If v is white, it is a descendant of u, so $f[v] < f[u]$. If v is black, $f[v] < f[u]$ as well. Thus for any edge $(u, v) \in E$, $f[v] < f[u]$.

[5] Program evaluation and review technique.

3.5.2 Strongly connected components

Run the depth-first search algorithm *twice*. After the first execution, reverse all the edges and run it a second time, processing the vertices adjacent to each vertex in decreasing order by finishing time.

Why does this work? The strongly connected components of a graph form a DAG. Reversing the edges has no effect inside a strongly connected component, but with the edges reversed, such a component is inaccessible from another component (all edges now go out), *as long as we start in the component that would be last in a topological order*. But we are porcessing the vertices in decreasing order by finishing time, that is, in topological order.