Illinois Institute of Technology
Department of Computer Science

# Lectures 18–19: November 5–10, 2014

CS 430 Introduction to Algorithms
Fall Semester, 2014

## 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. 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:

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

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 (gray) 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 of a graph $G$ starting at the vertex $s \in V[G]$:

```
function BFS($G$, $s$)
1: for all $u \in V[G] - \{s\}$ do
2:    $color[u] \leftarrow$ WHITE
3:    $d[u] \leftarrow \infty$
4:    $\pi[u] \leftarrow$ NIL
5: end for
6: $color[s] \leftarrow$ GRAY
7: $d[s] \leftarrow 0$
8: $\pi[s] \leftarrow$ NIL
9: $Q \leftarrow \{s\}$
10: while $Q \neq \emptyset$ do
11:    $u \leftarrow$ DEQUEUE ($Q$)
12:    for all $v \in Adj[u]$ do
13:       if $color[v] =$ WHITE then
14:          $color[v] \leftarrow$ GRAY
```

[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.

```
15:             $d[v] \leftarrow d[u] + 1$
16:             $\pi[v] \leftarrow u$
17:             ENQUEUE $(Q, v)$
18:         **end if**
19:     **end for**
20:     $color[u] \leftarrow$ BLACK
21: **end while**
```

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 $\pi$, 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.

The edges are of the following types:

| | |
|---|---|
| Black vertex→black vertex | Completely explored territory |
| Black vertex→gray vertex | Haven't yet finished with latter vertex |
| Gray vertex→gray vertex | Haven't yet finished with either vertex |
| Gray vertex→white vertex | Haven't yet finished with former, haven't seen latter |
| White vertex→white vertex | Haven't seen either vertex |

## 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 useful 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 $\pi$ pointers "in reverse" from $v$ back to $s$.

Two observations are crucial to prove the correctness of the BFS algorithm in finding the shortest paths 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.

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

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 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

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

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:

```
function DFS($G$)
 1: for all $u \in V[G]$ do
 2:    $color[u] \leftarrow$ WHITE
 3:    $\pi[u] \leftarrow$ NIL
 4: end for
 5: $time \leftarrow 0$
 6: for all $u \in V[G]$ do
 7:    if $color[u] =$ WHITE then
 8:       DFS-visit(u)
 9:    end if
10: end for
```

```
function DFS-visit($u$)
 1: $color[u] \leftarrow$ GRAY
 2: $d[u] \leftarrow time \leftarrow time + 1$
 3: for all $v \in Adj[u]$ do
 4:    if $color[v] =$ WHITE then
 5:       $\pi[v] \leftarrow u$
 6:       DFS-visit($v$)
 7:    end if
 8: end for
 9: $color[u] \leftarrow$ BLACK
10: $f[u] \leftarrow time \leftarrow time + 1$
```

### 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** connects the current vertex to a vertex in the subtree rooted at the current vertex (its “descendant”). 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 $d[u] < d[v] < f[u] < f[v]$; that is, the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are either disjoint or nested. 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 backing 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

[5]Program evaluation and review technique.

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 if and only if 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 $G$, an edge $u$ to $v$ means $v$ cannot be gray, since it would then be a back edge (and hence there would be 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 $(u, v) \in E$, $f[v] < f[u]$.

## 4 Minimum Spanning Trees

Consider a connected, undirected graph $G = (V, E)$. For many applications it is useful to find a subgraph $T = (V, E')$ of $G$, where $E'$ is the smallest collection of edges sufficient to make $T$ a connected graph, i.e. $T$ is a tree. Subgraphs such as $T$ are called *spanning trees* of $G$. This problem can be made more general by assigning each edge of $E$ a weight and finding the spanning tree whose edges have the smallest combined weight. Such a spanning tree is called a *minimum spanning tree.* It turns out that such trees can be found efficiently. We examine a number of algorithms for doing so.[6]

### 4.1 Prim's Algorithm

The first algorithm we examine for the minimum spanning tree problem is called Prim's algorithm. In this algorithm, the spanning tree is "grown" from a single initial vertex, called the *source vertex.* Each step of the algorithm involves adding a single edge and vertex to the existing tree. This is done in a greedy manner by adding the cheapest edge from the current tree to some "outside" vertex. This can be implemented efficiently by putting the vertices into a priority queue. The source vertex is initially assigned weight 0 and all other vertices are assigned infinite weight. These weights are an estimate of the distance from the vertex to the current tree. When a vertex $u$ is removed from the queue all its edges are examined. For each outgoing edge which goes to a vertex not already in the tree, the cost of including that edge is compared to the current weight of that vertex. If the examined edge is cheaper, then the distance estimate is revised by calling Decrease-Key to adjust the weight of that vertex. If the vertices each remember the cheapest edge connecting them to the tree, then the edges which should be added to the tree can be easily determined as well.

Now we show that this algorithm produces a minimum spanning tree. As with other greedy algorithms, we do this by looking at the algorithm's first mistake. Let this first mistake occur when we add some vertex $v$ to the tree $T$ using an edge $uv$ which is not in any optimal minimum spanning tree of the graph. Let $T_{opt}$ be an optimal tree which includes the edges in $T$. There must be a path from $u$ to $v$ in this tree. Since $u$ in inside $T$ and $v$ is not, at some point this path must cross from vertices in $T$ to vertices not in $T$. Let $wx$ be the first edge leaving the vertices of $T$ along this path. Clearly removing $wx$ from $T_{opt}$ and replacing it with $uv$ creates a spanning tree. Furthermore, since we choose to connect a vertex via the cheapest outgoing edge at each step, the edge $wx$ must cost at least as much as $uv$. Thus, the new spanning tree we create must cost no more than $T_{opt}$, making it a minimum spanning tree. However, since $T + uv$ is a subset of this tree, we must not have made a mistake adding $uv$. Therefore, the algorithm produces a minimum spanning tree.

What is the running time of this algorithm? If we use a Fibonacci heap to implement the priority queue, the vertices can be added to the queue in $O(V)$ time, the vertices can be extracted in a total of $O(V \log V)$ time,

[6]This problem is discussed in chapter 23 of CLRS, though it is given a slightly different treatment here.

and the keys can be decreased in a total of $O(E)$ time. Thus the entire algorithm runs in $O(E + V \log V)$ time.

## 4.2 Kruskal's Algorithm

Kruskal's algorithm is a different greedy approach to finding a minimum spanning tree. Rather than starting from a single source vertex as we do with Prim's algorithm, Kruskal's algorithm grows separate subtrees to grow until they are eventually connected to form a minimum spanning tree. Initially, each vertex is a trivial subtree. At each subsequent step, we add the cheapest edge which connects different subtrees.

First we show that this algorithm produces a minimum spanning tree, again by analyzing the first "mistake" made by the algorithm. Suppose that some set $S$ of edges has already been correctly added before some edge $uv$ not in any minimum spanning tree is added. Let $T_{opt}$ be a minimum spanning tree which uses the edges of $S$. There must be a path between vertices $u$ and $v$ in $T_{opt}$. Since $u$ and $v$ were in separate components before that edge was added, there must be some first edge $wx$ not in $S$ along this path. Swap $uv$ for this edge in $T_{opt}$. Since we choose edges in order of weight and have not chosen $wx$ yet, the resulting spanning tree must also be minimal and therefore the algorithm did not make a mistake.

Now we consider the running time of Kruskal's algorithm. First the algorithm sorts the edges by weight, in $O(E \log E)$ time. In order to keep track of which vertices are already connected, we keep track of the components with a disjoint set data structure. Thus, for initialization we build $|V|$ new sets (calling the Make-Set operation on each vertex) in $O(V)$ time. For each edge $(u, v)$ considered in increasing order, we determine if $u$ and $v$ are in the same tree by comparing Find-Set$(u)$ with Find-Set$(v)$. If $u$ and $v$ are indeed in different trees, we add the edge $(u, v)$ and Union the two trees. Since at worst we do 2 Find-Set operations per edge, this loop takes at most $O(E\alpha(V)) \leq O(E \log V)$ time. Thus the full algorithm takes $O(E \log E)$ time. This is asymptotically more than the running time of Prim's algorithm, but Kruskal's may be more practical for small problem instances since it does not require a Fibonacci heap, which introduces some very large constants.