Observation: For a digraph $e \leq n^2 = O(n^2)$. For an undirected graph $e \leq \binom{n}{2} = n(n-1)/2 = O(n^2)$.

A graph or digraph is allowed to have no edges at all. One interesting question is what the minimum number of edges that a connected graph must have.

We say that a graph is *sparse* if e is much less than n^2.

For example, the important class of *planar graphs* (graphs which can be drawn on the plane so that no two edges cross over one another) $e = O(n)$. In most application areas, very large graphs tend to be sparse. This is important to keep in mind when designing graph algorithms, because when n is really large and $O(n^2)$ running time is often unacceptably large for real-time response.

Lecture 22: Graphs Representations and BFS

(Thursday, April 16, 1998)

Read: Sections 23.1 through 23.3 in CLR.

Representations of Graphs and Digraphs: We will describe two ways of representing graphs and digraphs. First we show how to represent digraphs. Let $G = (V, E)$ be a digraph with $n = |V|$ and let $e = |E|$. We will assume that the vertices of G are indexed $\{1, 2, \ldots, n\}$.

Adjacency Matrix: An $n \times n$ matrix defined for $1 \leq v, w \leq n$.

$$A[v, w] = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise.} \end{cases}$$

If the digraph has weights we can store the weights in the matrix. For example if $(v, w) \in E$ then $A[v, w] = W(v, w)$ (the weight on edge (v, w)). If $(v, w) \notin E$ then generally $W(v, w)$ need not be defined, but often we set it to some "special" value, e.g. $A(v, w) = -1$, or ∞. (By ∞ we mean (in practice) some number which is larger than any allowable weight. In practice, this might be some machine dependent constant like `MAXINT`.)

Adjacency List: An array $Adj[1 \ldots n]$ of pointers where for $1 \leq v \leq n$, $Adj[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.

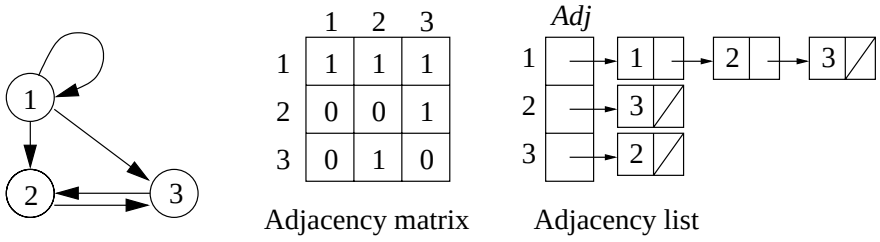

Figure 23: Adjacency matrix and adjacency list for digraphs.

We can represent undirected graphs using exactly the same representation, but we will store each edge twice. In particular, we representing the undirected edge $\{v, w\}$ by the two oppositely directed edges (v, w) and (w, v). Notice that even though we represent undirected graphs in the same way that we

represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

This can cause some complications. For example, suppose you write an algorithm that operates by marking edges of a graph. You need to be careful when you mark edge (v, w) in the representation that you also mark (w, v), since they are both the same edge in reality. When dealing with adjacency lists, it may not be convenient to walk down the entire linked list, so it is common to include *cross links* between corresponding edges.

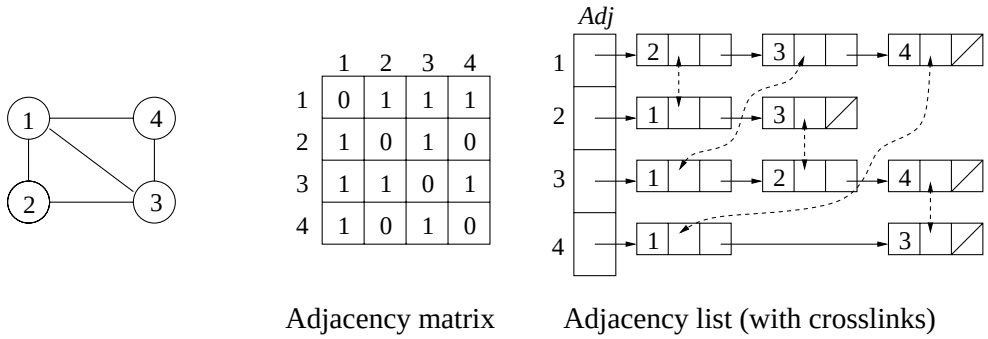

Figure 24: Adjacency matrix and adjacency list for graphs.

An adjacency matrix requires $\Theta(n^2)$ storage and an adjacency list requires $\Theta(n+e)$ storage (one entry for each vertex in Adj and each list has *outdeg*(v) entries, which when summed is $\Theta(e)$. For sparse graphs the adjacency list representation is more cost effective.

Shortest Paths: To motivate our first algorithm on graphs, consider the following problem. You are given an undirected graph $G = (V, E)$ (by the way, everything we will be saying can be extended to directed graphs, with only a few small changes) and a *source vertex* $s \in V$. The *length* of a path in a graph (without edge weights) is the number of edges on the path. We would like to find the shortest path from s to each other vertex in G. If there are ties (two shortest paths of the same length) then either path may be chosen arbitrarily.

The final result will be represented in the following way. For each vertex $v \in V$, we will store $d[v]$ which is the *distance* (length of the shortest path) from s to v. Note that $d[s] = 0$. We will also store a *predecessor (or parent) pointer* $\pi[v]$, which indicates the first vertex along the shortest path if we walk from v backwards to s. We will let $\pi[s] =$ NIL.

It may not be obvious at first, but these single predecessor pointers are sufficient to reconstruct the shortest path to any vertex. Why? We make use of a simple fact which is an example of a more general principal of many optimization problems, called the *principal of optimality*. For a path to be a shortest path, every subpath of the path must be a shortest path. (If not, then the subpath could be replaced with a shorter subpath, implying that the original path was not shortest after all.)

Using this observation, we know that the last edge on the shortest path from s to v is the edge (u, v), then the first part of the path *must* consist of a shortest path from s to u. Thus by following the predecessor pointers we will construct the *reverse* of the shortest path from s to v.

Obviously, there is simple brute-force strategy for computing shortest paths. We could simply start enumerating all simple paths starting at s, and keep track of the shortest path arriving at each vertex. However, since there can be as many as $n!$ simple paths in a graph (consider a complete graph), then this strategy is clearly impractical.

Here is a simple strategy that is more efficient. Start with the source vertex s. Clearly, the distance to each of s's neighbors is exactly 1. Label all of them with this distance. Now consider the unvisited

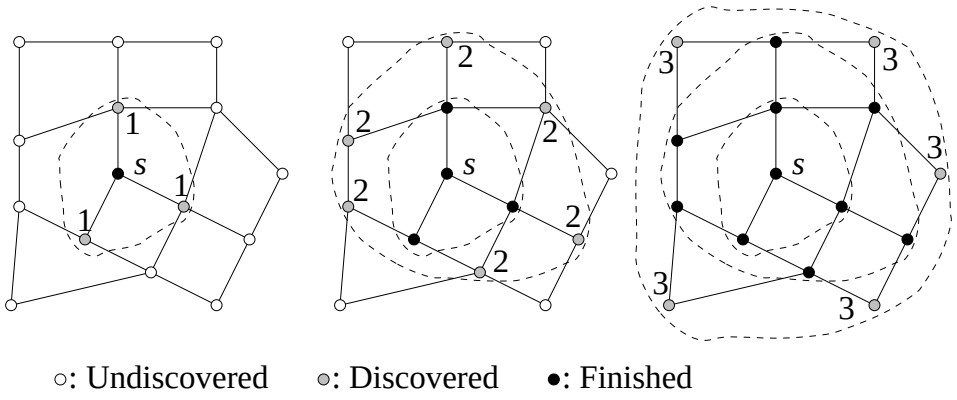

Figure 25: Breadth-first search for shortest paths.

neighbors of these neighbors. They will be at distance 2 from s. Next consider the unvisited neighbors of the neighbors of the neighbors, and so on. Repeat this until there are no more unvisited neighbors left to visit. This algorithm can be *visualized* as simulating a wave propagating outwards from s, visiting the vertices in bands at ever increasing distances from s.

Breadth-first search: Given an graph $G = (V, E)$, breadth-first search starts at some source vertex s and "discovers" which vertices are reachable from s. Define the *distance* between a vertex v and s to be the minimum number of edges on a path from s to v. Breadth-first search discovers vertices in increasing order of distance, and hence can be used as an algorithm for computing shortest paths. At any given time there is a "frontier" of vertices that have been discovered, but not yet processed. Breadth-first search is named because it visits vertices across the entire "breadth" of this frontier.

Initially all vertices (except the source) are colored white, meaning that they have not been seen. When a vertex has first been discovered, it is colored gray (and is part of the frontier). When a gray vertex is processed, then it becomes black.

Breadth-First Search

```
BFS(graph G=(V,E), vertex s) {
    int d[1..size(V)]                       // vertex distances
    int color[1..size(V)]                   // vertex colors
    vertex pred[1..size(V)]                 // predecessor pointers
    queue Q = empty                         // FIFO queue

    for each u in V {                       // initialization
        color[u] = white
        d[u]     = INFINITY
        pred[u]  = NULL
    }
    color[s] = gray                         // initialize source s
    d[s] = 0
    enqueue(Q, s)                           // put source in the queue
    while (Q is nonempty) {
        u = dequeue(Q)                      // u is the next vertex to visit
        for each v in Adj[u] {
            if (color[v] == white) {        // if neighbor v undiscovered
                color[v] = gray             // ...mark it discovered
                d[v]     = d[u]+1           // ...set its distance
                pred[v]  = u                // ...and its predecessor
```

```
                enqueue(Q, v)                    // ...put it in the queue
            }
        }
        color[u] = black                         // we are done with u
    }
}
```

The search makes use of a *queue*, a first-in first-out list, where elements are removed in the same order they are inserted. The first item in the queue (the next to be removed) is called the *head* of the queue. We will also maintain arrays $color[u]$ which holds the color of vertex u (either white, gray or black), $pred[u]$ which points to the predecessor of u (i.e. the vertex who first discovered u, and $d[u]$, the distance from s to u. Only the color is really needed for the search, but the others are useful depending on the application.

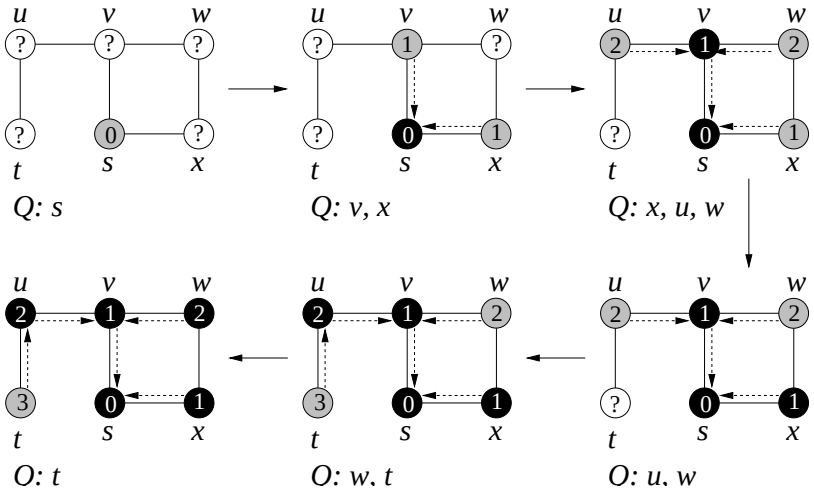

Figure 26: Breadth-first search: Example.

Observe that the predecessor pointers of the BFS search define an inverted tree. If we reverse these edges we get a rooted unordered tree called a *BFS tree* for G. (Note that there are many potential BFS trees for a given graph, depending on where the search starts, and in what order vertices are placed on the queue.) These edges of G are called *tree edges* and the remaining edges of G are called *cross edges*. It is not hard to prove that if G is an undirected graph, then cross edges always go between two nodes that are at most one level apart in the BFS tree. The reason is that if any cross edge spanned two or more levels, then when the vertex at the higher level (closer to the root) was being processed, it would have discovered the other vertex, implying that the other vertex would appear on the very next level of the tree, a contradiction. (In a directed graph cross edges will generally go down at most 1 level, but they may come up an arbitrary number of levels.)

Analysis: The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. Let $n = |V|$ and $e = |E|$. Observe that the initialization portion requires $\Theta(n)$ time. The real meat is in the traversal loop. Since we never visit a vertex twice, the number of times we go through the while loop is at most n (exactly n assuming each vertex is reachable from the source). The number of iterations through the inner for loop is proportional to $deg(u) + 1$. (The $+1$ is because even if $deg(u) = 0$, we need to spend a constant amount of time to set up the loop.) Summing up over all vertices we have the running time

$$T(n) = n + \sum_{u \in V} (deg(u) + 1) = n + \sum_{u \in V} deg(u) + n = 2n + 2e \in \Theta(n + e).$$

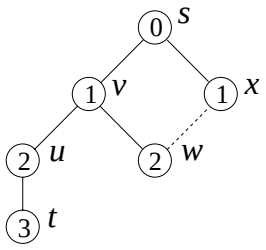

Figure 27: BFS tree.

For an directed graph the analysis is essentially the same.

Lecture 23: All-Pairs Shortest Paths

(Tuesday, April 21, 1998)

Read: Chapt 26 (up to Section 26.2) in CLR.

All-Pairs Shortest Paths: Last time we showed how to compute shortest paths starting at a designated source vertex, and assuming that there are no weights on the edges. Today we talk about a considerable generalization of this problem. First, we compute shortest paths not from a single vertex, but from every vertex in the graph. Second, we allow edges in the graph to have numeric *weights*.

Let $G = (V, E)$ be a directed graph with edge weights. If (u, v) E, is an edge of G, then the weight of this edge is denoted $W(u, v)$. Intuitively, this weight denotes the distance of the road from u to v, or more generally the cost of traveling from u to v. For now, let us think of the weights as being positive values, but we will see that the algorithm that we are about to present can handle negative weights as well, in special cases. Intuitively a negative weight means that you get paid for traveling from u to v. Given a path $\pi = \langle u_0, u_1, \ldots, u_k \rangle$, the *cost* of this path is the sum of the edge weights:

$$cost(\pi) = W(u_0, u_1) + W(u_1, u_2) + \cdots W(u_{k-1}, u_k) = \sum_{i=1}^{k} W(u_{i-1}, u_i).$$

(We will avoid using the term *length*, since it can be confused with the number of edges on the path.) The *distance* between two vertices is the cost of the minimum cost path between them.

We consider the problem of determining the cost of the shortest path between all pairs of vertices in a weighted directed graph. We will present two algorithms for this problem. The first is a rather naive $\Theta(n^4)$ algorithm, and the second is a $\Theta(n^3)$ algorithm. The latter is called the *Floyd-Warshall algorithm*. Both algorithms is based on a completely different algorithm design technique, called *dynamic programming*.

For these algorithms, we will assume that the digraph is represented as an adjacency matrix, rather than the more common adjacency list. Recall that adjacency lists are generally more efficient for sparse graphs (and large graphs tend to be sparse). However, storing all the distance information between each pair of vertices, will quickly yield a dense digraph (since typically almost every vertex can reach almost every other vertex). Therefore, since the output will be dense, there is no real harm in using the adjacency matrix.

Because both algorithms are matrix-based, we will employ common matrix notation, using i, j and k to denote vertices rather than u, v, and w as we usually do. Let $G = (V, E, w)$ denote the input digraph and its edge weight function. The edge weights may be positive, zero, or negative, but we assume that