GRAPH ALGORITHMS

It's an abstract notion, used to represent the idea of some kind of connection between pairs of objects. A graph consists of:

- A collection of "vertices", which I'll usually draw as small circles on the blackboard, and
- A collection of "edges", each connecting some two vertices. I'll usually draw these as curves on the blackboard connecting the given pair of vertices.

For this definition it doesn't matter what the vertices or edges represent -- that will be different depending on what application the graph comes from. It also doesn't matter how I draw the graph, the only part that matters is which pairs of vertices are connected with each other.

There are two different types of graph that we'll commonly see.

- In one type, known as an *undirected graph* it doesn't matter which end of the graph is which -- the relation between the two is symmetric. In that case I'll just draw the edges as an undecorated curve.

  For instance, suppose we draw a graph with one vertex for every person in the U.S., and one edge connecting any two people who have shaken hands with each other. If I've shaken hands with you, you've shaken hands with me, so each edge is symmetric. It seems to be true that graphs like this have very small {\em diameter}: you can connect any two people with a very short chain of handshakes.

- In the other type of graph, known as a *directed graph*, an edge goes from one of the vertices, towards the other. In this case I'll draw an arrowhead at the vertex towards which the edge is going. If we drew a graph like the handshake graph described above, but instead connected two people if one had written a letter to the other, the result would be directed: if I've written a letter to you, you may not have written a letter back to me.

Whenever we talk about graphs, we'll use n to denote the number of vertices, and m to denote the number of edges. This notation is confusing, but I'll stick with it because it's very standard. We'll also use V(G) to denote the set of vertices of a graph G, and E(G) to denote the set of edges.

Representation of Graphs

In order to perform graph algorithms in a computer, we have to decide how to store the graph. When I talk about graphs to you, I'll draw a diagram with circles and lines on the blackboard, but computers aren't very good at interpreting that sort of input. Instead we need a representation closer to the abstract definition of a graph. There are several possibilities, with different uses. I didn't go over all of them in the lecture (I left out the incidence list and incidence matrix.)

- **Object oriented representation**. The most obvious thing to do is just to copy the definition I gave of a graph. Have some structure for each vertex (representing whatever information you want to store there), and another structure for each edge (with pointers to the two vertices it connects). This representation is a little difficult to work with, because unless the edges are ordered more carefully it will be difficult to find the ones you want.

  As an example we might have a graph with four vertices (which I'll call A, B, C, and D) and four edges: (A,B), (C,D), (B,C), (C,A). The object oriented representation would just have a list or array of structures for each of these objects.

  The total space used by this representation is just O(m+n), since there is a constant amount of space (one structure) per vertex or edge. Most operations in this representation involve scanning the whole list of edges, and take time O(m).

- **Adjacency list** In the adjacency list representation, each vertex keeps a linked list of the neighboring vertices. The edges don't really appear at all. In the same graph above, the lists for each vertex would be:

  A: B, C
  B: A, C
  C: D, B, A

  This representation makes it much easier to find the edges connected to any particular vertex. Its space is still also small: O(m+n), since the total length of all the lists is 2m (each edge appears twice, once for each of its endpoints). It is also quite fast for many applications. The slowest operation that might commonly be used is testing whether a pair of vertices is connected by an edge; this has to be done by scanning through one of the lists, but you could speed it up by sorting the adjacency lists and using binary search. Another disadvantage is that each edge is listed twice, so if the edges carry any extra information such as a length it may be complicated to keep track of both copies and make sure they have the same information.

- **Incidence list**. By combining the adjacency list and the object oriented representation, we get something with the advantages of both. We just add to the object oriented representation a list, for each vertex, of pointers to the edges incident to it. If I don't specify a representation, this is probably what I have in mind. The space is a little larger than the previous two representations, but still O(m+n).
- **Adjacency matrix**. In some situations we are willing to use a somewhat larger data structure so that we can test very quickly whether an edge exists. We make an n x n matrix M[i,j], with rows and columns indexed by vertices. If edge (u,v) present, we put a one in cell M[u,v]; otherwise we leave M[u,v] zero. Finding the neighbors of a vertex involves scanning a row in O(n) time, but to test if an edge (u,v) exists just look in that entry, in constant time. To store extra information like edge lengths, you can just use more matrices. For the same graph above, we'd have a matrix

0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0

This is occasionally useful for performing graph computations by linear algebra. For instance, if you take the kth power of this matrix, an entry will be nonzero if and only if the corresponding vertices are connected by a path of k edges. For undirected graphs, the matrix will be symmetric.

- **Incidence matrix**. This is another matrix, but generally it is rectangular rather than square (n x m); the rows are indexed by vertices and the columns by edges. Just like the adjacency matrix, we put a one in a cell when the corresponding vertex and edge are incident. Therefore every column will have exactly two ones in it. For a directed graph, you can make a similar matrix in which every column has one +1 and one -1 entry. This matrix is usually not symmetric. For the same graph, the incidence matrix is

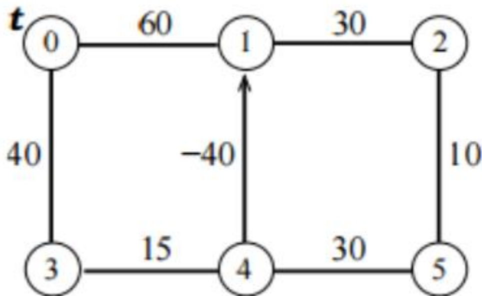
  1 0 1 0
  1 1 0 0
  0 1 1 1
  0 0 0 1

Shortest Paths

We are now going to turn to another basic graph problem: finding shortest paths in a weighted graph, and we will look at several algorithms based on Dynamic Programming. For an edge (i,j) in our graph, let's use len(i,j) to denote its length. The basic shortest-path problem is as follows: Definition 12.1 Given a weighted, directed graph G, a start node s and a destination node t, the s-t shortest path problem is to output the shortest path from s to t. The single-source shortest path problem is to find shortest paths from s to every node in G. The (algorithmically equivalent) single-sink shortest path problem is to find shortest paths from every node in G to t. We will allow for negative-weight edges (we'll later see some problems where this comes up when using shortest-path algorithms as a subroutine) but will assume no negative-weight cycles (else the shortest path can wrap around such a cycle infinitely often and has length negative infinity). As a shorthand, if there is an edge of length ℓ from i to j and also an edge of length ℓ from j to i, we will often just draw them together as a single undirected edge. So, all such edges must have positive weight.

The Bellman-Ford Algorithm

We will now look at a Dynamic Programming algorithm called the Bellman-Ford Algorithm for the single-sink (or single-source) shortest path problem.3 Let us develop the algorithm using the following example:



Compute the lengths of the shortest paths first, and afterwards we can easily reconstruct the paths themselves. The idea for the algorithm is as follows: 1. For each node v, find the length of the shortest path to t that uses at most 1 edge, or write down ∞ if there is no such path. This is easy: if v = t we get 0; if (v,t) ∈ E then we get len(v,t); else just put down ∞. 2. Now, suppose for all v we have solved for length of the shortest path to t that uses i − 1 or fewer edges. How can we use this to solve for the shortest path that uses i or fewer edges? Answer: the shortest path from v to t that uses i or fewer edges will first go to some neighbor x of v, and then take the shortest path from x to t that uses i−1 or fewer edges, which we've already solved for! So, we just need to take the min over all neighbors x of v. 3. How far do we need to go? Answer: at most i = n − 1 edges. Specifically, here is pseudocode for the algorithm. We will use d[v][i] to denote the length of the shortest path from v to t that uses i or fewer edges (if it exists) and infinity otherwise ("d" for "distance"). Also, for convenience we will use a base case of i = 0 rather than i = 1.

Progressive Alignment

- *Progressive alignment* is a variation of greedy algorithms with a somewhat more intelligent strategy for scheduling the merges
- Progressive alignment works well for close sequences, but deteriorates for distant sequences
    - Gaps in consensus string are permanent
    - Simplified representation of the alignments

Feng-Doolittle Progressive Alignment

Step 1: Compute all possible pairwise alignments

Step 2: Convert alignment scores to distances

Step 3: Construct a "guide tree" by clustering

Step 4: Progressive alignment based on the guide tree (bottom up)

Generating multiple alignment

- At each step, follow the guide tree and consider all possible pairwise alignments of sequences in the two candidate groups (3 cases):

Sequence vs. sequence

Sequence vs. group (the best matching sequence in the group determines the alignment)

group vs. group (the best matching pair of sequences determines the alignment)

- "Once a gap, always a gap"

gap is replaced by a neutral symbol X

X can be matched with any symbol, including a gap without penalty

- Align the two most similar sequences
- Following the guide tree, add in the next sequences, aligning to the existing alignment
- Insert gaps as necessary

ALGORITHM:

The Feng-Dolittle algorithm was one of the first progressive alignment algorithms. In overview, it is as follows:

(i) Calculate a diagonal matrix of $N(N-1)/2$ distances between all pairs of sequences by standard pairwise alignment, converting raw alignment scores to approximate pairwise 'distances'. N

(ii) Construct a guide tree from the distance matrix using the clustering algorithm.

(iii) Starting from the first node added to the tree, align the child nodes (which may be two sequences, a sequence and an alignment, or two alignments). Repeat for all other nodes in the order that they were added to the tree (i.e. from most similar pairs to least similar pairs) until all sequences have been aligned.

The method for converting alignment scores to distances does not need to be especially accurate, as the goal is only to create an approximate guide tree. Feng & Doolittle calculate the distance D as

$$D = -\log S_{\text{eff}} = -\log \frac{S_{\text{obs}} - S_{\text{rand}}}{S_{\max} - S_{\text{rand}}},$$

where S_{obs} is the observed pairwise alignment score; is the maximum score, S_{max} the average of the score of aligning either sequence to itself; S_{rand} and is the expected score for aligning two random sequences of the same length and residue composition. The last one, S_{rand}, may either be calculated by random shuffling of the two sequences, or by an approximate calculation given in Feng & Doolittle. The effective score S_{eff} can thus be viewed as a normalized percentage similarity; it is expected to roughly decay exponentially towards zero with increasing evolutionary distance, hence the to make the measure more approximately linear with evolutionary distance. Sobs rand Smax Srand S Seff − log Sequence-sequence alignments are done with the usual pairwise dynamic programming algorithm. A sequence is added to an existing group by aligning it pairwise to each sequence in the group in turn. The highest scoring pairwise alignment determines how the sequence will be aligned to the group. For aligning a group to a group, all sequence pairs between the two groups are tried; the best pairwise sequence alignment determines the alignment of the two groups. Thus, the scoring system is essentially the standard pairwise score with an affine gap penalty. After an alignment is completed, gap symbols are replaced with a neural X character. The rule allows pairwise sequence alignments to be used to guide the alignment of sequences to groups or groups to groups.

A problem with the Feng-Doolittle approach is that all alignments are determined by pairwise sequence alignments. Once an aligned group has been built up, it is advantageous to use position-specific information from the group's multiple alignment to align a new sequence to it. When we use SP score, and linear gap scoring, profile alignment is relative simple

Let us set

$$s(-,a) = s(a,-) = -g \text{ and } s(-,-) = 0.$$

Assume we have two multiple alignments (or profiles), one containing sequence 1 to n, and the other containing sequence n+1 to N. An alignment of these two profiles means that gaps are inserted in whole columns, so the alignment within one of the profiles is not changed. The score of the global alignment is then

$$\sum_i S(m_i) = \sum_i \sum_{k<l\leq N} s(m_i^k, m_i^l)$$
$$= \sum_i \sum_{k<l\leq n} s(m_i^k, m_i^l) + \sum_i \sum_{n<k<l\leq N} s(m_i^k, m_i^l) + \sum_i \sum_{k\leq n, n<l\leq N} s(m_i^k, m_i^l).$$

All we did was to split up the sum into two sums only concerning the two profiles and one sum containing all the cross terms. The first two sums are unaffected by the global alignment, because adding columns of gap characters to a profile adds zero to the score (0).

Therefore the optimal alignment of the two profiles can be obtained by only optimizing the last sum with the cross terms. This can be done exactly like a standard pairwise alignment, where columns are scored against columns by adding the pair scores. When one of the profiles consists of a single sequence only, it corresponds to aligning a single sequence to a profile. s(−,−) = One widely used implementation of profile-based progressive multiple alignment is the CLUSTALW program.

Problems:

- All alignments are completely determined by pairwise sequence alignment (restricted search space)
- No backtracking (subalignment is "frozen")
    - No way to correct an early mistake
    - Non-optimality: Mismatches and gaps at highly conserved region should be penalized more, but we can't tell where is a highly conserved region early in the process

Profile Alignment

- Aligning two alignments/profiles
- Treat each alignment as "frozen"
- Alignment them with a possible "column gap"

Iterative Refinement

- Re-assigning a sequence to a different cluster/profile
- Repeatedly do this for a fixed number of times or until the score converges
- Essentially to enlarge the search space