Course Notes, Spring 2009

1 Graphs

A graph G is a collection of vertices and edges. We normally will write this as $G = (V, E)$, where V is the vertex set and E is the edge set. We denote the edge between x and y either as $\{x, y\}$ or (x, y) (both notations are used – you can do whichever you like).

Lots of terminology is used when referring to graphs. Here's a partial list. Assume you have a graph $G = (V, E)$.

- A vertex v is *adjacent* to vertex y if $(v, y) \in E$.
- A vertex v is *incident* with an edge e if $e = (v, w)$ (i.e., if v is one of the endpoints of e).
- A *self-loop* is an edge (v, v).
- A *simple graph* has no self-loops.
- A *path* is a sequence of vertices, $v_1, v_2, \ldots, v_k$, so that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$. The length of the path is the number of edges in the path.
- A *simple path* is a path in which no vertex appears twice.
- A *cycle* is a path in which the beginning and end vertex are the same.
- A *simple cycle* is a cycle in which no vertex appears twice (except the beginning and end vertex).
- A graph is *connected* if there is a path between every two vertices.
- The *components* of a graph are the maximal connected subgraphs.
- The *degree* of a vertex v is the number of edges incident on v.
- A *clique* in a graph is a subset of vertices that are all pairwise adjacent.
- For $V_0 \subseteq V$, the subgraph of G induced by V is $G_0 = (V_0, E_0)$, where $E_0 = \{(x, y) : x \in V_0, y \in V_0, (x, y) \in E\}$. We denote this induced subgraph by $G|V_0$ ("G restricted to V_0").
- An *independent set* in a graph G is a subset $V_0 \subseteq V$ such that $G|V_0$ has no edges.
- A graph is said to be a *complete graph* if it is a clique.

2 Algorithms

What we are interested in is designing algorithms to solve problems. Problems are defined by *input* and *output*. Typical types of problems include optimization problems, but there are also problems that are decision problems (YES/NO). The following are examples of computational problems that you might pose or encounter:

1.
   - Input: Array A[1...n] of n integers and value B
   - Output: YES if there exists a j so that $A[j] = B$
2.
   - Input: graph $G = (V, E)$
   - Question: Does there exist a cycle of the graph that uses every vertex exactly once?
3.
   - Input: graph $G = (V, E)$
   - Question: does there exist a cycle in the graph that uses every edge exactly once?
4.
   - Input: array A[1...n] of n integers
   - Output: index j so that $A[j]$ is the maximum entry in A.
5.
   - Input: array $A[1...n]$ of n integers
   - Output: array $B[1...n]$ consisting of the same elements of A and satisfying $B_1 \leq B_2 \leq \ldots \leq B_n$.
6.
   - Input: array $A[1...n]$ of n integers
   - Output: k, the number of distinct entries in $A[1...n]$
7.
   - Input: Connected graph $G = (V, E)$ with positive weights on the edges, and two vertices v and w
   - Output: the length of the shortest path between v and w.

The main thing that we are interested in (for this class) is whether we can solve these problems in polynomial time. What do we mean by "polynomial time"? More generally, what do we mean by this?

3 Running time analyses

A "running time analysis" of an algorithm is a statement about how many operations it will use, in a *worst case scenario*, for an input of "size n". What is the size of an input? What is a worst case scenario? And what is counted, in the operations?

For operations, we'll count everything: I/O, assignments to variables, arithmetic operations (including incrementing variables), comparisons, and any other "basic" steps that you might consider. The size of an input is the amount of

storage you need to express the input in a reasonable fashion. So a graph with n vertices and m edges needs $O(n^2)$ space if you use an adjacency matrix, but only $O(n+m)$ space if you use an adjacency list.

Oops. What do I mean by $O(n^2)$ and $O(n+m)$? (By the way, $O(n^2)$ is pronounced "big-oh of n squared".)

A function $f(n)$ is $O(g(n))$ if there is a positive value C and some N, for all $n > N$, $f(n) \leq Cg(n)$. For an example, consider $f(n) = 7n$ and $g(n) = 3n^2$. Is $f(n)$ "big-oh" of $g(n)$? Yes, since setting $C = 3$ and $N = 1$ proves the inequality. What about $f(n) = 7n + 100$ and $g(n) = 3n^2$?

Try the next set of pairs:

1. $f(n) = 3n - 100$ and $g(n) = 5n + n^2$
2. $f(n) = 5n + n^2$ and $g(n) = 3n - 100$
3. $f(n) = n^n$ and $g(n) = 3^n$
4. $f(n) = 3^n$ and $g(n) = n^n$
5. $f(n) = n \log n$ and $g(n) = n$
6. $f(n) = n \log n$ and $g(n) = n^2$
7. $f(n) = n^{1.5}$ and $g(n) = n \log n$
8. $f(n) = n!$ and $g(n) = n^5$

Back to running time analyses, an algorithm has a running time that is $O(f(n))$ if there is a constant C such that for all large enough n, on all inputs of size n, the running time is never larger than $c \cdot f(n)$.

Some algorithms are said to be "'polynomial time", which means that their running times are big-oh of a polynomial. Other algorithms are "exponential time". For the purposes of most of the course, we'll focus on designing algorithms which are polynomial time (when we can), and not care too much (initially) about the polynomial. But some problems don't seem to allow for polynomial time algorithms! (We'll talk about those, too.)

4 Computational problems and how hard they are

In a similar fashion, some problems are said to be "polynomial time" because it is possible to design an algorithm which runs in polynomial time for the problem. For example, the following are all polynomial time:

1. Given two arrays, determine if they are identical.
2. Given two arrays, determine if they have the same entries (though perhaps in different orders). You need to consider the number of times each entry appears!

3. Given two arrays, determine the entries that appear in both arrays.

4. Given a graph, determine if it has a cycle that uses every edge exactly once.

5. Given a graph G with positive edge weights, and given two vertices in the graph, find the "shortest path" between the two vertices. (Here you define the length of the path as the sum of the edge weights.)

6. Given a tree T with DNA sequences of length k labelling each leaf, find additional sequences for the internal nodes, all of length k, so that you minimize the sum of the Hamming distances between the endpoints of each edge. (This is the Maximum Parsimony problem on a fixed tree.)

7. Given two DNA sequences S and T, and a cost for each insertion, deletion, and substitution, find a minimum cost transformation of S into T. (This is the Edit Distance problem, which is equivalent to the Pairwise Alignment problem.)

You may notice that although some of these problems are obviously easily seen as polynomial time... not all of them are. Sometimes it takes insight and skill to design an algorithm to solve a problem in polynomial time.

You may know some sorting algorithms. What are their running times? How do you analyze this?

- Selection sort
- Merge sort
- Bubble sort
- Quick sort (randomized algorithm with better performance on average than some other algorithms)

5 NP-hard problems

NP-hardness is a technical term, and we'll discuss this in class (later). For now, however, you should just realize that whenever a problem is said to be "NP-hard" it means you should *not* try to design a polynomial time algorithm for it. Instead, let yourself use exponential time (probably this is needed!), and consider *exhaustive search* methods.

To analyze the running time of an exhaustive search strategy, you will need to be able to do "combinatorial counting". We'll work on that in this class as well.

Examples of NP-hard problems include:

1. 3-colorability: given a graph $G = (V, E)$, determine if we can assign colors red, blue, and green to the vertices so that no edge connects vertices of different colors.

2. Hamiltonian cycle: given a graph $G = (V, E)$, determine if it has a cycle which uses every vertex exactly once.

3. Travelling salesman: given the complete graph $G = (V, E)$ with edge weights, determine the shortest Hamiltonial cycle in the graph.

4. Satisfiability: given a logical formula, determine if there is an assignment of truth values to the variables which makes it true.

5. DNA Maximum Parsimony: given a collection of DNA sequences for a set of species, find the evolutionary tree for the species with the total minimum number of mutations (this requires assigning sequences to the internal nodes as well).

6. Max clique: given a graph $G = (V, E)$, find the largest clique in the graph.

Sometimes you'll be satisfied with an approximate answer to an optimization problem. In such cases, it may be possible to design a polynomial time algorithm which produces a solution which is approximately correct. We'll discuss approximation algorithms and their guarantees, as well.

More often, however, you will need to develop a *heuristic*: something that will produce a good solution in many cases, but maybe never a provably correct solution! How do we evaluate these algorithms?

6 Research opportunities in this class

Computational biology research projects are available for students who can program (especially if you know any scripting language, like Perl or Python). My own research involves designing and studying new heuristics for phylogeny estimation or multiple sequence alignment. Please see me early in the semester if this interests you!