COMP 271: Design and Analysis of Algorithms[1]
Fall 2002

Lecture 1: Introduction and Example

Read: Course syllabus. Review Chapters 1–4 in CLR.

Why study algorithm design? Our text defines an *algorithm* to be any well-defined computational procedure that takes some values as *input* and produces some result as *output*. Like a recipe, an algorithm provides a step by step method for solving the problem.

A good understanding of algorithms is essential for a good understanding of the most basic element of computer science: *programming*. Unlike a program, an algorithm is a mathematical entity, that is independent of a specific programming language, machine, or compiler. In some sense, this course is all about the mathematical theory behind the design of good programs.

There are many facets to good program design. Good algorithm design is one of them (and an important one). To be really complete algorithm designer, it is important to be aware of programming and machine issues as well. Although we will emphasize the mathematics in this course, you should always maintain a skeptical sense about whether you can turn these theoretically efficient algorithms in actually efficient programs. For example, a theoretically efficient program that has poor paging performance, may run much slower than a theoretically inefficient program that has better paging performance. (Paging is an issue from operating systems that arises when virtual memory is used.)

However, before you start worrying about compiler, machine, or operating systems issues, you should start first with a good understanding of the mathematical aspect of your problem.

My favorite example of the importance of understanding good algorithm design arose from a conversation I had with a programmer with many years of experience who was working on an optimization algorithm for scheduling the flow of oil in a large network of pipelines in a oil-refinery. After working on the problem for months, the best solution he had was based on a brute-force search of the entire solution space, along with a number of clever heuristics to speed-up the computation locally. It was taking his program about 1 month of dedicated time on a powerful workstation to arrive at the optimal solution. A quick mathematical analysis revealed that, as a function of input size, his program was taking around $n!$ time.

He was asking me whether we could improve the heuristics that he used in the search. I suggested that first, he consider whether we might consider redesigning the entire program from scratch, with an eye to improving the basic algorithm.

After discussing his problem for an hour or so, I had discovered that the basic component of the algorithm required computing shortest paths in networks according to some complex cost criterion. However, shortest paths are one of the standard algorithms discussed in this course. On a network of size n, it can be solved in time that is linear in n. After reorganizing his approach, and employing Dijkstra's algorithm for computing shortest paths, he reported to me that he could now solve the same problem on the same workstation in about 1 minute. From 1 month to 1 minute is a speed-up by a factor of about 40,000. The resulting program was much shorter and much simpler than the monster that he had been trying to "tweak" for absolute efficiency.

An Example–Max-Dominance: For the next couple of lectures, we will introduce the principal elements of algorithm analysis through an example problem. To motivate the problem, suppose that you are planning on making purchasing plans for storage systems for a manufacturer of large-scale computer systems. You study the data produced by various manufacturers of storage systems, and you find that these systems are graded from two perspectives: capacity and speed. Obviously you want to purchase storage systems that are high in both capacity and speed, but you find that there are tradeoffs: the fastest systems often have lower capacity, the highest capacity systems often have lower speed. However, you know that you are definitely NOT interested in a system if its strictly *dominated* by another system which is bigger in both capacity and speed. Assuming that you know both the capacity and speed of each system, which systems are candidates for consideration?

This application may be too small to warrant writing a program (after all, how many different storage systems are in the world). But there are many applications where this type of dominance pruning is a common initial step before applying a very expensive selection criterion. In these applications, there may be thousands of objects to be considered (and there may be many dimensions, not just 2).

The first step before designing an algorithm is to make sure we have a clear statement of the problem. Let a point p in 2-dimensional space be given by its coordinates, $p = (p.x, p.y)$. A point q is said to *dominate* point p if $p.x \leq q.x$ and $p.y \leq q.y$. Given a set of n points, $P = \{p_1, p_2, \ldots, p_n\}$, a point is said to be *maximal* if it is dominated by no other point in P.

Max-Dominance Problem: Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$, each represented by its x and y coordinates, output the set of the maximal points of P, that is, those points p_i, such that p_i is not dominated by any other point of P.

Observe that our description of the problem so far has been at a fairly mathematical level. For example, we have intentionally not discussed issues as to how points are represented (e.g., using a structure with records for the x and y coordinates, or a 2-dimensional array) or the input and output formats. These would normally be important in a software specification. However, we would like to keep as many of the issues as vague as possible, unless they significantly affect the efficiency of the algorithm.

Brute Force Algorithm: To get the ball rolling, let's just consider a simple brute-force algorithm, with no thought to efficiency. Here is the simplest one that I can imagine. For each point p_i, test it against all other points p_j, $j \neq i$. If p_i is not dominated by any other point, then output it.

Brute Force

```
Boolean Dom(q, p) {                              // does q dominate p?
    return (p.x <= q.x) && (p.y <= q.y);
}

Max-Dom(P, n) {
    for i = 1 to n {
        is_max = TRUE;
        for j = 1 to n {
            if ((i != j) && Dom(P[j], P[i])) {
                is_max = FALSE;
                break;
            }
        }
        if (is_max) output P[i];
```

```
    }
}
```

There are no formal rules to the syntax of this pseudocode. In particular, do not assume that more detail is better. When writing pseudocode, you should omit details that tend to detract from the main ideas of the algorithm. How much detail is needed? As long as a competent programmer could, without difficulty or serious ambiguity, implement your algorithm as a problem in any standard programming language, then the description is sufficient.

Correctness: Normally, an algorithm requires a proof of its correctness. In this course these proofs will usually not be highly formal. The goal of correctness proofs in this class will be to convince a reasonably intelligent but skeptical person that the algorithm does what it claims. In this case, the correctness proof is really trivial, because the algorithm just implements the definition. If your algorithm made use of some clever trick, your correctness proof would have to explain why the trick works.

Analysis: The main purpose of our mathematical analyses will be be measure the execution time (and sometimes the space) of an algorithm. Obviously the running time of the algorithm is a function of many different things. To avoid dependencies on compiler, machine, or operating system, we will usually count some quantity that is reflective of the algorithm's running time. For example, in this problem we might count the number of steps of pseudocode that are executed.

Running time depends on input size. So we will define running time in terms of a function of input size. For this problem the input size is the number of points, n.

Different inputs of the same size may generally result in different execution times. (For example, in this problem, the number of times we execute the inner loop before breaking out depends not only on the size of the input, but the structure of the input.) There are two common criteria used in measuring running times:

Worst-case time: What is the maximum running time over all inputs of size n? Let $T(I)$ denote the running time on input I, and let $|I|$ denote the size of input I. Then the worst-case running time is given by:

$$W(n) = \max_{|I|=n} T(I).$$

The subscript "$|I| = n$" means that the maximum is taken over all (legal) inputs I whose size is n.

Average-case time: What is the average running time over all inputs of size n? More generally, for each input I, let $p(I)$ denote the probability of seeing this input. The average-case running time is the weight sum of running times, with the probability being the weight.

$$A(n) = \sum_{|I|=n} p(I)T(I).$$

We will usually work with worst-case running time because for many of the problems we will work with, average-case running time is difficult to compute, and it is difficult to specify probability distributions on inputs that are really meaningful for all applications. It turns out that for most of the algorithms we will consider, there will be little difference between worst-case and average-case times.

Lecture 2: Algorithm Analysis Background

Read: Review Chapters 1–4 in CLR.

Max-Dominance: Last time we introduced the max-dominance problem. Recall that a point $q = (q.x, q.y)$ dominates a point $p = (p.x, p.y)$ if $p.x \leq q.x$ and $p.y \leq q.y$.

Max-Dominance Problem: Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$, each represented by its x and y coordinates, output the set of the maximal points of P, that is, those points p_i, such that p_i is not dominated by any other point of P.

Last time we introduced a simple brute force algorithm that simply checked each point against every other point for dominance. Since there are n^2 point-to-point comparisons, the running time was on the order of n^2, or $O(n^2)$.

Understand the problem: The first step before approaching the solution of any algorithmic problem is to develop a good intuitive understanding of the mathematical aspects problem. In this case, let's start by considering an example in the figure below.

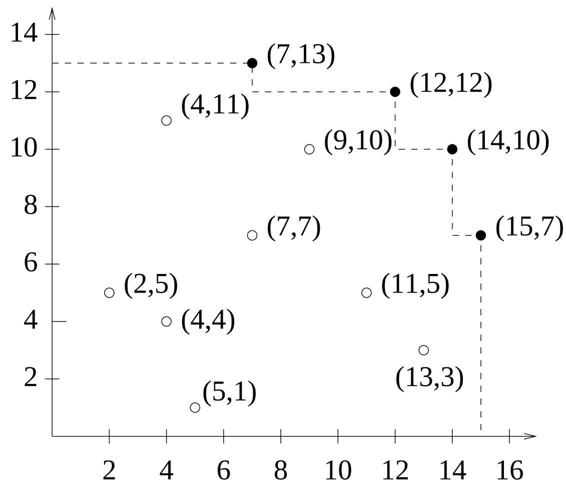


Figure 1: Maximal Points.

A couple of questions might occur to us. For example, could all of the points be maximal? (Yes, if the points lie on a line of slope -1, for example.) Must at least one point be maximal? (Yes, the point with the maximum x or maximum y coordinate must be maximal.) Can two maximal points have the same x- and y-coordinates? (No, for example, if they shared the same x-coordinate, then the one with the smaller y-coordinate could not be maximal.) If we output maximal points in ascending order with respect to x, then must the y coordinates be descending with respect to y? (Yes, because if both x and y were to increase, then the first point would not be maximal.)

A slightly smarter algorithm: Let's consider a slightly smarter approach to the problem. (However, this is not the best algorithm yet.) They key to much of good algorithm design is to avoid work that doesn't need to be done. One of the problems with the brute-force algorithm is that it compares each point against every other point. How can we reduce the work? Well, rather than testing whether a point is dominated by all other points, it suffices to test whether it is dominated just by points with larger x-coordinates. (Since if the x-coordinate is smaller, then the point cannot possibly dominate us.) Of course, we could also say the same symmetrically about the y-coordinate, but we will have to chose one or the other.

So this suggests the following approach. First sort the points in ascending order by x-coordinate. Then for each point, check it for dominance against all points that appear later in the list. Since sorting is a well-known problem, we will not give the sorting algorithm, but simply assume that the implementor choses an efficient sorting algorithm.

First Improvement

```
Max-Dom(P, n) {
    Sort P in ascending order by x-coordinate;
    for i = 1 to n {
        j = i+1;
        while (j <= n && (P[i].y > P[j].y)) j++;
        if (j > n) output P[i];
    }
}
```

Is this correct? It seems obvious. After all, as we mentioned before, if a point appears earlier in the sorted list then its x-coordinate is smaller and it cannot dominate a later point. But there is a subtlety that should be considered. What happens if two points have the same x-coordinate? The one with the larger y-coordinate will dominate the other. For this reason, it is important that our sorting algorithm break ties carefully. If two points have the same x-coordinate, then they should be sorted by y-coordinate. This is called *lexicographical order*. We assume that there are no duplicate points (although a good implementation should check for this).

What is the running time of this procedure? First, the sorting procedure takes $O(n \log n)$ time to run. This is a well-known fact from CMSC 251. The time is dominated by the inner while loop. The number of times the while loop is executed depends on how many points are visited before we find one that dominates the current point. An average-case analysis might be quite tricky, since it would depend on the underlying point distribution.

However, a worst-case analysis is quite a bit easier. In the worst-case, we stay in the while loop as long as possible. What does this mean? It means that every point is maximal (and we know that this is possible). So, if we count the number of iterations of the inner while loop under the assumption that it goes to the end of the list every time, how many iterations do we have? This can be expressed as a summation. For a given value of i, the inner loop is executed for $j = i+1, i+2, \ldots, n$. The simple rule to remember is to take the difference of the final value and the initial value and add one, thus the total is $n - (i+1) + 1 = n - i$. This is true for i running from 1 through n, so we have a total worst-case running time (ignoring constant factors) of:

$$T(n) = n \log n + [(n-1) + (n-2) + (n-3) + \ldots + 2 + 1].$$

Summations: Let's ignore the $n \log n$ term for now, and concentrate on the sum. In summation notation this is:

$$\sum_{i=1}^{n-1} i.$$

This is a well-known standard summation, called an *arithmetic series*[2] (see CLR page 43–45 for others). The solution is:

$$\frac{n(n-1)}{2}.$$

[2]The story goes that this is the famous summation that Gauss solved as a child. When a teacher wanted to keep his students busy for a while, he told them to add up all the numbers from 1 to 100. Gauss came back with the answer in a few minutes to the surprise of the teacher. The trick is to pair-up groups of large a small numbers, to

Solving summations is generally a combination of dirty-work and creativity. The dirty-work consists of memorizing standard summations (or knowing of good sources to find them). Our text has quite a few. Standard texts by Knuth and Purdom and Brown also have many. A second approach is to observe that any summation can be approximated by an integral. In particular, if f is an increasing function:

$$\int_{m-1}^{n} f(x)dx \leq \sum_{i=m}^{n} f(i) \leq \int_{m}^{n+1} f(x)dx$$

There are many math reference books with tables of integrals to help you solve these. Another trick to solving summations is to reorder the terms in an order that is easier to see the answer or to rearrange the terms get cancellations, or finally to bound complex functions with simpler functions to get bounds. A method of last resort is to work out a number of terms, and see whether you can see a pattern. If so, try to prove your answer by induction.

Asymptotics: Getting back to our analysis, the final running time of the algorithm is

$$T(n) = n \log n + \frac{n(n-1)}{2}.$$

We would like to simplify this, by extracting the fastest growing term. This term will point us to the bottleneck in performance as n becomes large.

Asymptotics is subject of CMSC 251. Here are the intuitive definitions. See CLR, Chapter 3 for the formalities.

$f(n) = O(g(n))$: $f(n)$ grows asymptotically no faster than $g(n)$. For example, $3n^2 = O(n^2)$, $4n^2 = O(n^4)$.

$f(n) = \Omega(g(n))$: $f(n)$ grows no slower than $g(n)$. For example, $n \log n = \Omega(n)$.

$f(n) = \Theta(g(n))$: $f(n)$ grows asymptotically at the same rate as $g(n)$. For example, $n^2 + n \log n = \Theta(n^2)$.

The simple rule that I have always applied (which works well in most situations) is the ratio rule. Given two functions $f(n)$ and $g(n)$, consider the limiting ratio:

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}.$$

Then, if this limit exists (and it almost always does) then

$$\begin{aligned} L = 0 \quad &\Rightarrow \quad f(n) = O(g(n)), \\ L = \infty \quad &\Rightarrow \quad f(n) = \Omega(g(n)), \\ L = \text{any nonzero constant} \quad &\Rightarrow \quad f(n) = \Theta(g(n)). \end{aligned}$$

Also recall that if $f(n) = \Theta(g(n))$ then it is automatically true that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. For historical reasons, it is customary to list running times using $O()$ rather than $\Theta()$, unless we really need to emphasize the asymptotic equality.

---

reduce the sum to a product:

$$(100 + 0) + (99 + 1) + (98 + 2) + \ldots + (51 + 49) + 50 \;=\; 50 \cdot 100 + 50 = 5050.$$

In the case of $T(n)$, the first term is growing as $n \log n$, the second term is growing as $n^2/2 - n/2$, which is dominated by $n^2/2$, so it looks as though the overall asymptotic complexity is $\Theta(n^2)$. To verify this we check:

$$\begin{aligned} L &= \lim_{n\to\infty} \frac{T(n)}{n^2} = \lim_{n\to\infty} \frac{n \log n + n(n-1)/2}{n^2} \\ &= \lim_{n\to\infty} \frac{n \log n + n^2/2 - n/2}{n^2} = \lim_{n\to\infty} \left(\frac{\log n}{n} + \frac{1}{2} - \frac{1}{2n} \right) \\ &= \lim_{n\to\infty} (0 + 1/2 - 0) = 1/2. \end{aligned}$$

Thus this is a $\Theta(n^2)$ and hence an $O(n^2)$ algorithm. This is no significant improvement over our brute-force algorithm. (Although there is an improvement in constant factors. In particular, this algorithm makes about half as many comparisons.)

Lecture 3: Divide and Conquer and Recurrences

Read: Review Chapters 1–4 in CLR.

Faster Algorithm for Max-Dominance: Recall the max-dominance problem from the last two lectures. So far we have introduced a simple brute-force algorithm that ran in $O(n^2)$ time, which operated by comparing all pairs of points. Last time we considered a slight improvement, which sorted the points by their x-coordinate, and then compared each point against the subsequent points in the sorted order. However, this improvement, only improved matters by a constant factor. The question we consider today is whether there is an approach that is significantly better.

A Major Improvement: The problem with the previous algorithm is that, even though we have cut the number of comparisons roughly in half, each point is still making lots of comparisons. Can we save time by making only one comparison for each point? The inner while loop is testing to see whether *any* point that follows $P[i]$ in the sorted list has a larger y-coordinate. This suggests, that if we knew which point among $P[i+1, \ldots, n]$ had the maximum y-coordinate, we could just test against that point.

How can we do this? Here is a simple observation. For any set of points, the point with the maximum y-coordinate is the maximal point with the smallest x-coordiante. This suggests that we can sweep the points backwards, from right to left. We keep track of the index j of the most recently seen maximal point. (Initially the rightmost point is maximal.) When we encounter the point $P[i]$, it is maximal if and only if $P[i].y > P[j].y$. This suggests the following algorithm.

Sort and Reverse Scan

```
Max_Dom(P, n) {
    Sort P in ascending order by x-coordinate;
    output P[n];                        // last point is always maximal
    j = n;
    for i = n-1 downto 1 {
        if (P[i].y > P[j].y) {          // is P[i] maximal?
            output P[i];                // yes..output it
            j = i;                      // P[i] has the largest y so far
        }
    }
}
```

The running time of the for-loop is obviously $O(n)$, because there is just a single loop that is executed $n-1$ times, and the code inside takes constant time. The total running time is dominated by the $O(n \log n)$ sorting time, for a total of $O(n \log n)$ time.

How much of an improvement is this? Probably the most accurate way to find out would be to code the two up, and compare their running times. But just to get a feeling, let's look at the ratio of the running times, ignoring constant factors:

$$\frac{n^2}{n \lg n} = \frac{n}{\lg n}.$$

(I use the notation $\lg n$ to denote the logarithm base 2, $\ln n$ to denote the natural logarithm (base e) and $\log n$ when I do not care about the base. Note that a change in base only affects the value of a logarithm function by a constant amount, so inside of O-notation, we will usually just write $\log n$.)

For relatively small values of n (e.g. less than 100), both algorithms are probably running fast enough that the difference will be practically negligible. (Rule 1 of algorithm optimization: Don't optimize code that is already fast enough.) On larger inputs, say, $n = 1,000$, the ratio of n to $\log n$ is about $1000/10 = 100$, so there is a 100-to-1 ratio in running times. Of course, we would need to factor in constant factors, but since we are not using any really complex data structures, it is hard to imagine that the constant factors will differ by more than, say, 10. For even larger inputs, say, $n = 1,000,000$, we are looking at a ratio of roughly $1,000,000/20 = 50,000$. This is quite a significant difference, irrespective of the constant factors.

Divide and Conquer Approach: One problem with the previous algorithm is that it relies on sorting. This is nice and clean (since it is usually easy to get good code for sorting without troubling yourself to write your own). However, if you really wanted to squeeze the most efficiency out of your code, you might consider whether you can solve this problem without invoking a sorting algorithm.

One of the basic maxims of algorithm design is to first approach any problem using one of the standard algorithm design paradigms, e.g. divide and conquer, dynamic programming, greedy algorithms, depth-first search. We will talk more about these methods as the semester continues. For this problem, divide-and-conquer is a natural method to choose. What is this paradigm?

Divide: Divide the problem into two subproblems (ideally of approximately equal sizes),

Conquer: Solve each subproblem recursively, and

Combine: Combine the solutions to the two subproblems into a global solution.

How shall we divide the problem? I can think of a couple of ways. One is similar to how *mergesort* operates. Just take the array of points $P[1..n]$, and split into two subarrays of equal size $P[1..n/2]$ and $P[n/2+1..n]$. Because we do NOT sort the points, there is no particular relationship between the points in one side of the list from the other.

Another approach, which is more reminiscent of *quicksort* is to select a random element from the list, called a *pivot*, $x = P[r]$, where r is a random integer in the range from 1 to n, and then partition the list into two sublists, those elements whose x-coordinates are less than or equal to x and those that greater than x. This will not be guaranteed to split the list into two equal parts, but on average it can be shown that it does a pretty good job.

Let's consider the first method. (The quicksort method will also work, but leads to a tougher analysis.) Here is more concrete outline. We will describe the algorithm at a very high level. The input will be a point array, and a point array will be returned. The key ingredient is a function that takes the maxima of two sets, and merges them into an overall set of maxima.

Divide-and-Conquer

```
Max_Dom(P, n) {
    if (n == 1) return {P[1]};          // one point is trivially maximal
    m = n/2;                            // midpoint of list
    M1 = Max_Dom(P[1..m], m);           // solve for first half
    M2 = Max_Dom(P[m+1..n], n-m);       // solve for second half
    return Max_Merge(M1, M2);           // merge the results
}
```

The general process is illustrated below.

The main question is how the procedure `Max_Merge()` is implemented, because it does all the work. Let us assume that it returns a list of points in *sorted order* according to x-coordinates of the maximal points. Observe that if a point is to be maximal overall, then it must be maximal in one of the two sublists. However, just because a point is maximal in some list, does not imply that it is globally maximal. (Consider point $(7, 10)$ in the example.) However, if it dominates all the points of the other sublist, then we can assert that it is maximal.

I will describe the procedure at a very high level. It operates by walking through each of the two sorted lists of maximal points. It maintains two pointers, one pointing to the next unprocessed item in each list. Think of these as *fingers*. Take the finger pointing to the point with the smaller x-coordinate. If its y-coordinate is larger than the y-coordinate of the point under the other finger, then this point is maximal, and is copied to the next position of the result list. Otherwise it is not copied. In either case, we move to the next point in the same list, and repeat the process. The result list is returned.

The details will be left as an exercise. Observe that because we spend a constant amount of time processing each point (either copying it to the result list or skipping over it) the total execution time of this procedure is $O(n)$.

Recurrences: How do we analyze recursive procedures like this one? If there is a simple pattern to the sizes of the recursive calls, then the best way is usually by setting up a *recurrence*, that is, a function which is defined recursively in terms of itself.

We break the problem into two subproblems of size roughly $n/2$ (we will say exactly $n/2$ for simplicity), and the additional overhead of merging the solutions is $O(n)$. We will ignore constant factors, writing $O(n)$ just as n, giving:

$$\begin{aligned} T(n) &= 1 && \text{if } n = 1, \\ T(n) &= 2T(n/2) + n && \text{if } n > 1. \end{aligned}$$

There are many techniques for solving recurrences. For the sort of recurrences that arise often in divide-and-conquer algorithms, there is a theorem with the ominous title of the *Master theorem* that solves most of these. It is presented on page 62 of our text, and is well worth noting.

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

P = [(9,10)(15,7)(2,14)(5,1)(13,3)(7,13)(14,10)(11,5)(4,11)(7,7)(16,4)(12,12)]

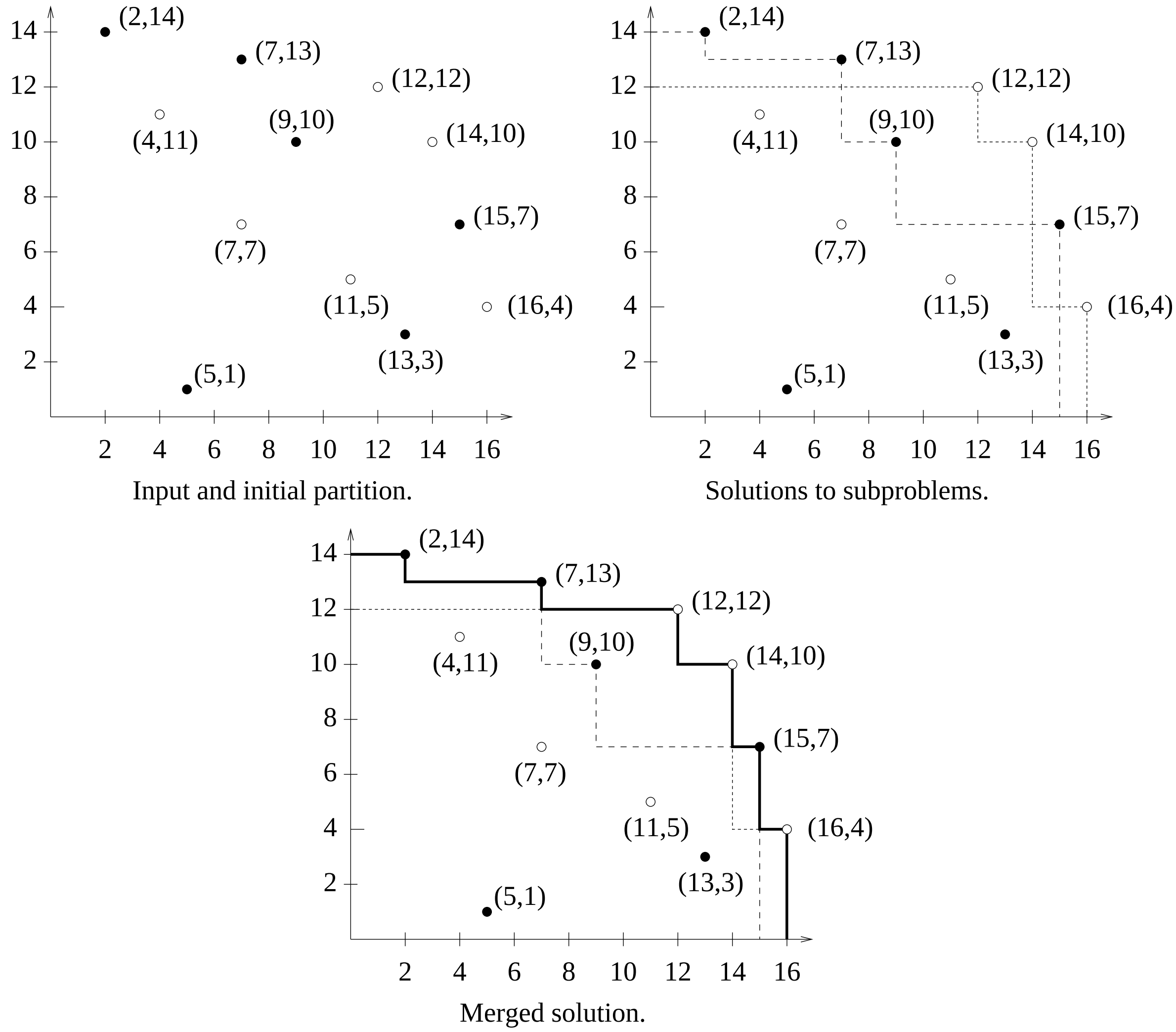


Figure 2: Divide and conquer approach.

where we interpret n/b to be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

(1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(2) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

(3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = \Theta(f(n))$.

In this case, $a = 2$, $b = 2$, $f(n) = n$. The function $n^{\log_b a}$ equals n, and so case (2) applies. Thus $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$.

The statement of the Master theorem can be quite intimidating, but the conditions are easy to establish. It is often easier to solve recurrences by visualizing the recursion tree. For example, in this problem the recursion tree for the problem is a balanced binary tree. At level i of the recursion tree (where the root is level 0) the value of n, that is, the size of the lists being processed is $n/2^i$ and there are 2^i such lists. The number of levels in the tree is the number of times that we can cut n in half before arriving at a list of size 1. This is $\lg n$.

The total work done on any level is equal to the total sizes of all the lists on this level, and hence is $O(n)$. If we add up all the times spent on all $\lg n$ levels, we get a total cost of $O(n \log n)$.

A third method of solving recurrences is by converting the recurrence into a summation. This can be done by a process of repeated expansion, until a pattern emerges.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/4) + n/2) + n \;=\; 4T(n/4) + 2n \\
&= 2(4T(n/4) + n/4) + 2n \;=\; 8T(n/8) + 3n \\
&= 2(8T(n/8) + n/8) + 3n \;=\; 16T(n/16) + 4n \\
&= \ldots \;=\; 2^k T(n/2^k) + kn.
\end{aligned}
$$

Since $T(1) = 1$, we set $k = \lg n$, so that $T(n/2^k) = T(1) = 1$. With this substitution we have

$$
\begin{aligned}
T(n) &= 2^{\lg n} T(1) + (\lg n) n \\
&= n + n \lg n \\
&= O(n \log n).
\end{aligned}
$$

Lecture 4: Issues in Sorting

Read: Review Chapts. 6 and 7; read Chapt. 8.

Review of Sorting: As you know, sorting is the problem of permuting a sequence of numbers (or elements from any total order, e.g. strings) into ascending order. Sorting is among the most fundamental problems in algorithm design. This is largely because sorting is often the first step in some more complex algorithm, as an initial stage in organizing data for faster retrieval subsequently.

You are probably familiar with the standard simple $O(n^2)$ sorting algorithms, such as *insertion sort* and *bubble sort*, and more efficient $O(n \log n)$ sorting algorithms such as *merge sort*, *quicksort*, and *heapsort*.

Without reviewing these algorithms again, here are a few interesting facts about these sorting algorithms.

Quicksort: Widely regarded as the *fastest* of the fast algorithms (because its inner loop compares elements against a single "pivot value" which can be stored in a register for fast access). This algorithm is $O(n \log n)$ in the expected case, and $O(n^2)$ in the worst case. The probability that the algorithm takes asymptotically longer (assuming that the pivot is chosen randomly) is extremely small for large n.

Mergesort: Mergesort is the only *stable* sorting algorithm of these three. This means that if two elements have the same value, then their relative position in the final sorted order is the same after sorting. This is a nice feature, if the list is already sorted according to a secondary key. The downside is the mergesort is the only algorithm of the three that requires additional array storage (ignoring a recursion stack).

Heapsort: Heapsort is based on a nice data structure, called a *heap*, which is a fast priority queue. Elements can be inserted into a heap in $O(\log n)$ time, and the smallest elements can be deleted in $O(\log n)$ time. If you only want to extract the k smallest values, a heap can allow you to do this is $O(n + k \log n)$ time. A heap has the additional advantage of being used in contexts where the priority of elements changes. Each change of priority can be processed in $O(\log n)$ time.

Often times I hear people say that they would rather use mergesort or heapsort, because they do not want to run the risk that quicksort might run in $O(n^2)$ time. To give some concept of just how low the probability is that quicksort will approach its worst-case time bound, consider the probability that quicksort takes $O(n^2)$ time on an input of size 1,000. The algorithm would have to make on the order of 1,000 unbalanced splits, each occurring with probability much less than 1/2, which occurs with probability less than $(1/2)^{1000} \approx 10^{-30}$. This is a unfathomably small quantity. For example, if you assume that every year, a single meteor large enough to smash your workstation strikes the earth at a random spot, then it is more likely that such a meteor will hit your workstation during the second that quicksort is running, than quicksort will take $O(n^2)$ time.

Lower Bounds for Comparison-Based Sorting: Can we sort faster than $O(n \log n)$ time? We will give an argument that, if the sorting algorithm has the property that the manner in which it permutes the elements is based solely on comparisons, then it is impossible to sort more efficiently. Such an algorithm is called a *comparison-based* sorting algorithm.

Virtually all known sorting algorithms are based on making comparisons (although we will see that exceptions exist in special cases). This does not preclude the possibility of sorting algorithms whose actions are determined by other sorts of operations. For example, the following instruction would be illegal in a *comparison-based* sorting algorithm.

```
Swap a[i] with a[(a[i]*2367723) % n];
```

We have "cheated" because the algorithm can perform some action that depends on the *contents* of an element, rather than a comparison. Recently people have discovered that if you could perform these sorts of operations, and you had virtually unlimited word size, then you could sort faster than $O(n \log n)$ time. Unfortunately, there have not been any *practical* algorithms discovered that sort faster than $O(n \log n)$ time, except in the special cases we will discuss later.

We want to show that any *comparison based* sorting algorithm for a input sequence $\langle a_1, a_2, \ldots, a_n \rangle$ must make at least $\Omega(n \log n)$ comparisons in the worst-case. This is really a difficult task if you think about it, because our proof cannot take advantage of any knowledge about the programming language being used, or the machine being used, or how the algorithm goes about deciding which elements to compare. The only fact that we can exploit is that the algorithm's actions are determined by the result of its comparisons.

Decision Tree Argument: In order to prove lower bounds, we need an abstract way of modeling "any possible" sorting algorithm (since we cannot imagine what sort of creativity an algorithm designer of the future may employ). Any comparison-based sorting algorithm can be viewed abstractly as a *decision tree.*

By our assumption, once we know the result of every comparison made by a sorting algorithm, the "action" of the algorithm is completely determined, meaning that we can infer the manner in which data items will be permuted for the final output. Each node of the decision tree represents a comparison made in the algorithm (e.g. $a_4 \leq a_7$), and the two branches represent "true" and "false". When we come to a leaf (i.e. the algorithm has made all the comparisons it needs) the manner in which the algorithm manipulates the sequence for the final output is fully determined. Given any sorting algorithm (e.g. mergesort), and given n, it is a fairly easy (but tedious) exercise to present a decision tree for the algorithm.

Let $T(n)$ denote the worst case running time of some sorting algorithm. Then for any given input of size n, it can make at most $T(n)$ comparisons (maybe fewer). This defines a decision tree whose height is at most $T(n)$. Any binary tree of height $T(n)$, has at most $2^{T(n)}$ leaves, implying that the sorting algorithm can *distinguish* among at most $2^{T(n)}$ different final actions. Let's call this quantity $A(n)$, for the number of different final actions the algorithm can take. Each action can be thought of as a specific way of permuting the input to get a sorted output.

How many possible results must any sorting algorithm distinguish between? If the input consists of n distinct numbers, then those numbers could be presented in any of $n!$ different permutations. For each different permutation, the algorithm must "unscramble" the numbers in an essentially different way. This implies that the number of different final results is at least as large as $n!$, that is, $A(n) \geq n!$.

Since $A(n) = 2^{T(n)}$ we get $2^{T(n)} \geq n!$, implying that

$$T(n) \geq \lg(n!).$$

We can use *Stirling's approximation* for $n!$ yielding:

$$T(n) \geq \lg\left(\frac{n}{e}\right)^n \;=\; n\lg n - n\lg e \;=\; \Omega(n\log n).$$

Thus we have, the following theorem.

Theorem: Any comparison-based sorting algorithm has worst-case running time $\Omega(n \log n)$.

Linear Time Sorting: The lower bound tells us that if we hope to sort numbers faster than in $O(n \log n)$ time, we cannot do it by making comparisons alone. Many applications involve sorting small integers (e.g. sorting characters, exam scores, last four digits of a social security number, etc.). We present three algorithms based on the theme of speeding up sorting in special cases, by *not* making comparisons.

Counting Sort: Counting sort assumes that each input is an integer in the range from 1 to k, where $k \in O(n)$. The algorithm sorts in $O(n)$ time, and is very cute.

The basic idea is to determine, for each element in the input, its *rank* in the list, that is, the number of elements that are less than or equal to it. Notice that once you know the rank of every element, you sort by simply copying each element to the appropriate location of the final sorted output array.

We use three arrays:

$A[1..n]$: Holds the initial input.

$B[1..n]$: Holds the sorted output.

$C[1..k]$: $C[i]$ contains the number of elements of A that are less than or equal to i.

The algorithm operates by first constructing C. Once this is done, it takes each element of A and places in its final position in the sorted list. How do we know where it goes? Well, we know that for each j, $C[A[j]]$ is the number of elements that are less than or equal to $A[j]$. Thus in a sorted list, $A[j]$ should be placed in this position. That is, $B[C[A[j]]] = A[j]$. We need to be careful if there are duplicates, since we do not want them to overwrite the same location of B. To do this, we decrement $C[A[j]]$ after copying. (This is tricky, read it carefully!)

Counting Sort

```
Counting_Sort(A[1..n], B[1..n], k) {
    for i = 1 to k do C[i] = 0;
    for j = 1 to n do C[A[j]]++;           // C[i] = no. of (A[j] == i)
    for i = 2 to k do C[i] += C[i-1];      // C[i] = no. of (A[j] <= i)
    for j = n downto 1 do B[C[A[j]]--] = A[j];
}
```

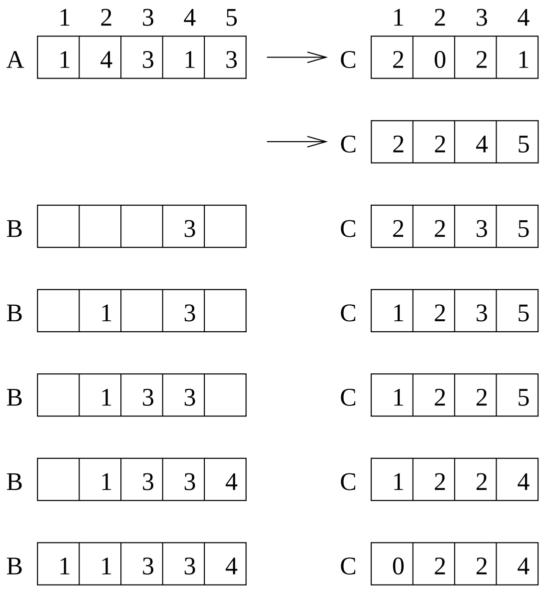


Figure 3: Counting Sort.

Clearly the algorithm runs in $O(n+k)$ time. If $k = O(n)$, then the total running time is $O(n)$.

Radix Sort: The main shortcoming of counting sort is that it is only really (due to space requirements) for small integers. If the integers are in the range from 1 to 1 million, we may not want to allocate an array of a million elements. Radix sort provides a nice way around this by sorting numbers one digit at a time.

The idea is very simple. Let's think of our list as being composed of n numbers, each having d decimal digits (or digits in any base). Let's suppose that we have access to a stable sorting algorithm, like `Counting_Sort`. To sort these numbers we can simply sort repeatedly, starting at the lowest order digit, and finishing with the highest order digit. Since the sorting algorithm is stable, we know that if the numbers are already sorted with respect to low order digits, and then later we sort with respect to high order digits, numbers having the same high order digit will remain sorted with respect to their low order digit.

Radix Sort

```
Radix_Sort(A[1..n], d) {
    for i = 1 to d do {
        Sort A (stably) with respect to i-th lowest order digit;
    }
}
```

Example:

$$\begin{array}{ccccccc} 246 & & 923 & & 923 & & 238 \\ 925 & \Longrightarrow & 925 & \Longrightarrow & 925 & \Longrightarrow & 246 \\ 238 & & 246 & & 238 & & 923 \\ 923 & & 238 & & 246 & & 925 \end{array}$$

The running time is clearly $O(d(n+k))$ where d is the number of digits, n is the length of the list, and k is the number of values a digit can have (which we assume to be $O(n)$).

Lecture 5: More Issues in Sorting

Read: Chapts 8 and 9.

Bucket Sort: Last time we studied some algorithms for sorting in $O(n)$ time, assuming that the numbers to be sorted are small integers. What if you are sorting items that are numeric values but not integers? It is still possible to achieve $O(n)$ expected time, if the numbers are uniformly distributed, and in particular, if the numbers do not bunch up into small clusters.

Here is the idea. We first determine the maximum and minimum of the numbers to sort. Let us assume for simplicity, that the range is the interval $[0, 1)$. (In general, if the numbers are in some other range, we can just scale them appropriately.)

Then we construct an array with n different entries indexed from 0 to $n-1$. Each element of this array is a pointer to the head of a linked list. Initially the lists are all empty. For each number $A[i]$ we insert it in the $\lfloor n \cdot A[i] \rfloor$-th list. Since $A[i]$ is in the range $[0, 1)$, $n \cdot A[i]$ is in the range from $[0, n)$, and so the floor is a number from 0 to $n-1$. Then we sort the numbers in each of the linked lists using *any* sorting algorithm, even a simple $O(n^2)$ algorithm. Finally we concatenate them into a final list.

Bucket Sort

```
Bucket_Sort(A[1..n])
{
    for i = 1 to n
        insert A[i] into B[floor(n*A[i])];
    for i = 0 to n-1
        sort list B[i] using insertion sort;
    return concatenation of B[0], B[1], ..., B[n-1];
}
```

What is the running time of bucket-sort? If the elements are uniformly distributed over the interval $[0, 1)$, then each list will have, in the expected case, 1 element.

More formally, let X_i denote the random variable which denotes the number of elements assigned to the i-th bucket. Because the elements are assumed to be uniformly distributed, each element has an equal probability of going into any bucket, or in particular, it has a

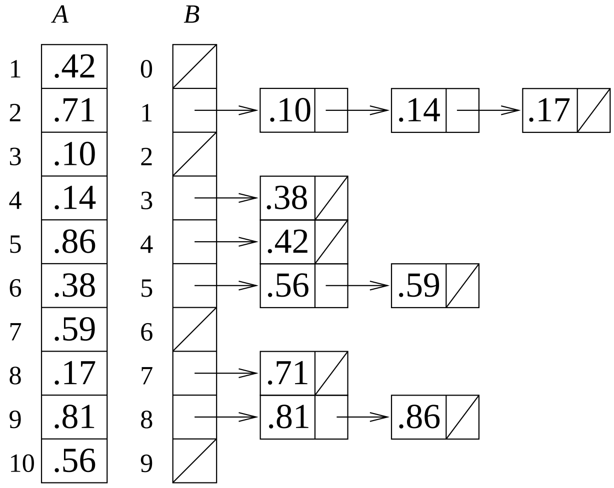


Figure 4: Bucket sort.

probability of $p = 1/n$ of going into the i-th bucket. The total number of items in the i-th bucket can be analyzed, by thinking of each element as a coin flip. With probability $p = 1/n$ the coin comes up heads, meaning that the number goes into this bucket. With probability $1 - 1/n$ the coin comes up tails, meaning that the item goes into another bucket. X_i is just the total number of heads.

This means that the variables X_i are all equally distributed with a *Binomial distribution*. In particular, it is equal to the number of heads after n flips, with a probability of $1/n$ of getting heads. From standard results on Binomial distribution, it follows that the expected number of heads, the expected bucket size is 1. In fact, the expected value of X_i^2 (this is the expected running time of an $O(n^2)$ sorting algorithm on each bucket) is also $O(1)$. (See the text for details.)

Thus, the total expected running time is $O(n)$, the time needed to run n sorts on each list, and the time to concatenate all the lists.

Selection: Consider the following problem. Given a sequence of numbers $a_1, a_2, \ldots, a_n$, and an integer i, $1 \le i \le n$, compute the i-th smallest element. This is called the *selection* problem. The *median* of a set of numbers is a special case of this problem, where $n = \lceil n/2 \rceil$.

Selection sounds like a very mundane problem, but because it is such a basic question, it is remarkable how many places that the problem arises in practice. Here is an example. In statistics, a very common problem is that of finding a good line of fit for a set of points in the plane. The most well-known method is called a *least squares fit.* For any line in the plane, you give it a *score* of how well it fits the points. The score of a line is defined to be the sum of squares of the distances of the points from the line. Thus if point p_i is distance d_i from the line, then this point constributes d_i^2 to the score. Among all possible lines in the plane, you take the line that has the lowest score. This is the *least squares estimator.*

But there is a problem with this method. It assumes that the errors small and evenly distributed on either side of the line of fit. In many applications (e.g. in computer vision) there are *outlying* data points that can confuse the estimator. Because errors are squared, even a relatively small number of outlying points can bias the results of the estimator.

An alternative approach which is more *robust* to outliers is called *LMS*, for the least *median* of squared errors. In this approach, the score of a line is not the *sum* of squared errors, but

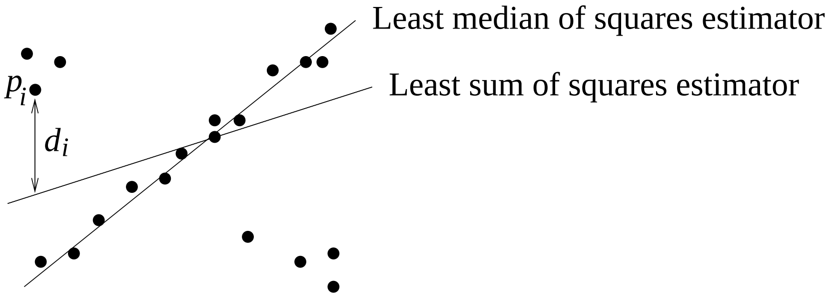


Figure 5: Least squares and LMS.

the *median* of the squared errors. This is much less affected by outliers. If a single point is very far off, it contributes a massive squared error. But a single point, no matter how far away cannot alter the median. If there are fewer than $n/2$ outlying points, then the median will not be negatively effected by the presence of outliers.

The LMS estimator is commonly approximated by trying many pairs of data points, for each computing the line passing through these points. For each line we compute its score, but first computing the distances from each point to the line, and taking the median of all these distances. The line with smallest score wins. This requires computing lots of medians, and so requires a fast median finding algorithm.

Randomized Selection: There is an obvious $O(n \log n)$ algorithm for this problem, by sorting the numbers into ascending order, and then returning the i-th element of the sorted array. The question is whether we can do better than this.

The answer is yes. There is an $O(n)$ time algorithm for this problem. Unfortunately, this algorithm is rather complicated, and not really practical. However, there is a simple $O(n)$ *expected-case* randomized algorithm. To say that the algorithm is *randomized* means that it has access to a random number generator. The algorithm always produces the correct answer, and, with (very) high probability, it runs in $O(n)$ time.

The idea is based on a modification of QuickSort. We assume the existence of a procedure, called `Partition` which is given a subarray $A[p..r]$. This procedure randomly selects an element x from this subarray, called the *pivot*. (In the example below, we will assume that it is the first element of the array. However, it is safer to generate the middle element, or even better, a random element of the array, since if the list is already sorted, selecting the first element can result in unbalanced splits.) It then rearranges the elements of the subarray into two subarrays, $A[p..q-1]$ contains the elements less than or equal to x, $A[q] = x$, and $A[q+1..r]$ contains the elements greater than or equal to x. (We will not give this algorithm. See Sect. 7.1. in the text.) The algorithm is so constructed that these two sublists will both be nonempty.

Next the algorithm determines the rank of the element x, $k = q - p + 1$. If $i = k$ we return the element $A[q]$. If $i < k$, then we recurse on the left sublist. If $i > k$, then we recurse on the right sublist, but we look for the $(i-k)$-th smallest element (because we have eliminated the k elements from consideration.

Randomized Select

```
procedure Select(A[p..r], i)
{
    if (p == r) return(A[p]);
    q = Randomized_Partition(A[p..r]);
    k = q - p + 1;
```

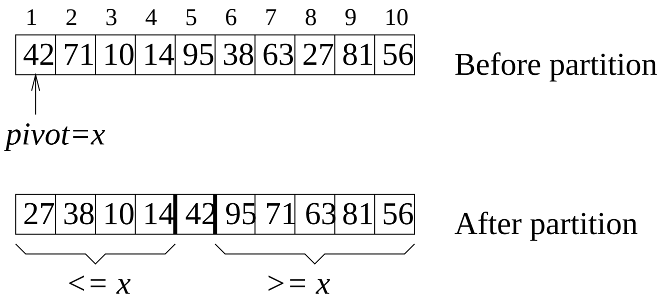


Figure 6: Partitioning.

```
    if (i == k) return(A[q]);
    if (i < k) return Select(A[p..q-1], i);
    else       return Select(A[q+1..r], i-k);
}
```

What is the running time of this algorithm? Because it is randomized it is most reasonable to determine its *expected* time complexity. Let's let $T(n)$ denote the expected case running time of the algorithm on a list of size n. In general we do not know what i is, so let's make the worst-case assumption, that whenever we partition the list, the i-th smallest element occurs on the LARGER side. (This will certainly give an upper bound on the expected case performance, no matter what i is.) We assume the running time of partition on a list of size n is $O(n)$.

There are n possible choices for the pivot element. Each is equally likely, with probability $1/n$. If x is the k-th smallest element of the list, then we create two sublists of sizes $k-1$ and $n-k$. (The partitioning algorithm given in the text has a slight bias, in that it is slightly more likely to create the split into 1 and $n-1$. To simplify things we will ignore this.) If we assume that we recurse on the larger of the two lists, we get the following recurrence.

$$T(n) \leq \frac{1}{n}\left(\sum_{k=1}^{n} T(\max(k-1, n-k))\right) + n.$$

The recurrence is rather messy, but it is worked out in detail in the text book. Basically, the recurrence is simplified to the form

$$T(n) \leq \frac{2}{n}\left(\sum_{k=n/2}^{n-1} T(k)\right) + n.$$

and then an induction argument is used to show that $T(n) \leq cn$ for some appropriately chosen constant c.

Deterministic Selection: Since we know there is a simple randomized algorithm for this problem, it is natural to ask whether there is a deterministic version of the algorithm. Basically, the randomized algorithm generates a random pivot element, and its running time will be good as long as this element is not too close to either extreme.

This raises the following question: Suppose you are given an unsorted list of n points and you wish to compute *any* element, so long as this element is not among the $n/20$ smallest elements or among the $n/20$ largest elements. Only 10% of the elements are forbidden, so it should be easy to generate one of the other 90% of elements. (Selecting a random pivot will succeed 90% of the time.) You have $O(n)$ time to do this in.

However this problem seems to be remarkably hard to do deterministically. Here is the best known deterministic approach.

(1) Divide the elements into roughly $n/5$ groups, each of size 5.

(2) Compute the median of each group (by any method you like).

(3) Compute the median of these $n/5$ group medians.

It is not too hard to show that the resulting element will not be among the $n/4$ smallest or $n/4$ largest elements. However, how do you implement step (3)? The reason we are doing this is to compute medians in the first place. The answer, is that you call the deterministic selection algorithm recursively. Since the list is of smaller size ($n/5$) the process will eventually terminate. However, proving that this is the case is a tricky process. We will not give the proof, but it is given in the text. By the way, why groups of size 5? It turns out that you want an odd number, to make median computation more symmetric. It also turns out that 3 does not work, but every larger odd number does work. So 5 is the simplest choice.

Lecture 6: Graphs and their Representation

Read: Sections 22.1 in CLR.

Graph Algorithms: For the next few weeks, we will be discussing algorithms on *graphs*, both undirected and directed graphs. Intuitively, a graph is a collection of vertices or nodes, connected by a collection of edges. Graphs are extremely important because they are a very flexible mathematical model for many application problems. Basically, any time you have a set of objects, and there is some "connection" or "relationship" or "interaction" between pairs of objects, a graph is a good way to model this. Examples of graphs in application include *communication* and *transportation networks*, *VLSI* and other sorts of *logic circuits*, *surface meshes* used for shape description in computer-aided design and geographic information systems, *precedence constraints* in scheduling systems. The list of application is almost too long to even consider enumerating it.

Most of the problems in computational graph theory that we will consider arise because they are of importance to one or more of these application areas. Furthermore, many of these problems form the basic building blocks from which more complex algorithms are then built.

Graphs and Digraphs: A *directed graph* (or digraph) $G = (V, E)$ consists of a finite set of *vertices* V (also called *nodes*) and E is a binary relation on V (i.e. a set of *ordered* pairs from V) called the *edges*.

For example, the figure below (left) shows a directed graph. Observe that *self-loops* are allowed by this definition. Some definitions of graphs disallow this. Multiple edges are not permitted (although the edges (v, w) and (w, v) are distinct). This shows the graph $G = (V, E)$ where $V = \{1, 2, 3\}$ and $E = \{(1, 1), (1, 2), (2, 3), (3, 2), (1, 3)\}$.

In an *undirected graph* (or just *graph*) $G = (V, E)$ the edge set consists of unordered pairs of distinct vertices (thus self-loops are not allowed). The figure above (right) shows the graph $G = (V, E)$, where $V = \{1, 2, 3, 4\}$ and the edge set consists of $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\}$.

We say that vertex w is *adjacent* to vertex v if there is an edge from v to w. In an undirected graph, we say that an edge is *incident* on a vertex if the vertex is an endpoint of the edge. In a directed graph we will often say that an edge either *leaves* or *enters* a vertex.

A digraph or undirected graph is said to be *weighted* if its edges are labelled with numeric weights. The meaning of the weight is dependent on the application, e.g. distance between vertices or flow capacity through the edge.

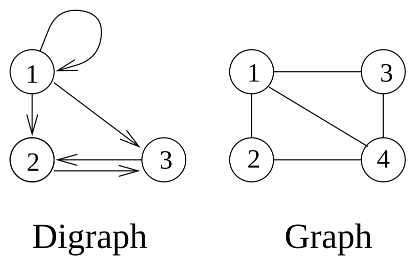


Figure 7: Digraph and graph example.

Observe that directed graphs and undirected graphs are different (but similar) objects mathematically. Certain notions (such as path) are defined for both, but other notions (such as connectivity) are only defined for one.

In a digraph, the number of edges coming out of a vertex is called the *out-degree* of that vertex, and the number of edges coming in is called the *in-degree*. In an undirected graph we just talk about the *degree* of a vertex, as the number of edges which are *incident* on this vertex. By the *degree* of a graph, we usually mean the maximum degree of its vertices.

In a directed graph, each edge contributes 1 to the in-degree of a vertex and contributes one to the out-degree of a vertex, and thus we have

Observation: For a digraph $G = (V, E)$,

$$\sum_{v \in V} \textit{in-deg}(v) = \sum_{v \in V} \textit{out-deg}(v) = |E|.$$

($|E|$ means the cardinality of the set E, i.e. the number of edges).

In an undirected graph each edge contributes once to the degree of two different vertices and thus we have

Observation: For an undirected graph $G = (V, E)$,

$$\sum_{v \in V} \textit{deg}(v) = 2|E|.$$

When referring to graphs and digraphs we will always let $n = |V|$ and $e = |E|$. Our textbook usually uses the abuse of notation $V = |V|$ and $E = |E|$. Beware, the sometimes V is a set, and sometimes it is a number. Some authors use $m = |E|$.

Because the running time of an algorithm will depend on the size of the graph, it is important to know how n and e relate to one another.

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

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, running time of $O(n^2)$ is often unacceptably large for real-time response.

Representations of Graphs and Digraphs: We will describe two ways of representing digraphs. We can represent undirected graphs using exactly the same representation, but we will double each edge, 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.

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.

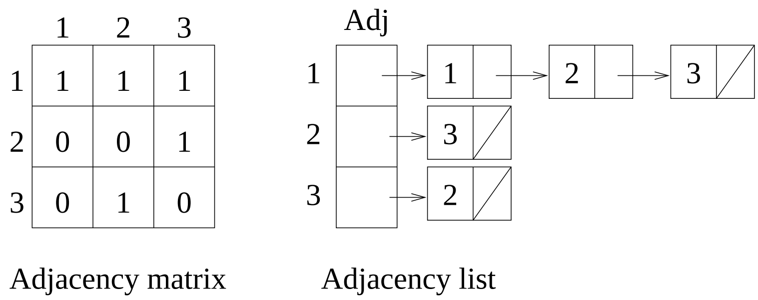


Figure 8: Adjacency matrix and adjacency list.

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 *out-deg*(v) entries, which when summed is $\Theta(e)$. For sparse graphs the adjacency list representation is more cost effective.

Adjacency matrices allow faster access to edge queries (e.g. is $(u, v) \in E$) and adjacency lists allow faster access to enumeration tasks (e.g. find all the vertices adjacent to v).

Paths and Cycles: A *path* in a directed graph is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ such that (v_{i-1}, v_i) is an edge for $i = 1, 2, \ldots, k$. The *length* of the path is the number of edges, k. We say that w is *reachable* from u if there is a path from u to w. A path is *simple* if all vertices are distinct.

(Q: What is the maximum number of simple paths of length k in a digraph? A: $n(n-1)(n-2)\ldots(n-k) = O(n^{k+1})$. This grows very rapidly with k, for example when $k = n-1$ this is $n!$. Thus, algorithms based on a brute-force enumeration of paths are rarely practical.)

A *cycle* is a path containing at least one edge and for which $v_0 = v_k$. A cycle is *simple* if, in addition, $v_1, \ldots, v_k$ are distinct. (Note: A self-loop counts as a simple cycle of length 1). A digraph is *acyclic* if it contains no cycles. A digraph is *strongly connected* if for any pair of vertices u and v, v is reachable from u (and vice versa).

In an undirected graph we define path and cycle exactly the same, but for a *simple cycle* we ADD the requirement that the cycle visit at least three vertices (to rule out the degenerate cycle $\langle u, w, u \rangle$. An undirected graph is said to be *acyclic* if it has no SIMPLE cycles. An undirected graph is *connected* if every vertex can reach every other vertex. In general, every disconnected graph is formed from some number of *connected components*.

A *forest* is an acyclic undirected graph. A *(free) tree* is a connected, acyclic undirected graph. A directed graph that is acyclic is called a *DAG*, for *directed acyclic graph*.

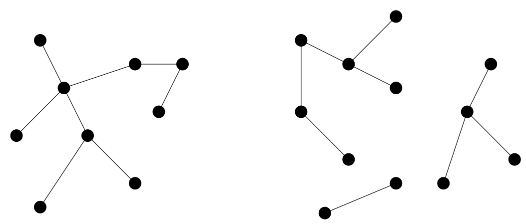


Figure 9: Free trees and forests.

Observe that a free tree is a minimally connected graph in the sense that the removal of any causes the resulting graph to be disconnected. Furthermore, there is a unique path between any two vertices in a free tree. The addition of any edge to a free tree results in the creation of a single cycle.

Subgraphs and special graphs: A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a subset $V' \subseteq V$, the subgraph *induced* by V' is the graph $G' = (V', E')$ where

$$E' = \{(u, v) \in E \mid u, v \in V'\}.$$

In other words, take all the edges of G that join pairs of vertices in V'.

An undirected graph that has the maximum possible number of edges is called a *complete graph*. Complete graphs are often denoted with the letter K. For example, K_5 is the complete graph on 5 vertices. Given a graph G, a subset of vertices $V' \subseteq V$ is said to form a *clique* if the subgraph induced by V' is complete. In other words, all the vertices of V' are adjacent to one another. A subset of vertices V' forms an *independent set* if the subgraph induced by V' has no edges.

The *complement* of a graph $G = (V, E)$, often denoted $\bar{G}$ is a graph on the same vertex set, but in which the edge set has been complemented. The *reversal* of a directed graph, often denoted G^R is a graph on the same vertex set in which all the edge directions have been reversed. This may also be called the *transpose* and denoted G^T.

Here is an interesting fact about graphs and their complements.

Observation: Given any undirected graph G, if G is not connected then $\bar{G}$ is connected.

Proof: Since G is not connected, consider two vertices a and b in different connected components. For every other vertex v, v cannot be adjacent to BOTH a and b. In the complement $\bar{G}$, a and b are adjacent to each other and every other vertex v is adjacent to at least one of a or b (possibly both).

Now, consider any two vertices u and v in $\bar{G}$. Either both are adjacent a (in which case there is a path between them), or both are adjacent to b (again, there is a path between them) or one is adjacent to a and the other to b (in which case there is a path from one to a then to b then to the other.

Lecture 7: Breadth-First Search

Read: Section 22.2 in CLR.

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* $s \in V$. Recall that the *length* of a path in a graph 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 answer will be encoded 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 $\pi[s] = \text{NIL}$.

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 will work. 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 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 (where the length of a path = number of edges on the path). 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. When a gray vertex is processed, then it becomes black.

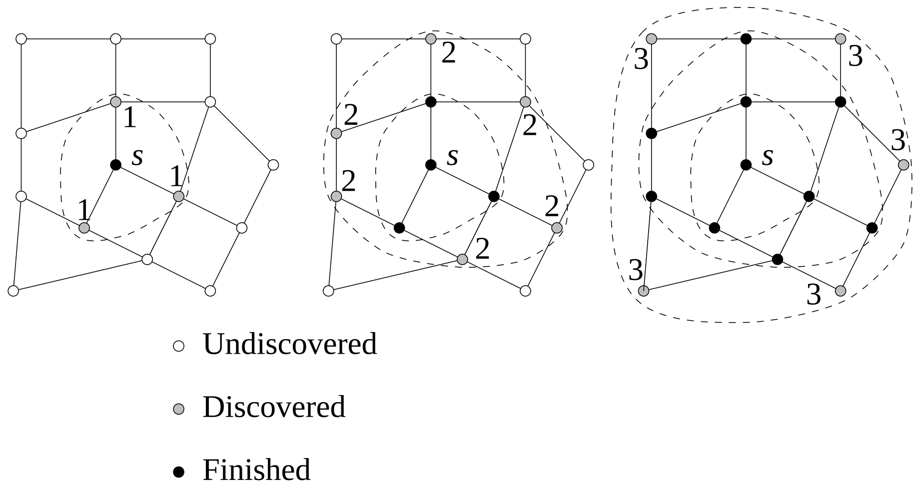


Figure 10: Breadth-first search: Conceptual picture.

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 $dist[u]$, the distance from s to u. Only the color is really needed for the search, but the others are useful depending on the application.

Breadth-First Search

```
BFS(G,s) {
    for each u in V - {s} {
        color[u] = white;
        d[u]     = infinity;
        pred[u]  = NIL;
    }
    color[s] = gray;
    d[s] = 0;
    Q = {s};
    while (Q is nonempty) {
        u = Dequeue[Q];
        for each v in Adj[u] {
            if (color[v] == white) {
                color[v] = gray;
                d[v]     = d[u]+1;
                pred[v]  = u;
                Enqueue(Q, v);
            }
        }
        color[u] = black;
    }
}
```

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

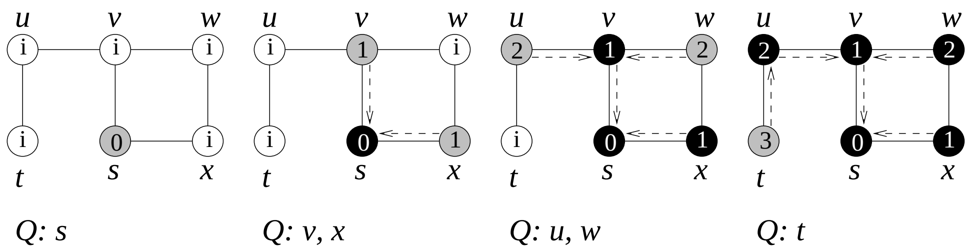


Figure 11: Breadth-first search: Example.

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.)

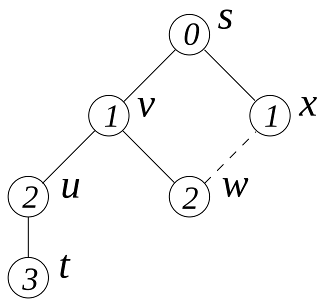


Figure 12: BFS tree.

Analysis: The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. We will do it in detail here. 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

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

For an directed graph the analysis is essentially the same.

Lecture 8: Depth-First Search and Topological Sort

Read: Sections 22.3 and 22.4 in CLR.

Depth-First Search: Breadth-first search is an example of a *graph traversal* algorithm. The job of any graph traversal is to visit the vertices of the graph, according to some strategy. In BFS, the traversal strategy is according to increasing levels of distance from the starting vertex. But there are other natural traversal strategies.

One of the major difficulties in designing algorithms for graphs is there lack of structure. For example, in sorting we saw that it was very easy to pose the problem in the divide-and-conquer paradigm. However, it is not at all clear how to meaningfully divide a graph into smaller subgraphs. The importance of any graph traversal strategy is its ability to impose some sort of structure on the graph. One nice observation we saw from BFS, is that it allows us to view the graph in terms of a tree (the BFS tree) along with some auxiliary edges (cross edges). Trees are much more structured objects than graphs are. For example, trees break up nicely into subtrees, upon which subproblems can be solved recursively. However, the rub will be with how the nontree edges are handled. The next traversal algorithm that we will study is called *depth-first search*, and it has the nice property that nontree edges have a good deal of mathematical structure.

Consider the problem of searching a castle for treasure. To solve it you might use the following strategy. As you enter a room of the castle, paint some graffiti on the wall to remind yourself that you were already there. Successively travel from room to room as long as you come to a place you haven't already been. When you return to the same room, try a different door leaving the room (assuming it goes somewhere you haven't already been). When all doors have been tried in a given room, then backtrack.

Notice that this algorithm is described recursively. In particular, when you enter a new room, you are beginning a new search. This is the general idea behind depth-first search.

Depth-First Search Algorithm: We assume we are given an undirected graph $G = (V, E)$. The same algorithm works for directed graphs (but the resulting structure imposed on the graph is different).

We use four auxiliary arrays. As before we maintain a color for each vertex: white means *undiscovered*, gray means *discovered* but not finished processing, and black means *finished*. As before we also store predecessor pointers, pointing back to the vertex that discovered a given vertex. We will also associate two numbers with each vertex. These are *time stamps*. When we first discover a vertex u store a counter in $d[u]$ and when we are finished processing a vertex we store a counter in $f[u]$. The purpose of the time stamps will be explained later. (Note: Do not confuse the discovery time $d[v]$ with the distance $d[v]$ from BFS.)

Depth-First Search

```
DFS(G) {
    for each u in V {
        color[u] = white;
        pred[u]  = nil;
    }
    time = 0;
    for each u in V
        if (color[u] == white) DFSVisit(u);
}

DFSVisit(u) {
    color[u] = gray;
    d[u] = ++time;
    for each v in Adj(u) do
        if (color[v] == white) {
            pred[v] = u;
            DFSVisit(v);
        }
```

```
        color[u] = black;
        f[u] = ++time;
    }
```

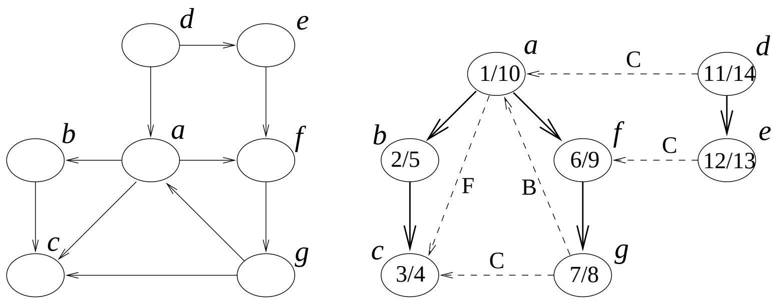


Figure 13: Depth-First Search tree.

Tree structure: DFS imposes a tree (actually a collection of trees, or a forest) on the structure of the graph. Tree edges are those edges $(pred[v], v)$ where DFS calls are made. For directed graphs the other edges of the graph can be classified as follows:

Back edges: (u, v) where v is a (not necessarily proper) ancestor of u in the tree. (Thus, a self-loop is considered to be a back edge).

Forward edges: (u, v) where v is a proper descendent of u in the tree.

Cross edges: (u, v) where u and v are not ancestors or descendents of one another (in fact, the edge may go between different trees of the forest).

It is not difficult to classify the edges of a DFS tree by analyzing the values of colors of the vertices and/or considering the time stamps. This is left as an exercise.

With undirected graphs, there are some important differences in the structure of the DFS tree. First, there is really no distinction between forward and back edges. So, by convention, they are all called back edges. Furthermore, it can be shown that there can be no cross edges. (Can you see why not?)

Time-stamp structure: There is also a nice structure to the time stamps. In CLR this is referred to as the *parenthesis structure*. In particular, the following are easy to observe.

- u is a descendent of v, if and only if $[d[u], f[u]]$ is a subinterval of $[d[v], f[v]]$.
- u is an ancestor of v, if and only if $[d[u], f[u]]$ contains $[d[v], f[v]]$.
- u is unrelated to v, if and only if $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint intervals.

Analysis: The running time of DFS is $\Theta(n + e)$. This is somewhat harder to see than the BFS analysis, because the recursive nature of the algorithm obscures things. Normally, recurrences are good ways to analyze recursively defined algorithms, but it is not true here, because there is no good notion of "size" that we can attach to each recursive call.

However, what saves us is the observation that each vertex is visited exactly once, and hence the call `DFSVisit()` is made exactly once for each vertex. We can just analyze each one individually and add up their running times. Ignoring the time spent in the recursive calls, we can see that each vertex u can be processed in $O(1 + outdeg(u))$ time. Thus the total time used in the procedure is

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

Cycles: DFS allows us to determine a number of things about a graph or digraph. For example, suppose you are given a graph or digraph. You run DFS. You can determine whether the graph contains any cycles very easily. We do this with the help of the following two lemmas.

Lemma: Given a digraph G, in any DFS tree of G, tree edges, forward edges, and cross edges all go from a vertex of higher finish time to a vertex of lower finish time. Back edges go from a vertex of lower finish time to a vertex of higher finish time.

Proof: For tree, forward, and back edges, the proof follows directly from the parenthesis lemma, presented last time. (E.g. for a forward edge (u, v), v is a descendent of u, and so v's start-finish interval is contained within u's, implying that v has an earlier finish time.) For a cross edge (u, v) we know that the two time instervals are disjoint, and since v must have been started before u, it also must be finished before u.

Lemma: A digraph is acyclic if and only if any DFS of G yields no back edges.

Proof: ($\Rightarrow$) We show the contrapositive. If there is a back edge (u, v), then v is an ancestor of u and by following tree edges from v to u we get a cycle.

($\Leftarrow$) Suppose there are no back edges. By the lemma above, each of the remaining types of edges, tree, forward, and cross all have the property that they go from vertices with higher finishing time to vertices with lower finishing time. Thus along any path, finish times decrease monotonically, implying there can be no cycle.

Beware: There is no simple relationship between the number of back edges and the number of cycles. For example, the DFS may only have a single back edge, but there may be an exponential number of simple cycles in the graph.

A similar theorem applies to undirected graphs, but it is much simpler to prove, since an undirected graph is acyclic if and only if it is a free tree, and the DFS of a graph consists only of tree edges and back edges.

Directed Acyclic Graph: A *directed acyclic graph* is often called a DAG for short DAG's arise in many applications where there are precedence or ordering constraints. (E.g. In scheduling problems.) For example, if there are a series of tasks to be performed, and certain tasks must precede other tasks (e.g. in construction you have to build the first floor before you build the second floor, but you can do the electrical wiring while you install the windows). In general a *constraint graph* is a DAG in which vertices are tasks and the edge (u, v) means that task u must be completed before task v begins.

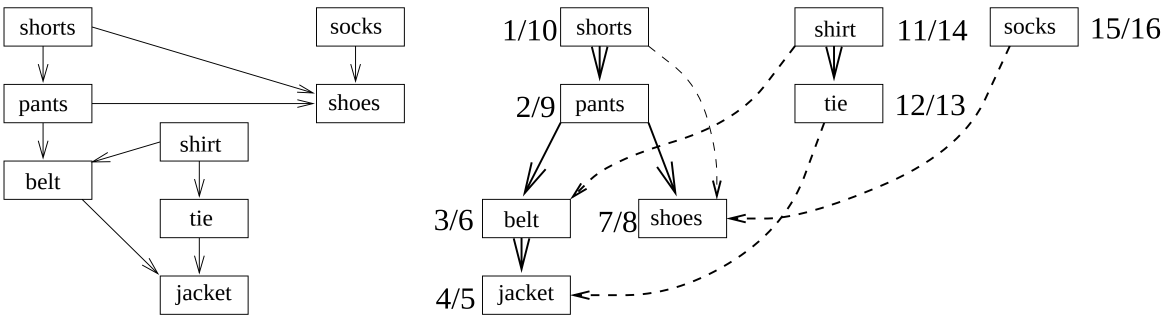
A *topological sort* of a DAG is a linear ordering of the vertices of the DAG such that for each edge (u, v), u appears before v in the ordering. Note that in general, there may be many legal topological orders for a given DAG.

To compute a topological ordering is actually very easy, given DFS. By the previous lemma, for every edge (u, v) in a DAG, the finish time of u is greater than the finish time of v. Thus, it suffices to output the vertices in reverse order of finishing time.

Topological Sort

```
TopSort(G) {
    Run DFS(G), computing finish times f[v] for each vertex v;
    As each vertex is finished, insert it onto the front of a linked list;
    Return the linked list;
}
```

As an example we consider the DAG presented in the text for Professor Bumstead's order of dressing. We do our depth-first search in a different order, and so we get a different final ordering. However both orderings are legitimate, given the precedence constraints.



Final order: socks, shirt, tie, shorts, pants, shoes, belt, jacket

Figure 14: Topological short.

The running time of topological sort is $O(n + e)$.

An interesting related question is how to determine which edges of a DAG are redundant, in the sense that removing these edges does not alter the partial order defined by the DAG.

Lecture X01: Articulation Points

Supplementary
This material is not covered in the text (except as Problem 22–2).

Articulation Points and Biconnected Components: Let $G = (V, E)$ be a **connected** undirected graph. Consider the following definitions.

Articulation Point (or Cut Vertex): Is any vertex whose removal (together with the removal of any incident edges) results in a disconnected graph.

Bridge: Is an edge whose removal results in a disconnected graph.

Biconnected: A graph is biconnected if it contains no articulation points. (In general a graph is k-connected, if k vertices must be removed to disconnect the graph.)

Biconnected graphs and articulation points are of great interest in the design of network algorithms, because these are the "critical" points, whose failure will result in the network becoming disconnected.

We give an algorithm for computing articulation points. An algorithm for computing bridges is simple modification to this procedure.

Articulation Points and DFS: In order to determine the articulation points of an undirected graph, we will call depth-first search, and use the tree structure provided by the search to aid us. In particular, let us ask ourselves if a vertex v is an articulation point, how would we know it by its structure in the DFS tree?

We assume that G is connected (if not, we can apply this algorithm to each individual connected component), so there is only one tree in the DFS forest. Recall that because G is undirected, the DFS tree will have only tree edges and back edges (no cross edges).

For now, let us consider the typical case of a vertex v, where v is not a leaf and v is not the root. Let's let $w_1, w_2, \ldots, w_k$ be the children of v. For each child there is a subtree of the DFS tree rooted at this child. If for some child, there is NO back edge going to a proper ancestor of v, then if we remove v, this subtree becomes disconnected from the rest of the graph, and hence v is an articulation point. On the other hand, if every one of the subtrees rooted at the children of v have back edges to proper ancestors of v, then if v is removed, the graph remains connected (the backedges hold everything together). This leads to the following.

Observation 1: An internal vertex v of the DFS tree (other than the root) is an articulation point if and only there is a subtree rooted at a child of v such that there is NO back edge from any vertex in this subtree to a proper ancestor of v.

What about the leaves? If v is a leaf, can it be an articulation point? Answer: No, because when you delete a leaf from a tree, the rest of the tree remains connected, thus even ignoring the back edges, the graph is connected after the deletion of a leaf from the DFS tree.

Observation 2: A leaf of the DFS tree is never an articulation point. Note that this is completely consistent with Observation 1, since a leaf will not have any subtrees in the DFS tree, so we can delete the word "internal" from Observation 1.

What about the root? Since there are no cross edges between the subtrees of the root if the root has two or more children then it is an articulation point (since its removal separates these two subtrees). On the other hand, if the root has only a single child, then (as in the case of leaves) its removal does not disconnect the DFS tree, and hence cannot disconnect the graph in general.

Observation 3: The root of the DFS is an articulation point if and only if it has two or more children.

Articulation Points by DFS: Observations 1, 2, and 3 provide us with a structural characterization of which vertices in the DFS tree are articulation points. How can we design an algorithm which tests these conditions? Checking that the root has multiple children is an easy exercise. Checking Observation 1 is the hardest, but we will exploit the structure of the DFS tree to help us.

The basic thing we need to check for is whether there is a back edge from some subtree to an ancestor of a given vertex. How can we do this? It would be too expensive to keep track of ALL the back edges from each subtree (because there may be $\Theta(e)$ back eges. A simpler scheme is to keep track of back edge that goes highest in the tree (in the sense of going closest to the root). If any back edge goes to an ancestor of v, this one will.

How do we know how close a back edge goes to the root? As we travel from v towards the root, observe that the discovery times of these ancestors of v get smaller and smaller (the root having the smallest discovery time of 1). So we keep track of the back edge (u, w) which has the SMALLEST value of $d[w]$.

Low: Define $Low[v]$ to be the minimum of $d[v]$ and the set

$$\{d[w] \mid (u, w) \text{is a back edge and } u \text{ is a descendent of } v\}.$$

The term "descendent" is used in the nonstrict sense, that is, u may be equal to v. Intuitively, $Low[v]$ is the highest you can get in the tree by taking any one backedge from either v or any of its descendents. (It is called "Low" not "High" because the discovery numbers are low.)

To compute $Low[v]$ we use the following simple rules: Suppose that we are performing DFS on the vertex v.

Initialization: $Low[v] = d[v]$.

Back edge (v, w): $Low[v] = \min(Low[v], d[w])$. Explanation: We have detected a new back edge coming out of v. If this goes to a lower d value than the previous back edge then make this the new low.

Tree edge (v, w): $Low[v] = \min(Low[v], Low[w])$. Explanation: Since w is in the subtree rooted at v any single back edge leaving the tree rooted at w is a single back edge for the tree rooted at v.

Observe that once $Low[w]$ is computed for all vertices w, we can test whether a given nonroot vertex v is an articulation point by Observation 1 as follows: v is an articulation point if and only if it has a child w in the DFS tree for which $Low[w] \geq d[v]$ (since if there were a back edge from either w or one of its descendents to an ancestor of v then we would have $Low[w] < d[v]$).

Algorithm for *Low*: There is one subtlety that we must watch for in designing the algorithm. When processing a vertex v, we need to know when a given edge (v, w) is a back edge. How do we do this? An almost correct answer is to test whether w is colored gray (since all gray vertices are ancestors of the current vertex). This is not quite correct because w may be the PARENT of v in the DFS tree and we are just seeing the "other half" of the tree edge between w and v (recalling that in constructing the adjacency list of an undirected graph we create two directed edges for each undirected edge). To test correctly for a back edge we use the predecessor pointer to check that w is not the parent of v in the DFS tree.

The complete algorithm for computing $Low[v]$ is as follows. The main procedure for DFS is the same as before, except that it calls the following routine rather than DFSvisit.

Articulation Points

```
ArtPt(v) {
    color[v] = gray;
    Low[v] = d[v] = ++time;
    for all w in Adj(v) do {
        if (color[w] == white) {          // (v,w) is a tree edge
            pred[w] = v;
            ArtPt(w);
            Low[v] = min(Low[v], Low[w]);
        }
        else if (w != pred[v]) {          // (v,w) back edge
            Low[v] = min(Low[v], d[w]);
        }
    }
    color[v] = black;
}
```

To actually compute the articulation points we add the following statement after the call `ArtPt(w)`.

```
    if (pred[v] == NULL) {                // v is the root
        if (this is v's second child) {
            output v;
        }
    }
    else if (Low[w] >= d[v]) {            // v is not the root
        output v;
    }
```

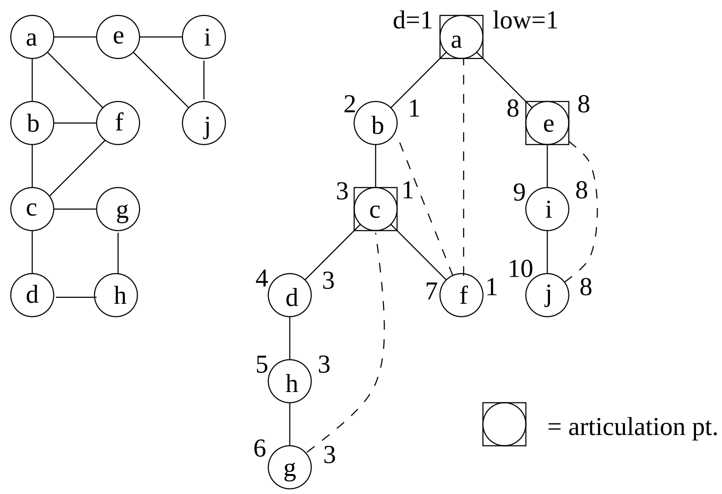


Figure 15: Articulation Points.

Lecture 9: Minimum Spanning Trees

Read: Chapt 23 in CLR.

Minimum Spanning Trees: A common problem in communications networks and circuit design is that of connecting together a set of nodes (communication sites or circuit components) by a network of minimal total length (where length is the sum of the lengths of connecting wires). We assume that the network is undirected. To minimize the length of the connecting network, it never pays to have any cycles (since we could break any cycle without destroying connectivity and decrease the total length). Since the resulting connection graph is connected, undirected, and acyclic, it is a *free tree*.

The computational problem is called the *minimum spanning tree* problem (MST for short). More formally, given an connected, undirected graph $G = (V, E)$, a *spanning tree* is an acyclic subset of edges $T \subseteq E$ that connects all the vertices together. Assuming that each edge (u, v) of G has a numeric weight or cost, $w(u, v)$, (may be zero or negative) we define the cost of a spanning tree T to be the sum of edges in the spanning tree

$$w(T) = \sum_{(u,v)\in T} w(u, v).$$

A *minimum spanning tree* (MST) is a spanning tree of minimum weight. Note that the minimum spanning tree may not be unique, but it is true that if all the edge weights are distinct, then the MST will be distinct (this is a rather subtle fact, which we will not prove). The figure below shows two minimum spanning trees for the same graph.

Steiner Minimum Trees: Before discussing an algorithm for the minimum spanning tree problem, we mention a generalization, called the *Steiner minimum tree* (SMT) problem. As with the MST problem, we are given an undirected graph $G = (V, E)$ with edge weights, and we are also given a subset of vertices $V' \subseteq V$, called the *terminals*. We wish to compute a connected, acyclic subgraph of G (i.e. a free tree) that includes all the terminals. Note that the MST is just a SMT where $V' = V$ (that is, all the vertices must be part of the tree). An example is shown below, where the terminals are shaded.

Although our algorithm for MST's may seem simple and straightforward, before you think that all problems involving spanning trees are easy, you should be aware that the best known

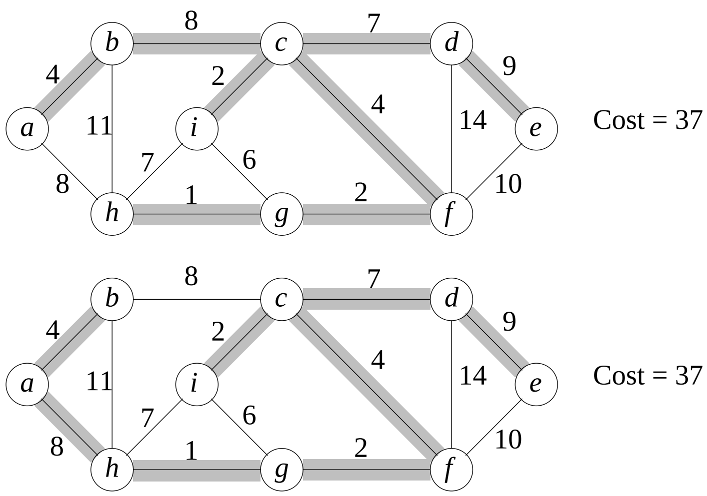


Figure 16: Minimum spanning tree.

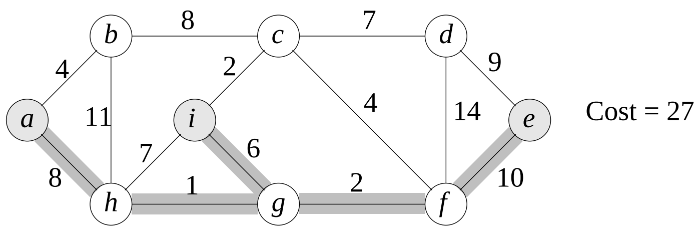


Figure 17: Steiner Minimum tree.

solution to the SMT problem runs in exponential time (and in fact it is an NP-complete problem).

Generic approach: We present two *greedy* algorithm (Kruskal's and Prim's algorithms) for computing a minimum spanning tree. A *greedy algorithm* is one that selects the cheapest of all options at each stage. We recall some basic facts about free trees. They are all quite easy to prove.

Lemma: A free tree with n vertices has exactly $n-1$ edges.

Lemma: There exists a unique path between any two vertices of a free tree.

Lemma: Adding any edge to a free tree creates a unique cycle. Breaking *any* edge on this cycle restores a free tree.

Lemma: Deleting any edge from a free tree breaks the tree into two subtrees.

The intuition behind the greedy MST algorithms is simple, we maintain a set of edges A, which will initially be empty, and we will add edges one at a time, until A equals the MST. A will always be a subset of the MST (thus we will never need to delete an edge, once it is added to A). We say that an edge (u,v) is *safe* if $A \cup \{(u,v)\}$ is still a subset of the MST. Note that in general, A forms a *forest*, since it never contains a cycle, but until we add the last edge, it will not be connected.

A generic greedy algorithm operates by repeatedly adding any *safe* edge to the current spanning tree.

Generic MST Algorithm

```
GenericMST(G, w) {
    A = Empty;
    while (A does not form a spanning tree do) {
        find an edge (u,v) that is safe for A;
        add (u,v) to A;
    }
    return A;
}
```

Lecture 10: Kruskal's Algorithm

Read: Chapt 23 in CLR.

Generic approach: We consider two algorithms for computing minimum spanning trees (Kruskal's and Prim's algorithms). They are both examples of greedy algorithms. A *greedy algorithm* is one that selects the cheapest of all options at each stage. Recall from last time that a free tree on n vertices has $n-1$ edges. The intuition behind each algorithm is to construct a set of edges A that will eventually form the spanning tree. We say that $A \subseteq E$ is *viable* if $A \subseteq T$, for some MST T. Thus a subset of edges is viable if it can be grown to form a complete minimum spanning tree. We want to avoid backtracking, so we will try to add edges in such a way that A is always viable.

Initially A is empty (this is certainly viable). At each step we insert one new edge into A. Certainly this newly inserted edge should have the property that it does not induce a cycle among the current set of edges. But we need more than that. We need to know that if we add this edge to A, then the new set is still viable. Such an edge is called *safe*, because it is safe to add it to the current set of edges.

What is safe? We consider the theoretical issues behind determining whether an edge is safe or not. Let S be a subset of the vertices $S \subseteq V$. A *cut* $(S, V - S)$ is just a partition of the vertices into two disjoint subsets. An edge (u, v) *crosses* the cut if one endpoint is in S and the other is in $V - S$. Given a subset of edges A, we say that a cut *respects* A if no edge in A crosses the cut.

It is not hard to see why respecting cuts are important to this problem. If we have computed a partial MST, and we wish to know which edges can be added that do NOT induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate.

An edge of E is a *light edge* crossing a cut, if among all edges crossing the cut, it has the minimum weight (the light edge may not be unique if there are duplicate edge weights). Intuition says that since all the edges that cross a respecting cut do not induce a cycle, then the lightest edge crossing a cut is a natural choice.

The main theorem which drives both algorithms is the following. It essentially says that we can always augment A by adding the minimum weight edge that crosses a cut which respects A. (It is stated in complete generality, so that it can be applied to both algorithms.)

MST Lemma: Let $G = (V, E)$ be a connected, undirected graph with real-valued weights on the edges. Let A be a viable subset of E (i.e. a subset of some MST), let $(S, V - S)$ be any cut that respects A, and let (u, v) be a light edge crossing this cut. Then the edge (u, v) is *safe* for A.

Proof: It will simplify the proof to assume that all the edge weights are distinct, but this is easy to overcome. Let T be any MST for G. If T contains (u, v) then we are done. Suppose that no MST contains (u, v). We will derive a contradiction.

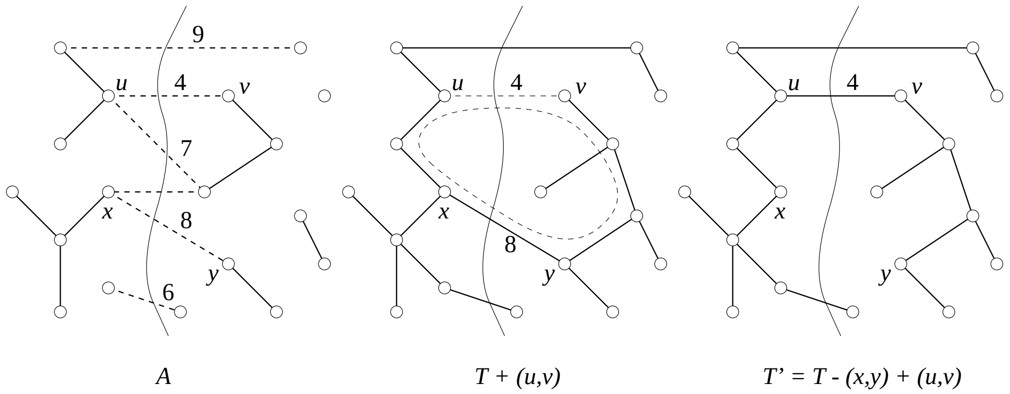


Figure 18: MST Lemma.

Add the edge (u, v) to T, thus creating a cycle. Since u and v are on opposite sides of the cut, and since any cycle must cross the cut an even number of times, there must be at least one other edge (x, y) in T that crosses the cut.

The edge (x, y) is NOT in A (because the cut respects A). By removing (x, y) we restore a spanning tree, call it T'. We have

$$w(T') = w(T) - w(x, y) + w(u, v).$$

Since (u, v) is lightest edge crossing the cut, we have $w(u, v) < w(x, y)$. Thus $w(T') < w(T)$. This contradicts the assumption that T was an MST.

Kruskal's Algorithm: Kruskal's algorithm works by attempting to add edges to the A in increasing order of weight (lightest edges first). If the next edge does not induce a cycle among the current set of edges, then it is added to A. If it does, then this edge is passed over, and we consider the next edge in order. Note that as this algorithm runs, the edges of A will induce a forest on the vertices. As the algorithm continues, the trees of this forest are merged together, until we have a single tree containing all the vertices.

Observe that this strategy leads to a correct algorithm. Why? Consider the edge (u, v) that Kruskal's algorithm seeks to add next, and suppose that this edge does not induce a cycle in A. Let A' denote the tree of the forest A that contains vertex u. Consider the cut $(A', V - A')$. Every edge crossing the cut is not in A', and so this cut respects A, and (u, v) is the light edge across the cut (because any lighter edge would have been considered earlier by the algorithm). Thus, by the MST Lemma, (u, v) is safe.

The only tricky part of the algorithm is how to detect whether the addition of an edge will create a cycle in A. We could perform a DFS on subgraph induced by the edges of A, but this will take too much time. We want a fast test that tells us whether u and v are in the same tree of A.

This can be done by a data structure (which we have not studied) called the disjoint set UNION-FIND data structure. This data structure supports three operations:

Create-Set: Create a set containing a single item.

Find-Set: Find the set that contains a given item, and

Union: Merge two sets into a common set.

You are not responsible for knowing how this data structure works (see Chapt 21 in CLR), but only how it is used as a black-box.

In Kruskal's algorithm, the vertices of the graph will be the elements to be stored in the sets, and the sets will be vertices in each tree of A. The set A can be stored as a simple list of edges.

Kruskal's Algorithm

```
Kruskal(G,w) {
    A = Empty;
    for each v in V do
        Create_Set(v);
    Sort the edges of E in increasing order by weight w;
    for each edge (u,v) in E (in order) {
        if (Find_Set(u) != Find_Set(v)) {
            Add (u,v) to A;
            Union(u, v);
        }
    }
    return A;
}
```

Disjoint Set Union-Find: Let us make a short digression to the question of how the union-find operations can be implemented efficiently. Recall that initially, every elements is in a set of its own. The two operations are (1) `Find_Set(v)` returns the set containing v, (2) `Union(u,v)` forms the union of the sets containing u and v.

In Chapt. 21, CLR presents a very efficient algorithm for this problem. The following algorithm is not as efficient asymptotically, but is easy to implement, and has an interesting analysis.

We will maintain two structures.

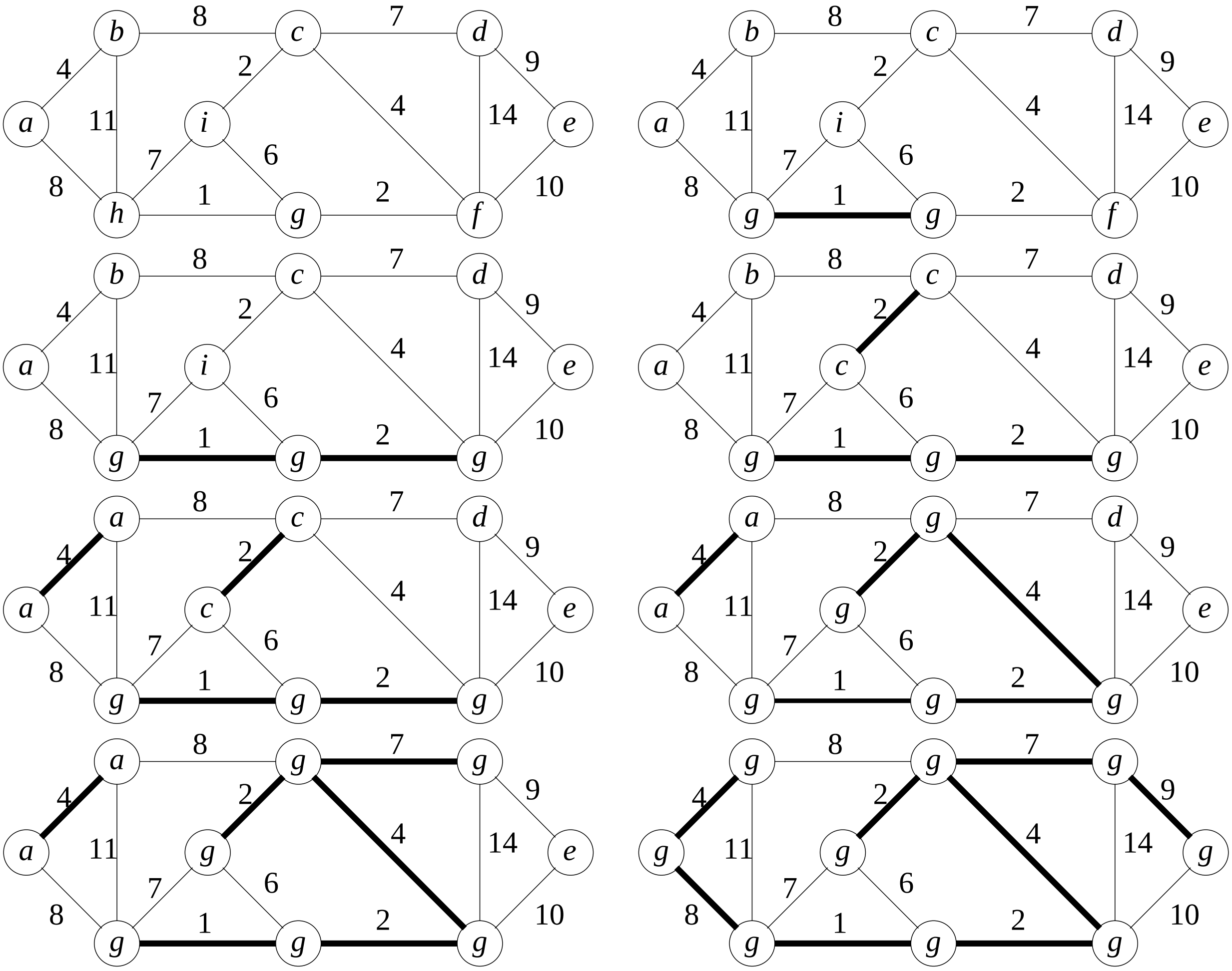


Figure 19: Kruskal's Algorithm.

- For each set, store (1) its *size* (the number of elements in the set), and (2) a *linked-list* of elements in this set.
- For each element, store a *label* indicating which set it belongs to (e.g. a pointer to the set structure described above).

`Find_Set(v)` just returns this label of v in $O(1)$ time. To perform `Union(u,v)`, we find the sets containing u and v (from their labels), merge their two linked lists into a single list (this can be done in $O(1)$ time), set the size of this new list to the sum of the two sizes, and then relabel all the elements of the *smaller* of the lists being merged. Clearly the union operation can be performed in $O(1+k)$ time, where k is the time to relabel the elements of the smaller list. If n is the number of items, we can perform at most $n-1$ unions before everyone is in the same set. The value of k could be as large as $n/2$

This idea of "relabeling the smaller part" is common to many merging/partitioning algorithms. How efficient is it? The real question is what is the *total* number of relabeling operations over the entire course of the algorithm (i.e. when all n elements have been merged into a single set). This is often called an *amortized analysis*, since we are studying the running time of a data structure totaled over a series of operations.

Theorem: Each element can be relabeled at most $\lg n$ times, implying a total of $O(n \log n)$ time spent relabeling.

Proof: Consider a single element v. The first time v is relabeled, it has been unioned into a set whose size is at least 2. The second time that v is relabeled, it must be merged with a set that is at least as large, so the new size must be at least 4. The third time v is relabeled, it must be merged with a set at least as large, so the new size must be at least 8. It is easy to see that if v is relabeled k times total, then its final set is of size 2^k. But since no set can be larger than n, we have $2^k \leq n$ implying that $k \leq \lg n$.

Lecture 11: Prim's Algorithm and Shortest Paths

Read: Chapt 23, 24 in CLR.

Prim's Algorithm: Prim's algorithm is another greedy algorithm for minimum spanning trees. It differs from Kruskal's algorithm only in how it selects the next *safe edge* to add at each step. Its running time is essentially the same as Kruskal's algorithm, $O((n+e)\log n)$. There are two reasons for studying Prim's algorithm. The first is to show that there is more than one way to solve a problem (an important lesson to learn in algorithm design), and the second is that Prim's algorithm looks very much like another greedy algorithm, called Dijkstra's algorithm, that we will study for a completely different problem, shortest paths. Thus, not only is Prim's a different way to solve the same MST problem, it is also the same way to solve a different problem. (Whatever that means!)

Different ways to grow a tree: Kruskal's algorithm worked by ordering the edges, and inserting them one by one into the spanning tree, taking care never to introduce a cycle. Intuitively Kruskal's works by merging or splicing two trees together, until all the vertices are in the same tree.

In contrast, Prim's algorithm builds the tree up by adding leaves one at a time to the current tree. We start with a root vertex r (it can be *any* vertex). At any time, the subset of edges A forms a single tree (in Kruskal's it formed a forest). We look to add a single vertex as a leaf to the tree. The process is illustrated in the following figure.

Observe that if we consider the set of vertices S currently part of the tree, and its complement $(V-S)$, we have a cut of the graph and the current set of tree edges A respects this cut.

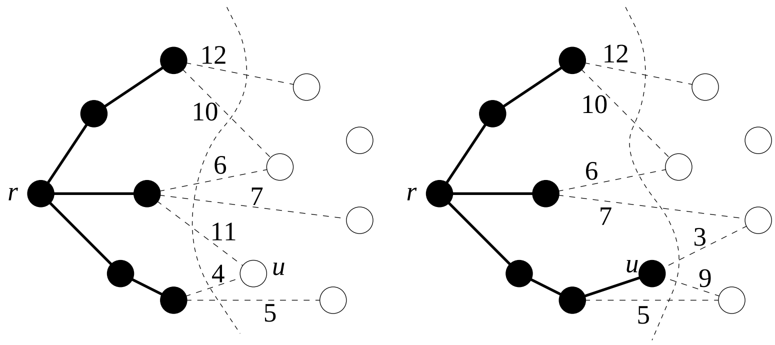


Figure 20: Prim's Algorithm.

Which edge should we add next? The MST Lemma from the previous lecture tells us that it is safe to add the *light edge*. In the figure, this is the edge of weight 4 going to vertex u. Then u is added to the vertices of S, and the cut changes. Note that some edges that crossed the cut before are no longer crossing it, and others that were not crossing the cut are.

It is easy to see, that the key question in the efficient implementation of Prim's algorithm is how to update the cut efficiently, and how to determine the light edge quickly?

To do this, we will make use of a data structure, that is often taught in CMSC 420, a *priority queue*. This is a data struture that stores a set of items, where each item is associated with a *key* value. The priority queue supports three operations.

`Insert(Q, u, key)`: Insert u with the key value *key* in Q.

`u = Extract_Min(Q)`: Extract the item with the minimum key value in Q.

`Decrease_Key(Q, u, new_key)`: Decrease the value of u's key value to *new_key*.

A priority queue can be implemented using the same heap data structure used in heapsort. All of the above operations can be performed in $O(\log n)$ time, where n is the number of items in the heap.

What do we store in the priority queue? At first you might think that we should store the edges that cross the cut, since this is what we are removing with each step of the algorithm. The problem is that when a vertex is moved from one side of the cut to the other, this results in a complicated sequence of updates.

There is a much more elegant solution, and this is what makes Prim's algorithm so nice. For each vertex in $u \in V - S$ (not part of the current spanning tree) we associate u with a key value `key[u]`, which is the weight of the lightest edge going from u to any vertex in S. We also store in `pred[u]` the end vertex of this edge in S. If there is no edge from u to a vertex in S, then we set its key value to $+\infty$. We will also need to know which vertices are in S and which are not. We do this by coloring the vertices in S black.

Here is Prim's algorithm. The root vertex r can be any vertex in V.

Prim's Algorithm

```
Prim(G,w,r) {
    for each u in V do {                  // initialization
        key[u] = +infinity;
        color[u] = white;
    }
    key[r] = 0;                           // start at root
```

```
    pred[r] = nil;
    Q = build initial queue with all vertices;
    while (Non_Empty(Q)) do {               // until all vertices in MST
        u = Extract_Min(Q);                 // vertex with lightest edge
        for each v in Adj[u] do {
            if ((color[v] == white) && (w(u,v) < key[v])) then {
                key[v] = w(u,v);            // new lighter edge out of v
                Decrease_Key(Q, v, key[v]);
                pred[v] = u;
            }
        }
        color[u] = black;
    }
    [The pred pointers define the MST as an inverted tree rooted at r]
}
```

The following figure illustrates Prim's algorithm. The arrows on edges indicate the predecessor pointers, and the numeric label in each vertex is the key value.

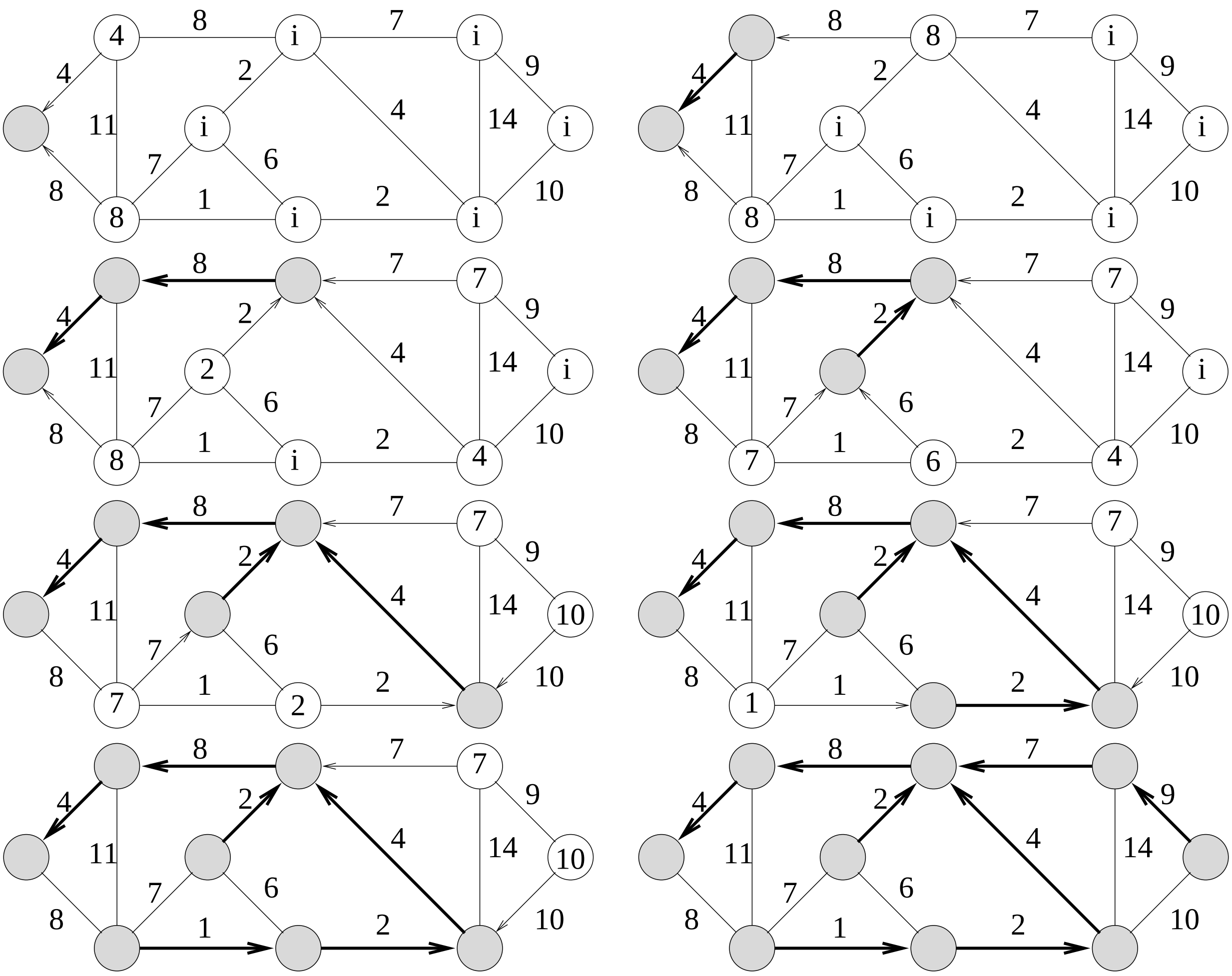


Figure 21: Prim's Algorithm.

To analyze Prim's algorithm, we account for the time spent on each vertex as it is extracted from the priority queue. It takes $O(\log n)$ to extact this vertex from the queue. For each incident edge, we spend potnetially $O(\log n)$ time decreasing the key of the neighboring vertex. Thus the time is $O(\log n + deg(u)\log n)$ time. The other steps of the update are constant time. So the overall running time is

$$\begin{aligned} T(n) &= \sum_{u \in V} (\log n + deg(u)\log n) \;=\; \sum_{u \in V} (1 + deg(u))\log n \\ &= \log n \sum_{u \in V} (1 + deg(u)) \;=\; (\log n)(n + 2e) \;=\; O((n+e)\log n). \end{aligned}$$

Shortest Paths: Let us leave the problem of minimum spanning trees, and consider the problem of computing shortest paths in a directed graph. We have already seen that breadth-first search is an $O(n+e)$ algorithm for finding shortest paths from a single source vertex to all other vertices, assuming that the graph has no edge weights. Suppose that the graph has edge weights, and we wish to compute the shortest paths from a single source vertex to all other vertices in the graph. (By the way, there are other formulations of the shortest path problem. One may want just the shortest path between a pair of vertices. It turns out there is no more efficient algorithm known for this problem, than just starting a single source algorithm from the starting vertex, and stopping when it reaches the destination. There is also a single sink problem, which can be solved in the transpose digraph. Computing all-pairs shortest paths can be solved by iterating a single-source algorithm over all vertices, but there are other global methods that are faster on dense graphs.)

Think of the vertices as cities, and the weights represent the cost of traveling from one city to another (nonexistent edges can be thought of a having infinite cost). When edge weights are present, we define the *length* of a path to be the sum of edge weights along the path. Define the *distance* between two vertices, u and v, $\delta(u,v)$ to be the length of the minimum length path from u to v. ($\delta(u,u) = 0$ by the empty path.)

Given a directed graph with nonnegative edge weights $G = (V, E)$ and a distinguished *source vertex*, $s \in V$, determine the distance from the source vertex to every vertex in the graph.

It is possible to have graphs with negative edges, but in order for the shortest path to be well defined, we need to add the requirement that there be no cycles whose total cost is negative (otherwise you make the path infinitely short by cycling forever through such a cycle). The text discusses the Bellman-Ford algorithm for finding shortest paths assuming negative weight edges but no negative-weight cycles are present. We will discuss a simple algorithm, called Dijkstra's algorithm, which assumes there are no negative edge weights.

We will stress the task of computing the minimum distance from the source to each vertex. Computing the actual path will be a fairly simple extension. As in breadth-first search, for each vertex we will have a pointer $pred[v]$ which points back to the source. By following the predecessor pointers backwards from any vertex, we will construct the reversal of the shortest path to v.

Optimality Structure and Relaxation: The principal observation about shortest paths (which is true for many optimization problems) is that any subpath of a shortest path must also be a shortest path. (Why? If there was a subpath which was NOT a shortest path, then by replacing this subpath with a shorter path, we could get a shorter global path.)

The basic structure of Dijkstra's algorithm is to maintain an *estimate* of the shortest path for each vertex, call this $d[v]$. (NOTE: Don't confuse $d[v]$ with the $d[v]$ in the DFS algorithm. They are completely different.) This value will always greater than or equal to the true shortest path distance from s to v, and we will always require that $d[v]$ equals the cost of some known path (unless $d[v] = \infty$ meaning that there is no path). Initially $d[s] = 0$ and all the other $d[v]$ values are set to ∞. As the algorithm goes on, and sees more and more vertices, it attempts to update $d[v]$ for each vertex in the graph, until all the $d[v]$ values converge to the true shortest distances.

The process by which an estimate is updated is called *relaxation*. Here is how relaxation works. Intuitively, if you can see that your solution is not yet reached an optimum value, then push it a little closer to the optimum. In particular, if you discover a path from s to v shorter than $d[v]$, then you need to update $d[v]$. This notion is common to many optimization algorithms.

Consider an edge from a vertex u to v whose weight is $w(u, v)$. Suppose that we have already computed current estimates on $d[u]$ and $d[v]$. We know that there is a path from s to u of weight $d[u]$. By taking this path and following it with the edge (u, v) we get a path to v of length $d[u]+w(u, v)$. If this path is better than the existing path of length $d[v]$ to v, we should take it. We should also remember that the shortest way back to the source is through u by updating the predecessor pointer.

Relaxing an edge

```
Relax(u,v) {
    if (d[u] + w(u,v)) < d[v] then {
        d[v] = d[u] + w(u,v);
        pred[v] = u;
    }
}
```

Observe that whenever we set $d[v]$ to a finite value, there is always evidence of a path of that length. Therefore $d[v] \geq \delta(s, v)$. If $d[v] = \delta(s, v)$, then further relaxations cannot change its value.

It is not hard to see that if we perform `Relax` repeatedly over all edges of the graph, the $d[v]$ values will eventually converge to the final true distance value from s. The cleverness of any shortest path algorithm is to perform the updates in a judicious manner, so the convergence is as fast as possible. Next time we will see how Dijkstra's algorithm does this.

Lecture 12: Dijkstra's Algorithm

Read: Chapt 24 in CLR.

Shortest Paths: Recall from last time the problem of computing the shortest paths in a digraph starting at a single source vertex. Recall that $\delta(u, v)$ is the distance, that is, the length of the shortest path between u and v. Given a *source vertex* s, we want to compute $\delta(s, u)$ for all vertices u in the digraph. Recall that $\delta(s, s) = 0$. Furthermore, we will encode the shortest paths using an inverted tree rooted at s. The algorithm that we will give is called *Dijkstra's algorithm*.

Recall that we maintain a distance estimate $d[u]$ for each vertex u in the digraph. This estimate has the properties that $d[u] \geq \delta(s, u)$, and that there exists a path of length $d[u]$ from s to u (as long as $d[u] \neq \infty$).

Recall that the main primitive used in computing shortest paths is called *relaxation*. Relaxing the edge (u, v) means setting $d[v] = \min(d[v], d[u] + w(u, v))$. Intuitively, this is because we know there exists a path from s to u of length $d[u]$, and hence adding the edge (u, v) to this, implies that there is a path of length $d[u]+w(u, v)$ to v. The whole question of how to design an efficient shortest path algorithm breaks down to one question: what is the best order in which to perform relaxations, so that the estimates converge most rapidly to the true distances.

Dijkstra's Algorithm: Dijkstra's algorithm, like many shortest path algorithms, is based on the notion of performing repeated relaxations. Dijkstra's algorithm operates by maintaining a subset of vertices, $S \subseteq V$, for which we claim we "know" the true distance, that is $d[v] = \delta(s, v)$. Initially $S =$, the empty set, and we set $d[s] = 0$ and all others to $+\infty$. One by one we select vertices from $V - S$ to add to S.

The set S can be implemented using an array of vertex colors. Initially all vertices are white, and we set `color[v] = black` to indicate that $v \in S$.

How do we select which vertex among the vertices of $V - S$ to add next to S? Here is where *greediness* comes in. For each vertex in $u \in V - S$, we have computed a distance estimate $d[u]$. The greedy thing to do is to take vertex for which $d[u]$ is minimum, that is, take the unprocessed vertex that is closest (by our estimate) to s. Later we will explain why this is the proper choice.

In order to perform this selection efficiently, we store the vertices of $V - S$ in a *priority queue* (e.g. a heap), where the key value of each vertex u is $d[u]$. Note the similarity with Prim's algorithm, although a different key value way used there. Also recall that if we implement the priority queue using a heap, we can perform the operations `Insert()`, `Extract_Min()`, and `Decrease_Key()`, each in $O(\log n)$ time.

Based on this, here is Dijkstra's algorithm. Compare this with Prim's algorithm. The two are remarkably similar, even though they solve completely different problems.

Dijkstra's Algorithm

```
Dijkstra(G,w,s) {
    for each u in V do {                        // initialization
        d[u] = +infinity;
        color[u] = white;
    }
    d[s] = 0;
    pred[s] = nil;
    Q = build initial queue with all vertices;
    while (Non_Empty(Q)) do {                   // until all vertices processed
        u = Extract_Min(Q);                     // select closest to s
        for each v in Adj[u] do {
            if (d[u] + w(u,v)) < d[v] then {    // Relax(u,v)
                d[v] = d[u] + w(u,v);
                Decrease_Key(Q, v, d[v]);
                pred[v] = u;
            }
        }
        color[u] = black;
    }
    [The pred pointers define an inverted shortest path tree]
}
```

Correctness: To see that Dijkstra's algorithm correctly gives the final true distances, we need to show that $d[v] = \delta(s, v)$ when the algorithm terminates. This is a consequence of the following lemma, which states that once a vertex u has been added to S (i.e. colored black), $d[u]$ is the true shortest distance from s to u. Since at the end of the algorithm, all vertices are in S, then all distance estimates are correct.

Lemma: When a vertex u is added to S, $d[u] = \delta(s, u)$.

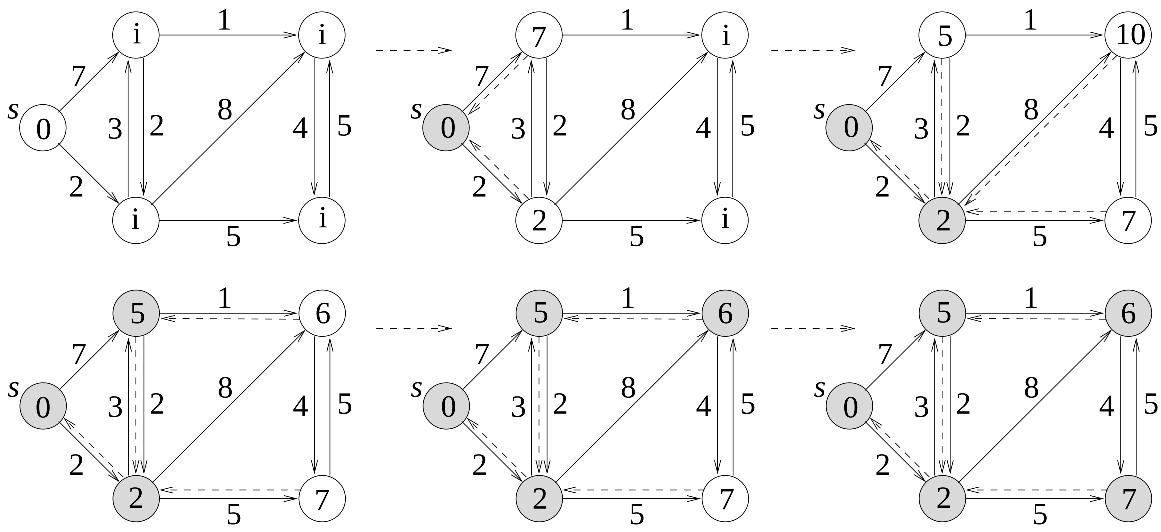


Figure 22: Dijkstra's Algorithm.

Proof: It will simplify the proof conceptually if we assume that all the edge weights are *strictly* positive (the general case of nonnegative edges is presented in the text).

Suppose to the contrary that at some point Dijkstra's algorithm *first* attempts to add a vertex u to S for which $d[u] \neq \delta(s,u)$. By our observations about relaxation, $d[u] > \delta(s,u)$. Consider the situation just prior to the insertion of u. Consider the true shortest path from s to u. Because $s \in S$ and $u \in V - S$, at some point this path must first take a jump out of S. Let (x,y) be the edge taken by the path, where $x \in S$ and $y \in V - S$ (it may be that $x = s$ and/or $y = u$).

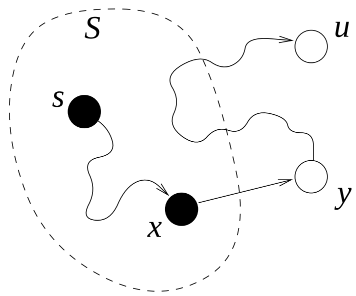


Figure 23: Correctness of Dijkstra's Algorithm.

We argue that $y \neq u$. Why? Since $x \in S$ we have $d[x] = \delta(s,x)$. (Since u was the first vertex added to S which violated this, all prior vertices satisfy this.) Since we applied relaxation to x when it was added, we would have set $d[y] = d[x] + w(x,y) = \delta(s,y)$. Thus $d[y]$ is correct, and by hypothesis, $d[u]$ is not correct, so they cannot be the same.

Now observe that since y appears midway on the path from s to u, and all subsequent edges are positive, we have $\delta(s,y) < \delta(s,u)$, and thus

$$d[y] = \delta(s,y) < \delta(s,u) \leq d[u].$$

Thus y would have been added to S *before* u, in contradiction to our assumption that u is the next vertex to be added to S.

Analysis: Because of the incredible similarity between the two algorithms, it follows from Prim's analysis, that Dijkstra's algorithm runs in $O((n+e)\log n)$ time. If Fibonacci heaps are used instead of standard binary heaps then the running time falls to $O(n\log n + e)$.

Lecture 13: Dynamic Programming

Read: The introduction to Chapter 15 in CLR. The material on the Knapsack Problem is not presented in our text, but is briefly discussed in Section 16.2.

Dynamic Programming: Today we move to an entirely new "paradigm" in solving optimization problems. So far we have seen the divide-and-conquer paradigm (useful in problems related to sorting or ordering), and for graph problems we have seen BFS and DFS and greedy algorithms.

Dynamic programming is used in optimization problems where we are typically looking for a solution that represents a set of decisions (e.g. a set of objects or a sequence of operations that minimize some objective function). The important elements to dynamic programming problems are that

Structure: Solutions to the optimization problems can be broken down into subproblems. Often problems can be broken into recursive subproblems (as in divide-and-conquer) but unlike divide-and-conquer there is often overlap between the subproblems.

Principle of Optimality: Subproblems must be solved optimally for the entire solution to be optimal.

Bottom-up computation: Computation is typically performed in a bottom-up manner (first solving smaller subproblems, and then combining these to form solutions to larger subproblems).

0-1 Knapsack Problem: Imagine that a burglar breaks into a museum and finds n items. Let v_i denote the value of the i-th item, and let w_i denote the weight of the i-th item. The burglar carries a knapsack capable of holding total weight W. The burglar wishes to carry away the most valuable subset items subject to the weight constraint.

For example, a burglar would rather steal diamonds before gold because the value per pound is better. But he would rather steal gold before lead for the same reason. We assume that the burglar cannot take a fraction of an object, so he/she must make a decision to take the object entirely or leave it behind. (There is a fractional version of the problem where the burglar can take a fraction of an object for a fraction of the value and weight. This is much easier to solve.)

More formally, given $\langle v_1, v_2, \ldots, v_n\rangle$ and $\langle w_1, w_2 \ldots, w_n\rangle$, and $W > 0$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of objects to "take") that maximizes

$$\sum_{i\in T} v_i,$$

subject to

$$\sum_{i\in T} w_i \le W.$$

Let us assume that the v_i's, w_i's and W are all positive integers. We will show that this problem can be solved in $O(nW)$ time. (Note that this is not very good if W is a large integer. But if we truncate our numbers to lower precision, this gives a reasonable approximation algorithm.)

Here is how we solve the problem. We construct an array $V[0..n, 0..W]$. For $1 \leq i \leq n$, and $0 \leq w \leq W$, the entry $V[i, w]$ we will store the maximum value of any subset of objects $\{1, 2, \ldots, i\}$ that can fit into a knapsack of weight w. If we can compute all the entries of this array, then the array entry $V[n, W]$ will contain the maximum value of all n objects that can fit into the entire knapsack of weight W.

To compute the entries of the array V we will imply an inductive approach. As a basis, observe that $V[0, w] = 0$ for $0 \leq w \leq W$ since if we have no items then we have no value. It will be convenient to "overload" our indexing operator so that $V[i, w] = -\infty$ whenever $w < 0$. Thus, a knapsack of negative capacity is made illegal by giving infinitely low value.

To compute $V[i, w]$ we consider two cases:

Leave object i: If we choose to not take object i, then the optimal value will come about by considering how to fill a knapsack of size w with the remaining objects $\{1, 2, \ldots, i-1\}$. This is just $V[i-1, w]$.

Take object i: If we take object i, then we gain a value of v_i but have used up w_i of our capacity. With the remaining $w - w_i$ capacity in the knapsack, we can fill it in the best possible way with objects $\{1, 2, \ldots, i-1\}$. This is just $v_i + V[i-1, w-w_i]$.

Since these are the only two possibilities, we can see that we have the following rule for constructing the array V:

$$
\begin{array}{lcll}
V[0, w] & = & 0 & \text{for } 0 \leq w \leq W, \\
V[i, w] & = & -\infty & \text{for } w < 0, \\
V[i, w] & = & \max(V[i-1, w], v_i + V[i-1, w-w_i]) & \text{for } 1 \leq i \leq n,\ 0 \leq w \leq W.
\end{array}
$$

Recall the rule above that $V[i, w] = -\infty$ if $w < 0$, so we do not have to worry about subscripting out of bounds in the last case.

An example is shown in the figure below. Let $v_1 = \$10$, $v_2 = \$40$, $v_3 = \$30$ and $v_4 = \$50$. Let $w_1 = 5$, $w_2 = 4$ $w_3 = 6$ and $w_4 = 3$. $W = 10$.

$V[i, w]$	$w = 0$	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Figure 24: 0-1 Knapsack Problem.

The final output is $V[4, 10] = 90$. This reflects the selection of items 2 and 4, of values \$40 and \$50, respectively and weights $4 + 3 \leq 10$.

It is very easy to take these rules an produce an algorithm that computes the maximum value for the knapsack in time proportional to the size of the array, which is $O((n+1)(W+1)) = O(nW)$.

Lecture 14: More Dynamic Programming

Read: Section 15.2.

0-1 Knapsack Problem: Last time we introduced the 0-1 Knapsack problem as an example of a simple dynamic programming problem. Recall that the problem was given object values $\langle v_1, v_2, \ldots, v_n \rangle$ and object weights $\langle w_1, w_2 \ldots, w_n \rangle$, and $W > 0$, and the problem was to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of objects to "take") that maximizes the *total value* $\sum_{i \in T} v_i$, subject to knapsack *weight constraint* $\sum_{i \in T} w_i \leq W$.

We showed that if we let $V[i, w]$ denote the maximum value of any subset of objects $\{1, 2, \ldots, i\}$ that can fit into a knapsack of weight capacity w, then we could solve this problem by computing $V[n, W]$ using the following recursive rule:

$$\begin{array}{lcll} V[0, w] & = & 0 & \text{for } 0 \leq w \leq W, \\ V[i, w] & = & -\infty & \text{for } w < 0, \\ V[i, w] & = & \max(V[i-1, w], v_i + V[i-1, w - w_i]) & \text{for } 1 \leq i \leq n,\ 0 \leq w \leq W. \end{array}$$

This suggests the following algorithm. As usual the rule above only computes the optimum value. To compute the actual subset, we could create an auxiliary boolean array $Take[i, w]$ which is 1 if we decide to take the i-th object in $V[i, w]$ and 0 otherwise. We will omit the details here but consider them with later dynamic programming problems.

0-1 Knapsack Problem

```
KnapSack(v, w, n, W) {
    for w = 0 to W do V[0,w] = 0;
    for i = 1 to n do {
        for w = 0 to W do {
            leave_val = V[i-1,w];                  // total value if we leave i
            if (w >= w[i])                         // enough capacity to take i
                take_val = v[i] + V[i-1, w - w[i]]; // total value if we take i
            else
                take_val = -INFINITY;              // cannot take i
                                                   // final value is max
            V[i,w] = max(leave_val, take_val);
        }
    }
    return V[n, W];
}
```

It is easy to see that the running time is $O(nW)$. Observe that this is fine if W is a small integer. If W is not an integer, it may be possible to scale and truncate W to a range of integer values, but this will generally not give an exact answer. The 0-1 knapsack problem is NP-complete (assuming W may be arbitrarily large) so it is not surprising that it is a harder problem to solve).

It is very easy to take these rules an produce an algorithm that computes the maximum value for the knapsack in time proportional to the size of the array, which is $O((n + 1)(W + 1)) = O(nW)$.

Chain Matrix Multiplication: This problem involves the question of determining the optimal sequence for performing a series of operations. This general class of problem is important in compiler design for code optimization and in databases for query optimization. We will study the problem in a very restricted instance, where the dynamic programming issues are easiest to see.

Suppose that we wish to multiply a series of matrices

$$A_1 A_2 \ldots A_n$$

Recall that matrix multiplication is associative but not commutative. This means that we are free to parenthesize the above multiplication however we like, but we are not free to rearrange the order of the matrices. Also recall that when two (nonsquare) matrices are being multiplied, there are restrictions on the dimensions. Recall that a $p \times q$ matrix has p horizontal rows and q vertical columns. You can multiply a $p \times q$ matrix A times a $q \times r$ matrix B, and the result will be a $p \times r$ matrix C. (The number of columns of A must equal the number of rows of B.) In particular for $1 \leq i \leq p$ and $1 \leq j \leq r$

$$C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j].$$

Observe that there are pr total entries in C and each takes $O(q)$ time to compute, thus the total time (e.g. number of multiplications) to multiply these two matrices is $p \cdot q \cdot r$.

Note that although any legal parenthesization will lead to a valid result, not all involve the same number of operations. Consider the case of 3 matrices: A_1 be 5×4, A_2 be 4×6 and A_3 be 6×2.

$$\begin{aligned} mult[((A_1 A_2) A_3)] &= (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180, \\ mult[(A_1 (A_2 A_3))] &= (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88. \end{aligned}$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence. The Chain Matrix Multiplication problem is: Given a sequence of matrices $A_1, A_2, \ldots, A_n$ and dimensions $p_0, p_1, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine the multiplication sequence that minimizes the number of operations.

IMPORTANT NOTE: The algorithm does not perform the multiplications, it just figures out the best order in which to perform the multiplications.

Naive Algorithm: We could write a procedure which tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is exponential in the number of items being parenthesized, roughly $\Omega(4^n/n^{3/2})$. This will not be practical except for very small n.

Dynamic Programming Solution: This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into subproblems, whose solutions can be combined to solve the global problem.

For convenience we can write $A_{i..j}$ to be the product of matrices i through j. It is easy to see that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix. In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is, for any k, $1 \leq k \leq n-1$,

$$A_{1..n} = A_{1..k} A_{k+1..n}.$$

Thus the problem of determining the optimal sequence of multiplications is broken up into 2 questions: how do we decide where to split the chain (what is k?) and how do we parenthesize the subchains $A_{1..k}$ and $A_{k+1..n}$? The subchain problems can be solved by recursively applying the same scheme. The former problem can be solved by just considering all possible values of k. Notice that this problem satisfies the principle of optimality, because if we want to find the optimal sequence for multiplying $A_{1..n}$ we must use the optimal sequences for $A_{1..k}$ and $A_{k+1..n}$. In other words, the subproblems must be solved optimally for the global problem to be solved optimally.

As with the 0-1 knapsack problem, we will store the solutions to the subproblems in an array. For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The optimum cost can be described by the following recursive definition. As a basis observe that if $i = j$ then the sequence contains only one matrix, and so the cost is 0. (There is nothing to multiply.) Thus, $m[i,i] = 0$. If $i < j$, then we are asking about the product $A_{i..j}$. This can be split by considering each k, $i \leq k < j$, as $A_{i..k}$ times $A_{k+1..j}$.

The optimum time to compute $A_{i..k}$ is $m[i,k]$, and the optimum time to compute $A_{k+1..j}$ is $m[k+1,j]$. We may assume that these values have been computed previously and stored in our array. Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix, and $A_{k+1..j}$ is a $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1} \cdot p_k \cdot p_j$. This suggests the following recursive rule for computing $m[i,j]$.

$$\begin{aligned} m[i,i] &= 0 \\ m[i,j] &= \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) \qquad \text{for } i < j. \end{aligned}$$

It is not hard to convert this rule into a procedure.

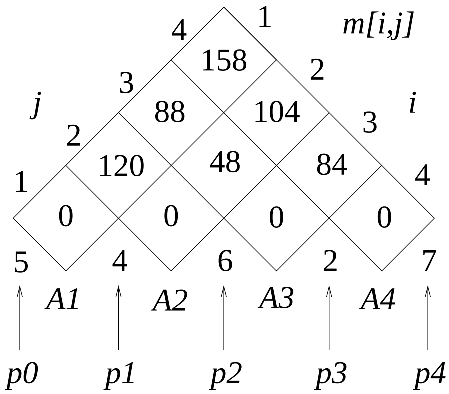


Figure 25: Chain Matrix Multiplication.

Extracting the final Sequence: We could implement the above rule, and this would tell us what the optimum number of operations is. The only trick in the implementation is that we compute array entries in a rather funny order, starting with the diagonal, and working further and further off the diagonal.

To extract the actual sequence is a fairly easy extension. The basic idea is to leave a "split marker" indicating what the best split is, that is, what value of k lead to the minimum value of m. We can maintain a parallel array $s[i,j]$ in which we will store the value of k providing the optimal split. Intuitively, this will tell us what multiplication to perform *last.* The entire sequence can be extracted recursively. Here is the algorithm for the optimal multiplication sequence.

Chain Matrix Multiplication

```
Matrix-Chain(p, n) {
    for i = 1 to n do m[i,i] = 0;              // initialize
    for l = 2 to n do {
        for i = 1 to n-l+1 do {
            j = i + l - 1;
            m[i,j] = INFINITY;
            for k = i to j-1 do {
```

```
                q = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j];
                if (q < m[i, j]) {
                    m[i,j] = q;
                    s[i,j] = k;
                }
            }
        }
    }
    return m and s;  (Optimum is in m[1,n].)
}
```

The actual multiplication code uses the $s[i,j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1..n]$, and that $s[i,j]$ is global to this recursive procedure. The procedure returns a matrix.

Extracting Optimum Sequence

```
Mult(i, j) {
    if (i > j) {
        X = Mult(A, s, i, s[i,j]);        // X = A[i]...A[k]
        Y = Mult(A, s, s[i,j]+1, j);      // Y = A[k+1]...A[j]
        return X*Y;                       // multiply matrices X and Y
    }
    else
        return A[i];
}
```

Lecture 15: All-Pairs Shortest Paths

Read: Chapt 25 (up to Section 25.2) in CLR.

All-Pairs Shortest Paths: We consider the problem of determining the length of the shortest path between all pairs of vertices in a weighted directed graph. Of course, we could use Dijkstra's algorithm applied to each vertex of the digraph. If we used the best version of Dijkstra's algorithm (with Fibonacci heaps) this would give an $O(n(n \log n + e)) = O(n^2 \log n + ne)$ time algorithm. If the digraph is dense, this is an $O(n^3)$ algorithm, and requires some effort to implement because of the underlying data structures.

We will show that there is an extremely simple $O(n^3)$ algorithm, called the Floyd-Warshall algorithm, which is based on dynamic programming. For dense digraphs, this algorithm is preferred over iterated Dijkstra's because of its simplicity. The Floyd-Warshall algorithm also has the advantage it works even on graphs with negative cost edges provided there are no negative cost cycles. For sparse digraphs, the iterated version of Dijkstra's algorithm is still better.

The Floyd-Warshall algorithm dates back to the early 60's. Warshall was interested in the weaker question of reachability, determine for each pair of vertices u and v, whether u can reach v. Floyd realized that the same technique could be used to compute shortest paths with only minor variations.

Unlike the previous algorithms, this algorithm assumes that the graph is represented by an adjacency matrix. The result is an $n \times n$ matrix of distances, which is dense.

The input is an $n \times n$ matrix of weights, which are based on the edge weights in the digraph:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ +\infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

The reason for setting $w_{ii} = 0$ is that intuitively the cost of getting from any vertex to itself is zero, even if there is a self-loop edge at this vertex.

The output will be an $n \times n$ distance matrix $D = d_{ij}$ where $d_{ij} = \delta(i,j)$, the shortest path length from vertex i to j. As before we will also compute an auxiliary matrix `pred[i,j]`, whose purpose will be to help derive the actual shortest path.

Dynamic Programming for Shortest Paths: Recall that every dynamic programming algorithm is based on certain basic elements.

Substructure: Decompose the problem of computing shortest paths into a collection of subproblems.

Optimal substructure: The subproblems should be solved optimally.

Bottom-up computation: Use solutions to small subproblems to solve larger subproblems.

The question is how to decompose the shortest path problem into subproblems in a meaningful way. There is one very natural way to do this. What is remarkable, is that this does *not* lead to the best solution. The Floyd-Warshall algorithm makes use of a much less natural decomposition into subproblems, but it leads to a more efficient algorithm.

Let us first sketch the natural way to break the problem into subproblems. For $0 \leq m \leq n-1$, define $d_{ij}^{(m)}$ to be the shortest path from vertex i to vertex j that contains at most m edges. Let $D^{(m)}$ denote the matrix whose entries are these values. The idea is to compute $D^{(0)}$ then $D^{(1)}$, and so on, up to $D^{(n-1)}$. Since we know that no shortest path can use more than $n-1$ (for otherwise it would have to repeat a vertex), we know that $D^{(n-1)}$ is the final distance matrix.

The question is, how do we compute these distance matrices? As a basis, for $m = 0$, we have

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases}$$

We can compute $D^{(m)}$ from $D^{(m-1)}$ by a process similar to relaxation. Consider the shortest path from i to j using at most m edges. This is the quantity $d_{ij}^{(m)}$. Either this path uses strictly less than m edges, in which case it is the same as $d_{ij}^{(m-1)}$. Otherwise it uses exactly m edges. In this case, the path uses $m-1$ edges to go from i to some vertex k, and then follows the edge (k,j) of weight w_{kj}. The path from i to k should be shortest using $\leq m-1$ edges (the principle of optimality) so the length of the resulting path is $d_{ik}^{(m-1)} + w_{kj}$. We do not know what k is, so we minimize over all possible choices. This gives the following rule:

$$\begin{aligned} d_{ij}^{(m)} &= \min\left(d_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{d_{ik}^{(m-1)} + w_{kj}\right\}\right). \\ &= \min_{1 \leq k \leq n}\left\{d_{ik}^{(m-1)} + w_{kj}\right\}, \end{aligned}$$

where the latter equality follows because $w_{jj} = 0$. The final output is the matrix $D^{(n-1)}$.

If we were to implement this rule, we would see that the running time is $O(n^4)$. Why? There are n possible values of m to consider, there are n^2 choices of i and j, and for each value of m, i, and j, we must minimize over n choices for k. Thus the running time is $O(n \cdot n^2 \cdot n) = O(n^4)$.

Lecture 16: Floyd-Warshall Algorithm

Read: Chapt 25 (up to Section 25.2) in CLR.

All-Pairs Shortest Paths: Last time we considered the problem of determining the length of the shortest path between all pairs of vertices in a weighted directed graph. We mentioned that there is an $O(n^3)$ dynamic programming algorithm, called the *Floyd-Warshall* algorithm. However, before presenting the Floyd-Warshall algorithm, we presented a somewhat more natural formulation of the problem from a dynamic programming perspective (mostly to show that there is more than one way to approach D.P. solutions).

We presented a dynamic programming scheme running in $O(n^4)$ time, based on computing shortest paths with the restriction that the path consists of at most $0, 1, 2, \ldots, n-1$ edges. Recall that $d_{ij}^{(m)}$ denotes the length of the shortest path from i to j, subject to the constraint that the path can use at most m edges. The recursive defining rule is:

$$\begin{aligned} d_{ij}^{(0)} &= \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases} \\ d_{ij}^{(m)} &= \min_{1 \le k \le n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}. \end{aligned}$$

Path Doubling: There is a quick and dirty way to improve this simple dynamic programming solution. (But this is still not as good as the Floyd-Warshall algorithm.) Here we computed $d^{(0)}$, then $d^{(1)}$, up to $d^{(n-1)}$. In each case we formed a path of length m by adding a single new edge to a path of length $m-1$. Can we "grow" paths faster?

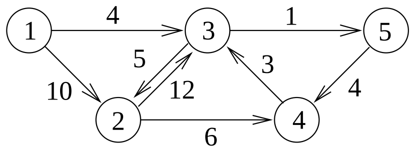
The answer is yes. In particular, if we know paths of length m between every pair of vertices in the digraph, then we can compute paths of length $2m$ by splicing these two paths together in all possible ways. This is called *path doubling.* Assuming we had computed $d^{(m)}$, we could then compute $d^{(2m)}$ by first considering the shortest path of length at most m from i to each possible vertex k, then the shortest path of length at most m from k to j. The new rule would be something like

$$d_{ij}^{(2m)} = \min_{1 \le k \le n} \left\{ d_{ik}^{(m)} + d_{kj}^{(m)} \right\}.$$

Since we double the lengths of paths at each stage, we would only have to compute $O(\log n)$ matrices, $d^{(1)}$, $d^{(2)}$, $d^{(4)}$, $d^{(8)}$, up to the next power of 2 larger than $n-1$. Because we are replacing the computation of n matrices with a computation of $\log n$ matrices, we replace a factor of n in the running time with $O(\log n)$. This would convert our $O(n^4)$ algorithm into an $O(n^3 \log n)$ algorithm.

Floyd-Warshall Algorithm: The genius of the Floyd-Warshall algorithm is in finding a completely different formulation for the various shortest path subproblems. At first the formulation may seem most unnatural, but it leads to a faster algorithm. As before, we will compute a set of matrices whose entries are $d_{ij}^{(k)}$. We will change the *meaning* of each of these entries. In particular, rather than having the superscript k restrict the *length* of the path, it will restrict the *possible locations* in the graph that the path can travel through on its way from i to j.

For a path $p = \langle v_1, v_2, \ldots, v_\ell \rangle$ we say that the vertices $v_2, v_3, \ldots, v_{\ell-1}$ are the *intermediate vertices*. Note that a path consisting of a single edge has no intermediate vertices. We define $d_{ij}^{(k)}$ to be the shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, \ldots, k\}$. In other words, we consider a path from i to j which either consists of the single edge (i, j), or it visits some intermediate vertices along the way, but these intermediate can only be chosen from $\{1, 2, \ldots, k\}$. The path is free to visit any subset of these



$$
\begin{array}{llll}
d_{14}^{(0)} = +\infty & \text{(no path)} & 1 \to \{\} \to 4 \\
d_{14}^{(1)} = +\infty & \text{(no path)} & 1 \to \{1\} \to 4 \\
d_{14}^{(2)} = 16 & (1,2,4) & 1 \to \{1,2\} \to 4 \\
d_{14}^{(3)} = 15 & (1,3,2,4) & 1 \to \{1,2,3\} \to 4 \\
d_{14}^{(4)} = 15 & (1,3,2,4) & 1 \to \{1,2,3,4\} \to 4 \\
d_{14}^{(5)} = 9 & (1,3,5,4) & 1 \to \{1,2,3,4,5\} \to 4
\end{array}
$$

Figure 26: Floyd-Warshall Formulation.

vertices, and to do so in any order. For example, in the digraph shown in the following figure, notice how the value of $d_{14}^{(k)}$ changes as k varies.

How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous entrie $d^{(k-1)}$. There are two basic ways we can get from vertex i to vertex j, assuming that the intermediate vertices are chosen from $\{1, 2, \ldots, k\}$:

We don't go through k at all: Then the shortest path from i to j uses only intermediate vertices $\{1, \ldots, k-1\}$ and hence the length of the shortest path is $d_{ij}^{(k-1)}$.

We do go through k: First observe that a shortest path does not pass through the same vertex twice, so we can assume that we pass through k exactly once (the assumption that there are no negative cost cycles is being used here). That is, we go from i to k, and then from k to j. In order for the overall path to be as short as possible we should take the shortest path from i to k, and the shortest path from k to j. (This is the principle of optimality.) Each of these paths uses intermediate vertices only in $\{1, 2, \ldots, k-1\}$. The length of the path is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

This gives us the following recursive rule for computing $d^{(k)}$:

$$
\begin{aligned}
d_{ij}^{(0)} &= w_{ij}, \\
d_{ij}^{(k)} &= \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) \qquad \text{for } k \geq 1.
\end{aligned}
$$

The final answer is $d_{ij}^{(n)}$ because this allows all possible vertices as intermediate vertices. Here is the complete algorithm. We have also included predecessor pointers for extracting the final shortest paths. We will discuss this later.

Floyd-Warshall Algorithm

```
Floyd_Warshall(w, n) {
    for i = 1 to n do {                       // initialize
        for j = 1 to n do {
            d[i,j] = w[i,j];
            pred[i,j] = nil;
        }
```

```
    }
    for k = 1 to n do {                         // intermediates {1,2,...,k}
        for i = 1 to n do {                     // consider all i, j
            for j = 1 to n do {
                if (d[i,k] + d[k,j]) < d[i,j] then {
                    d[i,j] = d[i,k] + d[k,j];   // new shorter path length
                    pred[i,j] = k;              // new path is through k
                }
            }
        }
    }
    return d[1..n, 1..n];                       // matrix of final distances
}
```

Clearly the algorithm's running time is $O(n^3)$. The space used by the algorithm is $O(n^2)$. Observe that we deleted all references to the superscript (k) in the code. It is left as an exercise that this does not affect the correctness of the algorithm.

Extracting Shortest Paths: The predecessor pointers `pred[i, j]` can be used to extract the final path. Here is the idea, whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set `pred[i, j] = k`. If the shortest path does not pass through any intermediate vertex, then `pred[i, j] = nil`. To find the shortest path from i to j, we consult `pred[i, j]`. If it is null, then the shortest path is just the edge (i, j). Otherwise, we recursively compute the shortest path from i to `pred[i, j]` and the shortest path from `pred[i, j]` to j.

Printing the Shortest Path

```
Path(i,j) {
    if pred[i,j] = nil                  // shortest path is edge
        output(i,j)
    else {                              // path goes through pred
        Path(i, pred[i,j]);             // go from i to pred
        Path(pred[i,j], j);             // go from pred to j
    }
}
```

Transitive Closure: Often times when dealing with nonweighted digraphs, a common problem is simply to determine which vertices can reach which others. This problem is called the *transitive closure* problem. The transitive closure of a digraph $G = (V, E)$ is a digraph $G^* = (V, E^*)$ defined on the same vertex set where

$$E^* = \{(i, j) \mid \text{there is a path from } i \text{ to } j\}.$$

Observe that with very little change, we can adapt the Floyd-Warshall algorithm to compute transitive closures also. We define $t_{ij}^{(k)}$ to be 1 if there is a path from i to j whose intermediate vertices are chosen from $\{1, 2, \ldots, k\}$. Otherwise $t_{ij}^{(k)} = 0$. Observe that

$$t_{ij}^{(0)} = \begin{cases} 1 & \text{if } i = j \text{ or } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

To compute $t_{ij}^{(k)}$ from t^{k-1} observe that $t_{ij}^{(k)} = 1$ if there is a path from i to j that passes through intermediate vertices $\{1, 2, \ldots, k-1\}$ $(t_{ij}^{(k-1)})$ or there is a path from i to k using these intermediate vertices ($t_{ik}^{(k-1)}$) and then a path from k to j ($t_{kj}^{(k-1)}$). This gives the following recursive rule:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$$

It is easy to modify the Floyd-Warshall algorithm above to compute transitive closure's as well. In fact, there is an extremely fast implementation of this algorithm if you assume that you can pack the 0-1 rows of an adjacency matrix into bit vectors. There is a variant of this algorithm in which Boolean operations are performed on entire bit vectors.

Lecture X02: Longest Common Subsequence

Supplementary
Read: Section 15.4 in CLR.

Strings and Genetics: Because of DNA research, there is a need to do string processing on extremely long strings. In order to do genetic comparisons, it is desrirable to determine how similar two strings are (presumably this determines how closely related two species are).

The method of comparing strings should be insensitive to random insertions and deletions of characters from some originating string. This suggests the following problem as a possible measure of similarity.

Longest Common Subsequence: Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Z = \langle z_1, z_2, \ldots, z_k \rangle$, we say that Z is a *subsequence* of X if there is a strictly increasing sequence of k indices $\langle i_1, i_2, \ldots, i_k \rangle$ $(1 \leq i_1 < i_2 < \ldots < i_k \leq n)$ such that $Z = \langle X_{i_1}, X_{i_2}, \ldots, X_{i_k} \rangle$.

For example, let $X = \langle ABRACADABRA \rangle$ and let $Z = \langle AADAA \rangle$, then Z is a suseqeunce of X.

Given two strings X and Y, the *longest common subsequence* of X and Y is a longest sequence Z which is both a subsequence of X and Y.

For example, let X be as before and let $Y = \langle YABBADABBADOO \rangle$. Then the longest common subsequence is $Z = \langle ABADABA \rangle$.

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$ determine a longest common subsequence. Note that it is not always unique. For example the LCS of $\langle A, B, C \rangle$ and $\langle B, A, C \rangle$ is either $\langle A, C \rangle$ or $\langle B, C \rangle$.

Dynamic Programming Solution: As with chained matrix multiplication we could try all possible subsequences (by generating all possible index vectors), but this would require exponential time (there are 2^m subsequences of X) (try to prove this).

Instead, we proceed as we did in matrix multiplication to mathematically derive a description of the structure of the optimal solution, and then convert this description into an algorithm.

We need to break the problem into smaller pieces. There are many ways to do this for strings, but it turns out for this problem that considering all pairs of *prefixes* will suffice for us. A *prefix* of a sequence is just an initial string of values, $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. X_0 is the empty sequence.

The idea intuitively will be to compute the longest common subsequence for EVERY PAIR of prefixes. Let $c[i,j]$ denote the length of the longest common subsequence of X_i and Y_i. Eventually we are interested in $c[m,n]$ since this will be the LCS of the two entire strings. The idea is to compute $c[i,j]$ assuming that we already know the values of $c[i',j']$ for $i' \leq i$ and $j' \leq j$ (but not both equal).

We begin with some observations.

Basis: $c[i,0] = c[j,0] = 0$. If either sequence is empty, then the longest common subsequence is empty.

Last characters match: Suppose $x_i = y_j$. Example: Let $X_i = \langle ABCA \rangle$ and let $Y_j = \langle DACA \rangle$. Since both end in A, the LCS MUST end in A (for otherwise we could make the sequence longer by adding A). Since the A is part of the LCS we may find the overall LCS by removing A from both sequences and taking the LCS of $X_{i-1} = \langle ABC \rangle$ and $Y_{j-1} = \langle DAC \rangle$ which is $\langle AC \rangle$ and then adding A to the end, giving $\langle ACA \rangle$ as the answer.

Thus, if $x_i = y_j$ then $c[i,j] = c[i-1, j-1] + 1$.

Last characters do not match: Suppose that $x_i \neq y_j$. In this case x_i and y_j CANNOT BOTH be in the LCS (since they would have to be the last character of the LCS). Thus either x_i is NOT part of the LCS, or y_j is NOT part of the LCS (and possibly BOTH are not part of the LCS).

In the first case the LCS of X_i and Y_j is the LCS of X_{i-1} and Y_j, which is $c[i-1,j]$. In the second case the LCS is the LCS of X_i and Y_{j-1} which is $c[i,j-1]$.

Thus, if $x_i \neq y_j$ then $c[i,j] = \max(c[i-1,j], c[i,j-1])$.

Combining these observations we have the following rule:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Implementing the Rule: The task now is to simply implement this rule. As with the chained matrix multiplication, this only computes the length of the longest sequence. It does not compute the actual sequence. This is a simple modification to the algorithm, which is given in the text.

Longest Common Subsequence

```
LCS(x[1..m], y[1..n]) {
    for i = 0 to m do c[i,0] = 0;
    for j = 0 to n do c[0,j] = 0;
    for i = 1 to m do {
        for j = 1 to n do {
            if (x[i] == y[j]) c[i,j] = c[i-1,j-1]+1
            else c[i,j] = max(c[i,j-1], c[i-1,j]);
        }
    }
    return c[m,n];
}
```

Example. Let $X = BACDB$ and let $Y = BDCB$ (so $LCS = BCB$).

The running time of the algorithm is

$$T(n,m) = \sum_{i=1}^{m} \sum_{j=1}^{n} 1 = \sum_{i=1}^{m} n = mn \in O(mn).$$

The algorithm also uses $O(mn)$ space. We can achieve a savings in space by observing that we only need the two most recent rows of the matrix c, $c[i,*]$ and $c[i-1,*]$. Thus, we can reduce the space to $O(n)$. In fact, if $n > m$ then we swap X and Y, so the space is $O(\min(m,n))$.

		0	1	2	3	4
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
5	B	0	1	2	2	3

Figure 27: Longest Common Subsequence.

Extracting the Actual Sequence: The trick for extracting the sequence is to first perform the entire computation, but at each step keep track of the choice which was made as to how the sequence was generated. Then starting at the final answer, $c[m, n]$, trace back to find how this answer was arrived at. Every time a diagonal step was taken by the algorithm corresponds to one additional character added to the LCS. (In the figure above we show these pointers only for the critical path.) Here is a more rigorous explanation.

There are three possible things that could happen to give the value to $c[i, j]$. We modify the above algorithm to record each event, by creating an array $b[i, j]$ which has 3 possible values. Replace the code:

```
if (x[i] == y[j]) c[i,j] = c[i-1,j-1]+1
else c[i,j] = max(c[i,j-1], c[i-1,j]);
```

with the following:

```
if (x[i] == y[j]) {
    c[i,j] = c[i-1,j-1]+1;
    b[i,j] = ADDXY;
}
else if (c[i-1,j] >= c[i,j-1]) {
    c[i,j] = c[i-1,j];
    b[i,j] = SKIPX;
}
else {
    c[i,j] = c[i,j-1];
    b[i,j] = SKIPY;
}
```

To recover the LCS we use the following recursive procedure.

Print LCS

```
PrintLCS(i, j) {
    if (i == 0 or j == 0) return;
    else if (b[i,j] == ADDXY) {
        PrintLCS(i-1, j-1);
        output(x[i]);
    }
```

```
        else if (b[i,j] == SKIPX)
            PrintLCS(i-1, j);
        else
            PrintLCS(i, j-1);
    }
```

Lecture 17: Greedy Algorithms: Huffman Coding

Read: Section 16.3 in CLR.

Greedy Algorithms: In many optimization algorithms a series of selections need to be made. In dynamic programming we saw one way to make these selections is in a "bottom-up" manner, by finding optimal solutions to subproblems and working up to larger subproblems. Sometimes an optimization problem has a particularly simple mathematical structure, that allows us to make decisions in a more "top-down" manner, by "greedily" making the best selection at each stage.

Because of their simplicity greedy algorithms are typically more efficient than dynamic programming solutions. However, unlike dynamic programming, which is applicable for a wide variety of problems, greedy algorithms only can be applied for a very special class of problems. Even when greedy algorithms do not produce the optimal solution, they often provide good heuristics for coming up with quick and dirty approximations.

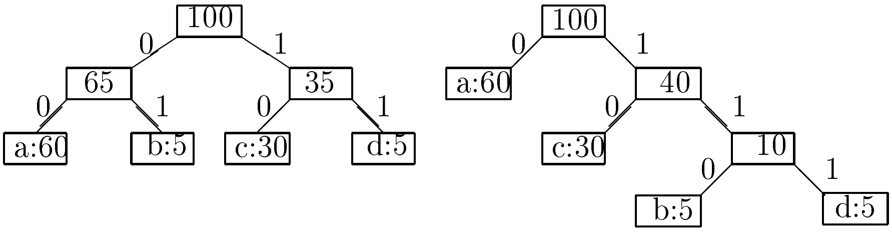
Huffman Codes: Huffman codes provide a method of compressing data. Normally when characters are coded using standard codes like ASCII, each character is represented by a fixed length code of bits (e.g. 8 bits per character). However, this is wasteful because characters that occur more often take as much space to encode as characters that are rarely used. By using variable length codes we can use shorter code words for more common letters, and longer code words for less common words.

For example, suppose we want to store messages made up of 4 characters:

	a	b	c	d
Frequency	60	5	30	5
Fixed-Length Code Word	00	01	10	11
Vriable-Length Code Word	0	110	10	111

Now, to store 100 of these characters using a fixed-length code would require $100 \cdot 2 = 200$ bits, but to store 100 characters with the given frequencies would only take $60 \cdot 1 + 5 \cdot 3 + 30 \cdot 2 + 5 \cdot 3 = 150$ bits, a 25% savings. Note that there is no requirement that codewords be ordered alphabetically.

Prefix Codes: Note that in both the fixed-length and variable-length codes given in the example above no codeword is a prefix of another. This is very important, because if codewords did share a common prefix, e.g. $a = 001$ and $b = 00101$, then when we see $0010110\ldots$ how do we know whether the first character of the encoded message is a or b? Codes that have this desirable property are called **prefix codes**. Observe that any binary prefix coding can be described by a tree in which the codewords are the leaves of the tree, and where a left branch means "0' and a right branch means "1". Here are the above two codes. The length of a codeword is just its depth in the tree.



Once we know the frequencies of the various characters, we can determine the total length of the encoded text. Let $f(c)$ denote the frequency of character c, and let $d_T(c)$ denote the length of the codeword (relative to some tree T). The number of bits needed to encode the text is

$$B(T) = \sum_{c \in C} f(c) d_T(c).$$

The **Huffman code** is the prefix code that minimizes this quantity.

Constructing a Huffman Code: Here is the intuition behind a greedy algorithm. We are going build the tree up from the leaf level. We know intuitively that infrequently occuring characters should appear at the bottom of the tree. We take the two characters with the smallest frequency of occurrence and create a subtree that has these two characters as leaves. We now imagine that this subtree represents a single character whose frequency is equal to the sum of frequencies of the two characters. This operation is called **merging**. After merging we effectively have one less character. We repeat this process until only one character remains. Here is an example (from the text).

Here is a loose description of the algorithm. Initially each character is represented by a leaf node. Each node has a field for left-child, right-child, and frequency.

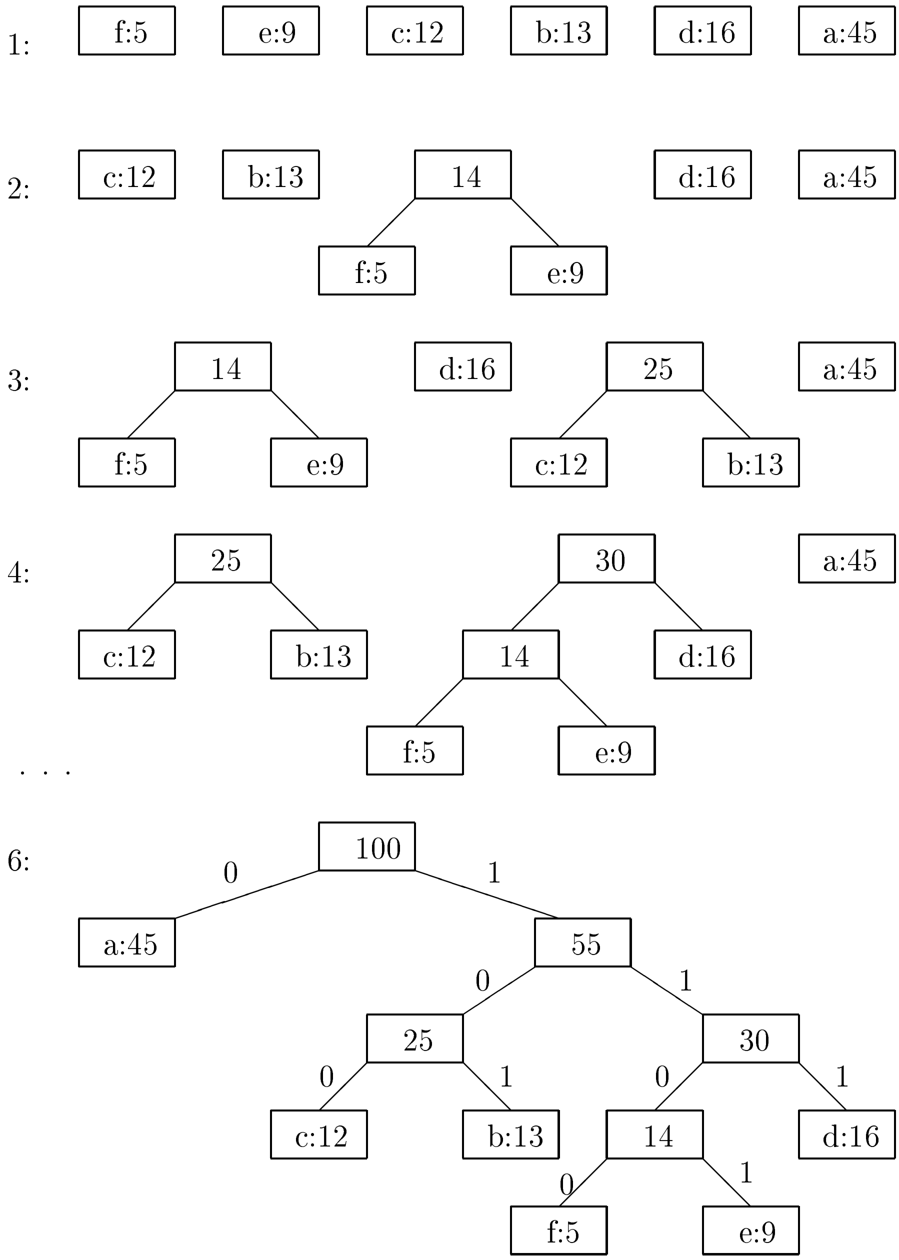
```
proc Huffman() {
    for i = 1 to n-1 do {
        let x and y be two characters with smallest frequency;
        create new node z with left child x,
                          with right child y,
                          with frequency f(z) = f(x) + f(y);
        replace characters x and y with z;
    }
}
```

The algorithm can be implemented in $O(n \log n)$ time by storing the characters ordered by frequency in a priority queue (e.g. a heap).

Correctness: The big question is why is this algorithm correct? It is obviously simple and easy to implement. Recall that the cost of any encoding tree T is $B(T) = \sum_c f(c) d_T(c)$. We will think in terms of the cost of trees, but everything we will say has an obvious counterpart regarding the structure of the code itself.

The idea is to show that any tree that differs from the one constructed by Huffman's algorithm can be converted into one that is equal to Huffman's tree without increasing its cost.

Claim: The tree for any optimal prefix code must be "full", meaning that every internal node has exactly two children.



1:
f:5
e:9
c:12
b:13
d:16
a:45
2:
c:12
b:13
14
d:16
a:45
f:5
e:9
3:
14
d:16
25
a:45
f:5
e:9
c:12
b:13
4:
25
30
a:45
c:12
b:13
14
d:16
f:5
e:9
. . .
6:
100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

Proof: If some internal node had only one child then we could simply get rid of this node and replace it with its one child. This would decrease the total cost of the encoding.

Claim: Consider the two characters, x and y with the smallest frequencies. Then there is an optimal code tree in which these two characters are siblings at the maximum depth in the tree.

Proof: Let T be any optimum prefix code tree, and let b and c be two siblings at the maximum depth of the tree. Assume without loss of generality that $f(b) \leq f(c)$ and $f(x) \leq f(y)$ (if this is not true, then rename these characters). Now, since x and y have the two smallest frequencies it follows that $f(x) \leq f(b)$ (they may be equal) and $f(y) \leq f(c)$ (may be equal). Because b and c are at the deepest level of the tree we know that $d(b) \geq d(x)$ and $d(c) \geq d(y)$.

Now switch the positions of x and b in the tree resulting in a different tree T' and see how the cost changes. The total cost of T' is

$$\begin{aligned}
B(T') &= B(T) - f(x)d(x) + f(x)d(b) - f(b)d(b) + f(b)d(x) \\
&= B(T) + f(x)(d(b) - d(x)) - f(b)(d(b) - d(x)) \\
&= B(T) - (f(b) - f(x))(d(b) - d(x)) \\
&\leq B(T) \qquad \text{because } (f(b) - f(x))(d(b) - d(x)) \geq 0.
\end{aligned}$$

Therefore T' is an optimum tree. By switching y with c we get a new tree T'' which by a similar argument is optimum. The final tree T'' satisfies the statement of the claim.

Claim: Huffman's algorithm produces the optimum prefix code tree.

Proof: The proof is by induction on the number of characters. When the number of characters is one, then the tree consists of a single leaf=root node, which is obviously optimum.

Assume inductively that when strictly fewer than n characters, Huffman's algorithm is guaranteed to produce the optimum tree. We want to show it is true with exactly n characters. Suppose we have exactly n characters. The previous claim states that we may assume that in the optimum tree, the two characters of lowest frequency x and y will be siblings at the lowest level of the tree. Remove x and y, replacing them with a new character z whose frequency is $f(z) = f(x) + f(y)$. Thus $n - 1$ characters remain. Consider ANY prefix code tree T made with this new set of $n - 1$ characters. We can convert it into a prefix code tree T' for the original set of characters by undoing the previous operation and replacing z with x and y (adding a "0" bit for x and a "1" bit for y). The cost of the new tree is

$$\begin{aligned}
B(T') &= B(T) - f(z)d(z) + f(x)(d(z) + 1) + f(y)(d(z) + 1) \\
&= B(T) - (f(x) + f(y))d(z) + (f(x) + f(y))(d(z) + 1) \\
&= B(T) + (f(x) + f(y))(d(z) + 1 - d(z) \\
&= B(T) + f(x) + f(y).
\end{aligned}$$

Since the change in cost depends in NO WAY on the structure of the tree T, to minimize the cost of the final tree T' we should build the tree T on $n - 1$ keys in the optimum way, but this exactly what Huffman's algorithm does (by induction). Thus the final tree is optimum.

Lecture 18: NP-Completeness

Read: Chapt 34.

Polynomial Time: Up until now all the algorithms we have seen have had the property that their running times are bounded above by some polynomial: Sorting is $O(n \log n) \leq O(n^2)$, Depth-First search is $O(n+e) \leq O(n^2)$, Kruskal's algorithm is $O((n+e)\log n) \leq O(n^3)$. In general any algorithm whose running time is $O(n^k)$ where k is a constant (independent of n) is a *polynomial time* algorithm. Examples of nonpolynomial time functions include $n^{(\log n)}$, 2^n, $n!$, and n^n.

(Beware: Some things that may "look" polynomial are not. For example, suppose some graph algorithm has a running time $O(n^k)$ where k is the number of articulation points in the graph. In the worst case the number of articulation points can be as high as $n-2$, so the running time is actually $O(n^{n-2})$ which is NOT polynomial time.)

We are interested in polynomial time for two reasons. The first is that it is generally felt that any algorithm that is NOT in polynomial time is not practical for large values of n. For example, 2^n grows so fast that for n equals, say 100, you have no hope of solving such a problem in reasonable time on any machine that will come into existence in the foreseeable future (for example, a dedicated parallel machine with a billion processors, which can perform operations at the rate of a billion per second, assuming maximum parallelism, would still take about 400 centuries to finish the task.)

The second reason is that although polynomial time contains some functions which cannot be called practical, e.g. $O(n^{100})$, polynomials have the nice property that they are closed under composition, thus if $f(n)$ is a polynomial, and $g(n)$ is a polynomial, then $f(g(n))$ is still a polynomial (whose degree is the product of the degree of f and g). Thus, as long as we make a polynomial number of calls to a polynomial time subroutine, we are assured that the final running time will be polynomial). This would not be true if we limited ourselves to, say polynomials of bounded degree.

We say that a PROBLEM is solvable in polynomial time (or more simply the problem is in polynomial time) if there exists an algorithm which solves the problem in polynomial time. We want to talk about a class of problems, called *NP-complete* problems, for which it is widely believed that none of these problems is solvable in polynomial time. To present the theory of NP-completeness it will be convenient to make a number of simplifying assumptions about the problems that we will be dealing with. The first assumption is that the problems are decision problems.

Decision Problems: A problem is called a *decision problem* if its output is a simple "yes" or "no" (or you may think of this as True/False, 0/1). For example, the normal minimum spanning tree problem is:

Optimization problem: Given a weighted graph, determine the spanning tree of minimum cost.

The corresponding decision problem might be

Decision problem: Given a weighted graph G and an integer k, does G have a spanning tree of weight at most k?

For almost all optimization problems there exists a corresponding simpler decision problem. Observe that if we had a subroutine for solving the optimization problem, solving the decision problem would be trivial. Thus if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must also be difficult.

The Class P: Define P to be the set of all decision problems Π that can be solved in polynomial time.

Encodings: We want to be able to argue that certain problems are NOT in the class P. In other words, it is computationally hard to solve the corresponding problem. One technical question that arises in this process is whether or not the difficulty in answering the problem is a function of how the input to the problem is encoded. For example, if I represented the graph by an adjacency list rather than an adjacency matrix would it make any difference. Obviously in this case the answer is NO, since in polynomial time we could convert one representation to another. Basically, the rule of thumb in deciding on the representation of an input is that there should be a polynomial time algorithm that converts your representation into any standard representation and vice versa. In general you should represent objects as succinctly as reasonably possible. For example, numbers should be represented in binary (or in some higher base) and not in unary.

One should also be careful about "loose" assumptions that we have been making throughout the semester. For example, we have said, "create an edge with capacity $+\infty$". Such statements need to be made much more carefully in the context of complexity theory, since it is not clear how to encode this as a string (without using an infinite number of bits). A better approach is to look at the graph, and based on the existing capacities, pick some capacity that is clearly large enough (e.g. sum all the capacities of the graph and use this value). Of course, the actual value to be chosen to be "large enough" depends on the actual problem you are dealing with.

Lecture 19: NP-Completeness: Certificates/Reduction

Read: Chapt 34.

Polynomial Time Verification and Certificates: Before talking about the class of NP-complete problems, it is helpful to introduce the notion of a verification algorithm. Some problems that are very hard to solve may be very easy to verify.

For example, consider the problem of determining whether a positive integer x is prime or not. If a number is NOT prime it is called *composite*. Presumably you test compositeness by dividing the number x by all smaller numbers $2 \leq y \leq x/2$ and testing whether y divides x, but this would take $O(x)$ time. However if x is encoded in binary, the length of the string x is $n = \lg x$, implying that $x = 2^n$. Thus the running time of this algorithm is $O(2^n)$ which is prohibitive if x has hundreds of digits (which is common in encryption applications).

On the other hand, if someone told you x is composite and here is the y which divides x to prove it. You could verify very quickly that y divides x (since the time to perform division is polynomial in the number of bits). The value y is called a *certificate* for the fact that x is composite.

Here is another example. Given an undirected graph G, a *Hamiltonian Cycle* is a simple cycle that visits every vertex of the graph exactly once. A graph is said to be *Hamiltonian* if it has such a cycle. The HAM-CYCLE problem is: Given a graph G, does it have a Hamiltonian cycle?

We could attempt to solve this problem brute force by trying all $n!$ permutations of the vertex set, and testing whether any of these is a cycle in the graph. In fact there is no polynomial time algorithm known for this problem. However, observe that if someone were to "give us" the correct permutation to test, then we could verify that this is correct very efficiently (in polynomial time). In this case the permutation is the *certificate*.

There is one important distinction to observe here. We call an instance I of a decision problem a *yes-instance* if the correct answer for it is YES, otherwise it is called a *no-instance*. The certificate allows you to verify that an instance is a *yes-instance*. However if the instance is a *no-instance* then there need not be a certificate to help show this. For example, if a graph does

NOT have a Hamiltonian cycle, it is not clear what "evidence" you could provide to someone that would convince them of this.

The class NP: We define the complexity class NP to be the set of all decision problems Π for which there is a polynomial time algorithm A (called a "proof verifier") with the following two properties:

1. For every *yes-instance* I of Π, there exists some structure S (the "certificate"), such that $A[I, S]$ outputs ACCEPT.
2. For every *no-instance* I of Π, and for every structure S, $A[I, S]$ outputs REJECT.

Note that since A must run in time polynomial in the length of the instance I, this implies a polynomial bound on the size of the structure S.

For example, compositeness is in NP. HAM-CYCLE (given a graph, does it contain a cycle that visits every vertex exactly once) is in NP. The certificate is a permutation of the vertices.

Intuitively, having certificates serves no real practical value, it is merely a mathematical device. It says that if you were capable of guessing the correct answer, then you could verify its correctness. Unfortunately, it doesn't tell you how to go about guessing. A nondeterministic computer is one that can guess answers. Thus, in polynomial time a nondeterministic machine can solve the HAM-CYCLE problem by guessing the path and verifying that it is a Hamiltonian cycle.

Observe that $P \subseteq NP$. In other words, if we can solve a problem in polynomial time, then we can certainly verify correctness in polynomial time. (More formally, we do not even need to see a certificate to solve the problem, we can solve it in polynomial time anyway).

However it is not known whether $P = NP$. It seems unreasonable to think that this should be so. In other words, just being able to verify that you have a correct solution does not help you in finding the actual solution very much. Most experts believe that $P \neq NP$, but no one has a proof of this.

Reductions: The class of NP-complete problems consists of a set of decision problems (a subset of the class NP) that no one knows how to solve efficiently, but if there were a polynomial time solution to even a single NP-complete problem, then every problem in NP would be solvable in polynomial time.

Let's suppose that Q and Q' are two problems. We do not know how to solve either in polynomial time, but we want to argue that if there were a way to solve Q in polynomial time, then there would be a way to solve Q'. In general how can you prove such a result (since you probably cannot solve Q in polynomial time anyway). The idea is a notion borrowed from mathematics called a *reduction*. The idea is that we reduce one problem to the other. Another way to think of this is to show that IF we had a subroutine for solving one problem we could use this subroutine to solve the other problem.

More formally, we say that a decision problem Π_1 is *polynomial-time reducible* to a decision problem Π_2 (written $\Pi_1 \propto \Pi_2$) if there is a polynomial time computable function f that maps instances of Π_1 to instances of Π_2, such that I is a *yes-instance* of Π_1 if and only if $f(I)$ is *yes-instance* of Π_2. In other words, f transforms each *yes-instance* of Π_1 to a *yes-instance* of Π_2 and each *no-instance* of Π_1 to a *no-instance* of Π_2.

Example: Consider the HAM-CYCLE problem and the Traveling Salesperson Problem (TSP). The TSP problem is: Given a complete weighted directed graph $G = (V, E)$ and a cost function $c(u, v)$ = cost of getting from u to v, and an integer k, does there exist a cycle of cost at most k in G that visits every vertex? The cycle need not be a simple cycle (this differs from the text's definition).

We will show that HAM-CYCLE $\propto$ TSP. In other words, we will show that IF we had a polynomial time algorithm for solving the TSP problem, we could use this algorithm to solve the HAM-CYCLE problem.

Here is the idea, let $G = (V, E)$ be the graph on which we want to solve the Hamiltonian cycle problem. We create want to create a complete directed graph G', and an integer k, such that G has a Hamiltonian cycle if and only if G' has a TSP tour of length at most k.

We create G' as follows. The vertex set for G' is the same as the vertex set for G, and the edge set for G' consists of all pairs of edges, and we set the weights:

$$\begin{aligned} c(u,v) &= 1 \qquad \text{if } (u,v) \in E \\ c(u,v) &= 2 \qquad \text{if } (u,v) \notin E. \end{aligned}$$

Finally, we set $k = n$ (where $n = |V|$). Observe that this transformation can be computed in linear time (and hence polynomial time).

Now, observe that if G has a Hamiltonian circuit, then every corresponding edge in G' will have weight 1, and the resulting cycle will have total weight n, thus a TSP tour of cost $k = n$ exists.

Conversely, if G' has a TSP cycle of weight n, this cycle has to visit every vertex, and so must use at least n edges. Since all edges have weight ≥ 1, this path must ONLY use edges of weight 1, and hence must only use edges of G. This tour cannot visit any vertex twice, because if it did it would need to use MORE than n edges, but we know its total cost is at most n.

Thus, by running our (hypothesized) polynomial time TSP algorithm on the graph G', we can determine whether or not G has a Hamiltonian cycle in polynomial time. Thus, TSP $\in P$ implies HAM-CYCLE $\in P$. (In fact both problems are NP-complete, so either claim is doubtful.)

Lecture 20: NP-Completeness: SAT and CLIQUE

Read: Chapt 34 (up through 34.5). Skim the latter half of 34.3 and all of 34.4.

Reductions: Last time we gave the following definition. We say that a decision problem Π_1 is *polynomial-time reducible* to decision problem Π_2 (written $\Pi_1 \propto \Pi_2$) if there is a polynomial time computable function f, such that for all instances I of Π_1, I is a *yes-instance* of Π_1 if and only if $f(I)$ is a *yes-instance* of Π_2.

Now, suppose we had a polynomial time algorithm for solving problem Π_2. How could we use this to solve problem Π_1? Let's say we are given an instance I of Π_1. The idea is to run the algorithm f on I giving a new instance $I' = f(I)$. Now we know that I is a *yes-instance* of Π_1 iff I' is a *yes-instance* of Π_2. So we run our polynomial time procedure for Π_2 on I'. If the procedure says "yes" then we know that $f(I)$ is a *yes-instance* of Π_2 which implies that I is a *yes-instance* of Π_1. If the procedure says "no" then we know that $f(I)$ is a *no-instance* of Π_2 which implies that I is *no-instance* of Π_1. In either case we get the answer we want. The whole thing runs in polynomial time, because f runs in polynomial time, and the procedure for Π_2 runs in polynomial time. Thus we have:

Lemma: If $\Pi_1 \propto \Pi_2$ and $\Pi_2 \in P$ then $\Pi_1 \in P$.

One important fact about reducibility is that it is transitive. In other words

Lemma: If $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$ then $\Pi_1 \propto \Pi_3$.

NP-completeness: It follows from the example above that if tomorrow we could show that TSP DID have a polynomial time solution, we could use that to solve HAM-CYCLE. This is the general structure of NP-complete problems. We have a collection of problems that can all be translated to one another by polynomial time reductions. Thus it follows that once one of them has been solved in polynomial time, all of them will be solvable in polynomial time.

Formally a decision problem Π is said to be *NP-complete* if:

(1) $\Pi \in NP$ and

(2) $\Pi' \propto \Pi$ for all $\Pi' \in NP$.

If Π satisfies only property (2) it is said to be *NP-hard*. Generally this means that it is even harder than an NP-complete problem.

An alternative way to show that a problem is NP-complete is to use transitivity.

Lemma: Π is NP-complete if

(1) $\Pi \in NP$ and

(2) $\Pi' \propto \Pi$ for some NP-complete problem Π'.

The reason is that all $\Pi'' \in NP$ are reducible to Π' (since Π' is NP-complete) and hence by transitivity Π'' is reducible to Π, implying that Π is NP-complete. For example, if we KNEW that HAM-CYCLE was NP-complete, the above reduction would imply that TSP is NP-complete (assuming that we could show that TSP is in NP).

Unfortunately, it appears to be almost impossible to prove that ONE problem is NP-complete, because the definition says that we have to be able to reduce EVERY problem in NP to this problem. There are infinitely many such problems, so how can we ever hope to do this? We will talk about this next time with Cook's theorem.

Cook's Theorem: A Boolean formula is a logical formula which consists of variables x_i, logical operations $\overline{x}$ meaning the negation of x, Boolean OR $(x \vee y)$ and Boolean AND $(x \wedge y)$. Given a Boolean formula, we say that it is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1. (As opposed to the case where no matter how you assign truth values the result is ALWAYS 0.)

Example:

$$(x_1 \wedge (x_2 \vee \overline{x_3})) \vee (\overline{x_2} \wedge x_3 \wedge \overline{x_1}).$$

By the assignment $x_1 = 1$, $x_2 = 1$, $x_3 = 0$ we make this formula true, and hence it is satisfiable.

Example:

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2}).$$

There is no assignment that makes this formula true, and hence it is NOT satisfiable.

Let SAT denote the problem of determining whether an input Boolean formula is satisfiable.

Cook's Theorem: SAT is NP complete.

Proof: $SAT \in NP$. To verify that some formula is in SAT, all we have to do is to "guess" a certificate. This is just an assignment of 0 or 1 values for all the variables and verify that the resulting formula evaluates to 1.

For all $\Pi \in NP$, $\Pi \propto SAT$. This is the hard part. The entire proof is long, but the idea is fairly simple. Since $\Pi \in NP$, there must be nondeterministic (guessing) computer that solves the problem Π. Deep down inside, every computer is just a collection of circuits, whose function can be described by a boolean formula. Cook (laboriously) figured out exactly how to set up a (very long) formula which exactly simulates the behavior of any such computer. (Cook used a very simple model of computation called a Turing machine.) The fact that the program runs in polynomial time, implied that Cook's formula was of polynomial size. Cook's formula was satisfiable if and only if the program ran in polynomial time and accepted its input.

In fact, it turns out that an even more restricted version of the Satisfiability problem, called 3-CNF-SAT is NP-complete.

A *literal* is a variable or its negation x or $\overline{x}$. A formula is in *3-CNF* (3 Conjuctive normal form) if it is the AND of clauses where each clause is the OR of 3 literals.

Example:

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_1} \vee x_4).$$

It turns out that 3-CNF-SAT, the problem of determining whether a formula in 3-CNF form is satisfiable, is also NP-complete.

NOTE: The problem of 2-CNF-SAT is solvable in polynomial time, so the 3 IS significant.

NP-completeness proofs: Now that we know that 3-CNF-SAT is NP-complete, we can use this fact to prove that other problems are NP-complete. We will start with the CLIQUE problem.

CLIQUE: Given an undirected graph $G = (V, E)$ and an integer k does G contain a completely connected subgraph with k vertices.

Claim: CLIQUE is NP-complete.

(CLIQUE $\in$ NP): Given G and k, to verify that G has a clique of size k we simply exhibit the k vertices that form the clique as the certificate. Given this certificate it is easy to verify that they form a clique.

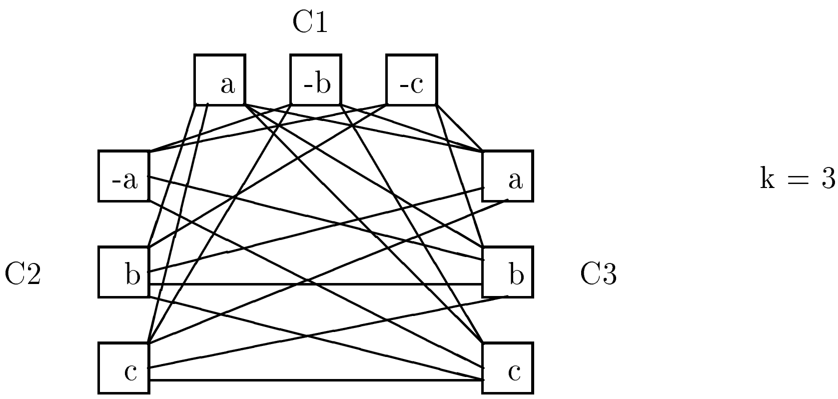
3-CNF-SAT $\propto$ CLIQUE: Given a Boolean formula F in 3-CNF form, we create an undirected graph as follows. Suppose that the graph has m clauses. For each clause, e.g. $(x_1 \vee \overline{x_2} \vee x_3)$ we create three new vertices, and label them x_1, $\overline{x_2}$, and x_3, respectively (note that if a given literal occurs in many clauses, there will be as many vertices having this same label).

We create the following edges. For every pair of literals l_i and l_j IN DIFFERENT CLAUSES, we connect them as long as $l_i \neq \overline{l_j}$. (i.e. as long as they are logically consistent). Let the resulting graph be G and let $k = m$ (the number of clauses). Output G and k.

Example: Suppose that we are given the 3-CNF formula:

$$(a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \wedge (a \vee b \vee c).$$

Note: This has the satsifying assignment $a = 0, b = 0, c = 1$, but our reduction will make NO use of this knowledge. (Remember it takes exponential time to derive this knowledge and our reduction must run in polynomial time.)



Note there does exist a 3-clique in this graph (in fact many). For example, there is a clique connecting the vertices $\overline{b}$ in C_1, with c in C_2 with c in C_3. Observe that in our earlier assignment all three of these literals are true.

We claim that F is satisfiable if and only if G has a clique of size k. If F is satisfiable, then each clause must have at least one true literal. By selecting one such literal from each clause, these vertices will form a clique in G (since they are all in different clauses and must all be consistent so they will all be connected).

Conversely, if G has a clique of size k. Observe that first that there are no two edges between vertices in the same clause, so for ANY clique in G we cannot use two vertices in the same clause, and hence if G has a clique of size k it must arise from connecting one vertex from each clause.

Second notice that all of literals corresponding to the vertices in these clauses are consistent with one another (i.e. we NEVER have a vertex labelled x and a vertex labelled $\overline{x}$ in the clique). Thus we can assign a truth value to make every one of these literals true. Thus, F is satisfiable.

Observe that our transformation did NOT attempt to solve the clique problem nor to solve the Satisfiability problem. Also observe that the transformation had NO knowledge of the solution to either problem. It did NOT know which variables should be set to 1 for satisfiability, it did NOT know which vertices will be in the clique. This is because computing these things would require exponential time (by the best known algorithms). We simply translated one into an equivalent instance of the other, mapping over the corresponding elements.

Lecture 21: NP-Completeness: Vertex Cover, Subset Sum

Read: Chapt 34 (up through 34.5).

Recap: Last time we gave a reduction from 3-SAT to CLIQUE. In other words, we converted a Boolean formula F into a graph G and integer k, such that F is satisfiable if and only if G has a clique of size k.

There are a couple things to emphasize about the construction. First, observe that the transformation did NOT attempt to solve the clique problem nor to solve the Satisfiability problem. It is a DETERMINISTIC, POLYNOMIAL time procedure, and hence does NOT have time to figure out what the clique is and what the satisfying assignment is.

However, in our proof of the correctness of the transformation, we hypothesized that either a clique existed or that there was a satisfying assignment. Keep these two things separate.

The transformation function only translates, without knowledge of the solutions. The proof of equivalence, makes use of the hypothesis that there exists a solution.

Another note, in NP-completeness proofs, you ALWAYS reduce the known NP-complete problem to the candidate problem that you are trying to prove is NP-complete. Not the other way around (which is more intuitive).

Today we give a few more examples.

Vertex Cover: A vertex cover in an undirected graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that every edge in G has at least one endpoint in V'. The vertex cover problem (VC) is: Given an undirected graph G and an integer k, does G have a vertex cover of size k?

Example: See page 949.

Claim: Vertex cover is NP-complete.

VC $\in$ NP: If a given graph G has a vertex cover of size k, then this can be verified in polynomial time. The certificate consists of the vertices in the vertex cover. It is easy to verify that every edge has one of these vertices as an endpoint.

CLIQUE $\propto$ VC: We want to reduce the known NP-complete problem CLIQUE to our problem VC. In other words, if we had a polynomial time subroutine for VC, then we could solve the known NP-complete problem CLIQUE in polynomial time.

Given a input for CLIQUE, (G, k), we want to produce an input for VC, (G', k'), such that G has a CLIQUE of size k if and only if G' has a vertex cover of size k'. (Thus if we could solve the VC problem, by applying our translation we could solve the CLIQUE problem.)

Intuition: The key idea is to think about an alternate notion of what it means for a set of vertices V' to be a vertex cover. Another way of thinking about this, is that for all the remaining vertices in the graph, $V - V'$, there is NO edge between two of these vertices. We know that to have a clique there must be an edge between EVERY pair of edges, and these are just opposites of one another. Define the *complement* of a graph G to be the graph $\overline{G} = (V, \overline{E})$ where

$$\overline{E} = \{(u, v) \mid (u, v) \notin E\}.$$

It follows that a subset V' is a vertex cover if and only if $V - V'$ is completely DISconnected in G if and only if $V - V'$ is completely CONNECTED in $\overline{G}$. Thus, a graph G has a vertex cover of size k, if and only if the complement graph $\overline{G}$ has a clique of size $|V| - k$.

This gives us the transformation. Given an instance of the clique problem (G, k) we compute $G' = \overline{G}$ and $k' = |V| - k$ and output (G', k'). This is computable in polynomial time. From the earlier comments it follows that the construction is correct.

Subset Sum: The subset sum problem is the following. Given a finite set S of positive numbers, and a *target* t, we want to know whether there exists a subset $S' \subseteq S$ which sums exactly to t.

This problem is a simplification of the knapsack problem. Suppose that in the knapsack problem the value of an item is equal to its weight. (Each item is worth its weight in gold?) The best possible solution is to exactly fill the knapsack, that is, is there a subset of objects whose total weight sums to the knapsack capacity W?

Example: $S = \{1, 3, 6, 8, 12, 15, 23, 32\}$ and $t = 19$. The subset $S' = \{1, 6, 12\}$ sums to t, so the answer is yes. If $t = 5$ the answer is no.

We showed that there is an dynamic programming solution to the knapsack problem that runs in $O(n \cdot W)$ time, but this is not a polynomial function of the input size. In particular, if we assume that each number is less than W and is represented in base 2, then the size of the input is $O(n \lg W)$. However, W is exponentially larger than $\lg W$. We will show in fact that the subset sum problem is NP-complete (implying that the Knapsack problem is NP-hard).

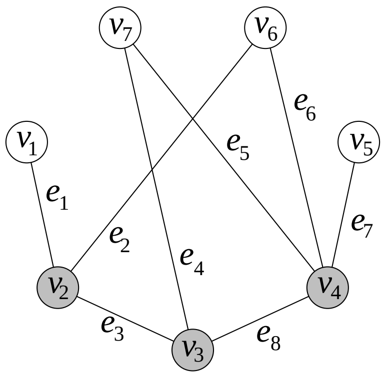
Claim: Subset sum (SS) is NP-complete.

SS $\in$ NP: Given S and t, the certificate is the subset S'. In polynomial time we can compute the sum of elements in S', and verify that it equals t.

VC $\propto$ SS: We want a polynomial time computable function f that maps an instance of the vertex cover (a graph G and integer k) to an instance of the the subset sum problem (set of integers S and target integer t) such that G has a vertex cover of size k if and only if S has a subset summing to t. Thus, if subset sum were solvable in polynomial time, so would vertex cover.

How can we encode the notion of selecting a subset of vertices that cover all the edges to that of selecting a subset of numbers that sums to t? In the vertex cover problem we are selecting vertices, and in the subset sum problem we are selecting numbers, so it seems logical that the reduction should map vertices into numbers. The constraint that these vertices should cover all the edges must be mapped to the constraint that the sum of the numbers should equal the target value.

Here is an idea, which does not work, but gives a sense of how to proceed. First number the edges of the graph 1 through e. Then represent each vertex v_i as a bit vector, where the j-th bit is set to 1 if and only if the edge e_j is incident to vertex v_i. Thus, these bit vectors, form the rows of an *incidence matrix* for the graph. An example is shown below, in which $k = 3$.



	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
v_1	1	0	0	0	0	0	0	0
v_2	1	1	1	0	0	0	0	0
v_3	0	0	1	1	0	0	0	1
v_4	0	0	0	0	1	1	1	1
v_5	0	0	0	0	0	0	1	0
v_6	0	1	0	0	0	1	0	0
v_7	0	0	0	1	1	0	0	0
$v_2 \vee v_3 \vee v_4$	1	1	1	1	1	1	1	1

Figure 28: Vertex cover and incidence matrix.

Now, suppose we take any subset of vertices and form the logical-or of the corresponding bit vectors. If the subset is a vertex cover, then every edge will be covered by at least one of these vertices, and so the logical or will be a bit vector of all 1's, $1111\ldots1$. Conversely, if the logical-or is a bit vector of 1's, then each edge has been covered by some vertex, implying that the vertices form a vertex cover. (Later we will consider how to encode the fact that there only allowed k vertices in the cover.)

Since bit vectors can be thought of as just a way of representing numbers in binary, this is starting to feel more like the subset sum problem. The target would be the number whose bit vector is all 1's. The problems are that logical-or is not the same as addition and we have no way of controlling how many vertices go into the vertex cover. We would like the addition process to simulate this logical-or reduction.

There are two ways in which addition differs significantly from logical-or. The first is the issue of carries. For example, the $1101 \vee 0011 = 1111$, but in binary $1101 + 0011 = 1000$.

To fix this, we recognize that we do not have to use a binary (base-2) representation. In fact, we can assume any base system we want. Observe that each column of the incidence matrix has at most two 1's in any column, because each edge is incident to at most two vertices. Thus, if use any base that is at least as large as base 3, we will never generate a carry to the next position. In fact we will use base 4 (for reasons to be seen below). Note that the base of the number system is just for own convenience of notation. Once the numbers have been formed, they will be converted into whatever form our machine assumes for its input representation, e.g. decimal.

The second problem is that an edge may generally be covered either once or twice in the vertex cover (if one or both of its endpoints is in the cover). So, the final sum of these numbers will be a number consisting of 1 and 2 digits, e.g. $1211 \ldots 112$. This does not provide us with a unique target value t. However, we know that no digit of our number should be a zero. What we will do will be to create e *slack values*. Each slack number will be of the form 00000100000 (that is, it will have a single 1 digit in it). Our target will be the number $2222 \ldots 222$ (all 2's). From our vertex cover we will get a number consisting of 1's and 2's, and then we will supplement this by adding in enough slack values to convert all the 1's to 2's.

There is one last issue. We are only allowed to place k vertices in the vertex cover. We will handle this by adding an additional column. For each vertex we will put a 1 in this column. For each slack variable we will put a 0. In the target, we will require that this column sum to the value k, the size of the vertex cover. Thus, to form the desired sum, we may only select k values from the vertex values.

Here is the final reduction. Given a graph G and integer k we create a series of n vertex values, $x_1, x_2, \ldots, x_n$ using base-4 notation. The value x_i is equal a 1 followed by e base-4 digits. The j-th digit is a 1 if edge e_j is incident to vertex v_i and 0 otherwise. Next create e slack values $y_1, y_2, \ldots, y_e$, where y_i is a 0 followed by e base-4 digits. The i-th digit of y_i is 1 and all others are 0. Finally let t be the base-4 number whose first digit is k and whose remaining e digits are all 2. (Note that if $k \geq 3$ then the first digit is not a legal base-4 value. We can let this most significant digit be as large as we like, because it cannot carry-over into any other positions.) Convert the x_i's y_j's and t to decimal notation. Output the set $S = \{x_1, \ldots, x_n, y_1, \ldots, y_e\}$ and t. Observe that all of this can be done in polynomial time. This is illustrated below.

We claim that G has a vertex cover of size k if and only if S has a subset that sums to t. If G has a vertex cover of size k, then we take the vertex values x_i corresponding to the vertices of the cover, and augment as necessary with slack values to form t. So S has a subset that sums to t. Conversely, if S has a subset that sums to t then we assert that it must select exactly k values from the vertex values (since the first digit must sum to k). These will form a vertex cover, since no column (edge) can be uncovered if we are to reach a total of 2 in each column.

It is worth noting that in this reduction, we needed to have large numbers. For example, the target value t is at least as large as $4^e \geq 4^n$ (assuming more edges than vertices). In our dynamic programming solution $W = t$, so our DP algorithm would run in $\Omega(n4^n)$ time, which is not polynomial time.

Lecture 22: Approximation Algorithms

Read: Chapt 35 (up through 35.2).

Approximation Algorithms: With NP-completeness we have seen that there are many important optimization problems that are likely to be quite hard to solve exactly. Since these are

		e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8
x_1	1	1	0	0	0	0	0	0	0
x_2	1	1	1	1	0	0	0	0	0
x_3	1	0	0	1	1	0	0	0	1
x_4	1	0	0	0	0	1	1	1	1
x_5	1	0	0	0	0	0	0	1	0
x_6	1	0	1	0	0	0	1	0	0
x_7	1	0	0	0	1	1	0	0	0
y_1	0	1	0	0	0	0	0	0	0
y_2	0	0	1	0	0	0	0	0	0
y_3	0	0	0	1	0	0	0	0	0
y_4	0	0	0	0	1	0	0	0	0
y_5	0	0	0	0	0	1	0	0	0
y_6	0	0	0	0	0	0	1	0	0
y_7	0	0	0	0	0	0	0	1	0
y_8	0	0	0	0	0	0	0	0	1
t	3	2	2	2	2	2	2	2	2

Vertex values
(take those in vertex cover)

Slack values
(take any to reach target)

vertex cover size

Figure 29: Vertex cover to subset sum reduction.

important problems, we cannot simply give up at this point, since people do need solutions to these problems. How do we cope with NP-completeness:

Use an exponential time algorithm: Even on the fastest parallel computers this approach is viable only for the smallest instances of these problems. There is little hope that even with improvements in technology, exponential problems of size much bigger than 100 can be expected to be solved, and certainly none of size 1,000.

General Optimization Methods: There are a number of techniques for solving very general combinatorial optimization problems. These go under names such as *branch-and-bound*, *simulated annealing*, *genetic algorithms*, and in fact *neural nets* can be used to solve combinatorial optimization problems. It usually cannot be proved that these methods operate in polynomial time, but for some optimization problems, they have been known to work fairly well. For others they do not do as well. You can learn about these methods in AI courses or courses on optimization theory (see the business school).

Approximation Algorithms: If we limit our attention to methods that have guaranteed polynomial time efficiency, and to prove that the solutions given are *close* to the optimal solutions, even though they may not be optimal. Currently many of the best algorithms for solving complex optimization problems consist of starting with the result of a good approximation algorithm, improved with the use of simulated annealing or other optimization method.

Performance Bounds: Most NP-complete problems have been stated as decision problems for theoretical reasons. However underlying most of these problems is a natural optimization problem. For example, the TSP optimization problem is to find the simple cycle of minimum cost in a digraph, the VC optimization problem is to find the vertex cover of minimum size, the CLIQUE optimization problem is to find the clique of maximum size. Note that sometimes we are minimizing and sometimes we are maximizing. An approximation algorithm is one that returns a legitimate answer, but not necessarily one of the smallest size.

How do we measure how good an approximation algorithm is? We say that an approximation algorithm has a *ratio bound* of $\rho(n)$ for any input of size n, if C is the cost of the solution produced by the approximation algorithm, and C^* is the cost of the optimal solution then

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

(We will assume that costs are always strictly positive quantities.) Observe that $\rho(n)$ is always greater than or equal to 1, and it is equal to 1 if and only if the approximate solution is the true optimum solution.

Sometimes it is more meaningful to measure the relative error. An approximation algorithm has the *relative error bound* $\epsilon(n)$ if

$$\frac{|C - C^*|}{C^*} \leq \epsilon(n).$$

Observe that $\epsilon(n) \leq \rho(n) - 1$. Sometimes the ratio does not depend on n, in which case we may just drop the n writing ρ or ϵ.

Different optimization problem (even different NP-complete optimization problems) have different properties with respect to approximations. Some can be approximated arbitrarily close in polynomial time. That is for any fixed $\epsilon > 0$ we can find a polynomial time algorithm whose relative error ratio is ϵ (but of course as ϵ goes to zero the running time approaches. Such an algorithm is called a *polynomial-time approximation scheme*. The running time might be something like $O(2^{1/\epsilon} n^2)$, so as ϵ gets smaller then running time is worse and worse.

If the running time depends only on a polynomial function of $1/\epsilon$ then it is called a *fully polynomial-time approximation scheme*. For example, a running time like $O((1/\epsilon)^2 n^3)$ would be such an example.

However, in many cases we cannot get better than a fixed constant factor. For example, some problems we no of no approximation algorithm that gives a better ratio than $\rho = 3/2$, no matter how much (polynomial) time we spend. Finally there are some problems where we cannot even guarantee a constant factor. E.g. the best ratio bound is $\rho(n) \in O(\log n)$, so for large n the performance gets worse and worse. In some cases it is known that finding an approximation to within a constant factor is itself an NP-complete problem.

Vertex Cover: We begin by showing that there is an approximation algorithm for vertex cover with a ratio bound of 2, that is, this algorithm will be guaranteed to find a vertex cover whose size is at most twice that of the optimal. Recall that the vertex cover problem seeks to find a set of vertices such that every edge in the graph touches one of these vertices. One quick observation to make is that for every edge (u, v) in the graph, one of its two vertices MUST be in the cover. The idea of this algorithm is to simply grab edges one by one and put BOTH vertices in our vertex cover (since we don't know which is the correct one, we pay the extra factor of 2 by putting both in). We then remove all edges that have been covered.

```
proc Approx-VC(G=(V,E)) {
    C = empty-set;
    E' = E;
    while E' is nonempty do {
        let (u,v) be any edge of E';   (*)
        add u and v to C;
        remove from E' all edges incident to u or v;
    }
    return(C);
}
```

Lecture 23: Approximations: VC and TSP

Read: Chapt 35 (up through 35.2).

Vertex Cover: We begin by showing that there is an approximation algorithm for vertex cover with a ratio bound of 2, that is, this algorithm will be guaranteed to find a vertex cover whose size is at most twice that of the optimal. Recall that the vertex cover problem seeks to find a set of vertices such that every edge in the graph touches one of these vertices. One quick observation to make is that for every edge (u, v) in the graph, one of its two vertices MUST be in the cover. The idea of this algorithm is to simply grab edges one by one and put BOTH vertices in our vertex cover (since we don't know which is the correct one, we pay the extra factor of 2 by putting both in). We then remove all edges that have been covered.

```
proc Approx-VC(G=(V,E)) {
    C = empty-set;
    E' = E;
    while E' is nonempty do {
        let (u,v) be any edge of E';   (*)
        add u and v to C;
        remove from E' all edges incident to u or v;
    }
```

```
    return(C);
}
```

Claim: Approx-VC has ratio bound of 2.

Proof: Consider the set C output by Approx-VC. Let C^* be the optimum VC. Let A be the set of edges selected by the line marked with `(*)` above. Observe that the size of C is exactly $2|A|$ because we add two vertices for each such edge. However note that in the optimum VC one of these two vertices MUST have been added to the VC, and thus the size of C^* is at least $|A|$. Thus we have:

$$\frac{|C|}{2} = |A| \leq |C^*| \qquad \Rightarrow \qquad \frac{|C|}{|C^*|} \leq 2.$$

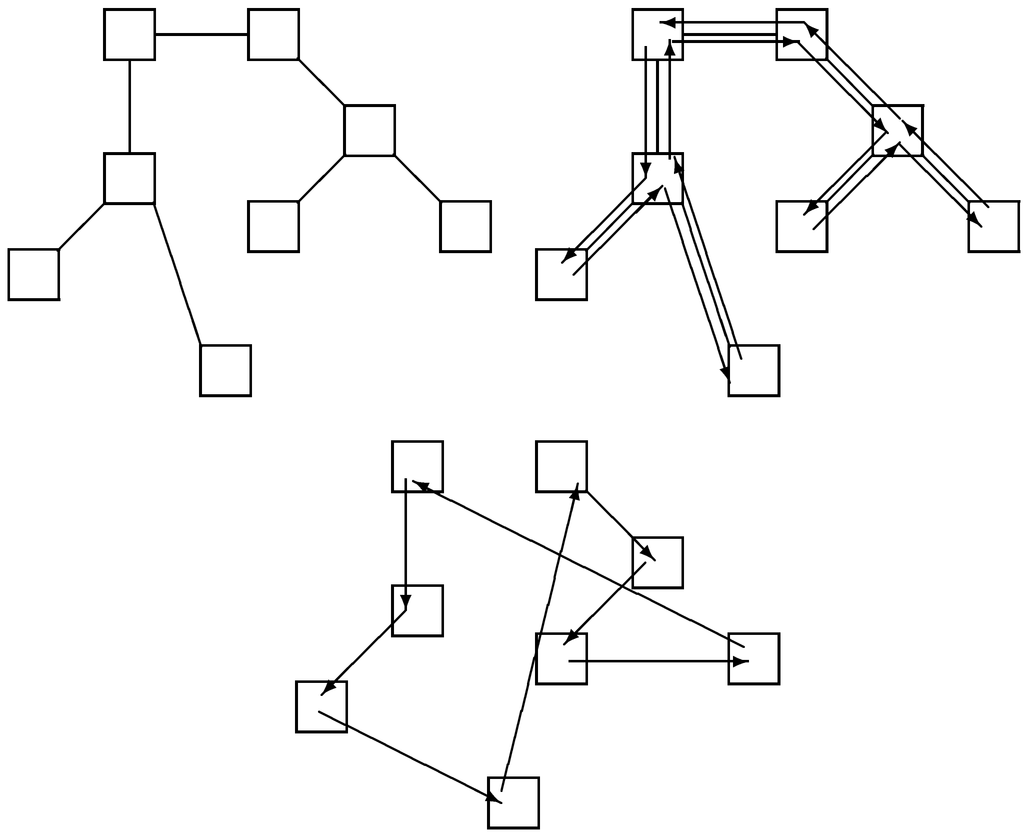
Traveling-Salesperson Approximation: Recall that we are given a complete undirected graph with nonnegative integer edge weights, and we want to find a Hamiltonian cycle (tour) of minimum cost. Given a set of edges A forming a tour we define $c(A)$ to be the sum of edge weights in A.

For most of the applications of TSP, it is true that it never helps to take an indirect route, that is, it is always better to go from u to v along the edge (u, v) than it is to go along a path. This condition is called the *triangle inequality* (because of its similarity to a famous theorem from geometry) and can be stated as follows: for all $u, v, w \in V$

$$c(u, w) \leq c(u, v) + c(v, w).$$

When the underlying cost function satisfies the triangle inequality there is a approximation algorithm for TSP with a ratio-bound of 2. The idea is to observe that a TSP (with one edge removed) is a spanning tree, and hence its cost cannot be smaller than the minimum spanning tree. However, given the minimum spanning tree (which we know is easy to compute) we can use it to form a TSP tour of twice its cost.

Here is how the algorithm works. Given any free tree there is a tour of the tree called a *twice around tour* that traverses the edges of the tree twice, once in each direction. See the figure below. This path is not simple because it revisits vertices, but we can make it simple by *short-cutting*, that is, we skip over previously visited vertices. The triangle inequality assures us that this will not increase the length of the path.



Here is the algorithm.

```
proc Approx-TSP(G=(V,E)) {
    compute a minimum spanning tree T for G;
    designate any vertex r to be the root of the tree;
    let L be the list of vertices visited by a preorder walk ot T;
    return(the Hamiltonian cycle that comes by visiting vertices
        in the order given by L);
}
```

Claim: Approx-TSP has a ratio bound of 2.

Proof: Let H denote the tour produced by this algorithm and let H^* be the optimum tour. Let T be the minimum spanning tree. As we said before, since we can remove any edge of H^* resulting in a spanning tree, and since T is the minimum cost spanning tree we have

$$c(T) \leq c(H^*).$$

Now observe that the twice around tour of T has cost $2c(T)$, since every edge in T is hit twice. By the triangle inequality, when we short-cut an edge of T to form H we do not increase the cost of the tour, and so we have

$$c(H) \leq 2c(T).$$

Combining these we have

$$\frac{c(H)}{2} \leq c(T) \leq c(H^*) \qquad \Rightarrow \qquad \frac{c(H)}{c(H^*)} \leq 2.$$

TSP Without Triangle Inequality: If we do not have the triangle inequality then it is not possible to achieve an approximation to TSP, unless the unlikely event that $P = NP$. To see this, we show that if there existed a polynomial time approximation algorithm for TSP that could guarantee a fixed ratio bound ρ then it would follow that $P = NP$.

To do this we reduce the Hamiltonian Cycle problem (HC) to this approximation problem by showing that a polynomial time solution to the approximation problem would imply a polynomial time solution to HC. (Note that the approximation problem is not "formally" an NP-complete problem because it is not a decision problem. In this case we say that it is NP-hard.)

Here is the reduction. Given a graph $G = (V, E)$ for which we want to determine the existence of a Hamiltonian cycle, construct a complete graph G' as follows. Let ρ be the ratio bound that our approximation algorithm guarantees. For each edge $(u, v) \in E$ we give the edge (u, v) the weight 1 in G', and if $(u, v) \notin E$ we give (u, v) the weight $\rho|V| + 1$. Now, call the approximation TSP algorithm on G'. If it returns a TSP of total weight less than or equal to $\rho|V|$ then we claim the original graph has a Hamiltonian cycle, and otherwise we claim the original graph does not have a Hamiltonian cycle.

Why is this? If the original graph G did contain a Hamiltonian cycle then the cost of the corresponding path in G' is $|V|$. Since the TSP approximation algorithm is required to find a simple cycle that visits all vertices of cost no more than ρ times the optimum, it must return a path with cost at most $\rho|V|$.

On the other hand, if the graph G' does not contain a Hamiltonian cycle, then any TSP tour that visits every vertex in G' must use at least one edge that was NOT in the original graph, and such an edge would have weight $\rho|V| + 1$. Since all the other edge weights are positive, the total cost of the final path is strictly bigger than $\rho|V|$, and so the approximation algorithm will return a value bigger than $\rho|V|$.

Lecture 24: Approximations: Set Cover

Read: Chapt 35, up to 35.3.

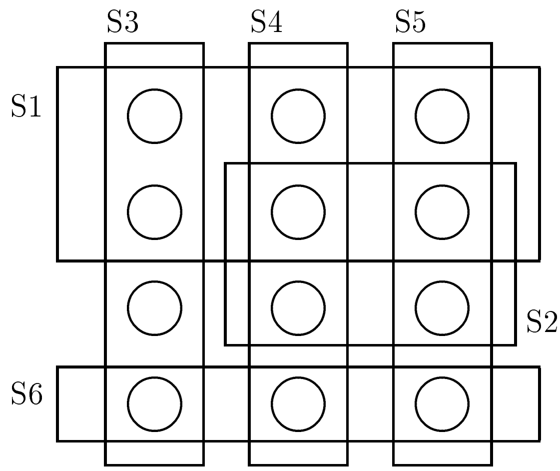
Set Cover: Set cover is an important optimization problem that is used to model resource-selection problems. It is a generalization of the vertex cover problem, and so is NP-complete. It is a very important problem in many applications, so there is a need for good approximations. We shall see that, unlike vertex cover and TSP (with triangle inequality) where we were able to achieve constant performace bounds, for this algorithm the best we can do is achieve a performance bound that grows slowly as a function of n.

The other interesting aspect of the argument is that it is an example of a clever combinatorial argument, that is interesting in its own right.

An instance of the problem consists of a pair (X, F) where X is a finite set (a domain of elements) and F is a family of subsets of X, such that every element of X belongs to at least one set of F. We say that a subset S of F *covers* its elements. The problem is to find the minimum-sized subset C of F (note that we are talking about a subset of sets here) such that

$$X = \cup_{S \in C} S.$$

We say that such a subset C *covers* X. Consider the following example. The optimum set cover consists of the three sets $\{S_3, S_4, S_5\}$.



Greedy Set Cover: A simple greedy approach to set cover works by at each stage selecting the set that covers the greatest number of "uncovered" elements.

```
proc Greedy-Set-Cover(X, F) {
    U = X;
    C = empty;
    while (U != empty) {
        select S in F that maximizes | S intersect U |;
        U = U - {S};
        add S to C;
    }
    return C;
}
```

For the example given earlier the greedy-set cover algorithm would select S_1 (since it covers 6 out of 12 elements), then S_6 (since it covers 3 out of the remaining 6), then S_2 (since it covers 2 of the remaining 3) and finally S_3.

How does it perform? It would be nice if we could argue that the greedy set cover algorithm produces a number of sets that is a constant factor from the optimum, but it turns out that this is not the case. (And counterexample exist.) However, we can show that this algorithm will never produce a solution that is more than a factor of $O(\log n)$ from the optimum solution where $n = |X|$ (in fact, more precisely $O(\log |S|)$ where S is the largest set in F).

The proof is based on a very clever combinatorial argument. The idea of many combinatorial arguments is to allocate counts according to one scheme and then combine these counts according to a different scheme. The proof that the greedy set cover is at most a log-factor of optimum uses the greedy set cover as one means to allocate counts, and then uses the optimum set cover as the other means to combine these counts.

Claim: Greedy set cover has the ratio bound $O(\log n)$ where n is the size of the largest set in F.

Proof: Let C denote the cover generated by the greedy-set cover algorithm and let C^* denote the optimum cover. Let S_i denote the i-th subset selected by the greedy algorithm. We want to count the number of elements in C, and we do this by a clever counting scheme. Whenever an element S_i is added to the greedy cover, we spread the cost of 1 that it incurs evenly among all the new elements that this set covers. For example, if the set covers 5 new elements (that have not been previously covered) then we assign each of these elements a cost of 1/5. Each element will be assigned a cost only ONCE (because it is only assigned a cost the first time it is covered). Let c_x be the cost assigned to element $x \in X$.

Lets let $SS_i = (S_1 \cup S_2 \cup \ldots \cup S_i)$. These are all the elements that have been covered by the i-th stage of the greedy algorithm. The set of newly covered elements is $S_i - SS_{i-1}$. Thus if x is first covered by S_i then

$$c_x = \frac{1}{|S_i - SS_{i-1}|}.$$

Since each set in C is assigned a total cost of 1, if we sum all the costs of all the elements we have

$$|C| = \sum_{x \in X} c_x.$$

Since C^* is a cover for X, it follows that

$$|C| = \sum_{x \in X} c_x \leq \sum_{S \in C^*} \sum_{x \in S} c_x.$$

The remainder of the proof will show that for any set S

$$\sum_{x \in S} c_x = O(\log n).$$

From this it will follow that $|C| \leq |C^*| \cdot O(\log n)$ from which we will have $|C|/|C^*| \in O(\log n)$, which will complete the proof.

SubClaim: For each $S \in F$

$$\sum_{x \in S} c_x = O(\log n),$$

where n is the size of S.

Proof: We make use of one important fact, that we have not mentioned since the beginning of the course. That is:

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} \in O(\log n).$$

$H(0) = 0$. This is called the Harmonic series, and it shows up in an important way in this proof.

Let $u_i = |S - SS_i|$ denote the number of elements of S that have NOT yet been covered by the i-th stage. Let $u_0 = |S|$. Let k be the least index such that $u_k = 0$. (In other words, by stage k all the elements of S have been covered.) Note that $u_0, u_1, \ldots, u_k$ is a nonincreasing sequence, and $u_{i-1} - u_i$ is the number of newly covered elements at stage i. If we add up all the costs assigned to the elements of S we have

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \frac{1}{|S_i - SS_{i-1}|}.$$

Because the greedy algorithm always selects the set that maximizes the number of newly covered elements, we know that

$$|S_i - SS_{i-1}| \geq |S - SS_{i-1}| = u_{i-1}.$$

Consequently we have

$$\sum_{x \in S} c_x \le \sum_{i=1}^{k} \frac{u_{i-1} - u_i}{u_{i-1}}.$$

If you write out the elements of this sum it can be varified that the elements are less than the corresponding elements of the Harmonic series. To see this, observe first that if $a < b$ then

$$H(b) - H(a) = \sum_{i=a+1}^{b} \frac{1}{i} \ge \frac{b-a}{b}.$$

Substituting u_{i-1} and u_i for b and a we have

$$\begin{aligned} \sum_{x \in S} c_x &\le \sum_{i=1}^{k} H(u_{i-1}) - H(u_i) \\ &= H(u_0) - H(u_k) \\ &= H(|S|) - U(0) \\ &= H(|S|) = H(n) \in O(\log n). \end{aligned}$$