CMSC 451 (NTU 520B): Design and Analysis of Computer Algorithms[1]
Fall 1999
Dave Mount

Lecture 1: Course Introduction

(Thursday, Sep 2, 1999)
Reading: Chapter 1 in CLR (Cormen, Leiserson, and Rivest).
Professor Carl Smith reviewed material from Chapter 1 in CLR.

Lecture 2: Asymptotics and Summations

(Tuesday, Sep 7, 1999)
Read: Review Chapters 1, 2, and 3 in CLR (Cormen, Leiserson, and Rivest).

What is an algorithm? Our text defines an *algorithm* to be any well-defined computational procedure that takes some values as *input* and produces some values as *output*. Like a cooking recipe, an algorithm provides a step-by-step method for solving a computational problem. Unlike programs, algorithms are not dependent on a particular programming language, machine, system, or compiler. They are mathematical entities, which can be thought of as running on some sort of *idealized computer* with an infinite random access memory and an unlimited word size. Algorithm design is all about the mathematical theory behind the design of good programs.

Why study algorithm design? Programming is a very complex task. There are a number of aspects of programming that make it so complex. The first is that most programming projects are very large, requiring the coordinated efforts of many people. (This is the topic a course like CMSC 435 in software engineering.) The next is that many programming projects involve storing and accessing large quantities of data efficiently. (This is the topic of courses on data structures and databases like CMSC 420 and 424.) The last is that many programming projects involve solving complex computational problems, for which simplistic or naive solutions may not be efficient enough. The complex problems may involve numerical data (the subject of courses on numerical analysis, like CMSC 466), but often they involve discrete data. This is where the topic of algorithm design and analysis is important.

Although the algorithms discussed in this course will often represent only a tiny fraction of the code that is generated in a large software system, this small fraction may be very important for the success of the overall project. An unfortunately common approach to this problem is to first design an inefficient algorithm and data structure to solve the problem, and then take this poor design and attempt to fine-tune its performance. The problem is that if the underlying design is bad, then often no amount of fine-tuning is going to make a substantial difference.

The focus of this course is on how to design good algorithms, and how to analyze their efficiency. We will study a number of different techniques for designing algorithms (divide-and-conquer, dynamic programming, depth-first search), and apply them to a number of different problems.

An understanding of good design techniques is critical to being able to good programs. In addition, it is important to be able to quickly analyze the running times of these designs (without expensive prototyping and testing). We will begin with a review of the analysis techniques, which were covered in the prerequisite course, CMSC 251. See Chapters 1-3 of CLR for more information.

Asymptotics: The formulas that are derived for the running times of program may often be quite complex. When designing algorithms, the main purpose of the analysis is to get a sense for the trend in the algorithm's running time. (An exact analysis is probably best done by implementing the algorithm and measuring CPU seconds.) We would like a simple way of representing complex functions, which captures the essential growth rate properties. This is the purpose of *asymptotics*.

Asymptotic analysis is based on two simplifying assumptions, which hold in most (but not all) cases. But it is important to understand these assumptions and the limitations of asymptotic analysis.

Large input sizes: We are most interested in how the running time grows for large values of n.

Ignore constant factors: The actual running time of the program depends on various constant factors in the implementation (coding tricks, optimizations in compilation, speed of the underlying hardware, etc). Therefore, we will ignore constant factors.

The justification for considering large n is that if n is small, then almost any algorithm is fast enough. People are most concerned about running times for large inputs. For the most part, these assumptions are reasonable when making comparisons between functions that have significantly different behaviors. For example, suppose we have two programs, one whose running time is $T_1(n) = n^3$ and another whose running time is $T_2(n) = 100n$. (The latter algorithm may be faster because it uses a more sophisticated and complex algorithm, and the added sophistication results in a larger constant factor.) For small n (e.g., $n \leq 10$) the first algorithm is the faster of the two. But as n becomes larger the relative differences in running time become much greater. Assuming one million operations per second.

n	$T_1(n)$	$T_2(n)$	$T_1(n)/T_2(n)$
10	0.001 sec	0.001 sec	1
100	1 sec	0.01 sec	100
1000	17 min	0.1 sec	10,000
10,000	11.6 days	1 sec	1,000,000

The clear lesson is that as input sizes grow, the performance of the asymptotically poorer algorithm degrades much more rapidly.

These assumptions are not always reasonable. For example, in any particular application, n is a fixed value. It may be the case that one function is smaller than another asymptotically, but for your value of n, the asymptotically larger value is fine. Most of the algorithms that we will study this semester will have both low constants and low asymptotic running times, so we will not need to worry about these issues.

To represent the running times of algorithms in a simpler form, we use *asymptotic notation*, which essentially represents a function by its fastest growing term and ignores constant factors. See Chapter 2 in CLR for the formal "c and n_0" definitions. However, for our purposes, the following definitions based on limits is much easier to apply, and holds for virtually all functions that arise as running times. (For strange functions where these limits do not exist, you should use the formal definitions instead.)

Let $f(n)$ and $g(n)$ be two functions, which we will assume to be positive. Suppose we want to assert that $f(n)$ and $g(n)$ grow at roughly the same rates for large n (ignoring constant factors). This would be equivalent to saying

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,$$

where c is some nonzero constant (not 0 and not ∞). In asymptotic notation we write $f(n) \in \Theta(g(n))$. Intuitively it means that $f(n)$ and $g(n)$ are asymptotically equivalent. Suppose we want to assert that $f(n)$ does not grow significantly faster than $g(n)$. Then the ratio $f(n)/g(n)$ should either approach a constant (they are equivalent) or 0 (if $g(n)$ grows faster than $f(n)$). In this case we say $f(n) \in O(g(n))$. (Our text uses $=$ rather than $\in$, but $O(g(n))$ is best thought of as a set of functions.) Here are the complete definitions:

Asymptotic Form	Relationship	Definition
$f(n) \in \Theta(g(n))$	$f(n) \equiv g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$
$f(n) \in O(g(n))$	$f(n) \preceq g(n)$	$0 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$
$f(n) \in \Omega(g(n))$	$f(n) \succeq g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}.$
$f(n) \in o(g(n))$	$f(n) \prec g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$
$f(n) \in \omega(g(n))$	$f(n) \succ g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$

For example $T(n) = (n^3 + 3n^2 + 2n)/6 \in \Theta(n^3)$ because

$$\lim_{n\to\infty} \frac{T(n)}{n^3} = \lim_{n\to\infty} \frac{(n^3 + 3n^2 + 2n)/6}{n^3} = \lim_{n\to\infty} \left(\frac{1}{6} + \frac{1}{2n} + \frac{1}{3n^2}\right) = \frac{1}{6},$$

and $0 < 1/6 < \infty$. (Note that it also follows that $T(n) \in O(n^3)$, and $T(n) \in \Omega(n^3)$.) Indeed, this is consistent with the informal notion of asymptotic notation, of ignoring the constant factor, and considering large values of n (since the largest power of n will dominate for large n). When dealing with limits, the following rule is nice to keep in mind.

L'Hôpital's rule: If $f(n)$ and $g(n)$ both approach 0 or both approach ∞ in the limit, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)},$$

where $f'(n)$ and $g'(n)$ denote the derivatives of f and g relative to n.

Some of the trickier asymptotic comparisons to make are those that involve exponentials and logarithms. Here are some simple rules of thumb to keep in mind.

Constants: Multiplicative and additive constants may be ignored. When constants appear in exponents or as the base of an exponential, they are significant. Thus $2n \equiv 3n$, but $n^2 \prec n^3$ and $2^n \prec 3^n$.

Logarithm base: Logarithms in different bases differ only by a constant factor, thus $\log_2 n \equiv \log_3 n$. Thus, we will often not specify logarithm bases inside asymptotic notation, as in $O(n \log n)$, since they do not matter.

Logs and powers: Remember that you can pull an exponent out of a logarithm as a multiplicative factor. Thus $n \log_2(n^2) = 2n \log_2 n \equiv n \log_2 n$.

Exponents and logs: Remember that exponentials and logs cancel one another. Thus $2^{\lg n} = n$.

Logs and polynomials: For any $a, b \geq 0$, $(\log n)^a \prec n^b$. (That is, logs grow more slowly than any polynomial.)

Polynomials and exponentials: For any $a \geq 0, b > 1$, $n^a \prec b^n$. (That is, polynomials grow more slowly than any exponential.)

Summations: There are some particularly important summations, which you should probably commit to memory (or at least remember their asymptotic growth rates). These are analogous to the basic formulas of integral calculus, and have a way of cropping up over and over again.

Constant Series: For integers a and b,

$$\sum_{i=a}^{b} 1 = \begin{cases} (b-a+1) & \text{if } b \geq a-1 \\ 0 & \text{otherwise.} \end{cases}$$

Notice that when $b = a - 1$, there are no terms in the summation (since the index is assumed to count upwards only), and the result is 0. Be careful to check that $b \geq a - 1$ before applying this rule.

Arithmetic Series: For $n \geq 0$,

$$\sum_{i=0}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

This is $\Theta(n^2)$.

Geometric Series: Let $x \neq 1$ be any constant (independent of n), then for $n \geq 0$,

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}.$$

If $0 < x < 1$ then this is $\Theta(1)$. If $x > 1$, then this is $\Theta(x^n)$, that is, the entire sum is proportional to the last element of the series.

Here are some more obscure ones, which come up from time to time. Not all of them are listed in CLR. A good source is the appendix in the book "The Analysis of Algorithms" by P. W. Purdom and C. A. Brown.

Quadratic Series: For $n \geq 0$,

$$\sum_{i=0}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{2n^3 + 3n^2 + n}{6}.$$

Linear-geometric Series: This arises in some algorithms based on trees and recursion. Let $x \neq 1$ be any constant, then for $n \geq 0$,

$$\sum_{i=0}^{n-1} i x^i = x + 2x^2 + 3x^3 \cdots + nx^n = \frac{(n-1)x^{(n+1)} - nx^n + x}{(x-1)^2}.$$

(What happens in the case where $x = 1$?) As n becomes large, this is dominated by the term $(n-1)x^{(n+1)}/(x-1)^2$. The multiplicative term $n - 1$ is very nearly equal to n for large n, and, since x is a constant, we may multiply this times the constant $(x-1)^2/x$ without changing the asymptotics. What remains is $\Theta(nx^n)$.

Harmonic Series: This arises often in probabilistic analyses of algorithms. It does not have an exact closed form solution, but it can be closely approximated. For $n \geq 0$,

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n.$$

There are also a few tips to learn about solving summations.

Summations with general bounds: When a summation does not start at the 1 or 0, as most of the above formulas assume, you can just split it up into the difference of two summations. For example, for $1 \leq a \leq b$

$$\sum_{i=a}^{b} f(i) \;=\; \sum_{i=0}^{b} f(i) - \sum_{i=0}^{a-1} f(i).$$

Approximate using integrals: Integration and summation are closely related. (Integration is in some sense a continuous form of summation.) Here is a handy formula. Let $f(x)$ be any *monotonically increasing function* (the function increases as x increases).

$$\int_0^n f(x)dx \;\leq\; \sum_{i=1}^{n} f(i) \;\leq\; \int_1^{n+1} f(x)dx.$$

Let us consider a simple example. Let $A[1..4n]$ be some array.

Sample Program Fragment

```
for i = n to 2n do {
    for j = 1 to i do {
        if (A[j] <= A[2j]) output "hello";
    }
}
```

In the worst case, how many times is the "hello" line printed as a function of n? In the worst case, the elements of A are in ascending order, implying that every time through the loop the string is output. Let $T(n)$ denote the number of times that the string is output. We can set up one nested summation for each nested loop, and then use the above rules to solve them.

$$T(n) = \sum_{i=n}^{2n} \sum_{j=1}^{i} 1.$$

The "1" is due to the fact that the string is output once for each time through the inner loop. Solving these from the inside out, we see that the last summation is a constant sum, and hence

$$T(n) = \sum_{i=n}^{2n} (i - 1 + 1) = \sum_{i=n}^{2n} i.$$

This is just an arithmetic series, with general bounds, which we can break into the difference of two arithmetic series, starting from 0.

$$\begin{aligned} T(n) &= \sum_{i=0}^{2n} i - \sum_{i=0}^{n-1} i \\ &= \frac{2n(2n+1)}{2} - \frac{n(n-1)}{2} \;=\; \frac{(4n^2+2n)-(n^2-n)}{2} \\ &= \frac{3}{2}(n^2+n) \;\in\; \Theta(n^2). \end{aligned}$$

At this point, it is a good idea to go back and test your solution for a few small values of n (e.g. $n = 0, 1, 2$) just to double-check that you have not made any math errors.

Lecture 3: Divide-and-Conquer and Recurrences

(Thursday, Sep 9, 1999)

Read: Chapter 4 from CLR. Constructive induction is not discussed in CLR.

Divide-and-Conquer Algorithms: An important approach to algorithm design is based on *divide-and-conquer*. Such an algorithm consists of three basic steps: divide, conquer, and combine. The most common example (described in Chapt 1 of CLR) is that of Mergesort. To sort a list of numbers you first split the list into two sublists of roughly equal size, sort each sublist, and then merge the sorted lists into a single sorted list.

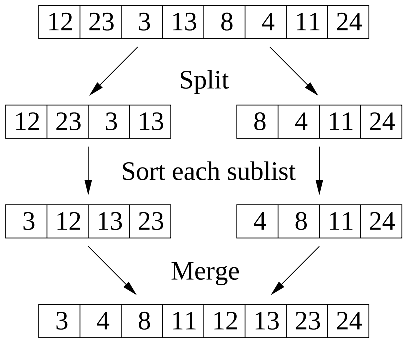


Figure 1: Mergesort Example.

Divide: Split the original problem of size n into a (typically 2, but maybe more) problems of roughly equal sizes, say n/b.

Conquer: Solve each subproblem recursively.

Combine: Combine the solutions to the subproblems to a solution to the original problem. The time to combine the solutions is called the *overhead*. We will assume that the running time of the overhead is some polynomial of n, say cn^k, for constants c and k.

This recursive subdivision is repeated until the size of the subproblems is small enough that the problem can be solved by brute-force.

For example, in Mergesort, we subdivide each problem into $a = 2$ parts, each part of size $n/2$ (implying that $b = 2$). Two sorted lists of size $n/2$ can be merged into a single sorted list of size n in $\Theta(n)$ time (thus $c = k = 1$). This is but one example. There are many more.

Analysis: How long does a divide-and-conquer algorithm take to run? Let $T(n)$ be the function that describes the running time of the algorithm on a subarray of length $n \geq 1$. For simplicity, let us assume that n is a power of 2. (The general analysis comes about by considering floors and ceilings, and is quite a bit messier. See CLR for some explanation.)

As a basis, when n is 1, the algorithm runs in constant time, namely $\Theta(1)$. Since we are ignoring constant factors, we can just say that $T(1) = 1$. Otherwise, if $n > 1$, then it splits the list into two sublists, each of size $n/2$, and makes two recursive calls on these arrays, each taking $T(n/2)$ time. (Do you see why?) As mentioned above, merging can be done in $\Theta(n)$ time, which we will just express as n (since we are ignoring constants). So the overall running time is described by the following *recurrence*, which is defined for all $n \geq 1$, where n is a power of 2:

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

This is is very well known recurrence. Let's consider some general methods for solving recurrences. See CLR for more methods.

Solving Recurrences by The Master Theorem: There are a number of methods for solving the sort of recurrences that show up in divide-and-conquer algorithms. The easiest method is to apply the *Master Theorem* that is given in CLR. Here is a slightly more restrictive version, but adequate for a lot of instances. See CLR for the more complete version of the Master Theorem.

Theorem: (Simplified Master Theorem) Let $a \geq 1$, $b > 1$ be constants and let $T(n)$ be the recurrence

$$T(n) = aT(n/b) + cn^k,$$

defined for $n \geq 0$.

Case (1): $a > b^k$ then $T(n)$ is $\Theta(n^{\log_b a})$.

Case (2): $a = b^k$ then $T(n)$ is $\Theta(n^k \log n)$.

Case (3): $a < b^k$ then $T(n)$ is $\Theta(n^k)$.

Using this version of the Master Theorem we can see that in our recurrence $a = 2$, $b = 2$, and $k = 1$, so $a = b^k$ and case (2) applies. Thus $T(n)$ is $\Theta(n \log n)$.

There many recurrences that cannot be put into this form. For example, the following recurrence is quite common: $T(n) = 2T(n/2) + n \log n$. This solves to $T(n) = \Theta(n \log^2 n)$, but the Master Theorem (either this form or the one in CLR will not tell you this.) For such recurrences, other methods are needed.

Expansion: A more basic method for solving recurrences is that of *expansion* (which CLR calls *iteration*). This is a rather painstaking process of repeatedly applying the definition of the recurrence until (hopefully) a simple pattern emerges. This pattern usually results in a summation that is easy to solve. If you look at the proof in CLR for the Master Theorem, it is actually based on expansion.

Let us consider applying this to the following recurrence. We assume that n is a power of 3.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T\left(\frac{n}{3}\right) + n \qquad \text{if } n > 1 \end{aligned}$$

First we expand the recurrence into a summation, until seeing the general pattern emerge.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{3}\right) + n \\ &= 2\left(2T\left(\frac{n}{9}\right) + \frac{n}{3}\right) + n \;=\; 4T\left(\frac{n}{9}\right) + \left(n + \frac{2n}{3}\right) \\ &= 4\left(2T\left(\frac{n}{27}\right) + \frac{n}{9}\right) + \left(n + \frac{2n}{3}\right) \;=\; 8T\left(\frac{n}{27}\right) + \left(n + \frac{2n}{3} + \frac{4n}{9}\right) \\ &\;\vdots \\ &= 2^k T\left(\frac{n}{3^k}\right) + \sum_{i=0}^{k-1} \frac{2^i n}{3^i} \;=\; 2^k T\left(\frac{n}{3^k}\right) + n \sum_{i=0}^{k-1} (2/3)^i. \end{aligned}$$

The parameter k is the number of expansions (not to be confused with the value of k we introduced earlier on the overhead). We want to know how many expansions are needed to arrive at the basis case. To do this we set $n/(3^k) = 1$, meaning that $k = \log_3 n$. Substituting this in and using the identity $a^{\log b} = b^{\log a}$ we have:

$$T(n) = 2^{\log_3 n} T(1) + n \sum_{i=0}^{\log_3 n - 1} (2/3)^i = n^{\log_3 2} + n \sum_{i=0}^{\log_3 n - 1} (2/3)^i.$$

Next, we can apply the formula for the geometric series and simplify to get:

$$\begin{aligned} T(n) &= n^{\log_3 2} + n \frac{1 - (2/3)^{\log_3 n}}{1 - (2/3)} \\ &= n^{\log_3 2} + 3n(1 - (2/3)^{\log_3 n}) = n^{\log_3 2} + 3n(1 - n^{\log_3 (2/3)}) \\ &= n^{\log_3 2} + 3n(1 - n^{(\log_3 2) - 1}) = n^{\log_3 2} + 3n - 3n^{\log_3 2} \\ &= 3n - 2n^{\log_3 2}. \end{aligned}$$

Since $\log_3 2 \approx 0.631 < 1$, $T(n)$ is dominated by the $3n$ term asymptotically, and so it is $\Theta(n)$.

Induction and Constructive Induction: Another technique for solving recurrences (and this works for summations as well) is to guess the solution, or the general form of the solution, and then attempt to verify its correctness through induction. Sometimes there are parameters whose values you do not know. This is fine. In the course of the induction proof, you will usually find out what these values must be. We will consider a famous example, that of the *Fibonacci numbers.*

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{for } n \geq 2. \end{aligned}$$

The Fibonacci numbers arise in data structure design. If you study AVL, or height balanced, trees in CMSC 420, you will learn that the minimum-sized AVL trees are produced by the recursive construction given below. Let $L(i)$ denote the number of leaves in the minimum-sized AVL tree of height i. To construct a minimum-sized AVL tree of height i, you create a root node whose children consist of a minimum-sized AVL tree of heights $i-1$ and $i-2$. Thus the number of leaves obeys $L(0) = L(1) = 1$, $L(i) = L(i-1) + L(i-2)$. It is easy to see that $L(i) = F_{i+1}$.

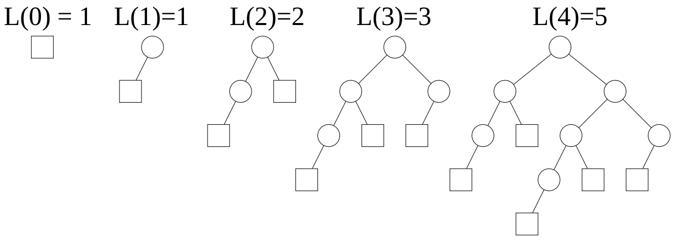


Figure 2: Minimum-sized AVL trees.

If you expand the Fibonacci series for a number of terms, you will observe that F_n appears to grow exponentially, but not as fast as 2^n. It is tempting to conjecture that $F_n \leq \phi^{n-1}$, for some real parameter ϕ, where $1 < \phi < 2$. We can use induction to prove this and derive a bound on ϕ.

Lemma: For all integers $n \geq 1$, $F_n \leq \phi^{n-1}$ for some constant ϕ, $1 < \phi < 2$.

Proof: We will try to derive the tightest bound we can on the value of ϕ.

Basis: For the basis cases we consider $n = 1$. Observe that $F_1 = 1 \leq \phi^0$, as desired.

Induction step: For the induction step, let us assume that $F_m \leq \phi^{m-1}$ whenever $1 \leq m < n$. Using this *induction hypothesis* we will show that the lemma holds for n itself, whenever $n \geq 2$.

Since $n \geq 2$, we have $F_n = F_{n-1} + F_{n-2}$. Now, since $n-1$ and $n-2$ are both strictly less than n, we can apply the induction hypothesis, from which we have

$$F_n \leq \phi^{n-2} + \phi^{n-3} \;=\; \phi^{n-3}(1+\phi).$$

We want to show that this is at most ϕ^{n-1} (for a suitable choice of ϕ). Clearly this will be true if and only if $(1+\phi) \leq \phi^2$. This is not true for all values of ϕ (for example it is not true when $\phi = 1$ but it is true when $\phi = 2$.)

At the critical value of ϕ this inequality will be an equality, implying that we want to find the roots of the equation

$$\phi^2 - \phi - 1 = 0.$$

By the quadratic formula we have

$$\phi \;=\; \frac{1 \pm \sqrt{1+4}}{2} \;=\; \frac{1 \pm \sqrt{5}}{2}.$$

Since $\sqrt{5} \approx 2.24$, observe that one of the roots is negative, and hence would not be a possible candidate for ϕ. The positive root is

$$\phi \;=\; \frac{1+\sqrt{5}}{2} \approx 1.618.$$

There is a very subtle bug in the preceding proof. Can you spot it? The error occurs in the case $n = 2$. Here we claim that $F_2 = F_1 + F_0$ and then we apply the induction hypothesis to both F_1 and F_0. But the induction hypothesis only applies for $m \geq 1$, and hence cannot be applied to F_0! To fix it we could include F_2 as part of the basis case as well.

Notice not only did we prove the lemma by induction, but we actually determined the value of ϕ which makes the lemma true. This is why this method is called *constructive induction*.

By the way, the value $\phi = \frac{1}{2}(1+\sqrt{5})$ is a famous constant in mathematics, architecture and art. It is the *golden ratio*. Two numbers A and B satisfy the golden ratio if

$$\frac{A}{B} = \frac{A+B}{A}.$$

It is easy to verify that $A = \phi$ and $B = 1$ satisfies this condition. This proportion occurs throughout the world of art and architecture.

Lecture 4: Sorting: Review

(Tuesday, Sep 14, 1999)

Read: Review Chapts. 7 and 8 and read Chapt. 9 in CLR.

Review of Sorting: Sorting is among the most basic computational problems in algorithm design. We are given a sequence of items, each associated with a given *key value*. The problem is to permute the items so that they are in increasing order by key. Sorting is important in algorithm because it is often the first step in some more complex algorithm, as an initial stage in organizing data for faster subsequent retrieval.

Sorting algorithms are usually divided into two classes, *internal sorting algorithms*, which assume that data is stored in an array, and *external sorting algorithm*, which assume that data is stored on tape or disk and can only be accessed sequentially. You are probably familiar with the standard simple $\Theta(n^2)$ sorting algorithms, such as *Insertion-sort* and *Bubblesort*. The efficient $\Theta(n \log n)$ sorting algorithms include as *Mergesort*, *Quicksort*, and *Heapsort*. The quadratic time algorithms are actually faster than the more complex $\Theta(n \log n)$ algorithms for small inputs, say less than about 20 keys. Among the slow sorting algorithms, insertion-sort has a reputation as being the better choice.

Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

In-place: The algorithm uses no additional array storage, and hence it is possible to sort very large lists without the need to allocate additional arrays.

Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

Here is a quick summary of the $\Theta(n \log n)$ algorithms. If you are not familiar with any of these, check out the descriptions in CLR.

Quicksort: It works recursively, by first selecting a random "pivot value" from the array. Then it partitions the array into elements that are less than and greater than the pivot. Then it recursively sorts each part.

Quicksort is widely regarded as the *fastest* of the fast sorting algorithms (on modern machines). One explanation is that its inner loop compares elements against a single pivot value, which can be stored in a register for fast access. The other algorithms compare two elements in the array. This is considered an *in-place* sorting algorithm, since it uses no other array storage. (It does implicitly use the system's recursion stack, but this is usually not counted.) It is *not stable*.

This algorithm is $\Theta(n \log n)$ in the *expected case*, and $\Theta(n^2)$ 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 also works recursively. It is a classical divide-and-conquer algorithm. The array is split into two subarrays of roughly equal size. They are sorted recursively. Then the two sorted subarrays are merged together in $\Theta(n)$ time.

Mergesort is the only *stable* sorting algorithm of these three. The downside is the Mergesort is the only algorithm of the three that requires additional array storage (ignoring the recursion stack), and thus it is *not in-place*. This is because the merging process merges the two arrays into a third array. Although it is possible to merge arrays in-place, it cannot be done in $\Theta(n)$ time.

Heapsort: Heapsort is based on a nice data structure, called a *heap*, which is an efficient implementation of a priority queue data structure. A priority queue supports the operations of inserting a key, and deleting the element with the smallest key value. A heap can be built for n keys in $\Theta(n)$ time, and the minimum key can be extracted in $\Theta(\log n)$ time. Heapsort is an *in-place* sorting algorithm, but it is *not stable*.

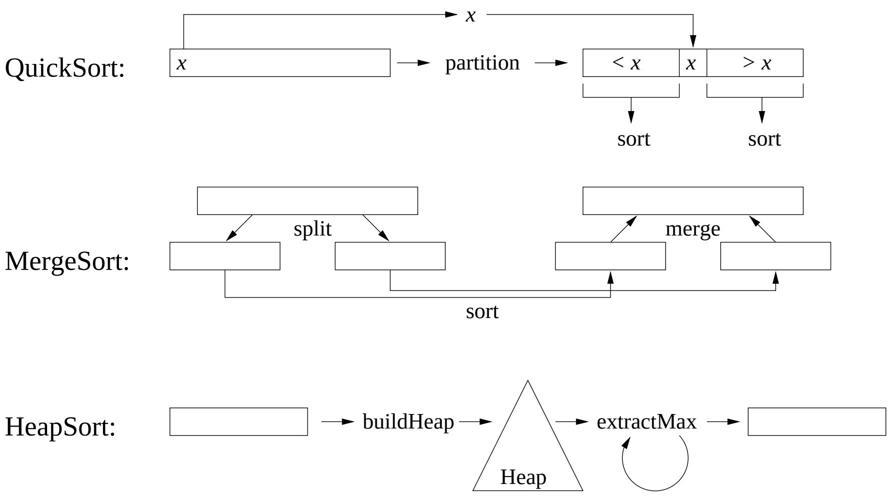


Figure 3: Common Sorting Algorithms.

Heapsort works by building the heap (ordered in reverse order so that the maximum can be extracted efficiently) and then repeatedly extracting the largest element. (Why it extracts the maximum rather than the minimum is an implementation detail, but this is the key to making this work as an in-place sorting algorithm.)

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 (key value) can be processed in $O(\log n)$ time.

Lower Bounds for Comparison-Based Sorting: The fact that $O(n \log n)$ sorting algorithms are the fastest around for many years, suggests that this may be the best that we can do. Can we sort in $o(n \log n)$ time (recall that this means asymptotic strictly faster $n \log n$ time)? We will give an argument that no sorting algorithm based on comparison can be faster than this.

Comparison-based Sorting Algorithm: The manner in which the algorithm permutes the elements is based solely on the results of the comparisons that the algorithm makes between the elements to be sorted.

Most general-purpose sorting algorithms are comparison-based (as are all the algorithms discussed above). We will see that exceptions exist in special cases. This does not preclude the possibility of sorting algorithms whose actions are determined by other operations.

We will show that any comparison-based sorting algorithm for a 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 and an input size n, can be viewed abstractly through a structure called a *decision tree*. We think of the execution of a sorting algorithm as a path from the root to some leaf in this tree. Here is the definition of a decision tree.

Internal node: Each internal node of the decision tree represents a comparison made in the algorithm (e.g. $a_4 : a_7$). The two branches represent $\leq$ or $>$, respectively. All input sequences in which $a_4 \leq a_7$ continue down the left branch and those with $a_4 > a_7$ continue along the right branch.

Leaf node: Each leaf node corresponds to a point in the algorithm where all the comparisons have been made. By definition of a comparison-based algorithm, the algorithms "action" (the permutation it generates) is completely determined at this point. The depth of a leaf node (distance from the root) is the number of decisions made so far. Certainly the running time of the algorithm must be at least this large.

Given any comparison-based sorting algorithm (e.g. Mergesort) and given n, it is a straightforward (but tedious) exercise to convert the algorithm into an equivalent decision tree. We will leave this as an exercise.

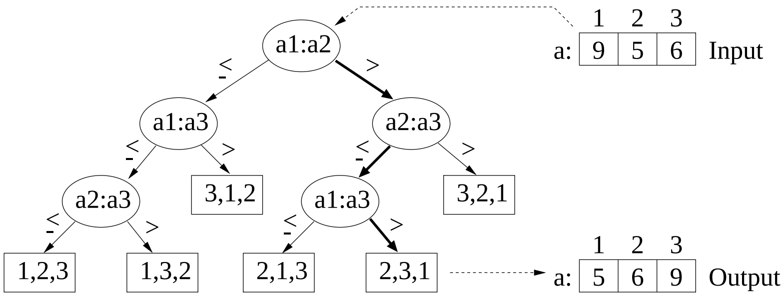


Figure 4: Decision Tree for Sorting.

For the analysis, we will need to use the following basic fact about binary trees. Define the *height* of a binary tree to be the length of the longest path (number of edges) from the root to any leaf.

Lemma: A binary tree with n leaves has height at least $\lg n$.

Proof: Notice that a complete binary tree of height h has 2^h leaves, and this is the largest number of leaves possible for any binary tree with this height. It follows that if n is the number of leaves of some tree of height h, then $n \leq 2^h$, implying that $h \geq \lg n$.

Now, here is our main result.

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

Proof: Consider any sorting algorithm and integer n, and consider the resulting decision tree. Let $T(n)$ denote the number of comparisons this algorithm makes in the worst case, that is, $T(n)$ is equal to the height of the decision tree.

How many leaves must the decision tree have? 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 permute the numbers in an essentially different way. This implies that the number of leaves in the decision tree is at least $n!$, implying by our lemma that the height of the tree is at least $\lg n!$.

We can apply *Stirling's approximation* for $n!$ (see CLR page 35) yielding:

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

This completes the proof.

This can also be generalized to show that the *average-case* time to sort is also $\Omega(n \log n)$.

Linear Time Sorting: This lower bound implies that if we hope to sort numbers faster than in $O(n \log n)$ time, we cannot do it by making comparisons alone. Next we consider the question of whether it is possible to sort without the use of comparisons. They answer is yes, but only under very restrictive circumstances.

Counting Sort: Counting sort assumes that each input is an integer in the range from 1 to k. The algorithm sorts in $\Theta(n+k)$ time. If k is known to be $\Theta(n)$, then this implies that the resulting sorting algorithm is $\Theta(n)$ time.

Define the *rank* of an element in an array to be the number of elements in the array that are less than or equal to it. The basic idea is to determine the rank of every element in the array. Once you know the ranks of the elements, you sort by simply copying each element to the appropriate location of the final sorted output array (but some care is needed if there are duplicates). The question is how to find the rank of an element without comparing it to the other elements of the array? Because the elements are over the range $\{1, 2, \ldots, k\}$, we can maintain an array that counts the number of occurrences of each element. Counting sort uses the following three arrays.

$A[1..n]$: Holds the initial input. $A[j]$ is the entire record, and $A[j].key$ is the integer key value on which to sort.

$B[1..n]$: Array of records which holds the sorted output.

$R[1..k]$: An array of integers. $R[x]$ will contain the rank of x in A, where $x \in [1..k]$.

The algorithm is remarkably simple, but deceptively clever. The algorithm operates by first constructing R. We do this in two steps. First we set $R[x]$ to be the number of elements of $A[j]$ whose key is equal to x. We can do this initializing R to zero, and then for each j, from 1 to n, we increment $R[A[j].key]$ by 1. Thus, if $A[j].key = 5$, then the 5th element of R is incremented, indicating that we have seen one more 5. To determine the number of elements that are less than or equal to x, we replace $R[x]$ with the sum of elements in the subarray $R[1..x]$. This is done by just keeping a running total of the elements of R.

Now $R[x]$ now contains the rank of x. This means that if $x = A[j].key$ then the final position of $A[j]$ should be at position $R[x]$ in the final sorted array. Thus, we set $B[R[x]] = A[j]$. Notice that this copies the entire record, not just the key value. There is a subtlety here however. 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 $R[i]$ after copying.

Counting Sort

```
CountingSort(int n, int k, array A, array B) {  // sort A[1..n] to B[1..n]
    for x = 1 to k do R[x] = 0                  // initialize R
    for j = 1 to n do R[A[j].key]++             // R[x] = #(A[j] == x)
    for x = 2 to k do R[x] += R[x-1]            // R[x] = rank of x
    for j = n downto 1 do {                     // move each element of A to B
        x = A[j].key                            // x = key value
        B[R[x]] = A[j]                          // R[x] is where to put it
        R[x]--                                  // leave space for duplicates
    }
}
```

There are four (unnested) loops, executed k times, n times, $k-1$ times, and n times, respectively, so the total running time is $\Theta(n+k)$ time. If $k = O(n)$, then the total running time is $\Theta(n)$. The figure below shows an example of the algorithm. You should trace through a few examples, to convince yourself how it works.

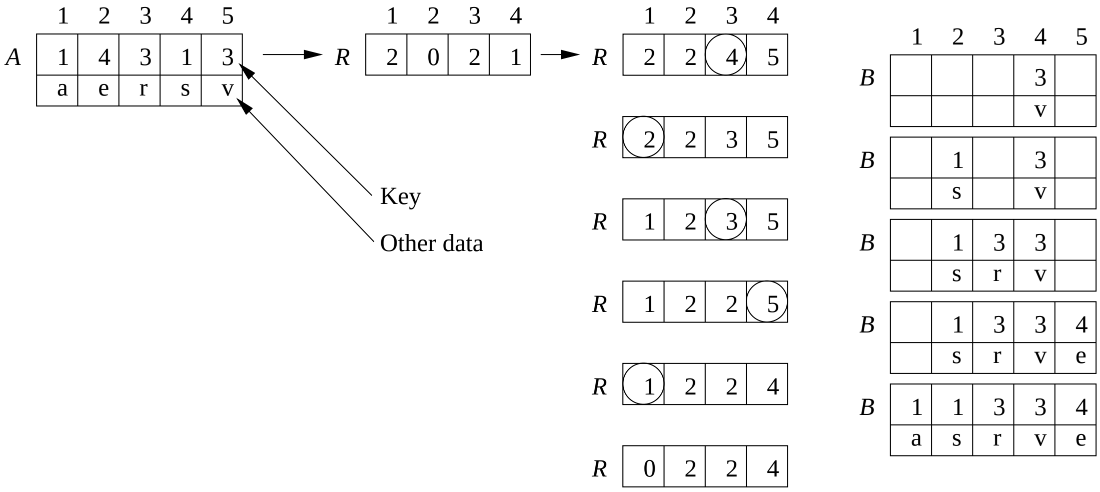


Figure 5: Counting Sort.

Obviously this not an in-place sorting algorithm (we need two additional arrays). However it is a stable sorting algorithm. I'll leave it as an exercise to prove this. (As a hint, notice that the last loop runs down from n to 1. It would not be stable if the loop were running the other way.)

Lecture 5: More on Sorting

(Thursday, Sep 16, 1999)
Read: Chapt. 9 in CLR.

RadixSort: Last time we discussed CountingSort, an $O(n+k)$ time algorithm for sorting n integers in the range from 1 to k. The main shortcoming of CountingSort is that (due to space requirements) it is only practical for a very small ranges of integers. If the integers are in the range from say, 1 to a million, we may not want to allocate an array of a million elements. RadixSort 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 integers, each having d decimal digits (or digits in any base). Let's suppose that we have access to any *stable* sorting algorithm, such as CountingSort. To sort these integers 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.

RadixSort

```
RadixSort(A, d) {
    for i = 1 to d do {
```

```
        Sort A (stably) with respect to i-th lowest order digit;
    }
}
```

Input								Output
576		49[4]		9[5]4		[1]76		176
494		19[4]		5[7]6		[1]94		194
194		95[4]		1[7]6		[2]78		278
296	$\Longrightarrow$	57[6]	$\Longrightarrow$	2[7]8	$\Longrightarrow$	[2]96	$\Longrightarrow$	296
278		29[6]		4[9]4		[4]94		494
176		17[6]		1[9]4		[5]76		576
954		27[8]		2[9]6		[9]54		954

Figure 6: Example of RadixSort.

The running time is $\Theta(d(n+k))$ where d is the number of digits, n is the length of the list, and k is the number of distinct values each digit may have. The value of k is 10 in this example (since we are dealing with decimal digits), but as we can see below, this can be adjusted.

A common application of this algorithm is for sorting integers over some range that is larger than n, but still polynomial in n. For example, suppose that you wanted to sort a list of integers in the range from 1 to n^2. First, you could subtract 1 so that they are now in the range from 0 to $n^2 - 1$. Observe that any number in this range can be expressed as 2-digit number, where each digit is over the range from 0 to $n-1$. In particular, given any integer L in this range, we can write $L = an + b$, where $a = \lfloor L/n \rfloor$ and $b = L \bmod n$. Now, we can think of L as the 2-digit number (a, b). So, we can radix sort these numbers in time $\Theta(2(n+n)) = \Theta(n)$. In general this works to sort any n numbers over the range from 1 to n^d, in $\Theta(dn)$ time.

BucketSort: CountingSort and RadixSort are only good for sorting small integers, or at least objects (like characters) that can be encoded as small integers. What if you want to sort a set of floating-point numbers? In the worst-case you are pretty much stuck with using one of the comparison-based sorting algorithms, such as QuickSort, MergeSort, or HeapSort. However, in special cases where you have reason to believe that your numbers are roughly uniformly distributed over some range, then it is possible to do better.

For example, suppose that you have a set of n floating-point numbers $A[i]$ that are roughly uniformly distributed over the range $[0, 1)$. To say that the values are *uniformly distributed* over this range means that for any interval $[a, b)$, where $0 \le a \le b \le 1$, the probability that an element of A falls within this interval is equal to the width of the interval $b - a$. For example, the probability that an element of A lies between 0.50 and 0.75 is $0.75 - 0.50 = 0.25$. Even the probability of this happening is larger, but only by a constant factor, then the complexity bounds for BucketSort still apply. If you have numbers over a different range, it is not a problem. In $\Theta(n)$ time you can find the maximum and minimum values and scale the constants used in the algorithm appropriately.

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 $n - 1$. We insert the items into the linked list in sorted order (thus, essentially simulating insertion sort). Finally we concatenate all the lists together to form the final sorted list.

BucketSort

```
BucketSort(A, n) {                          // sort A[1..n]
    allocate array B[0..n-1] and initialize each to NULL;
    for i = 1 to n {
        j = floor(n*A[i])                   // j = bucket index for A[i]
        insert A[i] into its sorted position in B[j]
    }
    return concatenation of B[0], B[1], ..., B[n-1];
}
```

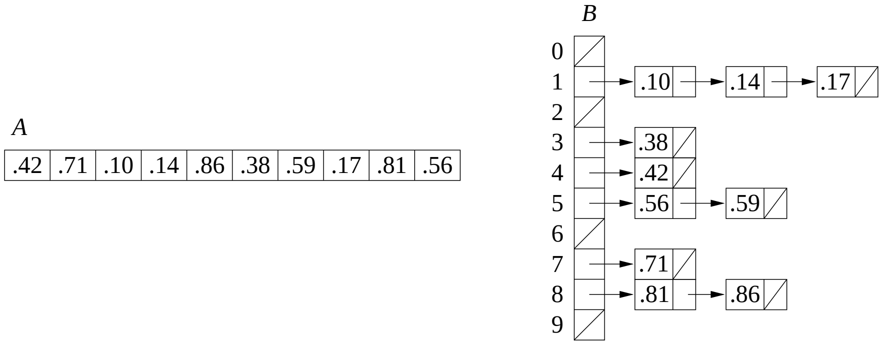


Figure 7: BucketSort.

What is the running time of BucketSort? Let's first consider how much time it takes to handle one of the lists. Let m denote the number of items that have been inserted into any one of these lists. In the worst case, when the ith item is inserted, it must be compared against each of the previous $i-1$ items. Thus the total worst-case insertion time would be

$$T(m) = \sum_{i=1}^{m} i = \frac{m(m+1)}{2} \in \Theta(m^2).$$

Clearly if the items are not uniformly distributed, and all of them fall into one bucket, the performance of the algorithm will be $\Theta(n^2)$, a disaster. But the expected time is much better.

Probabilistic Analysis: Here is a quick-and-dirty analysis. Since there are n buckets, and the items fall uniformly between them, we would expect around a constant number of items per bucket. Thus, the expected insertion time for each bucket is only a constant. Therefore the expected running time of the algorithm is $\Theta(n)$. This quick-and-dirty analysis is probably good enough to convince yourself of this algorithm's basic efficiency. A careful analysis involves understanding a bit about probabilistic analyses of algorithms. Since we haven't done any probabilistic analyses yet, let's try doing this one. (This one is rather typical.)

The first thing to do in a probabilistic analysis is to define a random variable that describes the essential quantity that determines the execution time. A *random variable* can be thought of as real variable that takes on values with certain random values. More formally, it is a function that maps some some sample space onto the reals. For $0 \le i \le n-1$, let X_i denote the random variable that indicates the number of elements assigned to the i-th bucket.

Since the distribution is uniform, all of the random variables X_i have the same probability distribution, so we may as well talk about a single random variable X, which will work for any bucket. As we argued above, the worst-case time to insert X items into a bucket is $\Theta(X^2)$, so what we like to know is the expected value of X^2, denoted $E[X^2]$.

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 probability of $p = 1/n$ of going into the ith bucket. So how many items do we expect will wind up in bucket i? We can analyze this by thinking of each element of A as being represented by a coin flip (with a biased coin, which has a different probability of heads and tails). With probability $p = 1/n$ the number goes into bucket i, which we will interpret as the coin coming up heads. With probability $1 - 1/n$ the item goes into some other bucket, which we will interpret as the coin coming up tails. Since we assume that the elements of A are independent of each other, X is just the total number of heads we see after making n tosses with this (biased) coin.

The number of times that a heads event occurs, given n independent trials in which each trial has two possible outcomes is a well-studied problem in probability theory. Such trials are called *Bernoulli trials* (named after the Swiss mathematician James Bernoulli). If p is the probability of getting a head, then the probability of getting k heads in n tosses is given by the following important formula

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k} \qquad \text{where} \qquad \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Although this looks messy, it is not too hard to see where it comes from. Basically p^k is the probability of tossing k heads, $(1-p)^{n-k}$ is the probability of tossing $n-k$ tails, and $\binom{n}{k}$ is the total number of different ways that the k heads could be distributed among the n tosses. This probability distribution (as a function of k, for a given n and p) is called the *binomial distribution*, and is denoted $b(k; n, p)$.

If you consult a standard textbook on probability and statistics (or look at Section 6.4 in CLR), then you will see the two important facts that we need to know about the binomial distribution. Namely, that its mean value $E[X]$ and its variance $\mathrm{Var}[X]$ are

$$E[X] = np \qquad \text{and} \qquad \mathrm{Var}[X] = E[X^2] - E^2[X] = np(1-p).$$

We want to determine $E[X^2]$. By the above formulas and the fact that $p = 1/n$ we can derive this as

$$E[X^2] \;=\; \mathrm{Var}[X] + E^2[X] \;=\; np(1-p) + (np)^2 \;=\; \frac{n}{n}\left(1 - \frac{1}{n}\right) + \left(\frac{n}{n}\right)^2 \;=\; 2 - \frac{1}{n}.$$

Thus, for large n the time to insert the items into any one of the linked lists is a just shade less than 2. Summing up over all n buckets, gives a total running time of $\Theta(2n) = \Theta(n)$. This is exactly what our quick-and-dirty analysis gave us, but now we know it is true with confidence.

Lecture 6: Dynamic Programming: Longest Common Subsequence

(Tuesday, Sep 21, 1999)
Read: Section 16.3 in CLR.

Dynamic Programming: We begin discussion of an important algorithm design technique, called *dynamic programming* (or DP for short). The technique is among the most powerful for designing algorithms for optimization problems. (This is true for two reasons. Dynamic programming solutions are based on a few common elements. Dynamic programming problems are typically optimization problems (find the minimum or maximum cost solution, subject to various constraints). The technique is related to divide-and-conquer, in the sense that it breaks problems down into smaller problems that it solves recursively. However, because of the somewhat different nature of dynamic programming problems, standard divide-and-conquer solutions are not usually efficient. The basic elements that characterize a dynamic programming algorithm are:

Substructure: Decompose your problem into smaller (and hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems. (Unlike divide-and-conquer problems, it is not usually sufficient to consider one decomposition, but many different ones.)

Table-structure: Store the answers to the subproblems in a table. This is done because (typically) subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem.

Bottom-up computation: Combine solutions on smaller subproblems to solve larger subproblems, and eventually to arrive at a solution to the complete problem. (Our text also discusses a top-down alternative, called *memoization*.)

The most important question in designing a DP solution to a problem is how to set up the subproblem structure. This is called the *formulation* of the problem. Dynamic programming is not applicable to all optimization problems. There are two important elements that a problem must have in order for DP to be applicable.

Optimal substructure: This is sometimes called the *principle of optimality*. It states that for the global problem to be solved optimally, each subproblem should be solved optimally.

Polynomially many subproblems: An important aspect to the efficiency of DP is that the total number of subproblems to be solved should be at most a polynomial number.

Strings: One important area of algorithm design is the study of algorithms for character strings. There are a number of important problems here. Among the most important has to do with efficiently searching for a substring or generally a pattern in large piece of text. (This is what text editors and functions like "grep" do when you perform a search.) In many instances you do not want to find a piece of text exactly, but rather something that is "similar". This arises for example in genetics research. Genetic codes are stored as long DNA molecules. The DNA strands can be broken down into a long sequences each of which is one of four basic types: C, G, T, A. But exact matches rarely occur in biology because of small changes in DNA replication. For this reason, it is of interest to compute similarities between strings that do not match exactly. One common method of measuring the degree of similarity between two strings is to compute their longest common subsequence.

Longest Common Subsequence: Let us think of character strings as sequences of characters. 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 $\langle i_1, i_2, \ldots, i_k \rangle$ ($1 \le i_1 < i_2 < \ldots < i_k \le n$) 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 subsequence 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 $\langle ABC \rangle$ and $\langle BAC \rangle$ is either $\langle AC \rangle$ or $\langle BC \rangle$.

Dynamic Programming Solution: The simple brute-force solution to the problem would be to try all possible subsequences from one string, and search for matches in the other string, but this is hopelessly inefficient, since there are an exponential number of possible subsequences.

Instead, we will derive a dynamic programming solution. In typical DP fashion, 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 will be to compute the longest common subsequence for every possible pair of prefixes. Let $c[i,j]$ denote the length of the longest common subsequence of X_i and Y_j. 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 $x_i = y_j$. For example: Let $X_i = \langle ABCA \rangle$ and let $Y_j = \langle DACA \rangle$. Since both end in A, we claim that the LCS *must* also end in A. (We will leave the proof as an exercise.) 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. (At first you might object: But how did you know that these two A's matched with each other. The answer is that we don't, but it will not make the LCS any smaller if we do.)

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 $c[i,j-1]$. We do not know which is the case, so we try both and take the one that gives us the longer LCS.

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. We concentrate only on computing the maximum *length* of the LCS. Later we will see how to extract the actual sequence. We will store some helpful pointers in a parallel array, $b[0..m, 0..n]$. The code and an example are shown below.

Build LCS Table

```
LCS(char x[1..m], char y[1..n]) {                    // compute LCS table
    int c[0..m, 0..n]
                                                     // initialize column 0
    for i = 0 to m do { c[i,0] = 0; b[i,0] = SKIPX; }
                                                     // initialize row 0
```

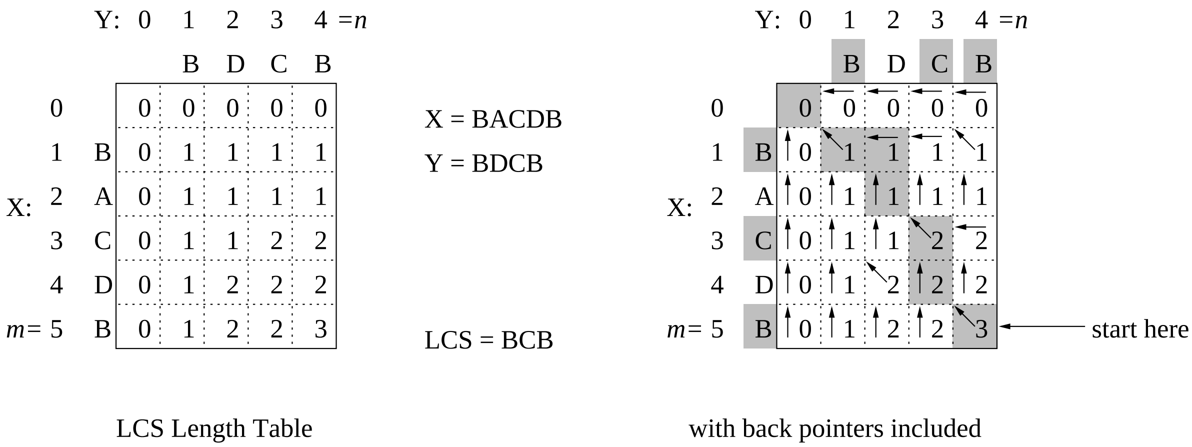


Figure 8: Longest common subsequence example.

```
    for j = 0 to n do { c[0,j] = 0; b[0,j] = SKIPY; }

    for i = 1 to m do {                          // fill rest of table
        for j = 1 to n do {
            if (x[i] == y[j]) {                  // take X[i] (=Y[j]) for LCS
                c[i,j] = c[i-1,j-1]+1;   b[i,j] = addXY;
            }
            else if (c[i-1,j] >= c[i,j-1]) {     // X[i] not in LCS
                c[i,j] = c[i-1,j];       b[i,j] = skipX;
            }
            else {                               // Y[j] not in LCS
                c[i,j] = c[i,j-1];       b[i,j] = skipY;
            }
        }
    }
    return c[m,n];                               // return length of LCS
}
```

Extracting the LCS

```
                                                 // extract the LCS
getLCS(char x[1..m], char y[1..n], int b[0..m,0..n]) {
    LCS = empty string;
    i = m; j = n;                                // start at lower right
    while(i != 0 && j != 0) {                    // go until upper left
        switch b[i,j] {
            case addXY:                          // add X[i] (=Y[j])
                add x[i] (or equivalently y[j]) to front of LCS
                i--;   j--;   break
            case skipX: i--;  break              // skip X[i]
            case skipY: j--;  break              // skip Y[j]
        }
    }
    return LCS
}
```

The running time of the algorithm is clearly $O(mn)$ since there are two nested loops with m and n iterations, respectively. The algorithm also uses $O(mn)$ space.

Extracting the Actual Sequence: Extracting the final LCS is done by using the back pointers stored in $b[0..m, 0..n]$. Intuitively $b[i,j] = add_{XY}$ means that $X[i]$ and $Y[j]$ together form the last character of the LCS. So we take this common character, and continue with entry $b[i-1, j-1]$ to the northwest ($\nwarrow$). If $b[i,j] = skip_X$, then we know that $X[i]$ is not in the LCS, and so we skip it and go to $b[i-1, j]$ above us ($\uparrow$). Similarly, if $b[i,j] = skip_Y$, then we know that $Y[j]$ is not in the LCS, and so we skip it and go to $b[i, j-1]$ to the left ($\leftarrow$). Following these back pointers, and outputting a character with each diagonal move gives the final subsequence.

Lecture 7: Dynamic Programming: Chain Matrix Multiplication

(Thursday, Sep 23, 1999)
Read: Section 16.1 of CLR.

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

$$A_1 A_2 \ldots A_n$$

Matrix multiplication is an associative but not a commutative operation. 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. A $p \times q$ matrix has p rows and q 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 $1 \le i \le p$ and $1 \le j \le r$,

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

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 $p \cdot q \cdot r$.

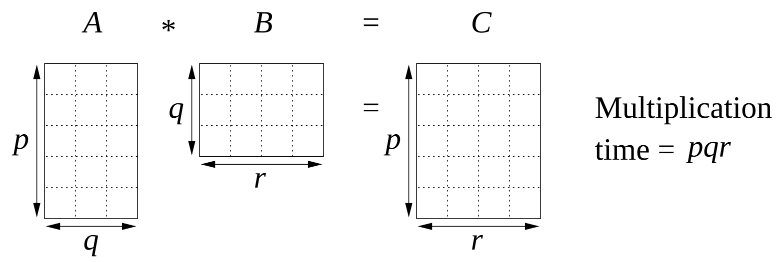


Figure 9: Matrix Multiplication.

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.

$$\begin{aligned}\text{multCost}[((A_1A_2)A_3)] &= (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180, \\ \text{multCost}[(A_1(A_2A_3))] &= (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88.\end{aligned}$$

Even for this small example, considerable savings can be achieved by reordering the evaluation sequence.

Chain Matrix Multiplication Problem: 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 order of multiplication (say, as a evaluation tree) that minimizes the number of operations.

Important Note: This 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 very large. If you have just one item, then there is only one way to parenthesize. If you have n items, then there are $n-1$ places where you could break the list with the outermost pair of parentheses, namely just after the 1st item, just after the 2nd item, etc., and just after the $(n-1)$st item. When we split just after the kth item, we create two sublists to be parenthesized, one with k items, and the other with $n-k$ items. Then we could consider all the ways of parenthesizing these. Since these are independent choices, if there are L ways to parenthesize the left sublist and R ways to parenthesize the right sublist, then the total is $L \cdot R$. This suggests the following recurrence for $P(n)$, the number of different ways of parenthesizing n items:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

This is related to a famous function in combinatorics called the *Catalan numbers* (which in turn is related to the number of different binary trees on n nodes). In particular $P(n) = C(n-1)$, where $C(n)$ is the nth Catalan number:

$$C(n) = \frac{1}{n+1}\binom{2n}{n}.$$

Applying Stirling's formula, we find that $C(n) \in \Omega(4^n/n^{3/2})$. Since 4^n is exponential and $n^{3/2}$ is just polynomial, the exponential will dominate, implying that function grows very fast. Thus, 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 result of multiplying 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, $1 \leq k \leq n-1$,

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

Thus the problem of determining the optimal sequence of multiplications is broken up into two 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.

So, let us think about the problem of determining the best value of k. At this point, you may be tempted to consider some clever ideas. For example: since we want matrices with small dimensions, pick the value of k that minimizes p_k. Although this is not a bad idea, it turns out that it does not work in this case. Instead, (as in almost all dynamic programming solutions), we will do the *dumb* thing. We will consider *all possible* values of k, and take the best of them.

Notice that this problem satisfies the principle of optimality, because once we decide to break the sequence into the product $A_{1..k} \cdot A_{k+1..n}$, we should compute each subsequence optimally. That is, for the global problem to be solved optimally, the subproblems must be solved optimally as well.

Dynamic Programming Formulation: We will store the solutions to the subproblems in a table, and build the table in a bottom-up manner. 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 formulation.

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, $m[i,i] = 0$.

Step: 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 $m[i,j]$.

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

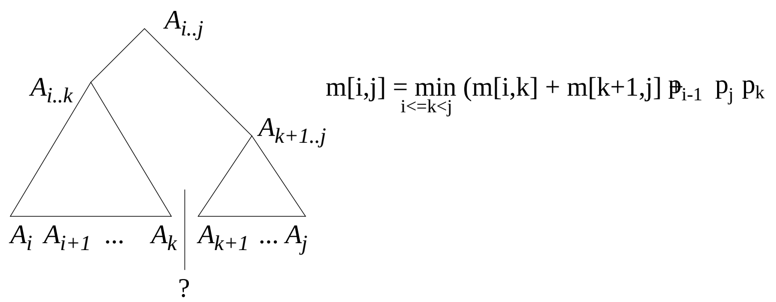


Figure 10: Dynamic Programming Formulation.

It is not hard to convert this rule into a procedure, which is given below. The only tricky part is arranging the order in which to compute the values. In the process of computing $m[i,j]$ we will need to access values $m[i,k]$ and $m[k+1,j]$ for k lying between i and j. This suggests that we should organize things our computation according to the number of matrices in the subchain. Let $L = j - i + 1$ denote the length of the subchain being multiplied. The subchains of length 1 ($m[i,i]$) are trivial. Then we build up by computing the subchains of lengths

$2, 3, \ldots, n$. The final answer is $m[1, n]$. We need to be a little careful in setting up the loops. If a subchain of length L starts at position i, then $j = i + L - 1$. Since we want $j \leq n$, this means that $i + L - 1 \leq n$, or in other words, $i \leq n - L + 1$. So our loop for i runs from 1 to $n - L + 1$ (to keep j in bounds).

Chain Matrix Multiplication

```
Matrix-Chain(array p[1..n], int n) {
    array s[1..n-1,2..n]
    for i = 1 to n do m[i,i] = 0;                  // initialize
    for L = 2 to n do {                            // L = length of subchain
        for i = 1 to n-L+1 do {
            j = i + L - 1;
            m[i,j] = INFINITY;
            for k = i to j-1 do {                  // check all splits
                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[1,n] (final cost) and s (splitting markers);
}
```

The array $s[i, j]$ will be explained later. It is used to extract the actual sequence. The running time of the procedure is $\Theta(n^3)$. We'll leave this as an exercise in solving sums, but the key is that there are three nested loops, and each can iterate at most n times.

Extracting the final Sequence: 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 $m[i, j]$. We can maintain a parallel array $s[i, j]$ in which we will store the value of k providing the optimal split. For example, suppose that $s[i, j] = k$. This tells us that the best way to multiply the subchain $A_{i..j}$ is to first multiply the subchain $A_{i..k}$ and then multiply the subchain $A_{k+1..j}$, and finally multiply these together. Intuitively, $s[i, j]$ tells us what multiplication to perform *last*. Note that we only need to store $s[i, j]$ when we have at least two matrices, that is, if $j > i$.

The actual multiplication algorithm 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)                          // basis case
        return A[i];
    else {
        k = s[i,j]
        X = Mult(i, k)                   // X = A[i]...A[k]
        Y = Mult(k+1, j)                 // Y = A[k+1]...A[j]
        return X*Y;                      // multiply matrices X and Y
    }
}
```

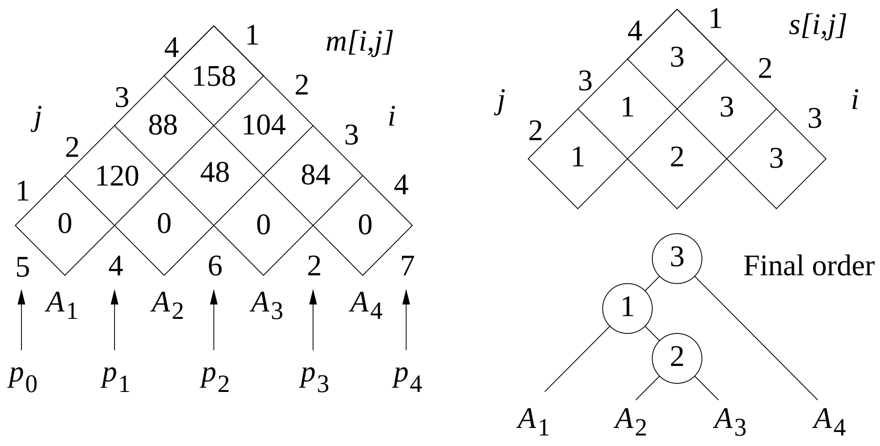


Figure 11: Chain Matrix Multiplication Example.

In the figure below we show an example. This algorithm is tricky, so it would be a good idea to trace through this example (and the one given in the text). The initial set of dimensions are $\langle 5, 4, 6, 2, 7\rangle$ meaning that we are multiplying A_1 (5×4) times A_2 (4×6) times A_3 (6×2) times A_4 (2×7). The optimal sequence is $((A_1(A_2A_3))A_4)$.

Lecture 8: Dynamic Programming: Memoization and Triangulation

(Tuesday, Sep 28, 1999)
Read: Section 16.2 and 16.4 of CLR.

Recursive Implementation: We have described dynamic programming as a method that involves the "bottom-up" computation of a table. However, the recursive formulations that we have derived have been set up in a "top-down" manner. Must the computation proceed bottom-up? Consider the following recursive implementation of the chain-matrix multiplication algorithm. The call `Rec-Matrix-Chain(p, i, j)` computes and returns the value of $m[i,j]$. The initial call is `Rec-Matrix-Chain(p, 1, n)`. We only consider the cost here.

Recursive Chain Matrix Multiplication

```
Rec-Matrix-Chain(array p, int i, int j) {
    if (i == j) m[i,j] = 0;                         // basis case
    else {
        m[i,j] = INFINITY;                          // initialize
        for k = i to j-1 do {                       // try all splits
            cost = Rec-Matrix-Chain(p, i, k) +
                   Rec-Matrix-Chain(p, k+1, j) + p[i-1]*p[k]*p[j];
            if (cost < m[i,j]) m[i,j] = cost;       // update if better
        }
    }
    return m[i,j];                                  // return final cost
}
```

(Note that the table $m[1..n, 1..n]$ is not really needed. We show it just to make the connection with the earlier version clearer.) This version of the procedure certainly looks much simpler, and more closely resembles the recursive formulation that we gave previously for this problem. So, what is wrong with this?

The answer is the running time is much higher than the $\Theta(n^3)$ algorithm that we gave before. In fact, we will see that its running time is *exponential* in n. This is unacceptably slow.

Let $T(n)$ denote the running time of this algorithm on a sequence of matrices of length n. (That is, $n = j - i + 1$.) If $i = j$ then we have a sequence of length 1, and the time is $\Theta(1)$. Otherwise, we do $\Theta(1)$ work and then consider all possible ways of splitting the sequence of length n into two sequences, one of length k and the other of length $n - k$, and invoke the procedure recursively on each one. So we get the following recurrence, defined for $n \geq 1$. (We have replaced the $\Theta(1)$'s with the constant 1.)

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k)) & \text{if } n \geq 2. \end{cases}$$

Claim: $T(n) \geq 2^{n-1}$.

Proof: The proof is by induction on n. Clearly this is true for $n = 1$, since $T(1) = 1 = 2^0$. In general, for $n \geq 2$, the induction hypothesis is that $T(m) \geq 2^{m-1}$ for all $m < n$. Using this we have

$$\begin{aligned} T(n) &= 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k)) \;\geq\; 1 + \sum_{k=1}^{n-1} T(k) \\ &\geq 1 + \sum_{k=1}^{n-1} 2^{k-1} \;=\; 1 + \sum_{k=0}^{n-2} 2^k \\ &= 1 + (2^{n-1} - 1) \;=\; 2^{n-1}. \end{aligned}$$

In the first line we simply ignored the $T(n-k)$ term, in the second line we applied the induction hypothesis, and in the last line we applied the formula for the geometric series.

Why is this so much worse than the dynamic programming version? If you "unravel" the recursive calls on a reasonably long example, you will see that the procedure is called repeatedly with the same arguments. The bottom-up version evaluates each entry exactly once.

Memoization: Is it possible to retain the nice top-down structure of the recursive solution, while keeping the same $O(n^3)$ efficiency of the bottom-up version? The answer is yes, through a technique called *memoization*. Here is the idea. Let's reconsider the function `Rec-Matrix-Chain()` given above. It's job is to compute $m[i, j]$, and return its value. As noted above, the main problem with the procedure is that it recomputes the same entries over and over. So, we will fix this by allowing the procedure to compute each entry exactly once. One way to do this is to initialize every entry to some *special value* (e.g. UNDEFINED). Once an entries value has been computed, it is never recomputed.

Memoized Chain Matrix Multiplication

```
Mem-Matrix-Chain(array p, int i, int j) {
    if (m[i,j] != UNDEFINED) return m[i,j];        // already defined
    else if (i == j) m[i,j] = 0;                   // basis case
    else {
        m[i,j] = INFINITY;                         // initialize
        for k = i to j-1 do {                      // try all splits
```

```
            cost = Mem-Matrix-Chain(p, i, k) +
                   Mem-Matrix-Chain(p, k+1, j) + p[i-1]*p[k]*p[j];
            if (cost < m[i,j]) m[i,j] = cost;       // update if better
        }
    }
    return m[i,j];                                  // return final cost
}
```

This version runs in $O(n^3)$ time. Intuitively, this is because each of the $O(n^2)$ table entries is only computed once, and the work needed to compute one table entry (most of it in the for-loop) is at most $O(n)$.

Memoization is not usually used in practice, since it is generally slower than the bottom-up method. However, in some DP problems, many of the table entries are simply not needed, and so bottom-up computation may compute entries that are never needed. In these cases memoization may be a good idea. If you have know that most of the table will not be needed, here is a way to save space. Rather than storing the whole table explicitly as an array, you can store the "defined" entries of the table in a hash table, using the index pair (i, j) as the hash key. (See Chapter 12 in CLR for more information on hashing.)

Polygons and Triangulations: Let's consider a geometric problem that outwardly appears to be quite different from chain-matrix multiplication, but actually has remarkable similarities. We begin with a number of definitions. Define a *polygon* to be a piecewise linear closed curve in the plane. In other words, we form a cycle by joining line segments end to end. The line segments are called the *sides* of the polygon and the endpoints are called the *vertices*. A polygon is *simple* if it does not cross itself, that is, if the sides do not intersect one another except for two consecutive sides sharing a common vertex. A simple polygon subdivides the plane into its *interior*, its *boundary* and its *exterior*. A simple polygon is said to be *convex* if every interior angle is at most 180 degrees. Vertices with interior angle equal to 180 degrees are normally allowed, but for this problem we will assume that no such vertices exist.

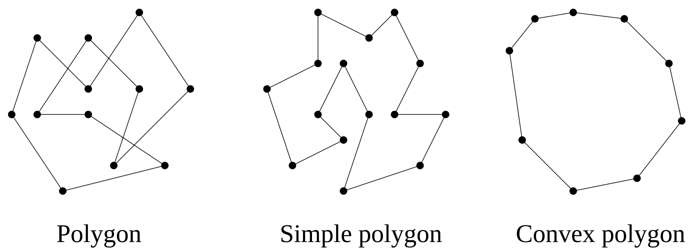


Figure 12: Polygons.

Given a convex polygon, we assume that its vertices are labeled in counterclockwise order $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$. We will assume that indexing of vertices is done modulo n, so $v_0 = v_n$. This polygon has n sides, $\overline{v_{i-1}v_i}$.

Given two nonadjacent sides v_i and v_j, where $i < j$, the line segment $\overline{v_iv_j}$ is a *chord*. (If the polygon is simple but not convex, we include the additional requirement that the interior of the segment must lies entirely in the interior of P. Any chord subdivides the polygon into two polygons: $\langle v_i, v_{i+1}, \ldots, v_j \rangle$, and $\langle v_j, v_{j+1}, \ldots, v_i \rangle$. A *triangulation* of a convex polygon is a maximal set T of chords. In other words, every chord that is not in T intersects the interior of some chord in T. It is easy to see that such a set of chords subdivides the interior of the polygon into a collection of triangles with pairwise disjoint interiors (and hence the name *triangulation*). It is not hard to prove (by induction) that every triangulation of an n-sided polygon consists of $n-3$ chords and $n-2$ triangles.

Triangulations are of interest for a number of reasons. Many geometric algorithm operate by first decomposing a complex polygonal shape into triangles. Then an algorithm can be applied triangle by triangle. Define the *dual graph* of the triangulation to be a graph whose vertices are the triangles, and in which two vertices share a side if the two triangles share a common chord. Observe that the dual graph is a free tree. Hence algorithms for traversing trees can be used for traversing the triangles of a triangulation.

In general, given a convex polygon, there are many possible triangulations. In fact, the number is exponential in n, the number of sides. Which triangulation is the "best"? There are many criteria that are used depending on the application. One criterion is to imagine that you must "pay" for the ink you use in drawing the triangulation, and you want to minimize the amount of ink you use. This suggests the following interesting problem:

Minimum-weight convex polygon triangulation: Given a convex polygon determine the triangulation that minimizes the sum of the perimeters of its triangles.

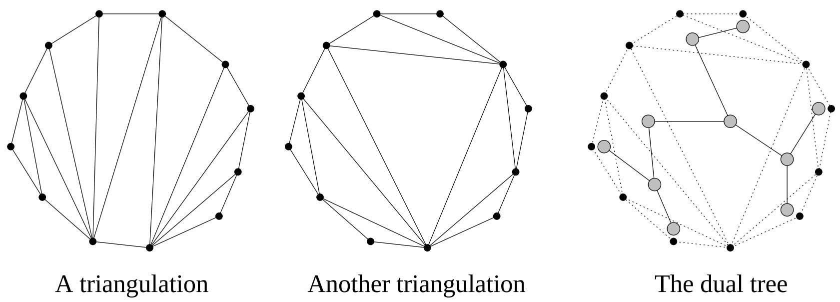


Figure 13: Triangulations.

Given three distinct vertices v_i, v_j, v_k, we define the *weight* of the associated triangle by the weight function

$$w(v_i, v_j, v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|,$$

where $|v_i v_j|$ denotes the length of the line segment $\overline{v_i v_j}$.

Correspondence to Binary Trees: One explanation behind the similarity of triangulations and the the chain matrix multiplication algorithm is to observe that both are fundamentally related to binary trees. In the case of the chain matrix multiplication, the associated binary tree is the evaluation tree for the multiplication, where the leaves of the tree correspond to the matrices, and each node of the tree is associated with a product of a sequence of two or more matrices. To see that there is a similar correspondence here, consider an $(n+1)$-sided convex polygon $P = \langle v_0, v_1, \ldots, v_n \rangle$, and fix one side of the polygon (say $\overline{v_0 v_n}$). Now consider a rooted binary tree whose root node is the triangle containing side $\overline{v_0 v_n}$, whose internal nodes are the nodes of the dual tree, and whose leaves correspond to the remaining sides of the tree. Observe that partitioning the polygon into triangles is equivalent to a binary tree with n leaves, and vice versa. This is illustrated in the figure below. Note that every triangle is associated with an internal node of the tree and every edge of the original polygon, except for the distinguished starting side $\overline{v_0 v_n}$, is associated with a leaf node of the tree.

Once you see this connection. Then the following two observations follow easily. Observe that the associated binary tree has n leaves, and hence (by standard results on binary trees) $n-1$ internal nodes. Since each internal node other than the root has one edge entering it, there

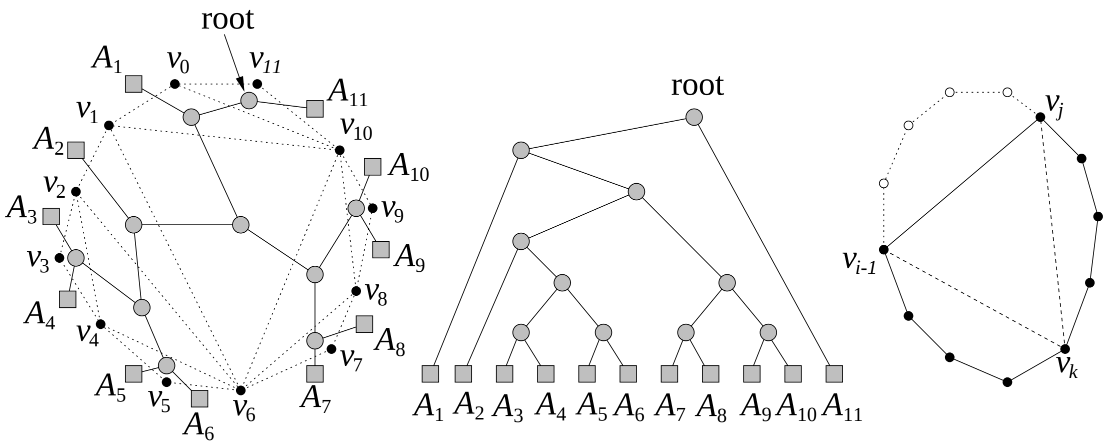


Figure 14: Triangulations and tree structure.

are $n-2$ edges between the internal nodes. Each internal node corresponds to one triangle, and each edge between internal nodes corresponds to one chord of the triangulation.

Dynamic Programming Solution: Let us consider an $(n+1)$-sided polygon $P = \langle v_0, v_1, \ldots, v_n \rangle$. To derive a DP formulation we need to define a set of subproblems from which we can derive the optimum solution. For $1 \le i \le j \le n$, define $t[i,j]$ to be the weight of the minimum weight triangulation for the subpolygon $\langle v_{i-1}, v_i, v_{i+1}, \ldots, v_j \rangle$, assuming that the chord $\overline{v_{i-1}v_j}$ is already present in the triangulation. (The reason that we start with v_{i-1} rather than v_i is to keep the structure as similar as possible to the chain-matrix multiplication problem. We will see this below.) Observe that if we can compute this quantity for all such i and j, then the weight of the minimum weight triangulation of the entire polygon will be $t[1,n]$.

As a basis case, we define the weight of the trivial "2-sided polygon" to be zero, implying that $t[i,i]$ (the line segment $\overline{v_{i-1}v_i}$) to be 0. In general, to compute $t[i,j]$, consider the subpolygon $\langle v_{i-1}, v_i, v_{i+1}, \ldots, v_j \rangle$, where $i < j$. One of the chords of this polygon is the side $\overline{v_{i-1}v_j}$. We may split this subpolygon by introducing a triangle whose base is this chord, and whose third vertex is any vertex v_k, where $i \le k \le j-1$. This subdivides the polygon into the subpolygons $\langle v_{i-1}, \ldots v_k \rangle$ and $\langle v_k, \ldots v_j \rangle$ whose minimum weights are already known to us as $t[i,k]$ and $t[k+1,j]$. In addition we should consider the weight of the newly added triangle $\triangle v_{i-1}v_kv_j$. Thus, we have the following recursive rule:

$$t[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k \le j-1}(t[i,k] + t[k+1,j] + w(v_{i-1}v_kv_j)) & \text{if } i < j. \end{cases}$$

Note that this has exactly the same structure as the recursive definition used in the chain matrix multiplication algorithm. The same $\Theta(n^3)$ algorithm can be applied with only minor changes.

Lecture 9: Greedy Algorithms: Huffman Coding

(Thursday, Sep 30, 1999)
Read: Section 17.3 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. Namely, the optimal solution is described in a recursive manner, and then is computed "bottom-up". Dynamic programming is a powerful technique, but it often leads to algorithms with higher than desired running times. Today we will consider an alternative design technique, called *greedy algorithms*. This method typically leads to simpler and faster algorithms, but it is not as powerful or as widely applicable as dynamic programming. We will give some examples of problems that can be solved by greedy algorithms. (Later in the semester, we will see that this technique can be applied to a number of graph problems as well.) Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (nonoptimal solution strategies), are are often used in finding good approximations.

Huffman Codes: Huffman codes provide a method of encoding data efficiently. Normally when characters are coded using standard codes like ASCII, each character is represented by a fixed-length *codeword* of bits (e.g. 8 bits per character). Fixed-length codes are popular, because its is very easy to break a string up into its individual characters, and to access individual characters and substrings by direct indexing. However, fixed-length codes may not be the most efficient from the perspective of minimizing the total quantity of data.

Consider the following example. Suppose that we want to encode strings over the (rather limited) 4-character alphabet $C = \{a, b, c, d\}$. We could use the following fixed-length code:

Character	a	b	c	d
Fixed-Length Codeword	00	01	10	11

A string such as "abacdaacac" would be encoded by replacing each of its characters by the corresponding binary codeword.

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	10

The final 20-character binary string would be "00010010110000100010".

Now, suppose that you "knew" the relative probabilities of characters in advance. (This might happen by analyzing many strings over a long period of time. In applications like data compression, where you want to encode one file, you can just scan the file and determine the exact frequencies of all the characters.) You can use this knowledge to encode strings differently. Frequently occuring characters are encoded using fewer bits and less frequent characters are encoded using more bits. For example, suppose that characters are expected to occur with the following probabilities. We could design a *variable-length code* which would do a better job.

Character	a	b	c	d
Probability	0.60	0.05	0.30	0.05
Variable-Length Codeword	0	110	10	111

Notice that there is no requirement that the alphabetical order of character correspond to any sort of ordering applied to the codewords. Now, the same string would be encoded as follows.

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

Thus, the resulting 17-character string would be "01100101110010010". Thus, we have achieved a savings of 3 characters, by using this alternative code. More generally, what would be the expected savings for a string of length n? For the 2-bit fixed-length code, the length of the encoded string is just $2n$ bits. For the variable-length code, the expected length of a single encoded character is equal to the sum of code lengths times the respective probabilities of their occurrences. The expected encoded string length is just n times the expected encoded character length.

$$n(0.60 \cdot 1 + 0.05 \cdot 3 + 0.30 \cdot 2 + 0.05 \cdot 3) \;=\; n(0.60 + 0.15 + 0.60 + 0.15) \;=\; 1.5n.$$

Thus, this would represent a 25% savings in expected encoding length.

The question that we will consider today is how to form the best code, assuming that the probabilities of character occurrences are known.

Prefix Codes: One issue that we didn't consider in the example above is whether we will be able to *decode* the string, once encoded. In fact, this code was chosen quite carefully. Suppose that instead of coding the character 'a' as 0, we had encoded it as 1. Now, the encoded string "111" is ambiguous. It might be "d" and it might be "aaa". How can we avoid this sort of ambiguity? You might suggest that we add separation markers between the encoded characters, but this will tend to lengthen the encoding, which is undesirable. Instead, we would like the code to have the property that it can be uniquely decoded.

Note that in both the variable-length codes given in the example above no codeword is a *prefix* of another. This turns out to be the key property. Observe that if two codewords did share a common prefix, e.g. $a \rightarrow 001$ and $b \rightarrow 00101$, then when we see $00101\ldots$ how do we know whether the first character of the encoded message is a or b. Conversely, if no codeword is a prefix of any other, then as soon as we see a codeword appearing as a prefix in the encoded text, then we know that we may decode this without fear of it matching some longer codeword. Thus we have the following definition.

Prefix Code: An assignment of codewords to characters so that no codeword is a prefix of any other.

Observe that any binary prefix coding can be described by a binary tree in which the codewords are the leaves of the tree, and where a left branch means "0" and a right branch means "1". The code given earlier is shown in the following figure. The length of a codeword is just its depth in the tree. The code given earlier is a prefix code, and its corresponding tree is shown in the following figure.

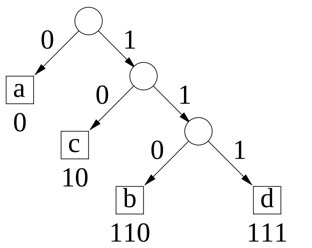


Figure 15: Prefix codes.

Decoding a prefix code is simple. We just traverse the tree from root to leaf, letting the input character tell us which branch to take. On reaching a leaf, we output the corresponding character, and return to the root to continue the process.

Expected encoding length: Once we know the probabilities of the various characters, we can determine the total length of the encoded text. Let $p(x)$ denote the probability of seeing character x, and let $d_T(x)$ denote the length of the codeword (depth in the tree) relative to some prefix tree T. The expected number of bits needed to encode a text with n characters is given in the following formula:

$$B(T) = n \sum_{x \in C} p(x) d_T(x).$$

This suggests the following problem:

Optimal Code Generation: Given an alphabet C and the probabilities $p(x)$ of occurrence for each character $x \in C$, compute a prefix code T that minimizes the expected length of the encoded bit-string, $B(T)$.

Note that the optimal code is not unique. For example, we could have complemented all of the bits in our earlier code without altering the expected encoded string length. There is a very simple algorithm for finding such a code. It was invented in the mid 1950's by David Huffman, and is called a *Huffman code.*. By the way, this code is used by the Unix utility `pack` for file compression. (There are better compression methods however. For example, `compress`, `gzip` and many others are based on a more sophisticated method called the *Lempel-Ziv coding*.)

Huffman's Algorithm: Here is the intuition behind the algorithm. Recall that we are given the occurrence probabilities for the characters. We are going build the tree up from the leaf level. We will take two characters x and y, and "merge" them into a single *super-character* called z, which then replaces x and y in the alphabet. The character z will have a probability equal to the sum of x and y's probabilities. Then we continue recursively building the code on the new alphabet, which has one fewer character. When the process is completed, we know the code for z, say 010. Then, we append a 0 and 1 to this codeword, given 0100 for x and 0101 for y.

Another way to think of this, is that we merge x and y as the left and right children of a root node called z. Then the subtree for z replaces x and y in the list of characters. We repeat this process until only one super-character remains. The resulting tree is the final prefix tree. Since x and y will appear at the bottom of the tree, it seem most logical to select the two characters with the smallest probabilities to perform the operation on. The result is Huffman's algorithm. It is illustrated in the following figure.

The pseudocode for Huffman's algorithm is given below. Let C denote the set of characters. Each character $x \in C$ is associated with a occurrence probability $x.prob$. Initially, the characters are all stored in a *priority queue* Q. Recall that this data structure can be built initially in $O(n)$ time, and we can extract the element with the smallest key in $O(\log n)$ time and insert a new element in $O(\log n)$ time. The objects in Q are sorted by probability. Note that with each execution of the for-loop, the number of items in the queue decreases by one. So, after $n-1$ iterations, there is exactly one element left in the queue, and this is the root of the final prefix code tree.

Huffman's Algorithm

```
Huffman(int n, character C[1..n]) {
    Q = C;                          // priority queue sorted by probability
    for i = 1 to n-1 {
        z = new internal tree node;
        z.left = x = Q.extractMin();    // extract the two smallest probabilities
        z.right = y = Q.extractMin();
        z.prob = x.prob + y.prob;       // z's probability is sum of x and y
        Q.insert(z);                    // insert z into priority queue
    }
    return the last element left in Q as the root;
}
```

Correctness: The big question that remains is why is this algorithm correct? Recall that the cost of any encoding tree T is $B(T) = \sum_x p(x)d_T(x)$. Our approach will be 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. First, observe that the Huffman tree is a *full binary tree*, meaning that every internal node has exactly two children. It would never

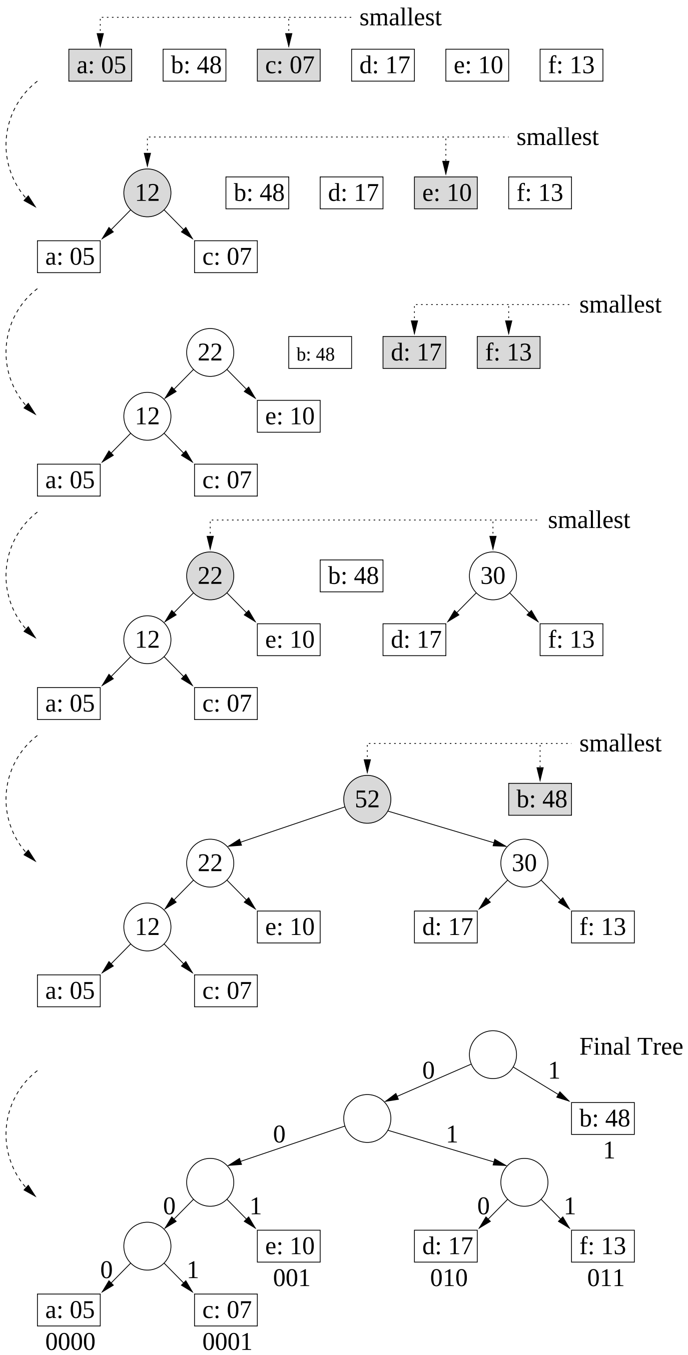


Figure 16: Huffman's Algorithm.

pay to have an internal node with only one child (since such a node could be deleted), so we may limit consideration to full binary trees.

Claim: Consider the two characters, x and y with the smallest probabilities. 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 optimal prefix code tree, and let b and c be two siblings at the maximum depth of the tree. Assume without loss of generality that $p(b) \le p(c)$ and $p(x) \le p(y)$ (if this is not true, then rename these characters). Now, since x and y have the two smallest probabilities it follows that $p(x) \le p(b)$ (they may be equal) and $p(y) \le p(c)$ (may be equal). Because b and c are at the deepest level of the tree we know that $d(b) \ge d(x)$ and $d(c) \ge d(y)$. Thus, we have $p(b) - p(x) \ge 0$ and $d(b) - d(x) \ge 0$, and hence their product is nonnegative. Now switch the positions of x and b in the tree, resulting in a new tree T'. This is illustrated in the following figure.

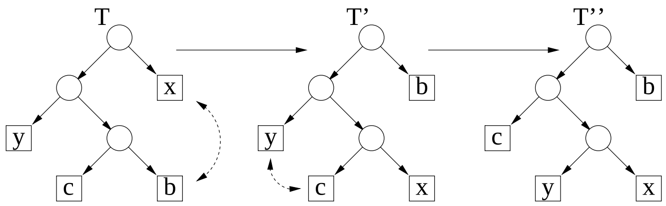


Figure 17: Correctness of Huffman's Algorithm.

Let's see how the cost changes. The cost of T' is

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

Thus the cost does not increase, implying that T' is an optimal tree. By switching y with c we get a new tree T'' which by a similar argument is also optimal. The final tree T'' satisfies the statement of the claim.

The above theorem asserts that the first step of Huffman's algorithm is essentially the proper one to perform. The complete proof of correctness for Huffman's algorithm follows by induction on n (since with each step, we eliminate exactly one character).

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

Proof: The proof is by induction on n, the number of characters. For the basis case, $n = 1$, the tree consists of a single leaf node, which is obviously optimal.

Assume inductively that when strictly fewer than n characters, Huffman's algorithm is guaranteed to produce the optimal 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 optimal tree, the two characters of lowest probability 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 probability is $p(z) = p(x) + p(y)$. 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

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

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 need to build the tree T on $n-1$ characters optimally. By induction, this exactly what Huffman's algorithm does. Thus the final tree is optimal.

Lecture 10: Greedy Algorithms: Activity Selection and Fractional Knapack

(Tuesday, Oct 5, 1999)
Read: Section 17.1 and 17.2 in CLR.

Activity Scheduling: Last time we showed one greedy algorithm, Huffman's algorithm. Today we consider a couple more examples. The first is called *activity scheduling* and it is a very simple scheduling problem. We are given a set $S = \{1, 2, \ldots, n\}$ of n *activities* that are to be scheduled to use some resource, where each activity must be started at a given start time s_i and ends at a given finish time f_i. For example, these might be lectures that are to be given in a lecture hall, where the lecture times have been set up in advance, or requests for boats to use a repair facility while they are in port.

Because there is only one resource, and some start and finish times may overlap (and two lectures cannot be given in the same room at the same time), not all the requests can be honored. We say that two activities i and j are *noninterfering* if their start-finish intervals do not overlap, that is $[s_i, f_i) \cap [s_j, f_j) = \emptyset$. The *activity scheduling problem* is to select a maximum-size set of mutually noninterfering activities for use of the resource. (Notice that there are many other criteria that we might have considered instead. For example, we might want to maximize the total utilization time for the facility, instead.)

So how do we schedule the largest number of activities on the resource? Intuitively, we do not like long activities, because they occupy the resource and keep us from honoring other requests. This suggests the following greedy strategy: repeated select the job with the smallest duration $(f_i - s_i)$ and schedule it, provided that it does not interfere with any previously scheduled activities. This turn out to be nonoptimal. (See Problem 17.1-3 in CLR).

Here is a simple greedy algorithm that does work. The intuition is the same. Since we do not like jobs that take a long time, let us select the job that finishes first and schedule it. Then, among all jobs that do not interfere with this first job, we schedule the one that finishes first, and so on. We begin by assuming that the activities have all be sorted by finish times, so that

$$f_1 \le f_2 \le \cdots \le f_n,$$

(and of course the s_i's are sorted in parallel). The pseudocode is presented below, and assumes that this sorting has already been done. The output is the list A of scheduled activities. The variable *prev* holds the index of the most recently scheduled activity at any time, in order to determine interferences.

Greedy Activity Scheduler

```
schedule(int n, int s[1..n], int f[1..n]) {
    // we assume f[1..n] is already sorted
    A = <1>;  prev = 1;                   // schedule activity 1 first
    for i = 2 to n {
        if (s[i] >= f[prev]) {            // no interference?
            append i to A;  prev = i;     // schedule i next
        }
    }
    return A;
}
```

It is clear that the algorithm is quite simple and efficient. The most costly activity is that of sorting the activities by finish time, so the total running time is $\Theta(n \log n)$. The figure below shows an example. The final output would be $\{1, 4, 7\}$.

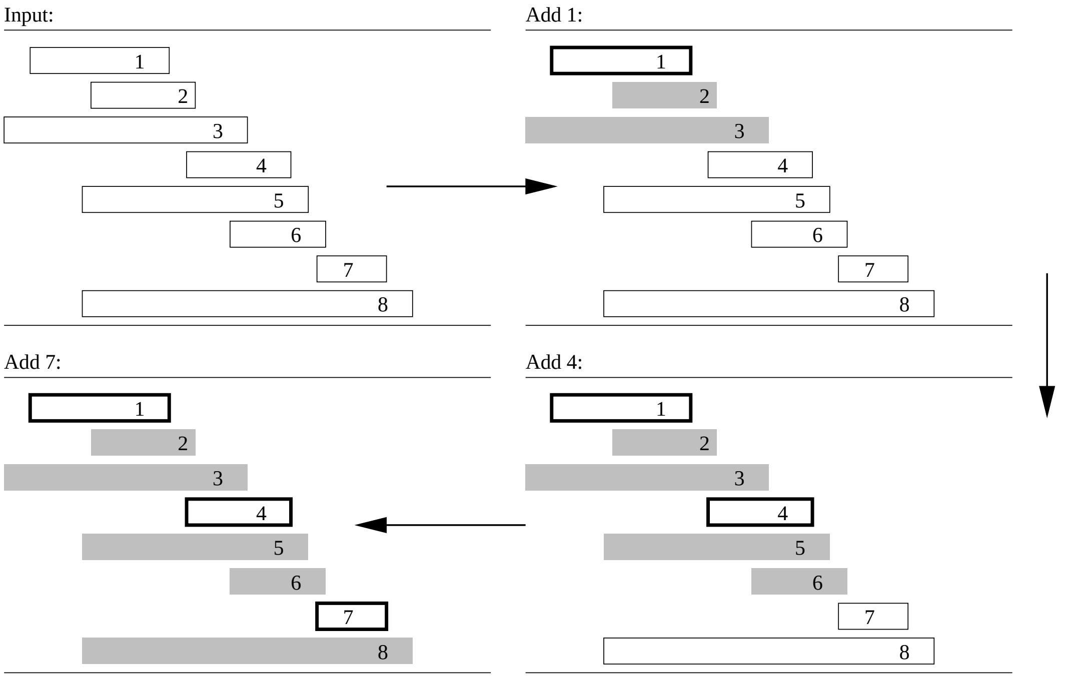


Figure 18: Activity Scheduler.

Correctness: Our proof of correctness is based on showing that the first choice made by the algorithm is the best possible, and then using induction to show that the algorithm is globally optimal. The proof's structure is noteworthy, because many greedy correctness proofs are based on the same idea: show that any other solution can be converted into the greedy solution without increasing its cost.

Claim: Let $S = \{1, 2, \ldots, n\}$ be a set of activities to be scheduled, sorted by increasing finish times. Then there is an optimal schedule in which activity 1 is scheduled first.

Proof: Let A be an optimal schedule. Let x be the activity in A with the smallest finish time. If $x = 1$ then we are done. Otherwise, we form a new schedule A' by replacing x with activity 1. We claim that this is a feasible schedule (i.e., it has no interfering activities). This is because $A - \{x\}$ cannot have any other activities that start before x finishes, since otherwise these activities would interfere with x. Since 1 is by definition the first activity to finish, it has an earlier finish time than x, and so it cannot interfere with any of the activities in $A - \{x\}$. Thus, A' is a feasible schedule. Clearly A and A' contain the same number of activities, implying that A' is also optimal.

Claim: The greedy algorithm gives an optimal solution to the activity scheduling problem.

Proof: The proof is by induction on the number of activities. For the basis case, if there are no activities, then the greedy algorithm is trivially optimal.

For the induction step, let us assume that the greedy algorithm is optimal on any set of activities of size strictly smaller than $|S|$, and we prove the result for S. Let S' be the set of activities that do not interfere with activity 1. That is

$$S' = \{i \in S \mid s_i \geq f_1\}.$$

Observe that any solution for S' can be made into a solution for S by simply adding activity 1, and vice versa. Since 1 is in the optimal schedule (by the above claim), it follows that to produce an optimal solution for the overall problem, we should first schedule 1 and then append the optimal schedule for S'. But by induction (since $|S'| < |S|$), this is exactly what the greedy algorithm does.

Fractional Knapsack Problem: The classical *(0-1) knapsack problem* is a famous optimization problem. A thief is robbing a store, and finds n items which can be taken. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but has a knapsack that can only carry W total pounds. Which items should he take? (The reason that this is called 0-1 knapsack is that each item must be left (0) or taken (1) in it entirety. It is not possible to take part of an item or multiple copies of an item.) This optimization problem arises in industrial packing applications. For example, you may want to ship some subset of items on a truck of limited capacity.

In contrast, in the *fractional knapsack problem* the setup is exactly the same, but the thief is allowed to take any *fraction* of an item for a fraction of the weight and a fraction of the value. So, you might think of each object as being a sack of gold, which you can partially empty out before taking.

The 0-1 knapsack problem is hard to solve, and in fact it is an NP-complete problem (meaning that there probably doesn't exist an efficient solution). However, there is a very simple and efficient greedy algorithm for the fractional knapsack problem. Let $\rho_i = v_i/w_i$ denote the *value-per-pound ratio*. We sort the items in decreasing order of ρ_i, and add them in this order. If the item fits, we take it all. At some point there is an item that does not fit in the remaining space. We take as much of this item as possible, thus filling the knapsack entirely. This is illustrated in the figure below.

It is easy to see that the greedy algorithm is optimal for the fractional problem. Given a room with sacks of gold, silver, and bronze, you would obviously take as much gold as possible, then take as much silver as possible, and then as much bronze as possible. But it would never benefit you to take a little less gold so that you could replace it with an equal volume of bronze.

You can also see why the greedy algorithm may not be optimal in the 0-1 case. Consider the example shown in the figure. If you were to sort the items by ρ_i, then you would first take the items of weight 5, then 20, and then (since the item of weight 40 does not fit) you would settle for the item of weight 30, for a total value of $\$30 + \$100 + \$90 = \$220$.

On the other hand, if you had been less greedy, and ignored the item of weight 5, then you could take the items of weights 20 and 40 for a total value of $\$100 + \$160 = \$260$. This feature

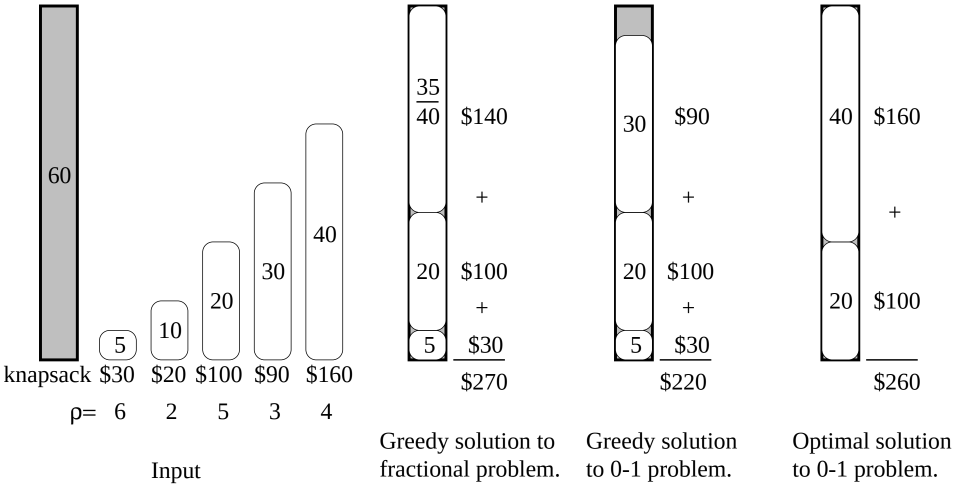


Figure 19: Knapsack Problem.

of "delaying gratification" in order to come up with a better overall solution is your indication that the greedy solution is not optimal.

Lecture 11: Graphs: Definitions and Representations

(Thursday, Oct 7, 1999)

Read: Review Sections 5.4, 5.5. Read 23.1 and 23.2 CLR.

Graph Algorithms: We are now beginning a major new section of the course. We will be discussing algorithms for both directed and undirected 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:

Most of you have encountered the notions of directed and undirected graphs in other courses, so we will give a quick overview here. Please read Sections 5.4, 5.5, and 23.1 in CLR for further information.

Definition: A *directed graph* (or *digraph*) $G = (V, E)$ consists of a finite set V, called the *vertices* or *nodes*, and E, a set of *ordered pairs*, called the *edges* of G. (Another way of saying this is that E is a binary relation on V.)

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).

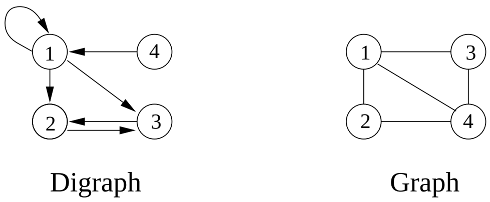


Figure 20: Digraph and graph example.

Definition: An *undirected graph* (or *graph*) $G = (V, E)$ consists of a finite set V of vertices, and a set E of *unordered pairs* of distinct vertices, called the edges. (Note that self-loops are not allowed).

Note 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) may only be defined for one, or may be defined differently.

We say that vertex v is *adjacent* to vertex u if there is an edge (u, v). In a directed graph, given the edge $e = (u, v)$, we say that u is the *origin* of e and v is the *destination* of e. In undirected graphs u and v are the *endpoints* of the edge. The edge e is *incident* (meaning that it touches) both u and v.

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 incident edges. By the *degree* of a graph, we usually mean the maximum degree of its vertices.

When discussing the size of a graph, we typically consider both the number of vertices and the number of edges. The number of vertices is typically written as n or V, and the number of edges is written as m or E or e. Here are some basic combinatorial facts about graphs and digraphs. We will leave the proofs to you. Given a graph with V vertices and E edges then:

In a graph:

- $0 \leq E \leq \binom{n}{2} = n(n-1)/2 \in O(n^2)$.
- $\sum_{v \in V} deg(v) = 2E$.

In a digraph:

- $0 \leq E \leq n^2$.
- $\sum_{v \in V} in\text{-}deg(v) = \sum_{v \in V} out\text{-}deg(v) = E$.

Notice that generally the number of edges in a graph may be as large as quadratic in the number of vertices. However, the large graphs that arise in practice typically have much fewer edges. A graph is said to be *sparse* if $E \in \Theta(V)$, and *dense*, otherwise. When giving the running times of algorithms, we will usually express it as a function of both V and E, so that the performance on sparse and dense graphs will be apparent.

Paths and Cycles: A *path* in a graph digraph 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. A path is *simple* if all vertices and all the edges are distinct. A *cycle* is a path containing at least one edge and for which $v_0 = v_k$. A cycle is *simple* if its vertices (except v_0 and v_k) are distinct and all its edges are distinct.

A graph or digraph is said to be *acyclic* if it contains no simple cycles. An acyclic connected graph is called a *free tree* or simply *tree* for short. (The term "free" is intended to emphasize the fact that the tree has no root, in contrast to a *rooted tree*, as is usually seen in data structures.) An acyclic undirected graph (which need not be connected) is a collection of free trees, and is (naturally) called a *forest*. An acyclic digraph is called a *directed acyclic graph*, or *DAG* for short.

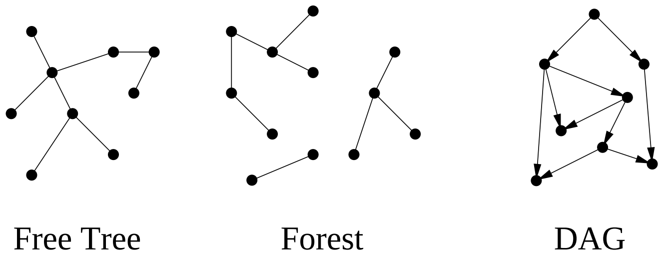


Figure 21: Free trees, forests, and DAG's.

We say that w is *reachable* from u if there is a path from u to w. Note that every vertex is reachable from itself by a trivial path that uses zero edges. An undirected graph is *connected* if every vertex can reach every other vertex. (Connectivity is a bit messier for digraphs, and we will define it later.) The subsets of mutually reachable vertices partition the vertices of the graph into disjoint subsets, called the *connected components* of the graph.

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

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

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

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 $1 \le v \le n$, $Adj[v]$ 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.

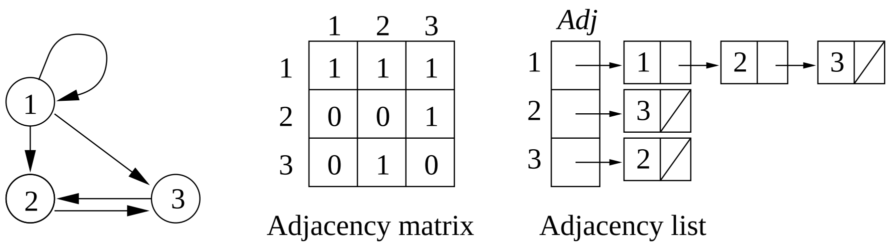
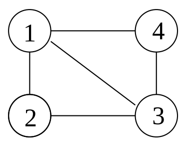


Figure 22: Adjacency matrix and adjacency list for digraphs.

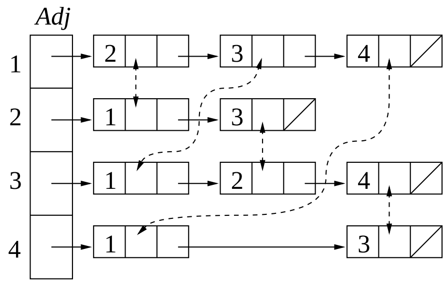
We can represent undirected graphs using exactly the same representation, but we will store each edge twice. In particular, we representing the undirected edge $\{v, w\}$ by the two oppositely directed edges (v, w) and (w, v). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

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



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Adjacency matrix



Adjacency list (with crosslinks)

Figure 23: Adjacency matrix and adjacency list for graphs.

An adjacency matrix requires $\Theta(V^2)$ storage and an adjacency list requires $\Theta(V + E)$ storage. The V arises because there is one entry for each vertex in Adj. Since each list has $out\text{-}deg(v)$ entries, when this is summed over all vertices, the total number of adjacency list records is $\Theta(E)$. For sparse graphs the adjacency list representation is more space efficient.

Graph Traversals: There are a number of approaches used for solving problems on graphs. One of the most important approaches is based on the notion of systematically visiting all the vertices and edge of a graph. The reason for this is that these traversals impose a type of tree structure (or generally a forest) on the graph, and trees are usually much easier to reason about than general graphs.

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

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

It may not be obvious at first, but these single predecessor pointers are sufficient to reconstruct the shortest path to any vertex. (Can you see why? Notice that the path will be constructed in reverse.)

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, there may be more than $n!$ simple paths in a graph (consider a complete graph). Thus, this strategy is clearly impractical.

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

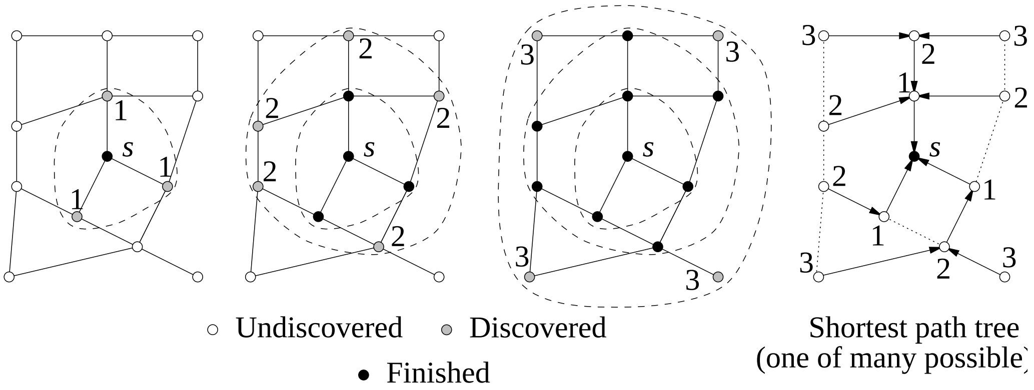


Figure 24: Breadth-first search for shortest paths.

This is the basis for the breadth-first search algorithm. Next time we will discuss this and its natural dual, depth-first search.

Lecture 12: Breadth-First and Depth-First Search

(Tuesday, Oct 12, 1999)
Read: Sections 23.2 and 23.3 in CLR.

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

Initially all vertices (except the source) are colored white, meaning that they are *undiscovered*. When a vertex has first been *discovered*, it is colored gray (and is part of the frontier). When a gray vertex is *processed*, then it becomes black.

The search makes use of a *queue*, a first-in first-out list, where elements are removed in the same order they are inserted. The first item in the queue (the next to be removed) is called the *head* of the queue. We will also maintain arrays $color[u]$ which holds the color of vertex u (either white, gray or black), $pred[u]$ which points to the predecessor of u (i.e. the vertex who first discovered u, and $d[u]$, the distance from s to u. Only the color is really needed for the search (in fact it is only necessary to know whether a node is nonwhite). We include all this information, because some applications of BFS use this additional information.

Breadth-First Search

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

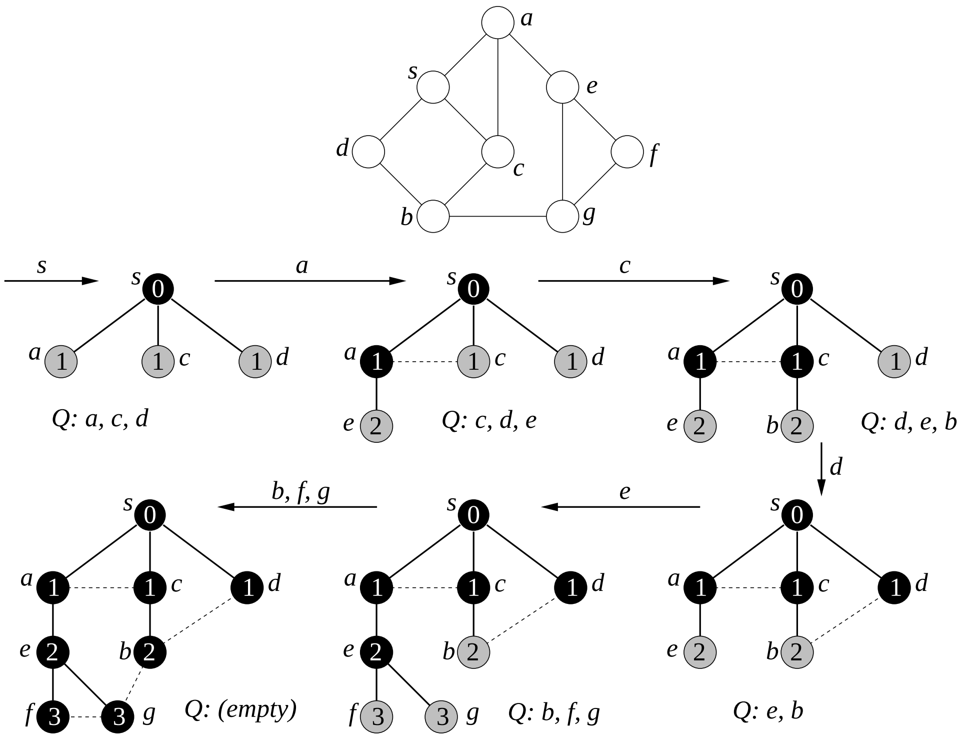


Figure 25: Breadth-first search: Example.

Observe that the predecessor pointers of the BFS search define an *inverted tree* (an acyclic directed graph in which the source is the root, and every other node has a unique path to the root). If we reverse these edges we get a rooted unordered tree called a *BFS tree* for G. (Note that there are many potential BFS trees for a given graph, depending on where the search starts, and in what order vertices are placed on the queue.) These edges of G are called *tree edges* and the remaining edges of G are called *cross edges*.

It is not hard to prove that if G is an undirected graph, then cross edges always go between two nodes that are at most one level apart in the BFS tree. (Can you see why this must be true?)

Analysis: The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. As done in CLR $V = |V|$ and $E = |E|$. Observe that the initialization portion requires $\Theta(V)$ 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 V (exactly V 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

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

The analysis is essentially the same for directed graphs.

Depth-First Search: 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 directed graph $G = (V, E)$. The same algorithm works for undirected 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.) The algorithm is shown below. As with BFS, DFS induces a tree structure. We will discuss this tree structure further below.

Depth-First Search

```
DFS(G) {                                    // main program
    for each u in V {                       // initialization
        color[u] = white;
        pred[u]  = nil;
    }
```

```
    time = 0;
    for each u in V
        if (color[u] == white)            // found an undiscovered vertex
            DFSVisit(u);                  // start a new search here
}

DFSVisit(u) {                             // start a search at u
    color[u] = gray;                      // mark u visited
    d[u] = ++time;
    for each v in Adj(u) do
        if (color[v] == white) {          // if neighbor v undiscovered
            pred[v] = u;                  // ...set predecessor pointer
            DFSVisit(v);                  // ...visit v
        }
    color[u] = black;                     // we're done with u
    f[u] = ++time;
}
```

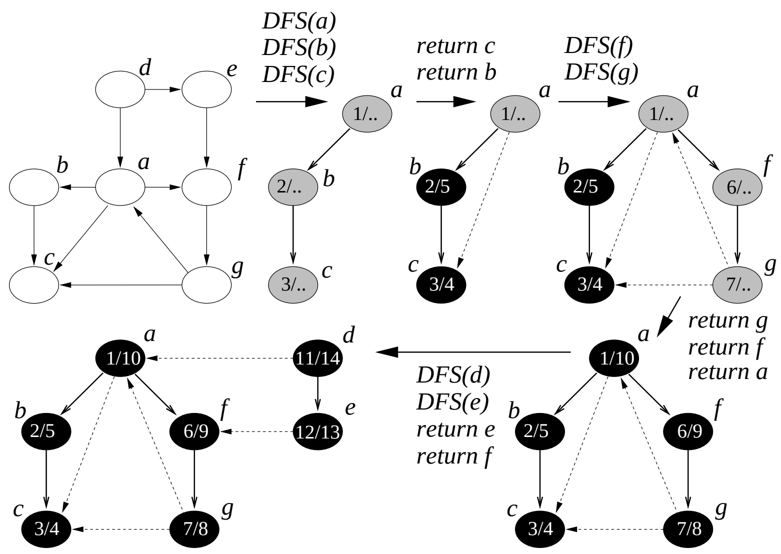


Figure 26: Depth-First search tree.

Analysis: The running time of DFS is $\Theta(V+E)$. 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.

First observe that if we ignore the time spent in the recursive calls, the main DFS procedure runs in $O(V)$ time. Observe that each vertex is visited exactly once in the search, 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 $O(1+\mathit{outdeg}(u))$ time. Thus the total time used in the procedure is

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

A similar analysis holds if we consider DFS for undirected graphs.

Tree structure: DFS naturally imposes a tree structure (actually a collection of trees, or a forest) on the structure of the graph. This is just the recursion tree, where the edge (u, v) arises when processing vertex u we call `DFSVisit(v)` for some neighbor v. 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* by convention. 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.

Lemma: (Parenthesis Lemma) Given a digraph $G = (V, E)$, and any DFS tree for G and any two vertices $u, v \in V$.

- u is a descendent of v if and only if $[d[u], f[u]] \subseteq [d[v], f[v]]$.
- u is an ancestor of v if and only if $[d[u], f[u]] \supseteq [d[v], f[v]]$.
- u is unrelated to v if and only if $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.

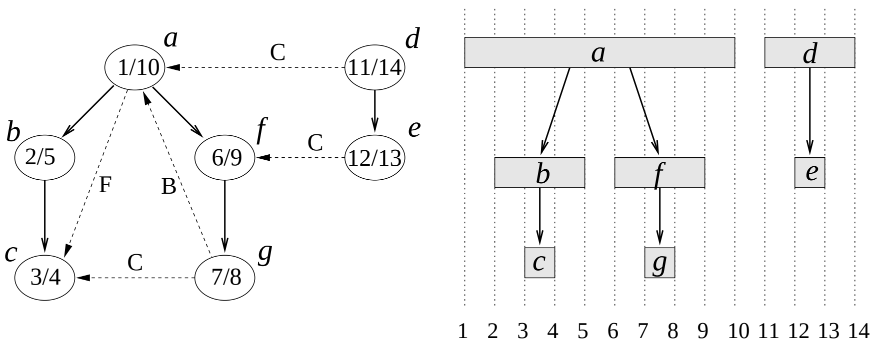


Figure 27: Parenthesis Lemma.

Lecture 13: Topological Sort and Strong Components

(Thursday, Oct 14, 1999)
Read: Sects. 23.3–5 in CLR.

Cycles: The time stamps given by DFS allow 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 $G = (V, E)$, consider any DFS forest of G, and consider any edge $(u, v) \in E$. If this edge is a tree, forward, or cross edge, then $f[u] > f[v]$. If the edge is a back edge then $f[u] \leq f[v]$.

Proof: For tree, forward, and back edges, the proof follows directly from the parenthesis lemma. (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 intervals are disjoint. When we were processing u, v was not white (otherwise (u, v) would be a tree edge), implying that v was started before u. Because the intervals are disjoint, v must have also finished before u.

Lemma: Consider a digraph $G = (V, E)$ and any DFS forest for G. G has a cycle if and only the DFS forest has a back edge.

Proof: ($\Leftarrow$) 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.

($\Rightarrow$) We show the contrapositive. 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: No back edges means no cycles. But you should not infer that there is some simple relationship between the *number* of back edges and the *number* of cycles. For example, a DFS tree may only have a single back edge, and there may anywhere from one up to an exponential number of simple cycles in the graph.

A similar theorem applies to undirected graphs, and is not hard to prove.

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. 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 *precedence 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. To do this we run a (stripped down) DFS, and when each vertex is finished we add it to the front of a linked list. The final linked list order will be the final topological order. This is given below.

Topological Sort

```
TopSort(G) {
    for each (u in V) color[u] = white;     // initialize
    L = new linked_list;                    // L is an empty linked list
    for each (u in V)
```

```
        if (color[u] == white) TopVisit(u);
    return L;                               // L gives final order
}

TopVisit(u) {                               // start a search at u
    color[u] = gray;                        // mark u visited
    for each (v in Adj(u))
        if (color[v] == white) TopVisit(v);
    Append u to the front of L;             // on finishing u add to list
}
```

This is typical example of DFS is used in applications. Observe that the structure is essentially the same as the basic DFS procedure, but we only include the elements of DFS that are needed for this application.

As an example we consider the DAG presented in CLR for Professor Bumstead's order of dressing. Bumstead lists the precedences in the order in which he puts on his clothes in the morning. We do our depth-first search in a different order from the one given in CLR, and so we get a different final ordering. However both orderings are legitimate, given the precedence constraints. As with depth-first search, the running time of topological sort is $\Theta(V + E)$.

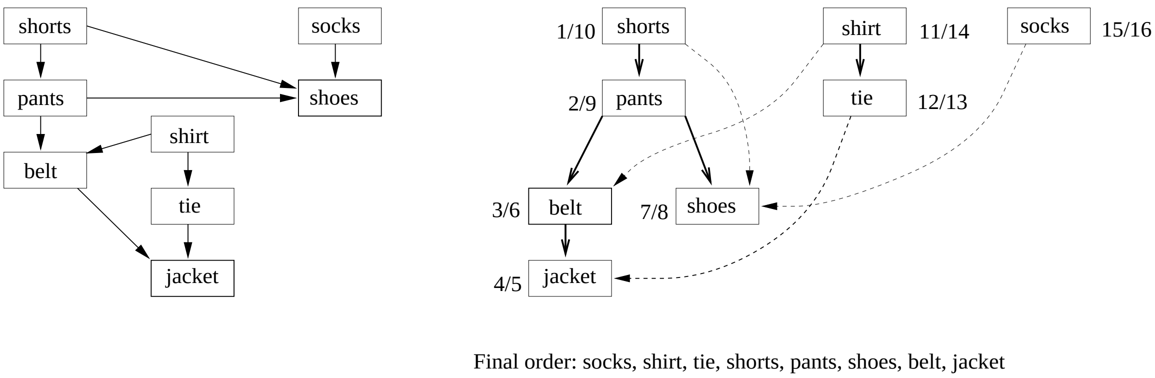


Figure 28: Topological sort.

Strong Components: Next we consider a very important connectivity problem with digraphs. When digraphs are used in communication and transportation networks, people want to know that there networks are complete in the sense that from any location it is possible to reach any other location in the digraph. A digraph is *strongly connected* if for every pair of vertices, $u, v \in V$, u can reach v and vice versa.

We would like to write an algorithm that determines whether a digraph is strongly connected. In fact we will solve a generalization of this problem, of computing the *strongly connected components* (or *strong components* for short) of a digraph. In particular, we partition the vertices of the digraph into subsets such that the induced subgraph of each subset is strongly connected. (These subsets should be as large as possible, and still have this property.) More formally, we say that two vertices u and v are *mutually reachable* if u and reach v and vice versa. It is easy to see that mutual reachability is an equivalence relation. This equivalence relation partitions the vertices into equivalence classes of mutually reachable vertices, and these are the strong components.

Observe that if we merge the vertices in each strong component into a single *super vertex*, and joint two supervertices (A, B) if and only if there are vertices $u \in A$ and $v \in B$ such that

$(u, v) \in E$, then the resulting digraph, called the *component digraph*, is necessarily acyclic. (Can you see why?)

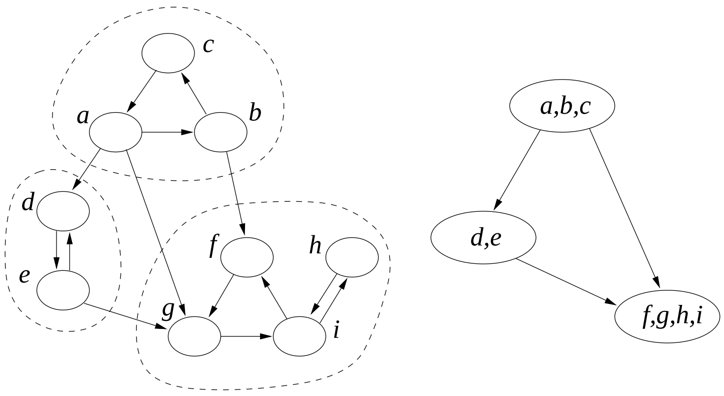


Figure 29: Strong Components.

The algorithm that we will present is an algorithm designer's "dream" (and an algorithm student's nightmare). It is amazingly simple and efficient, but it is so clever that it is very difficult to even see how it works. We will give some of the intuition that leads to the algorithm, but will not prove the algorithm's correctness formally. See CLR for a formal proof.

Strong Components and DFS: By way of motivation, consider the DFS of the digraph shown in the following figure (left). By definition of DFS, when you enter a strong component, every vertex in the component is reachable, so the DFS does not terminate until all the vertices in the component have been visited. Thus all the vertices in a strong component must appear in the same tree of the DFS forest. Observe that in the figure each strong component is just a subtree of the DFS forest. Is it always true for any DFS? Unfortunately the answer is no. In general, many strong components may appear in the same DFS tree. (See the DFS on the right for a counterexample.) Does there always exist a way to order the DFS such that it is true? Fortunately, the answer is yes.

Suppose that you knew the component digraph in advance. (This is ridiculous, because you would need to know the strong components, and that is the problem we are trying to solve. But humor me for a moment.) Further suppose that you computed a *reversed topological order* on the component digraph. That is, (u, v) is an edge in the component digraph, then v comes *before* u in this reversed order (not after as it would in a normal topological ordering). Now, run DFS, but every time you need a new vertex to start the search from, select the next available vertex according to this reverse topological order of the component digraph.

Here is an informal justification. Clearly once the DFS starts within a given strong component, it must visit every vertex within the component (and possibly some others) before finishing. If we do not start in reverse topological, then the search may "leak out" into other strong components, and put them in the same DFS tree. For example, in the figure below right, when the search is started at vertex a, not only does it visit its component with b and c, but the it also visits the other components as well. However, by visiting components in reverse topological order of the component tree, each search cannot "leak out" into other components, because other components would have already have been visited earlier in the search.

This leaves us with the intuition that if we could somehow order the DFS, so that it hits the strong components according to a reverse topological order, then we would have an easy

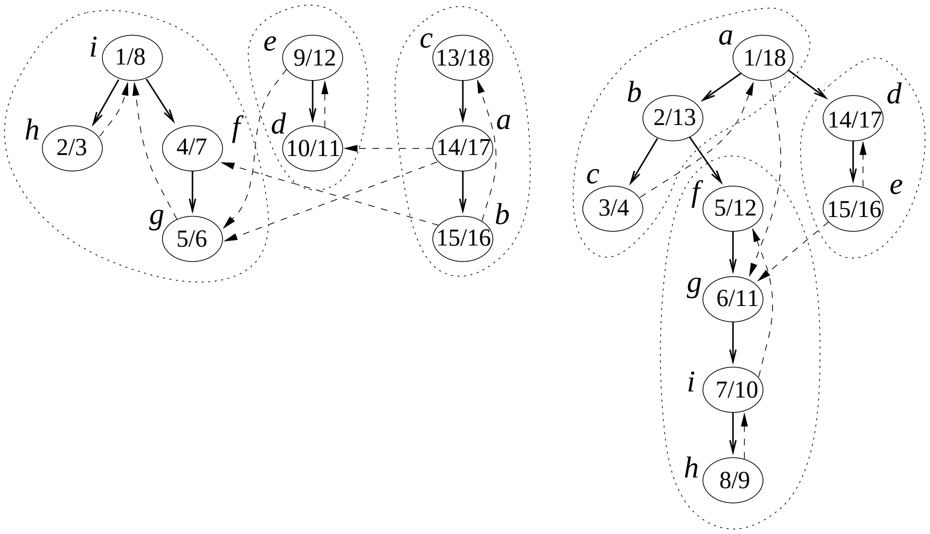


Figure 30: Two depth-first searches.

algorithm for computing strong components. However, we do not know what the component DAG looks like. (After all, we are trying to solve the strong component problem in the first place). The "trick" behind the strong component algorithm is that we can find an ordering of the vertices that has essentially the necessary property, without actually computing the component DAG.

The Plumber's Algorithm: I call this algorithm the plumber's algorithm (because it avoids leaks). Unfortunately it is quite difficult to understand why this algorithm works. I will present the algorithm, and refer you to CLR for the complete proof. First recall that G^R (what CLR calls G^T) is the digraph with the same vertex set as G but in which all edges have been reversed in direction. Given an adjacency list for G, it is possible to compute G^R in $\Theta(V + E)$ time. (I'll leave this as an exercise.)

Observe that the strongly connected components are not affected by reversing all the digraph's edges. If u and v are mutually reachable in G, then certainly this is still true in G^R. All that changes is that the component DAG is completely reversed. The ordering trick is to order the vertices of G according to their finish times in a DFS. Then visit the nodes of G^R in decreasing order of finish times. All the steps of the algorithm are quite easy to implement, and all operate in $\Theta(V + E)$ time. Here is the algorithm.

Strong Components

```
StrongComp(G) {
    Run DFS(G), computing finish times f[u] for each vertex u;
    Compute R = Reverse(G), reversing all edges of G;
    Sort the vertices of R (by CountingSort) in decreasing order of f[u];
    Run DFS(R) using this order;
    Each DFS tree is a strong component;
}
```

Correctness: Why visit vertices in decreasing order of finish times? Why use the reversal digraph? It is difficult to justify these elements formally. Here is some intuition, though. Recall that the main intent is to visit the strong components in a reverse topological order. The problem is how

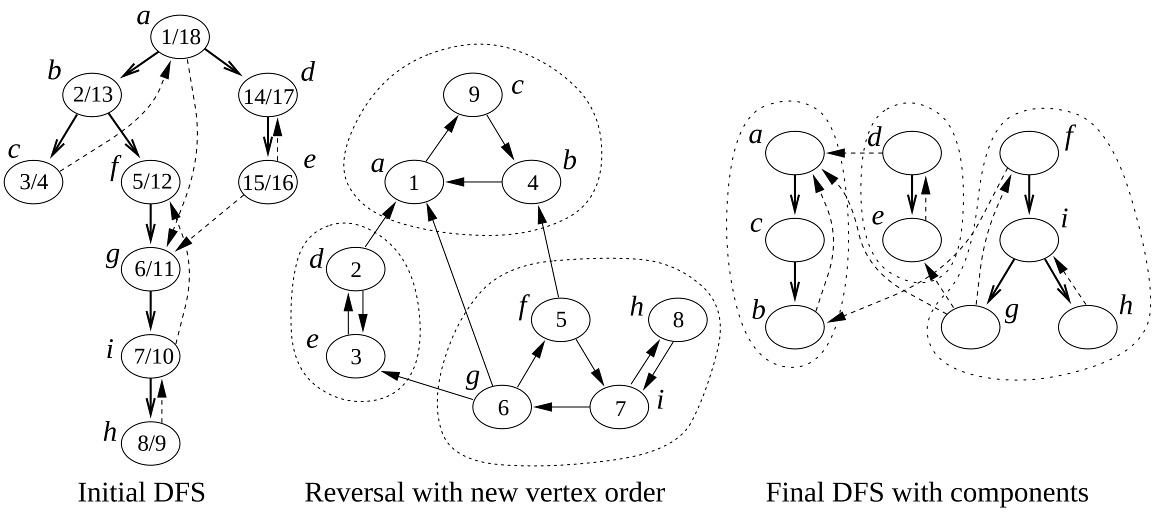


Figure 31: Strong Components Algorithm

to order the vertices so that this is true. Recall from the topological sorting algorithm, that in a DAG, finish times occur in reverse topological order (i.e., the first vertex in the topological order is the one with the highest finish time). So, if we wanted to visit the components in reverse topological order, this suggests that we should visit the vertices in increasing order of finish time, starting with the lowest finishing time. This is a good starting idea, but it turns out that it doesn't work. The reason is that there are many vertices in each strong component, and they all have different finish times. For example, in the figure above observe that in the first DFS (on the left) the lowest finish time (of 4) is achieved by vertex c, and its strong component is first, not last, in topological order.

It is tempting to give up in frustration at this point. But there is something to notice about the finish times. If we consider the *maximum finish time* in each component, then these are related to the topological order of the component graph. In fact it is possible to prove the following (but we won't).

Lemma: Consider a digraph G on which DFS has been run. Label each component with the maximum finish time of all the vertices in the component, and sort these in decreasing order. Then this order is a topological order for the component digraph.

For example, in the previous figure, the maximum finish times for each component are 18 (for $\{a, b, c\}$), 17 (for $\{d, e\}$), and 12 (for $\{f, g, h, i\}$). The order $\langle 18, 17, 12\rangle$ is a valid topological order for the component digraph.

The problem is that this is not what we wanted. We wanted a reverse topological order for the component digraph. So, the final trick is to reverse the digraph. This does not change the component graph, but it reverses the topological order, as desired.

Lecture 14: Minimum Spanning Trees and Kruskal's Algorithm

(Tuesday, Oct 19, 1999)
Read: Chapt 24 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 a 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 three spanning trees for the same graph, where the shaded rectangles indicate the edges in the spanning tree. The one on the left is not a minimum spanning tree, and the other two are. (An interesting observation is that not only do the edges sum to the same value, but in fact the same set of edge weights appear in the two MST's. Is this a coincidence? We'll see later.)

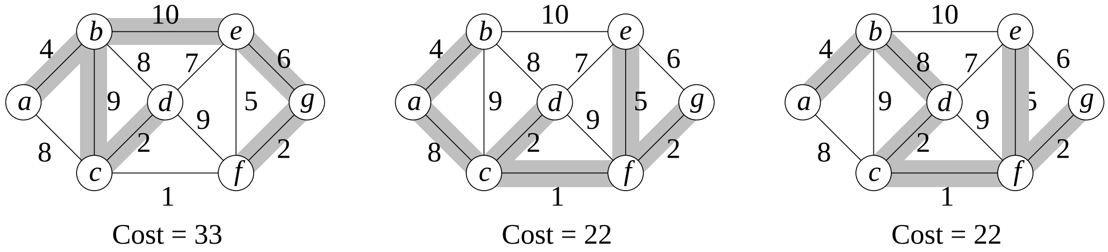


Figure 32: Spanning trees (the middle and right are minimum spanning trees.

Steiner Minimum Trees: Minimum spanning trees are actually mentioned in the U.S. legal code. The reason is that AT&T was a government supported monopoly at one time, and was responsible for handling all telephone connections. If a company wanted to connect a collection of installations by an private internal phone system, AT&T was required (by law) to connect them in the minimum cost manner, which is clearly a spanning tree ...or is it?

Some companies discovered that they could actually reduce their connection costs by opening a new bogus installation. Such an installation served no purpose other than to act as an intermediate point for connections. An example is shown in the figure below. On the left, consider four installations that that lie at the corners of a 1×1 square. Assume that all edge lengths are just Euclidean distances. It is easy to see that the cost of any MST for this configuration is 3 (as shown on the left). However, if you introduce a new installation at the center, whose distance to each of the other four points is $1/\sqrt{2}$. It is now possible to connect these five points with a total cost of of $4/\sqrt{2} = 2\sqrt{2} \approx 2.83$. This is better than the MST.

In general, the problem of determining the lowest cost interconnection tree between a given set of nodes, assuming that you are allowed additional nodes (called *Steiner points*) is called

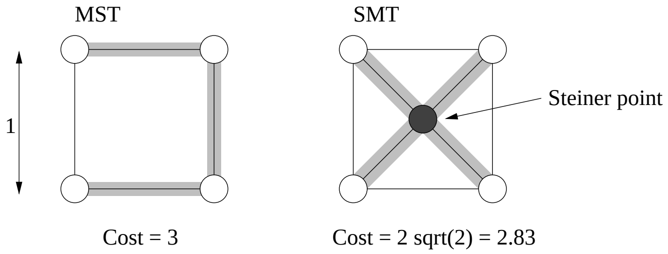


Figure 33: Steiner Minimum tree.

the *Steiner minimum tree* (or SMT for short). An interesting fact is that although there is a simple greedy algorithm for MST's (as we will see below), the SMT problem is much harder, and in fact is NP-hard. (By the way, the US Legal code is rather ambiguous on the point as to whether the phone company was required to use MST's or SMT's in making connections.)

Generic approach: We will present two *greedy* algorithms (Kruskal's and Prim's algorithms) for computing a minimum spanning tree. Recall that a *greedy algorithm* is one that builds a solution by repeated selecting the cheapest (or generally locally optimal choice) among all options at each stage. An important characteristic of greedy algorithms is that once they make a choice, they never "unmake" this choice. Before presenting these algorithms, let us review 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.
- There exists a unique path between any two vertices of a free tree.
- Adding any edge to a free tree creates a unique cycle. Breaking *any* edge on this cycle restores a free tree.

Let $G = (V, E)$ be an undirected, connected graph whose edges have numeric edge weights (which may be positive, negative or zero). The intuition behind the greedy MST algorithms is simple, we maintain a subset of edges A, which will initially be empty, and we will add edges one at a time, until A equals the MST. We say that a subset $A \subseteq E$ is *viable* if A is a subset of edges in *some* MST. We say that an edge $(u, v) \in E - A$ is *safe* if $A \cup \{(u, v)\}$ is viable. In other words, the choice (u, v) is a safe choice to add so that A can still be extended to form an MST. Note that if A is viable it cannot contain a cycle. A generic greedy algorithm operates by repeatedly adding any *safe* edge to the current spanning tree.

When is an edge 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. 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.

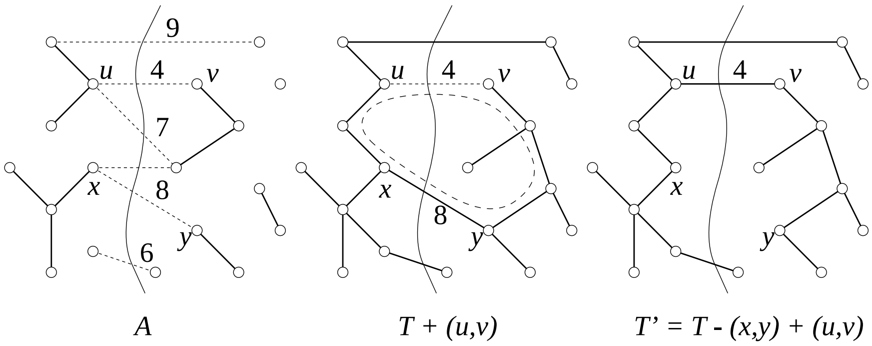


Figure 34: MST Lemma.

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(u): Create a set containing a single item v.

Find-Set(u): Find the set that contains a given item u.

Union(u, v): Merge the set containing u and the set containing v into a common set.

You are not responsible for knowing how this data structure works (see Chapt 22 in CLR). You may use it as a "black-box". For our purposes it suffices to know that each of these operations can be performed in $O(\log n)$ time, on a set of size n. (The Union-Find data structure is quite interesting, because it can actually perform a sequence of n operations much faster than $O(n \log n)$ time. However we will not go into this here. $O(\log n)$ time is fast enough for its use in Kruskal's algorithm.)

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. The algorithm an an example are shown below.

Kruskal's Algorithm

```
Kruskal(G=(V,E),w) {
    A = {};                                 // initially A is empty
    for (each u in V) Create_Set(u);        // create set for each vertex
    Sort E in increasing order by weight w;
    for each ((u,v) from the sorted list) {
        if (Find_Set(u) != Find_Set(v)) {   // if u and v are in different trees
            Add (u,v) to A;
            Union(u, v);
        }
    }
    return A;
}
```

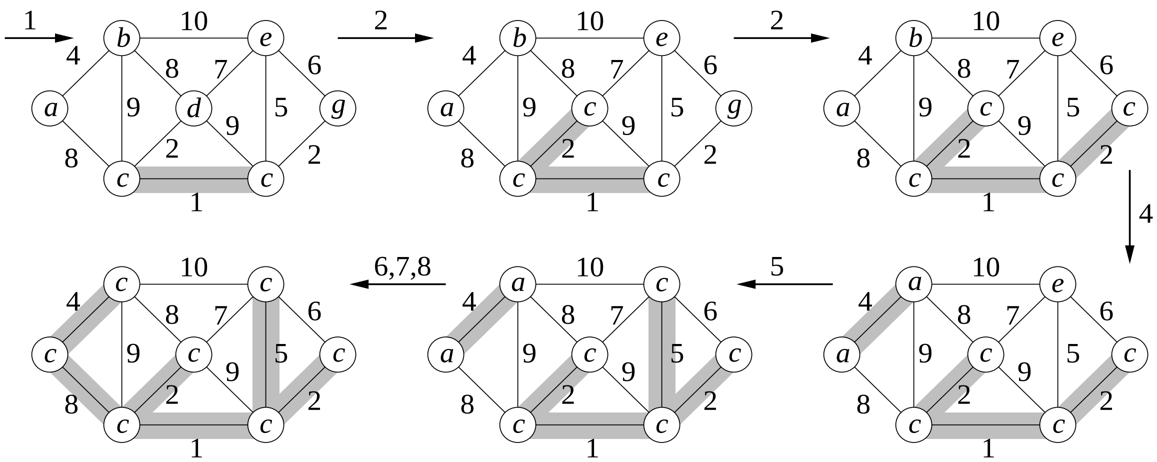


Figure 35: Kruskal's Algorithm.

Analysis: How long does Kruskal's algorithm take? As usual, let V be the number of vertices and E be the number of edges. Since the graph is connected, we may assume that $E \geq V - 1$. Observe that it takes $\Theta(E \log E)$ time to sort the edges. The for-loop is iterated E times, and each iteration involves a constant number of accesses to the Union-Find data structure on a collection of V items. Thus each access is $\Theta(V)$ time, for a total of $\Theta(E \log V)$. Thus the total running time is the sum of these, which is $\Theta((V + E) \log V)$. Since V is asymptotically no larger than E, we could write this more simply as $\Theta(E \log V)$.

Lecture 15: Prim's and Baruvka's Algorithms for MSTs

(Thursday, Oct 21, 1999)

Read: Chapt 24 in CLR. Baruvka's algorithm is not described 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, $O((V + E) \log V)$. 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.

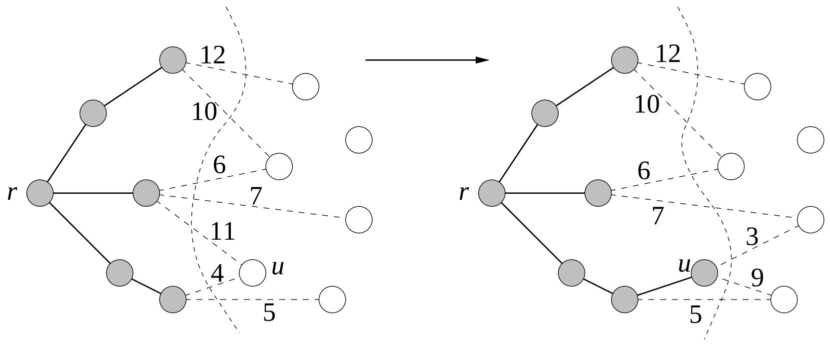


Figure 36: Prim's Algorithm.

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. 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 questions 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 *priority queue* data structure. Recall that this is the data structure used in HeapSort. This is a data structure that stores a set of items, where each item is associated with a *key* value. The priority queue supports three operations.

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

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

`decreaseKey(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 not edge from u to a vertex in $V - S$, 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) {                 // initialization
        key[u] = +infinity;
        color[u] = white;
    }
    key[r] = 0;                         // start at root
    pred[r] = nil;
    Q = new PriQueue(V);                // put vertices in Q
    while (Q.nonEmpty()) {              // until all vertices in MST
        u = Q.extractMin();             // vertex with lightest edge
        for each (v in Adj[u]) {
            if ((color[v] == white) && (w(u,v) < key[v])) {
                key[v] = w(u,v);        // new lighter edge out of v
                Q.decreaseKey(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.

To analyze Prim's algorithm, we account for the time spent on each vertex as it is extracted from the priority queue. It takes $O(\log V)$ to extract this vertex from the queue. For each incident edge, we spend potentially $O(\log V)$ time decreasing the key of the neighboring vertex.

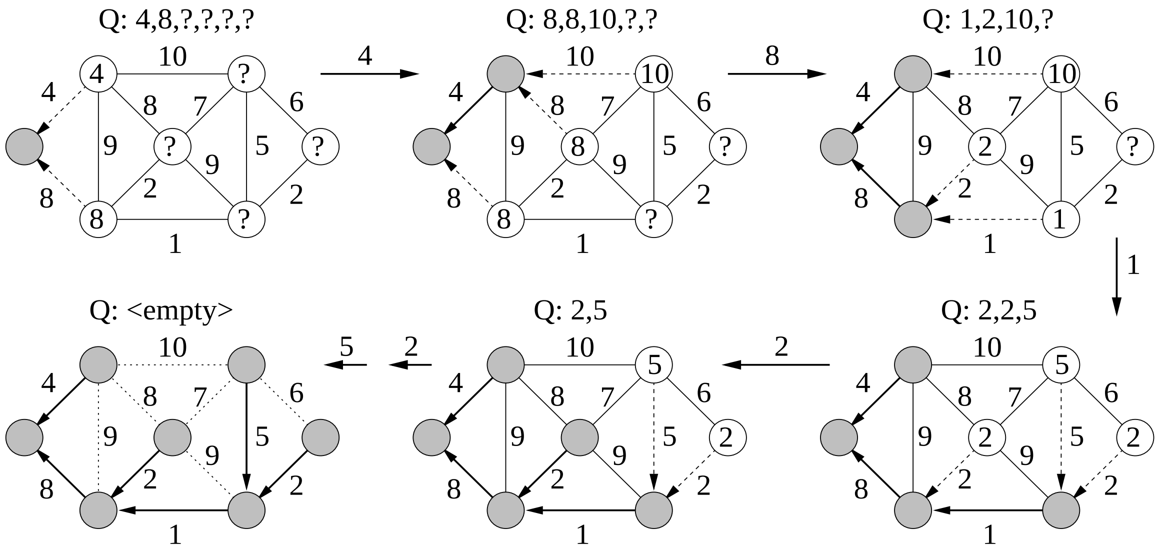


Figure 37: Prim's Algorithm.

Thus the time is $O(\log V + deg(u)\log V)$ time. The other steps of the update are constant time. So the overall running time is

$$\begin{aligned} T(V,E) &= \sum_{u\in V}(\log V + deg(u)\log V) \;=\; \sum_{u\in V}(1+deg(u))\log V \\ &= \log V \sum_{u\in V}(1+deg(u)) \;=\; (\log V)(V+2E) \;=\; \Theta((V+E)\log V). \end{aligned}$$

Since G is connected, V is asymptotically no greater than E, so this is $\Theta(E\log V)$. This is exactly the same as Kruskal's algorithm.

Baruvka's Algorithm: We have seen two ways (Kruskal's and Prim's algorithms) for solving the MST problem. So, it may seem like complete overkill to consider yet another algorithm. This one is called Baruvka's algorithm. It is actually the oldest of the three algorithms (invented in 1926, well before the first computers). The reason for studying this algorithm is that of the three algorithms, it is the easiest to implement on a parallel computer. Unlike Kruskal's and Prim's algorithms, which add edges one at a time, Baruvka's algorithm adds a whole set of edges all at once to the MST.

Baruvka's algorithm is similar to Kruskal's algorithm, in the sense that it works by maintaining a collection of disconnected trees. Let us call each subtree a *component*. Initially, each vertex is by itself in a one-vertex component. Recall that with stage of Kruskal's algorithm, we add the lightest-weight edge that connects two different components together. To prove Kruskal's algorithm correct, we argued (from the MST Lemma) that the lightest such edge will be *safe* to add to the MST.

In fact, a closer inspection of the proof reveals that the cheapest edge leaving *any* component is always safe. This suggests a more parallel way to grow the MST. Each component determines the lightest edge that goes from inside the component to outside the component (we don't care where). We say that such an edge *leaves* the component. Note that two components might select the same edge by this process. By the above observation, all of these edges are safe, so we may add them all at once to the set A of edges in the MST. As a result, many components will be merged together into a single component. We then apply DFS to the edges of A, to identify the new components. This process is repeated until only one component remains. A fairly high-level description of Baruvka's algorithm is given below.

Baruvka's Algorithm

```
Baruvka(G=(V,E), w) {
    initialize each vertex to be its own component;
    A = {};                                 // A holds edges of the MST
    do {
        for (each component C) {
            find the lightest edge (u,v) with u in C and v not in C;
            add {u,v} to A (unless it is already there);
        }
        apply DFS to graph H=(V,A), to compute the new components;
    } while (there are 2 or more components);
    return A;                               // return final MST edges
```

There are a number of unspecified details in Baruvka's algorithm, which we will not spell out in detail, except to note that they can be solved in $\Theta(V+E)$ time through DFS. First, we may apply DFS, but only traversing the edges of A to compute the components. Each DFS tree will correspond to a separate component. We label each vertex with its component number as part of this process. With these labels it is easy to determine which edges go between components (since their endpoints have different labels). Then we can traverse each component again to determine the lightest edge that leaves the component. (In fact, with a little more cleverness, we can do all this without having to perform two separate DFS's.) The algorithm is illustrated in the figure below.

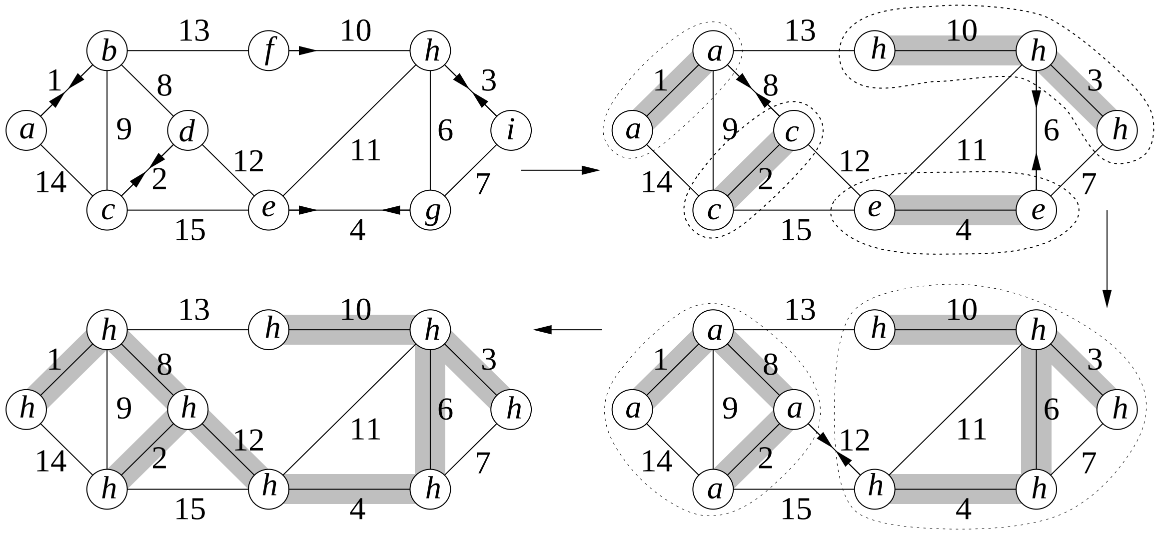


Figure 38: Baruvka's Algorithm.

Analysis: How long does Baruvka's algorithm take? Observe that because each iteration involves doing a DFS, each iteration (of the outer do-while loop) can be performed in $\Theta(V+E)$ time. The question is how many iterations are required in general? We claim that there are never more than $O(\log n)$ iterations needed. To see why, let m denote the number of components at some stage. Each of the m components, will merge with at least one other component. Afterwards the number of remaining components could be a low as 1 (if they all merge together), but never higher than $m/2$ (if they merge in pairs). Thus, the number of components decreases by at least half with each iteration. Since we start with V components, this can happen at most $\lg V$ time, until only one component remains. Thus, the total running time is $\Theta((V + E) \log V)$ time. Again, since G is connected, V is asymptotically no larger than E, so we can write this more succinctly as $\Theta(E \log V)$. Thus all three algorithms have the same asymptotic running time.

Lecture 16: Review for the Midterm

(Thursday, Oct 26, 1999)

General Remarks: Recall that our overall goal in this course is to provide a understanding of the tools used in designing and analyzing efficient algorithms. So far the main topics have included issues related to sorting, dynamic programming, greedy algorithms, and a bit on graph algorithms.

Material from Text: You are only responsible for material that has been covered in class or on class assignments. However it is always a good idea to see the text to get a better insight into some of the topics we have covered. See the class web page (in the lecture's section) for the relevant chapters to read. The only material that was not covered in CLR was the material on constructive induction and Baruvka's algorithm.

Cheat Sheet: Recall that the exam is closed-book, closed-notes, but you are allowed a sheet of notes (front and back). Use this sheet to write down things like:

Important formulas: E.g., summations such as $\sum i$, $\sum c^i$, the Master Theorem for recurrences, and maybe a few of the major recurrences like $T(n) = 2T(n/2) + n = \Theta(n \log n)$.

Definitions: For example, what is a strong component or a minimum spanning tree. Be sure to note whether the concept is defined for graphs or digraphs, weighted or unweighted.

Basic lemmas: Such as, a tree with n vertices has $n - 1$ edges, and the MST lemma.

Short descriptions of algorithms: For each algorithm specify the input and any assumptions on input structure, the output and asymptotic running time. Also write down a brief description of the algorithm (full pseudo-code is less important than something that makes sense to you). But do not rely on this. It is important that you have an intimate working knowledge of the algorithm.

After you do this, then *learn* everything on your cheat sheet so well that you don't need to look at it during the exam. Also, for each algorithm, be sure that you can work it out on an example without consulting your cheat sheet. Memorization of the text of the algorithm is not really important. What is important is that you understand how the algorithm works internally, since this is what you will need to answer questions about the algorithm.

Topics: Analysis: Summations, asymptotics, recurrences. Expect at least one question that requires solving a summation or recurrence, or analyzing an algorithm's running time.

Sorting:

Basic Sorting: Review the basic sorting algorithms from 251 (MergeSort, HeapSort, QuickSort). Also recall that given a list of n numbers, the k-th smallest can be found in $\Theta(n)$ time.

Sorting Lower bounds: Assuming comparisons are used, you cannot sort faster than $\Omega(n \log n)$ time. Understand how decision trees work.

Counting sort: You can sort n integers in the range 1 to k in $\Theta(n + k)$ time.

Radix sort: You can sort integers that have been broken into d digits in the range 0 to k in $\Theta(d(n + k))$ time. We showed that this can be used to sort n integers in the range 1 to n^d in $\Theta(dn)$ time.

Bucket Sort: You can sort n numbers uniformly distributed over some range in $\Theta(n)$ expected time.

Dynamic Programming: We covered DP algorithms for LCS (sequence structured), chain matrix-multiplication and minimum weight triangulation (both tree structured). We also presented Knapsack on a homework. These are common DP structures. Recall that DP involves constructing a table in a bottom-up manner, but memoization can be used to building things in a top-down manner.

Greedy Algorithms: We covered Huffman coding, activity scheduling, and fractional knapsack. Also, all the MST algorithms we saw are also examples of greedy algorithms. Greedy algorithms are generally very simple (just make the best choice at each stage). The main issue is proving their correctness.

Graph algorithms: We began by discussing graph and digraph representations, adjacency lists and adjacency matrices. We have also presented a number of algorithms.

BFS: Traverse a graph in breadth-first order. Recall that this is similar to DFS, but instead of storing unprocessed vertices in a stack, we use a queue. BFS can be applied to omputing the shortest paths from a single source in a digraph without edge weights in $\Theta(V + E)$ time.

DFS: Traverses a graph in depth-first order. Useful for many problems in $\Theta(V + E)$ time.

Cycle Detection: Determine whether a graph or digraph has a cycle. (Look for any backedge.)

Topological Sort: Sort the vertices of a DAG in topological order. (Vertices finish in reverse topological order.)

Connected components: Break a graph into its connected components. (Each DFS tree is a connected component.)

Strong components: Break a digraph into its strongly connected components. (This was the weird algorithm that reversed the digraph edges.)

MST: Compute the minimum spanning tree of a weighted graph. We presented three algorithms that run in $\Theta((V + E) \log V)$ time: Kruskal's, Prim's, and Baruvka's algorithms.

Possible Questions: You should expect a question of the form: work through some algorithm on a given input, showing intermediate results. I'll pick the algorithm at random, because I want you to learn them all. For some more complex algorithms (e.g. chain matrix multiplication) I may only ask you to do one iteration. You should also expect a series of short-answer problems (as appear in the practice problems). Although the answers are short, some of the problems may require a lot of thought. Don't worry about skipping a problem (or guessing an answer) if you don't see the answer right away.

The other problems will involve giving proofs, solving summations or recurrences, designing and analyzing algorithms. These are much more unpredictable. I try to ask at least a couple of questions that involve a very simple modification to a homework problem, so be sure you understand the solutions to all the homework problems (including the practice problems). Other questions usually involve a slight modification to a problem that we have worked on.

Finally, I usually have at least one (usually the last) challenging problem. Don't be upset if you cannot solve this problem. (But at least be sure to read it, since sometimes the problem is not as challenging as I think it is, and there may be a very simple solution.) If you cannot see how to solve the problem, feel free to write down your wild ideas or observations. You might get some partial credit if your idea is good.

When it comes to grading, I tend to give more credit for "insight" than for "memorization". For example, if a problem involves spanning trees, you will not get any credit for writing down Prim's algorithm. (You could just copy that out of your cheat sheet.) However, if you can say something intelligent about the mathematical structure of the particular problem, or about why some approach is not useful, you may get some partial credit (depending on how much your observation indicates about your understanding of the problem).

Lecture 17: Midterm Exam

(Thursday, Oct 28, 1999)
Midterm exam. No lecture today.

Lecture 18: Midterm Review

(Tuesday, Nov 2, 1999)
Today's lecture was a review of the midterm exam and discussion of the programming project.

Lecture 19: Dijkstra's Algorithm for Shortest Paths

(Thursday, Nov 4, 1999)
Read: Chapt 25 in CLR.

Shortest Paths: Consider the problem of computing shortest paths in a directed graph. We have already seen that breadth-first search is an $O(V + E)$ 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 single pair of vertices. Most algorithms for this problem are variants of the single-source algorithm that we will present. 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.

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 considering path of 0 edges from u to itself.)

The *single source shortest path* problem is as follows. We are given a directed graph with *nonnegative* edge weights $G = (V, E)$ and a distinguished *source vertex*, $s \in V$. The problem is to 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 greedy 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.

Shortest Paths and Relaxation: 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.) Intuitively $d[v]$ will be the length of the shortest path *that the algorithm knows of* from s to v. This, value will always greater than or equal to the true shortest path distance from s to v. Initially, we know of no paths, so $d[v] = \infty$. 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]) {          // is the path through u shorter?
        d[v] = d[u] + w(u,v);            // yes, then take it
        pred[v] = u;                     // record that we go through 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(u,v)` 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.

Dijkstra's Algorithm: Dijkstra's algorithm 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 $S = \emptyset$, 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 the vertex for which $d[u]$ is minimum, that is, take the unprocessed vertex that is closest (by our estimate) to s. Later we will justify 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 is 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()`, on a priority queue of size n each in $O(\log n)$ time. Each vertex "knows" its location in the priority queue (e.g. has a cross reference link to the priority queue entry), and each entry in the priority queue "knows" which vertex it represents. It is important when implementing the priority queue that this cross reference information is updated.

Here is Dijkstra's algorithm. Note the remarkable similarity to Prim's algorithm.

Dijkstra's Algorithm

```
Dijkstra(G,w,s) {
    for each (u in V) {                         // initialization
        d[u] = +infinity;
        color[u] = white;
        pred[u] = nil;
    }
    d[s] = 0;
    Q = build initial queue with all vertices;
    while (Non_Empty(Q)) {                      // until all vertices processed
        u = Extract_Min(Q);                     // select closest to s
        for each (v in Adj[u]) {
            if (d[u] + w(u,v) < d[v]) {         // 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]
}
```

Notice that the coloring is not really used by the algorithm, but it has been included to make the connection with the correctness proof a little clearer. Because of the similarity between this and Prim's algorithm, the running time is the same, namely $\Theta(E \log V)$.

Correctness: Recall that $d[v]$ is the distance value assigned to vertex v by Dijkstra's algorithm, and let $\delta(s, v)$ denote the length of the true shortest path from s to v. 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, $d[u] = \delta(s, u)$.

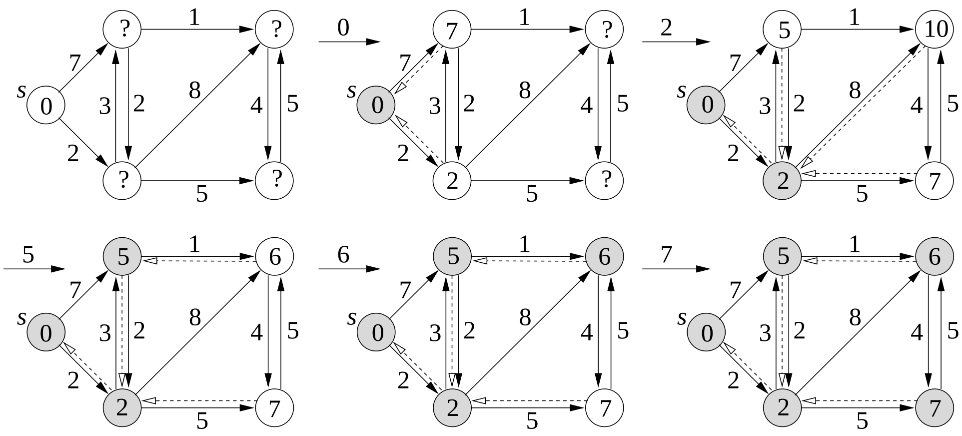


Figure 39: Dijkstra's Algorithm.

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, $d[u]$ is never less than $\delta(s,u)$, thus we have $d[u] > \delta(s,u)$. 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 jump out of S. Let (x, y) be the edge taken by the path, where $x \in S$ and $y \in V - S$. (Note that it may be that $x = s$ and/or $y = u$).

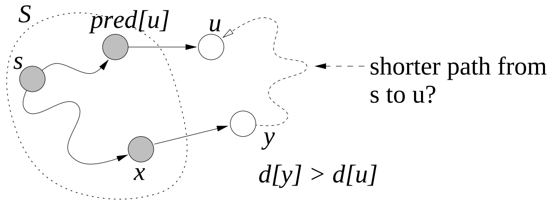


Figure 40: Correctness of Dijkstra's Algorithm.

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 somewhere along the shortest path from s to u (but not at u) and all subsequent edges following y are of positive weight, we have $\delta(s,y) < \delta(s,u)$, and thus

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

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.

Lecture 20: More Shortest Paths: Bellman-Ford and Floyd-Warshall

(Tuesday, Nov 9, 1999)

Read: Section 25.3 and 26.2 in CLR.

Bellman-Ford Algorithm: We saw that Dijkstra's algorithm can solve the single-source shortest path problem, under the assumption that the edge weights are nonnegative. We also saw that shortest paths are undefined if you have cycles of total negative cost. What if you have negative edge weights, but no negative cost cycles? We shall present the Bellman-Ford algorithm, which solves this problem. This algorithm is slower that Dijkstra's algorithm, running in $\Theta(VE)$ time. In our version we will assume that there are no negative cost cycles. The one presented in CLR actually contains a bit of code that checks for this. (Check it out.)

Recall that we are given a graph $G = (V, E)$ with numeric edge weights, $w(u, v)$. Like Dijkstra's algorithm, the Bellman-Ford algorithm is based on performing repeated relaxations. (Recall that relaxation updates shortest path information along a single edge. It was described in our discussion of Dijkstra's algorithm.) Dijkstra's algorithm was based on the idea of organizing the relaxations in the best possible manner, namely in increasing order of distance. Once relaxation is applied to an edge, it need never be relaxed again. This trick doesn't seem to work when dealing with graphs with negative edge weights. Instead, the Bellman-Ford algorithm simply applies a relaxation to *every* edge in the graph, and repeats this $V - 1$ times.

Bellman-Ford Algorithm

```
BellmanFord(G,w,s) {
    for each (u in V) {                     // standard initialization
        d[u] = +infinity;
        pred[u] = nil;
    }
    d[s] = 0;
    for i = 1 to V-1 {                      // repeat V-1 times
        for each (u,v) in E {               // relax along each edge
            Relax(u,v);
        }
    }
}
```

The $\Theta(VE)$ running time is pretty obvious, since there are two main nested loops, one iterated $V - 1$ times and the other iterated E times. The interesting question is how and why it works.

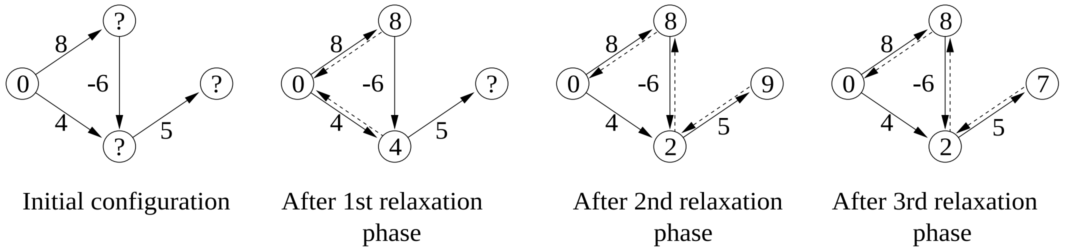


Figure 41: Bellman-Ford Algorithm.

Correctness of Bellman-Ford: I like to think of the Bellman-Ford as a sort of "BubbleSort analogue" for shortest paths, in the sense that shortest path information is propagated sequentially along each shortest path in the graph. Consider any shortest path from s to some other vertex u: $\langle v_0, v_1, \ldots, v_k \rangle$ where $v_0 = s$ and $v_k = u$. Since a shortest path will never visit the same vertex twice, we know that $k \le V - 1$, and hence the path consists of at most $V - 1$ edges.

Since this is a shortest path we have $\delta(s, v_i)$ (the true shortest path cost from s to v_i) satisfies

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i).$$

We assert that after the ith pass of the "for-i" loop that $d[v_i] = \delta(s, v_i)$. The proof is by induction on i. Observe that after the initialization (pass 0) we have $d[v_1] = d[s] = 0$. In general, prior to the ith pass through the loop, the induction hypothesis tells us that $d[v_{i-1}] = \delta(s, v_{i-1})$. After the ith pass through the loop, we have done a relaxation on the edge (v_{i-1}, v_i) (since we do relaxations along all the edges). Thus after the ith pass we have

$$d[v_i] \;\leq\; d[v_{i-1}] + w(v_{i-1}, v_i) \;=\; \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \;=\; \delta(s, v_i).$$

Recall from Dijkstra's algorithm that $d[v_i]$ is never less than $\delta(s, v_i)$ (since each time we do a relaxation there exists a path that witnesses its value). Thus, $d[v_i]$ is in fact equal to $\delta(s, v_i)$, completing the induction proof.

In summary, after i passes through the for loop, all vertices that are i edges away (along the shortest path tree) from the source have the correct distance values stored in $d[u]$. Thus, after the $(V - 1)$st iteration of the for loop, all vertices u have the correct distance values stored in $d[u]$.

All-Pairs Shortest Paths: We consider the generalization of the shortest path problem, to computing shortest paths between all pairs of vertices. Let $G = (V, E)$ be a directed graph with edge weights. If (u, v) E, is an edge of G, then the weight of this edge is denoted $w(u, v)$. Recall that the *cost* of a path is the sum of edge weights along the path. The *distance* between two vertices $\delta(u, v)$ is the cost of the minimum cost path between them. We will allow G to have negative cost edges, but we will not allow G to have any negative cost cycles.

We consider the problem of determining the cost of the shortest path between all pairs of vertices in a weighted directed graph. We will present a $\Theta(n^3)$ algorithm, called the *Floyd-Warshall algorithm*. This algorithm is based on *dynamic programming*.

For this algorithm, we will assume that the digraph is represented as an adjacency matrix, rather than the more common adjacency list. Although adjacency lists are generally more efficient for sparse graphs, storing all the inter-vertex distances will require $\Omega(n^2)$ storage, so the savings is not justified here. Because the algorithm is matrix-based, we will employ common matrix notation, using i, j and k to denote vertices rather than u, v, and w as we usually do.

Input Format: The input is an $n \times n$ matrix w of edge weights, which are based on the edge weights in the digraph. We let w_{ij} denote the entry in row i and column j of w.

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

Setting $w_{ij} = \infty$ if there is no edge, intuitively means that there is no direct link between these two nodes, and hence the direct cost is infinite. The reason for setting $w_{ii} = 0$ is that the cost of getting from any vertex to should be 0, since we have no distance to travel. Note that in digraphs it is possible to have self-loop edges, and so $w(i, i)$ may generally be nonzero. It cannot be negative (since there are no negative cost cycles), and if it is positive, then there is no point in using it as part of any shortest path.

The output will be an $n \times n$ distance matrix $D = d_{ij}$ where $d_{ij} = \delta(i, j)$, the shortest path cost from vertex i to j. Recovering the shortest paths will also be an issue. To help us do this, we will also compute an auxiliary matrix $mid[i, j]$. The value of $mid[i, j]$ will be a vertex that is somewhere along the shortest path from i to j. If the shortest path travels directly from i to j without passing through any other vertices, then $mid[i, j]$ will be set to *null*. We will see later than using these values it will be possible to reconstruct any shortest path in $\Theta(n)$ time.

Floyd-Warshall Algorithm: 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. The Floyd-Warshall algorithm runs in $\Theta(n^3)$ time. The genius of the algorithm is in finding a clever formulation for the shortest path subproblem. 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* of this path. Note that a path consisting of a single edge has no intermediate vertices.

Formulation: 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 among $\{1, 2, \ldots, k\}$. The path is free to visit any subset of these vertices, and to do so in any order. For example, in the digraph shown in the following figure, notice how the value of $d_{32}^{(k)}$ changes as k varies.

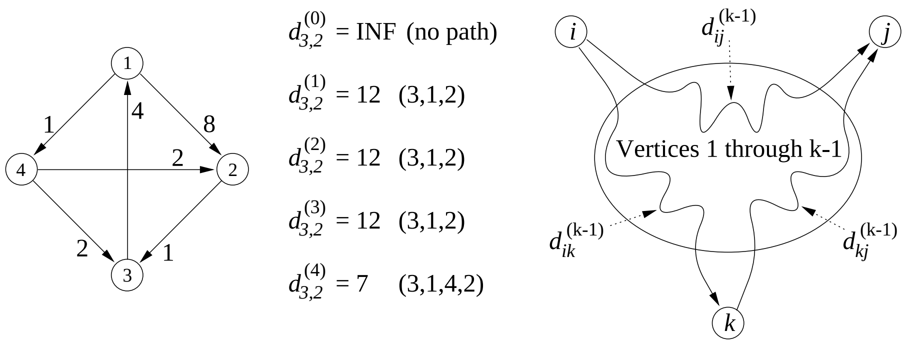


Figure 42: Floyd-Warshall Formulation.

Floyd-Warshall Update Rule: How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous matrix $d^{(k-1)}$? There are two basic cases, depending on the ways that we might get from vertex i to vertex j, assuming that the intermediate vertices are chosen from $\{1, 2, \ldots, k\}$:

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 $d_{ij}^{(k-1)}$.

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. Since 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 suggests the following recursive rule (the DP formulation) 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. We could write a recursive program to compute $d_{ij}^{(k)}$, but this will be prohibitively slow because the same value may be reevaluated many times. Instead, we compute it by storing the values in a table, and looking the values up as we need them. Here is the complete algorithm. We have also included mid-vertex pointers, $mid[i,j]$ for extracting the final shortest paths. We will discuss them later.

Floyd-Warshall Algorithm

```
Floyd_Warshall(int n, int w[1..n, 1..n]) {
    array d[1..n, 1..n]
    for i = 1 to n do {                           // initialize
        for j = 1 to n do {
            d[i,j] = W[i,j]
            mid[i,j] = null
        }
    }
    for k = 1 to n do                             // use intermediates {1..k}
        for i = 1 to n do                         // ...from i
            for j = 1 to n do                     // ...to j
                if (d[i,k] + d[k,j]) < d[i,j]) {
                    d[i,j] = d[i,k] + d[k,j]      // new shorter path length
                    mid[i,j] = k                  // new path is through k
                }
    return d                                      // matrix of final distances
}
```

Clearly the algorithm's running time is $\Theta(n^3)$. The space used by the algorithm is $\Theta(n^2)$. 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. (Hint: The danger is that values may be overwritten and then used later in the same phase. Consider which entries might be overwritten and then reused, they occur in row k and column k. It can be shown that the overwritten values are equal to their original values.) An example is shown in the following figure.

Extracting Shortest Paths: The mid-vertex pointers $mid[i,j]$ 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 $mid[i,j] = k$. If the shortest path does not pass through any intermediate vertex, then $mid[i,j] = null$. To find the shortest path from i to j, we consult $mid[i,j]$. If it is *null*, then the shortest path is just the edge (i,j). Otherwise, we recursively compute the shortest path from i to $mid[i,j]$ and the shortest path from $mid[i,j]$ to j.

Printing the Shortest Path

```
Path(i,j) {
    if (mid[i,j] == null)                        // path is a single edge
        output(i,j)
    else {                                       // path goes through mid
        Path(i, mid[i,j])                        // print path from i to mid
```

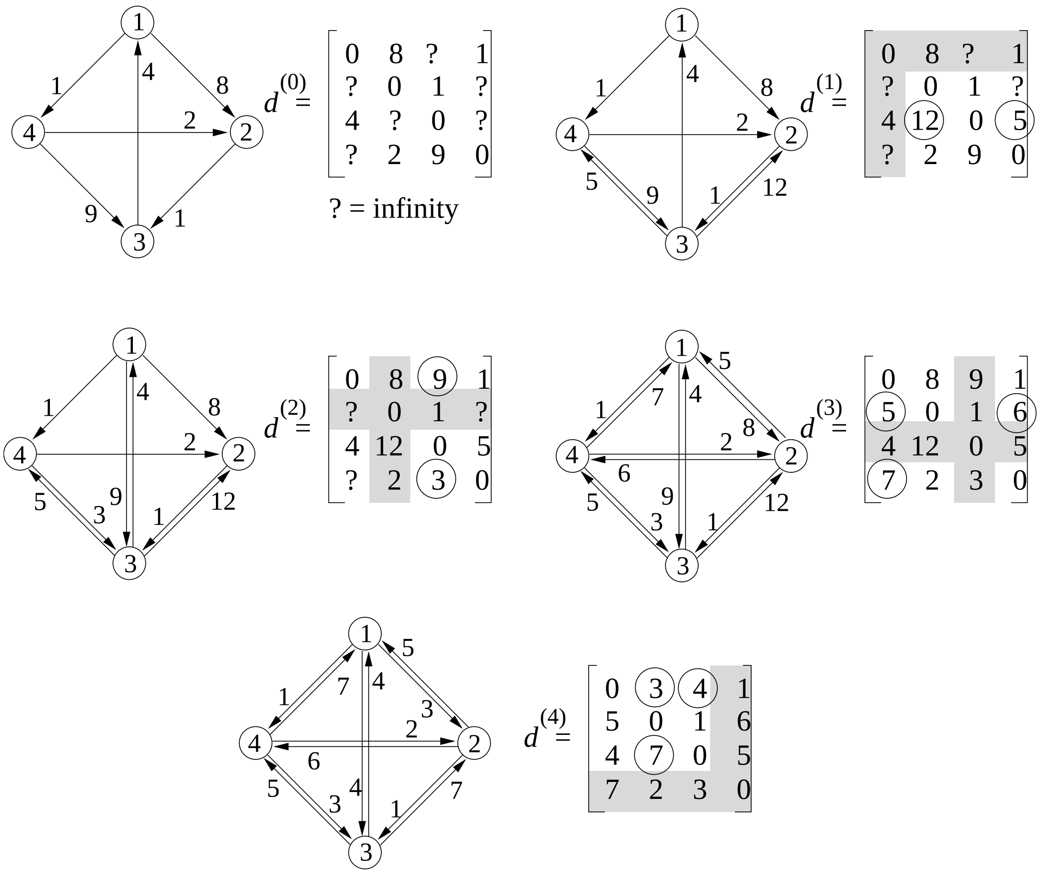


Figure 43: Floyd-Warshall Example. Newly updates entries are circled.

```
            Path(mid[i,j], j)                       // print path from mid to j
        }
    }
```

Lecture 21: NP-Completeness: Languages and NP

(Thursday, Nov 11, 1999)
Read: Chapt 36, Up through section 36.2.

Complexity Theory: At this point of the semester we have been building up your "bag of tricks" for solving algorithmic problems. Hopefully when presented with a problem you now have a little better idea of how to go about solving the problem. What sort of design paradigm should be used (divide-and-conquer, DFS, greedy, dynamic programming), what sort of data structures might be relevant (trees, heaps, graphs) and what representations would be best (adjacency list, adjacency matrices), what is the running time of your algorithm.

All of this is fine if it helps you discover an acceptably efficient algorithm to solve your problem. The question that often arises in practice is that you have tried every trick in the book, and nothing seems to work. Although your algorithm can solve small problems reasonably efficiently (e.g. $n \leq 20$) the really large applications that you want to solve (e.g. $n = 1,000$ or $n = 10,000$) your algorithm never terminates. When you analyze its running time, you realize that it is running in exponential time, perhaps $n^{\sqrt{n}}$, or 2^n, or $2^{(2^n)}$, or $n!$, or worse!

Near the end of the 60's where there was great success in finding efficient solutions to many combinatorial problems, but there was also a growing list of problems for which there seemed to be no known efficient algorithmic solutions. People began to wonder whether there was some unknown paradigm that would lead to a solution to these problems, or perhaps some proof that these problems are inherently hard to solve and no algorithmic solutions exist that run under exponential time.

Near the end of the 60's a remarkable discovery was made. Many of these hard problems were interrelated in the sense that if you could solve any one of them in polynomial time, then you could solve all of them in polynomial time. This discovery gave rise to the notion of NP-completeness, and created possibly the biggest open problems in computer science: is P = NP? We will be studying this concept over the next few lectures.

This area is a radical departure from what we have been doing because the emphasis will change. The goal is no longer to prove that a problem *can* be solved efficiently by presenting an algorithm for it. Instead we will be trying to show that a problem *cannot* be solved efficiently. The question is how to do this?

Laying down the rules: We need some way to separate the class of efficiently solvable problems from inefficiently solvable problems. We will do this by considering problems that can be solved in polynomial time.

When designing algorithms it has been possible for us to be rather informal with various concepts. We have made use of the fact that an intelligent programmer could fill in any missing details. However, the task of proving that something cannot be done efficiently must be handled much more carefully, since we do not want leave any "loopholes" that would allow someone to subvert the rules in an unreasonable way and claim to have an efficient solution when one does not really exist.

We have measured the running time of algorithms using worst-case complexity, as a function of n, the size of the input. We have defined input size variously for different problems, but the bottom line is the number of bits (or bytes) that it takes to represent the input using any *reasonably efficient encoding*. By a reasonably efficient encoding, we assume that there is not some significantly shorter way of providing the same information. For example, you could write numbers in unary notation $11111111_1 = 100_2 = 8$ rather than binary, but that would be unacceptably inefficient. You could describe graphs in some highly inefficient way, such as by listing all of its cycles, but this would also be unacceptable. We will assume that numbers are expressed in binary or some higher base and graphs are expressed using either adjacency matrices or adjacency lists.

We will usually restrict numeric inputs to be integers (as opposed to calling them "reals"), so that it is clear that arithmetic can be performed efficiently. We have also assumed that operations on numbers can be performed in constant time. From now on, we should be more careful and assume that arithmetic operations require at least as much time as there are bits of precision in the numbers being stored.

Up until now all the algorithms we have seen have had the property that their worst-case running times are bounded above by some *polynomial* in the input size, n. A *polynomial time algorithm* is any algorithm that runs in time $O(n^k)$ where k is some constant that is independent of n. A problem is said to be *solvable in polynomial time* if there is a polynomial time algorithm that solves it.

Some functions that do not "look" like polynomials (such as $O(n \log n)$) are bounded above by polynomials (such as $O(n^2)$). Some functions that do "look" like polynomials are not. For example, suppose you have an algorithm which inputs a graph of size n and an integer k and runs in $O(n^k)$ time. Is this a polynomial? No, because k is an input to the problem, so the user is allowed to choose $k = n$, implying that the running time would be $O(n^n)$ which is *not* a polynomial in n. The important thing is that the exponent must be a *constant independent of* n.

Of course, saying that all polynomial time algorithms are "efficient" is untrue. An algorithm whose running time is $O(n^{1000})$ is certainly pretty inefficient. Nonetheless, if an algorithm runs in worse than polynomial time (e.g. 2^n), then it is certainly not efficient, except for very small values of n.

Decision Problems: Many of the problems that we have discussed involve *optimization* of one form or another: find the shortest path, find the minimum cost spanning tree, find the minimum weight triangulation. For rather technical reasons, most NP-complete problems that we will discuss will be phrased as 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, accept/reject).

We will phrase many optimization problems in terms of decision problems. For example, the minimum spanning tree decision problem might be: Given a weighted graph G and an integer k, does G have a spanning tree whose weight is at most k?

This may seem like a less interesting formulation of the problem. It does not ask for the weight of the minimum spanning tree, and it does not even ask for the edges of the spanning tree that achieves this weight. However, our job will be to show that certain problems *cannot* be solved efficiently. If we show that the simple decision problem cannot be solved efficiently, then the more general optimization problem certainly cannot be solved efficiently either.

Language Recognition Problems: Observe that a decision problem can also be thought of as a language recognition problem. We could define a language L

$$L = \{(G, k) \mid G \text{ has a MST of weight at most } k\}.$$

This set consists of pairs, the first element is a graph (e.g. the adjacency matrix encoded as a string) followed by an integer k encoded as a binary number. At first it may seem strange expressing a graph as a string, but obviously anything that is represented in a computer is broken down somehow into a string of bits.

When presented with an input string (G, k), the algorithm would answer "yes" if $(G, k) \in L$ implying that G has a spanning tree of weight at most k, and "no" otherwise. In the first case we say that the algorithm "accepts" the input and otherwise it "rejects" the input.

Given any language, we can ask the question of how hard it is to determine whether a given string is in the language. For example, in the case of the MST language L, we can determine membership easily in polynomial time. We just store the graph internally, run Kruskal's algorithm, and see whether the final optimal weight is at most k. If so we accept, and otherwise we reject.

Definition: Define P to be the set of all languages for which membership can be tested in polynomial time. (Intuitively, this corresponds to the set of all decisions problems that can be solved in polynomial time.)

Note that languages are sets of strings, and P is a set of languages. P is defined in terms of how hard it is computationally to recognized membership in the language. A set of languages that is defined in terms of how hard it is to determine membership is called a *complexity class*. Since we can compute minimum spanning trees in polynomial time, we have $L \in$ P.

Here is a harder one, though.

$$M = \{(G, k) \mid G \text{ has a simple path of length at least } k\}.$$

Given a graph G and integer k how would you "recognize" whether it is in the language M? You might try searching the graph for a simple paths, until finding one of length at least k. If you find one then you can accept and terminate. However, if not then you may spend a lot of time searching (especially if k is large, like $n - 1$, and no such path exists). So is $M \in$ P? No one knows the answer. In fact, we will show that M is NP-complete.

In what follows, we will be introducing a number of classes. We will jump back and forth between the terms "language" and "decision problems", but for our purposes they mean the same things. Before giving all the technical definitions, let us say a bit about what the general classes look like at an intuitive level.

P: This is the set of all decision problems that can be *solved* in polynomial time. We will generally refer to these problems as being "easy" or "efficiently solvable". (Although this may be an exaggeration in many cases.)

NP: This is the set of all decision problems that can be *verified* in polynomial time. (We will give a definition of this below.) This class contains P as a subset. Thus, it contains a number of easy problems, but it also contains a number of problems that are believed to be very hard to solve. The term NP does *not* mean "not polynomial". Originally the term meant "nondeterministic polynomial time". But it is bit more intuitive to explain the concept from the perspective of verification.

NP-hard: In spite of its name, to say that problem is NP-hard does *not* mean that it is hard to solve. Rather it means that if we could solve this problem in polynomial time, then we could solve *all* NP problems in polynomial time. Note that for a problem to be NP hard, it does not have to be in the class NP. Since it is widely believed that all NP problems are not solvable in polynomial time, it is widely believed that no NP-hard problem is solvable in polynomial time.

NP-complete: A problem is NP-complete if (1) it is in NP, and (2) it is NP-hard. That is, NPC = NP $\cap$ NP-hard.

The figure below illustrates one way that the sets P, NP, NP-hard, and NP-complete (NPC) *might* look. We say *might* because we do not know whether all of these complexity classes are distinct or whether they are all solvable in polynomial time. There are some problems in the figure that we will not discuss. One is *Graph Isomorphism*, which asks whether two graphs are identical up to a renaming of their vertices. It is known that this problem is in NP, but it is not known to be in P. The other is QBF, which stands for *Quantified Boolean Formulas*. In this problem you are given a boolean formula with quantifiers ($\exists$ and $\forall$) and you want to know whether the formula is true or false. This problem is beyond the scope of this course, but may be discussed in an advanced course on complexity theory.

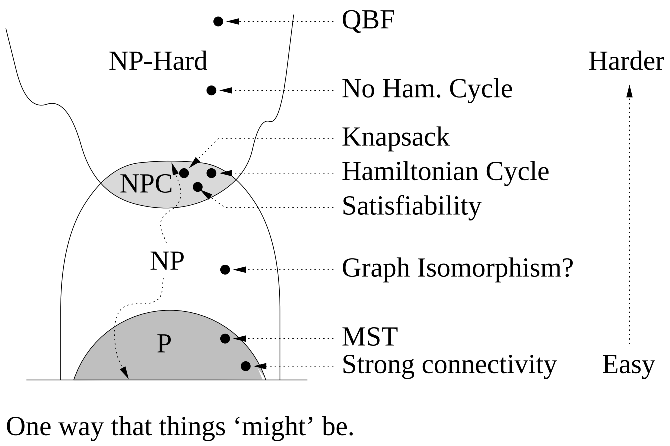


Figure 44: Structure of P, NP, and related complexity classes.

Polynomial Time Verification and Certificates: Before talking about the class of NP-complete problems, it is important to introduce the notion of a verification algorithm. Many language recognition problems that may be very hard to solve, but they have the property that it is easy to *verify* whether a string is in the language.

Consider the following problem, called the *Hamiltonian cycle problem.* Given an undirected graph G, does G have a cycle that visits every vertex exactly once. (There is a similar problem on directed graphs, and there is also a version which asks whether there is a path that visits all vertices.) We can describe this problem as a language recognition problem, where the language is

$$HC = \{(G) \mid G \text{ has a Hamiltonian cycle}\},$$

where (G) denotes an encoding of a graph G as a string. The Hamiltonian cycle problem seems to be much harder, and there is no known polynomial time algorithm for this problem. For example, the figure below shows two graphs, one which is Hamiltonian and one which is not.

However, suppose that a graph did have a Hamiltonian cycle. Then it would be a very easy matter for someone to convince us of this. They would simply say "the cycle is $\langle v_3, v_7, v_1, \ldots, v_{13}\rangle$". We could then inspect the graph, and check that this is indeed a legal cycle and that it visits all the vertices of the graph exactly once. Thus, even though we know of no efficient way to *solve* the Hamiltonian cycle problem, there is a very efficient way to *verify* that a given graph is in HC. The given cycle is called a *certificate*. This is some piece of information which allows us to verify that a given string is in a language.

More formally, given a language L, and given $x \in L$, a *verification algorithm* is an algorithm which given x and a string y called the *certificate*, can verify that x is in the language L using this certificate as help. If x is not in L then there is nothing to verify.

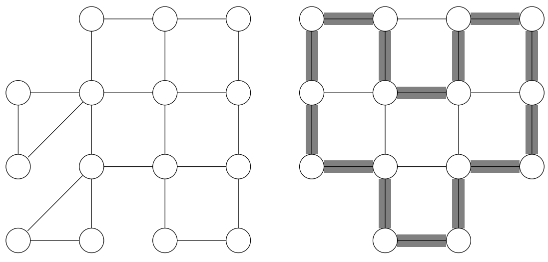


Figure 45: Hamiltonian cycle.

Note that not all languages have the property that they are easy to verify. For example, consider the following languages:

$$
\begin{aligned}
UHC &= \{(G) \mid G \text{ has a unique Hamiltonian cycle}\} \\
\overline{HC} &= \{(G) \mid G \text{ has no Hamiltonian cycle}\}.
\end{aligned}
$$

Suppose that a graph G is in the language UHC. What information would someone give us that would allow us to verify that G is indeed in the language? They could give us an example of the unique Hamiltonian cycle, and we could verify that it is a Hamiltonian cycle, but what sort of certificate could they give us to convince us that this is the *only* one? They could give another cycle that is NOT Hamiltonian, but this does not mean that there is not another cycle somewhere that is Hamiltonian. They could try to list every other cycle of length n, but this would not be at all efficient, since there are $n!$ possible cycles in general. Thus, it is hard to imagine that someone could give us some information that would allow us to efficiently convince ourselves that a given graph is in the language.

The class NP:

Definition: Define NP to be the set of all languages that can be verified by a polynomial time algorithm.

Why is the set called "NP" rather than "VP"? The original term NP stood for "nondeterministic polynomial time". This referred to a program running on a *nondeterministic computer* that can make guesses. Basically, such a computer could nondeterministically guess the value of certificate, and then verify that the string is in the language in polynomial time. We have avoided introducing nondeterminism here. It would be covered in a course on complexity theory or formal language theory.

Like P, NP is a set of languages based on some complexity measure (the complexity of verification). Observe that P $\subseteq$ NP. In other words, if we can solve a problem in polynomial time, then we can certainly verify membership 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. Next time we will define the notions of NP-hard and NP-complete.

Lecture 22: NP-Completeness: Reductions

(Tuesday, Nov 16, 1999)

Read: Chapt 36, through Section 36.4.

Summary: Last time we introduced a number of concepts, on the way to defining NP-completeness. In particular, the following concepts are important.

Decision Problems: are problems for which the answer is either yes or no. NP-complete problems are expressed as decision problems, and hence can be thought of as language recognition problems, assuming that the input has been encoded as a string. We *encode* inputs as strings. For example:

$$\begin{aligned} HC &= \{G \mid G \text{ has a Hamiltonian cycle}\} \\ MST &= \{(G, x) \mid G \text{ has a MST of cost at most } x\}. \end{aligned}$$

P: is the class of all decision problems which can be solved in polynomial time, $O(n^k)$ for some constant k. For example $MST \in$ P but HC is not known (and suspected not) to be in P.

Certificate: is a piece of evidence that allows us to *verify* in polynomial time that a string is in a given language. For example, suppose that the language is the set of Hamiltonian graphs. To convince someone that a graph is in this language, we could supply the certificate consisting of a sequence of vertices along the cycle. It is easy to access the adjacency matrix to determine that this is a legitimate cycle in G. Therefore $HC \in$ NP.

NP: is defined to be the class of all languages that can be *verified* in polynomial time. Note that since all languages in P can be solved in polynomial time, they can certainly be verified in polynomial time, so we have P $\subseteq$ NP. However, NP also seems to have some pretty hard problems to solve, such as HC.

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

Before discussing reductions, let us just consider the following question. Suppose that there are two problems, A and B. You know (or you strongly believe at least) that it is impossible to solve problem A in polynomial time. You want to prove that B cannot be solved in polynomial time. How would you do this? We want to show that

$$(A \notin \text{P}) \Rightarrow (B \notin \text{P}).$$

To do this, we could prove the contrapositive,

$$(B \in \text{P}) \Rightarrow (A \in \text{P}).$$

In other words, to show that B is not solvable in polynomial time, we will suppose that there is an algorithm that solves B in polynomial time, and then derive a contradiction by showing that A can be solved in polynomial time.

How do we do this? Suppose that we have a subroutine that can solve any instance of problem B in polynomial time. Then all we need to do is to show that we can use this subroutine to solve problem A in polynomial time. Thus we have "reduced" problem A to problem B. It is important to note here that this supposed subroutine is really a *fantasy*. We know (or strongly believe) that A cannot be solved in polynomial time, thus we are essentially proving that the subroutine cannot exist, implying that B cannot be solved in polynomial time. (Be sure that you understand this, this the basis behind all reductions.)

Example: 3-Colorability and Clique Cover: Let us consider an example to make this clearer. The following problem is well-known to be NP-complete, and hence it is strongly believed that the problem cannot be solved in polynomial time.

3-coloring (3Col): Given a graph G, can each of its vertices be labeled with one of 3 different "colors", such that no two adjacent vertices have the same label.

Coloring arises in various partitioning problems, where there is a constraint that two objects cannot be assigned to the same set of the partition. The term "coloring" comes from the original application which was in map drawing. Two countries that share a common border should be colored with different colors. It is well known that planar graphs can be colored with 4 colors, and there exists a polynomial time algorithm for this. But determining whether 3 colors are possible (even for planar graphs) seems to be hard and there is no known polynomial time algorithm. In the figure below we give two graphs. One which can be colored with 3 colors, and one that cannot.

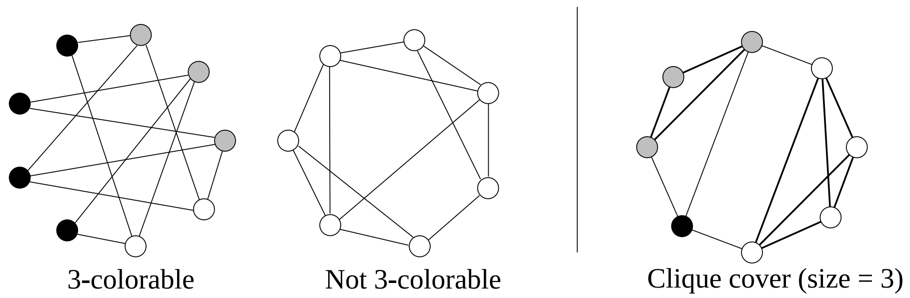


Figure 46: 3-coloring and Clique Cover.

The 3Col problem will play the role of problem A, which we strongly suspect to not be solvable in polynomial time. For our problem B, consider the following problem. Given a graph $G = (V, E)$, we say that a subset of vertices $V' \subseteq V$ forms a *clique* if for every pair of vertices $u, v \in V'$ $(u, v) \in E$. That is, the subgraph induced by V' is a complete graph.

Clique Cover (CCov): Given a graph G and an integer k, can we find k subsets of vertices $V_1, V_2, \ldots, V_k$, such that $\bigcup_i V_i = V$, and that each V_i is a clique of G.

The clique cover problem arises in applications of clustering. We put an edge between two nodes if they are similar enough to be clustered in the same group. We want to know whether it is possible to cluster all the vertices into k groups.

Suppose that you want to solve the CCov problem, but after a while of fruitless effort, you still cannot find a polynomial time algorithm for the CCov problem. How can you prove that CCov is likely to not have a polynomial time solution? You know that 3Col is NP-complete, and hence experts believe that 3Col $\notin$ P. You feel that there is some connection between the CCov problem and the 3Col problem. Thus, you want to show that

$$(\text{3Col} \notin \text{P}) \Rightarrow (\text{CCov} \notin \text{P}),$$

which you will show by proving the contrapositive

$$(\text{CCov} \in \text{P}) \Rightarrow (\text{3Col} \in \text{P}).$$

To do this, you assume that you have access to a subroutine $CCov(G, k)$. Given a graph G and an integer k, this subroutine returns true if G has a clique cover of size k and false otherwise, and furthermore, this subroutine runs in polynomial time. How can we use this "alleged" subroutine to solve the well-known hard 3Col problem? We want to write a polynomial time subroutine for 3Col, and this subroutine is allowed to call the subroutine $CCov(G, k)$ for any graph G and any integer k.

Both problems involve partitioning the vertices up into groups. The only difference here is that in one problem the number of cliques is specified as part of the input and in the other the number of color classes is fixed at 3. In the clique cover problem, for two vertices to be in the same group they must be adjacent to each other. In the 3-coloring problem, for two vertices to be in the same color group, they must not be adjacent. In some sense, the problems are almost the same, but the requirement adjacent/non-adjacent is exactly reversed.

We claim that we can *reduce* the 3-coloring problem to the clique cover problem as follows. Given a graph G for which we want to determine its 3-colorability, output the pair $(\overline{G}, 3)$ where $\overline{G}$ denotes the complement of G. We can then feed the pair $(\overline{G}, 3)$ into a subroutine for clique cover. This is illustrated in the figure below.

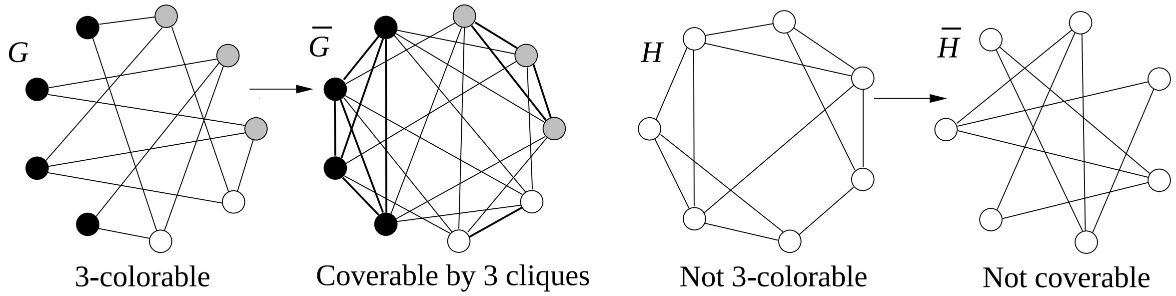


Figure 47: Clique covers in the complement.

Claim: A graph G is 3-colorable if and only if its complement $\overline{G}$ has a clique-cover of size 3. In other words,

$$G \in \text{3Col iff } (\overline{G}, 3) \in \text{CCov}.$$

Proof: ($\Rightarrow$) If G 3-colorable, then let V_1, V_2, V_3 be the three color classes. We claim that this is a clique cover of size 3 for $\overline{G}$, since if u and v are distinct vertices in V_i, then $\{u, v\} \notin E(G)$ (since adjacent vertices cannot have the same color) which implies that $\{u, v\} \in E(\overline{G})$. Thus every pair of distinct vertices in V_i are adjacent in $\overline{G}$.

($\Leftarrow$) Suppose $\overline{G}$ has a clique cover of size 3, denoted V_1, V_2, V_3. For $i \in \{1, 2, 3\}$ give the vertices of V_i color i. We assert that this is a legal coloring for G, since if distinct vertices u and v are both in V_i, then $\{u, v\} \in E(G)$ (since they are in a common clique), implying that $\{u, v\} \notin E(\overline{(}G)$. Hence, two vertices with the same color are not adjacent.

Polynomial-time reduction: We now take this intuition of reducing one problem to another through the use of a subroutine call, and place it on more formal footing. Notice that in the example above, we converted an instance of the 3-coloring problem (G) into an equivalent instance of the Clique Cover problem $(\overline{G}, 3)$.

Definition: We say that a language (i.e. decision problem) L_1 is *polynomial-time reducible* to language L_2 (written $L_1 \leq_P L_2$) if there is a polynomial time computable function f, such that for all x, $x \in L_1$ if and only if $f(x) \in L_2$.

In the previous example we showed that

$$\text{3Col} \leq_P \text{CCov}.$$

In particular we have $f(G) = (\overline{G}, 3)$. Note that it is easy to complement a graph in $O(n^2)$ (i.e. polynomial) time (e.g. flip 0's and 1's in the adjacency matrix). Thus f is computable in polynomial time.

Intuitively, saying that $L_1 \le_P L_2$ means that "if L_2 is solvable in polynomial time, then so is L_1." This is because a polynomial time subroutine for L_2 could be applied to $f(x)$ to determine whether $f(x) \in L_2$, or equivalently whether $x \in L_1$. Thus, in sense of polynomial time computability, L_1 is "no harder" than L_2.

The way in which this is used in NP-completeness is exactly the converse. We usually have strong evidence that L_1 is not solvable in polynomial time, and hence the reduction is effectively equivalent to saying "since L_1 is not likely to be solvable in polynomial time, then L_2 is also not likely to be solvable in polynomial time." Thus, this is how polynomial time reductions can be used to show that problems are as hard to solve as known difficult problems.

Lemma: If $L_1 \le_P L_2$ and $L_2 \in P$ then $L_1 \in P$.

Lemma: If $L_1 \le_P L_2$ and $L_1 \notin P$ then $L_2 \notin P$.

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

Lemma: If $L_1 \le_P L_2$ and $L_2 \le_P L_3$ then $L_1 \le_P L_3$.

The reason is that if two functions $f(x)$ and $g(x)$ are computable in polynomial time, then their composition $f(g(x))$ is computable in polynomial time as well. It should be noted that our text uses the term "reduction" where most other books use the term "transformation". The distinction is subtle, but people taking other courses in complexity theory should be aware of this.

NP-completeness: The set of NP-complete problems are all problems in the complexity class NP, for which it is known that if any one is solvable in polynomial time, then they all are, and conversely, if any one is not solvable in polynomial time, then none are. This is made mathematically formal using the notion of polynomial time reductions.

Definition: A language L is *NP-hard* if:

$$L' \le_P L \text{ for all } L' \in \text{NP}.$$

Definition: A language L is *NP-complete* if: $L \in \text{NP}$, and L is NP-hard.

An alternative (and usually easier way) to show that a problem is NP-complete is to use transitivity.

Lemma: L is NP-complete if

(1) $L \in NP$ and

(2) $L' \le_P L$ for some known NP-complete language L'.

The reason is that all $L'' \in NP$ are reducible to L' (since L' is NP-complete and hence NP-hard) and hence by transitivity L'' is reducible to L, implying that L is NP-hard.

This gives us a way to prove that problems are NP-complete, once we know that one problem is NP-complete. 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 showed that there is one problem called SAT (short for boolean satisfiability) that is NP-complete. To prove a second problem is NP-complete, all we need to do is to show that our problem is in NP (and hence it is reducible to SAT), and then to show that we can reduce SAT (or generally some known NPC problem) to our problem. It follows that our problem is equivalent to SAT (with respect to solvability in polynomial time). This is illustrated in the figure below.

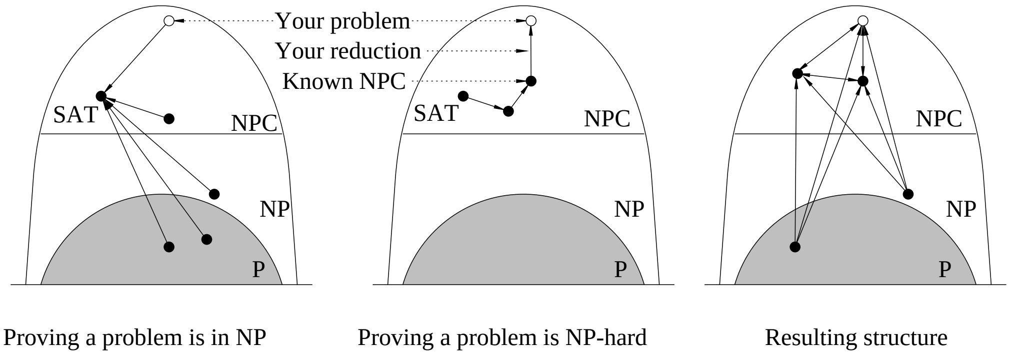


Figure 48: Structure of NPC and reductions.

Lecture 23: Cook's Theorem and Other Reductions

(Thursday, Nov 18, 1999)

Read: Section 36.5. The reduction given here is similar, but not the same as the reduction given in the text.

Recap: So far we introduced the definitions of NP-completeness. Recall that we mentioned the following topics:

P: is the set of decision problems (or languages) that are solvable in polynomial time.

NP: is the set of decision problems (or languages) that can be verified in polynomial time,

Polynomial reduction: $L_1 \leq_P L_2$ means that there is a polynomial time computable function f such that $x \in L_1$ if and only if $f(x) \in L_2$. A more intuitive to think about this, is that if we had a subroutine to solve L_2 in polynomial time, then we could use it to solve L_1 in polynomial time.

Polynomial reductions are transitive, that is, if $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

NP-Hard: L is NP-hard if for all $L' \in$ NP, $L' \leq_P L$. Thus, if we could solve L in polynomial time, we could solve all NP problems in polynomial time.

NP-Complete: L is NP-complete if (1) $L \in$ NP and (2) L is NP-hard.

The importance of NP-complete problems should now be clear. If any NP-complete problems (and generally any NP-hard problem) is solvable in polynomial time, then every NP-complete problem (and in fact every problem in NP) is also solvable in polynomial time. Conversely, if we can prove that any NP-complete problem (and generally any problem in NP) cannot be solved in polynomial time, then every NP-complete problem (and generally every NP-hard problem) cannot be solved in polynomial time. Thus all NP-complete problems are equivalent to one another (in that they are either all solvable in polynomial time, or none are).

An alternative way to show that a problem is NP-complete is to use transitivity of $\leq_P$.

Lemma: L is NP-complete if

(1) $L \in NP$ and
(2) $L' \leq_P L$ for some NP-complete language L'.

Note: The *known* NP-complete problem L' is being reduced to *candidate* NP-complete problem L. Keep this order in mind.

Cook's Theorem: Unfortunately, to use this lemma, we need to have *at least one* NP-complete problem to start the ball rolling. Stephen Cook showed that such a problem existed. Cook's theorem is quite complicated to prove, but we'll try to give a brief intuitive argument as to why such a problem might exist.

For a problem to be in NP, it must have an efficient verification procedure. Thus virtually all NP problems can be stated in the form, "does there exists X such that $P(X)$", where X is some structure (e.g. a set, a path, a partition, an assignment, etc.) and $P(X)$ is some property that X must satisfy (e.g. the set of objects must fill the knapsack, or the path must visit every vertex, or you may use at most k colors and no two adjacent vertices can have the same color). In showing that such a problem is in NP, the certificate consists of giving X, and the verification involves testing that $P(X)$ holds.

In general, any set X can be described by choosing a set of objects, which in turn can be described as choosing the values of some boolean variables. Similarly, the property $P(X)$ that you need to satisfy, can be described as a boolean formula. Stephen Cook was looking for the *most* general possible property he could, since this should represent the *hardest* problem in NP to solve. He reasoned that computers (which represent the most general type of computational devices known) could be described entirely in terms of boolean circuits, and hence in terms of boolean formulas. If any problem were hard to solve, it would be one in which X is an assignment of boolean values (true/false, 0/1) and $P(X)$ could be any boolean formula. This suggests the following problem, called the *boolean satisfiability problem.*

SAT: Given a boolean formula, is there some way to assign truth values (0/1, true/false) to the variables of the formula, so that the formula evaluates to true?

A boolean formula is a logical formula which consists of variables x_i, and the 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.)

For example,

$$(x_1 \wedge (x_2 \vee \overline{x}_3)) \wedge ((\overline{x}_2 \wedge \overline{x}_3) \vee \overline{x}_1)$$

is satisfiable, by the assignment $x_1 = 1$, $x_2 = 0$, $x_3 = 0$ On the other hand,

$$(\overline{x}_1 \vee (x_2 \wedge x_3)) \wedge (x_1 \vee (\overline{x}_2 \wedge \overline{x}_3)) \wedge (x_2 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

is not satisfiable. (Observe that the last two clauses imply that one of x_2 and x_3 must be true and the other must be false. This implies that neither of the subclauses involving x_2 and x_3 in the first two clauses can be satisfied, but x_1 cannot be set to satisfy them either.)

Cook's Theorem: SAT is NP complete.

We will not prove this theorem. The proof would take about a a full lecture (not counting the week or so of background on Turing machines). In fact, it turns out that a even more restricted version of the satisfiability problem is NP-complete. A *literal* is a variable or its negation x

or $\overline{x}$. A formula is in *3-conjunctive normal form* (3-CNF) if it is the boolean-and of clauses where each clause is the boolean-or of exactly 3 literals. For example

$$(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \overline{x}_3 \vee \overline{x}_4)$$

is in 3-CNF form. *3SAT* is the problem of determining whether a formula in 3-CNF is satisfiable. It turns out that it is possible to modify the proof of Cook's theorem to show that 3SAT is also NP-complete.

As an aside, note that if we replace the 3 in 3SAT with a 2, then everything changes. If a boolean formula is given in 2SAT, then it is possible to determine its satisfiability in polynomial time. (It turns out that the problem can be reduced to computing the strong components in a directed graph.) Thus, even a seemingly small change can be the difference between an efficient algorithm and none.

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

Independent Set (IS): Given an undirected graph $G = (V, E)$ and an integer k does G contain a subset V' of k vertices such that no two vertices in V' are adjacent to one another.

For example, the graph shown in the figure below has an independent set (shown with shaded nodes) of size 4. The independent set problem arises when there is some sort of selection problem, but there are mutual restrictions pairs that cannot both be selected. (For example, a company party where an employee and his/her immediate supervisor cannot both be invited.)

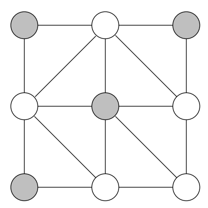


Figure 49: Independent Set.

Note that if a graph has an independent set of size k, then it has an independent set of all smaller sizes. So the corresponding optimization problem would be to find an independent set of the largest size in a graph. Often the vertices have weights, so we might talk about the problem of computing the independent set with the largest total weight. However, since we want to show that the problem is hard to solve, we will consider the simplest version of the problem.

Claim: IS is NP-complete.

The proof involves two parts. First, we need to show that IS $\in$ NP. The certificate consists of the k vertices of V'. We simply verify that for each pair of vertex $u, v \in V'$, there is no edge between them. Clearly this can be done in polynomial time, by an inspection of the adjacency matrix.

Secondly, we need to establish that IS is NP-hard, which can be done by showing that some known NP-complete problem (3SAT) is polynomial-time reducible to IS, that is, 3SAT $\leq_P$ IS. Let F be a boolean formula in 3-CNF form. We wish to find a polynomial time computable function f that maps F into a input for the IS problem, a graph G and integer k. That is,

$f(F) = (G, k)$, such that F is satisfiable if and only if G has an independent set of size k. This will mean that if we can solve the independent set problem for G and k in polynomial time, then we would be able to solve 3SAT in polynomial time.

An important aspect to reductions is that we do not attempt to solve the satisfiability problem. (Remember: It is NP-complete, and there is not likely to be any polynomial time solution.) So the function f must operate without knowledge of whether F is satisfiable. The idea is to *translate* the similar elements of the satisfiable problem to corresponding elements of the independent set problem.

What is to be selected?

3SAT: Which variables are to be assigned the value true, or equivalently, which literals will be true and which will be false.

IS: Which vertices will be placed in V'.

Requirements:

3SAT: Each clause must contain at least one true valued literal.

IS: V' must contain at least k vertices.

Restrictions:

3SAT: If x_i is assigned true, then $\overline{x}_i$ must be false, and vice versa.

IS: If u is selected to be in V', and v is a neighbor of u, then v cannot be in V'.

We want a function f, which given any 3-CNF boolean formula F, converts it into a pair (G, k) such that the above elements are translated properly. Our strategy will be to turn each literal into a vertex. The vertices will be in *clause clusters* of three, one for each clause. Selecting a true literal from some clause will correspond to selecting a vertex to add to V'. We will set k equal to the number of clauses, to force the independent set subroutine to select one true literal from each clause. To keep the IS subroutine from selecting two literals from one clause and none from some other, we will connect all the vertices in each clause cluster with edges. To keep the IS subroutine from selecting a literal and its complement to be true, we will put an edge between each literal and its complement.

A formal description of the reduction is given below. The input is a boolean formula F in 3-CNF, and the output is a graph G and integer k.

3SAT to IS Reduction

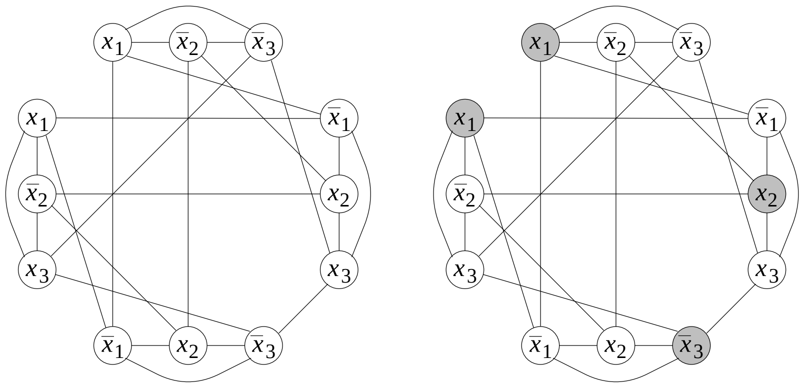
```
k ← number of clauses in F;
for each clause C in F {
    create a clause cluster of 3 vertices from the literals of C;
}
for each clause cluster (x_1, x_2, x_3) {
    create an edge (x_i, x_j) between all pairs of vertices in the cluster;
}
for each vertex x_i {
    create edges between x_i and all its complement vertices x̄_i;
}
return (G, k);
```

If F has k clauses, then G has exactly $3k$ vertices. Given any reasonable encoding of F, it is an easy programming exercise to create G (say as an adjacency matrix) in polynomial time. We claim that F is satisfiable if and only if G has an independent set of size k.

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

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3).$$

The reduction produces the graph shown in the following figure and sets $k = 4$.



The reduction Correctness (x1=x2=1, x3=0)

Figure 50: 3SAT to IS Reduction for $(x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (x_1 \vee \overline{x}_2 \vee x_3)$.

In our example, the formula is satisfied by the assignment $x_1 = 1$, $x_2 = 1$, and $x_3 = 0$. Note that this implies that the first literal of the first and last clauses are 1, the second literal of the second clause is 1, and the third literal of the third clause is 1. Observe that by selecting the corresponding vertices from the clusters, we get an independent set of size $k = 4$.

Correctness Proof: We claim that F is satisfiable if and only if G has an independent set of size k. If F is satisfiable, then each of the k clauses of F must have at least one true literal. Let V' denote the corresponding vertices from each of the clause clusters (one from each cluster). Because we take vertices from each cluster, there are no inter-cluster edges between them, and because we cannot set a variable and its complement to both be true, there can be no edge of the form $(x_i, \overline{x}_i)$ between the vertices of V'. Thus, V' is an independent set of size k.

Conversely, if G has an independent set V' of size k. First observe that we must select a vertex from each clause cluster, because there are k clusters, and we cannot take two vertices from the same cluster (because they are all interconnected). Consider the assignment in which we set all of these literals to 1. This assignment is logically consistent, because we cannot have two vertices labeled x_i and $\overline{x}_i$ in the same cluster. Finally the transformation clearly runs in polynomial time. This completes the NP-completeness proof.

Observe that our reduction did not attempt to solve the IS problem nor to solve the 3SAT. Also observe that the reduction had *no knowledge* of the solution to either problem. (We did not assume that the formula was satisfiable, nor did we assume we knew which variables to set to 1.) This is because computing these things would require exponential time (by the best known algorithms). Instead the reduction simply *translated* the input from one problem into an equivalent input to the other problem, while preserving the critical elements to each problem.

Lecture 24: Clique, Vertex Cover, and Dominating Set

(Tuesday, Nov 23, 1999)

Read: Chapt 36 (up through 36.5). The dominating set proof is not given in our text.

Recap: Last time we gave a reduction from 3SAT (satisfiability of boolean formulas in 3-CNF form) to IS (independent set in graphs). Today we give a few more examples of reductions. Recall that show that a problem is NP-complete we need to show (1) that the problem is in NP (i.e. we can verify when an input is in the language), and (2) that the problem is NP-hard, by showing that some known NP-complete problem can be reduced to this problem (there is a polynomial time function that transforms an input for one problem into an equivalent input for the other problem).

Some Easy Reductions: We consider some closely related NP-complete problems next.

Clique (CLIQUE): The *clique problem* is: given an undirected graph $G = (V, E)$ and an integer k, does G have a subset V' of k vertices such that for each distinct $u, v \in V'$, $\{u, v\} \in E$. In other words, does G have a k vertex subset whose induced subgraph is complete.

Vertex Cover (VC): 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?

Dominating Set (DS): A *dominating set* in a graph $G = (V, E)$ is a subset of vertices V' such that every vertex in the graph is either in V' or is adjacent to some vertex in V'. The *dominating set problem* (DS) is: given a graph $G = (V, E)$ and an integer k, does G have a dominating set of size k?

Don't confuse the clique (CLIQUE) problem with the clique-cover (CC) problem that we discussed in an earlier lecture. The clique problem seeks to find a single clique of size k, and the clique-cover problem seeks to partition the vertices into k groups, each of which is a clique.

We have discussed the facts that cliques are of interest in applications dealing with clustering. The vertex cover problem arises in various servicing applications. For example, you have a compute network and a program that checks the integrity of the communication links. To save the space of installing the program on every computer in the network, it suffices to install it on all the computers forming a vertex cover. From these nodes all the links can be tested. Dominating set is useful in facility location problems. For example, suppose we want to select where to place a set of fire stations such that every house in the city is within 2 minutes of the nearest fire station. We create a graph in which two locations are adjacent if they are within 2 minutes of each other. A minimum sized dominating set will be a minimum set of locations such that every other location is reachable within 2 minutes from one of these sites.

The CLIQUE problem is obviously closely related to the independent set problem (IS): Given a graph G does it have a k vertex subset that is completely *disconnected*. It is not quite as clear that the vertex cover problem is related. However, the following lemma makes this connection clear as well.

Lemma: Given an undirected graph $G = (V, E)$ with n vertices and a subset $V' \subseteq V$ of size k. The following are equivalent:

(i) V' is a clique of size k for the complement, $\overline{G}$.

(ii) V' is an independent set of size k for G.

(iii) $V - V'$ is a vertex cover of size $n - k$ for G.

Proof:

(i) $\Rightarrow$ (ii): If V' is a clique for $\overline{G}$, then for each $u, v \in V'$, $\{u, v\}$ is an edge of $\overline{G}$ implying that $\{u, v\}$ is not an edge of G, implying that V' is an independent set for G.

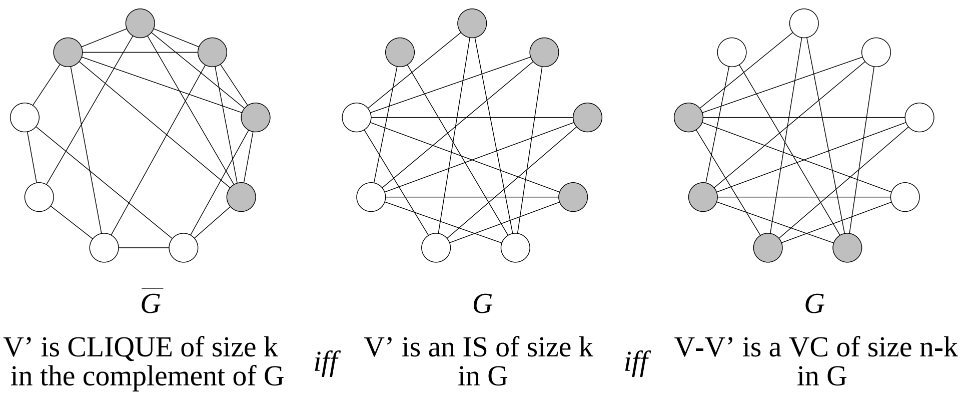


Figure 51: Clique, Independent set, and Vertex Cover.

(ii) $\Rightarrow$ (iii): If V' is an independent set for G, then for each $u, v \in V'$, $\{u, v\}$ is not an edge of G, implying that every edge in G is incident to a vertex in $V - V'$, implying that $V - V'$ is a VC for G.

(iii) $\Rightarrow$ (i): If $V - V'$ is a vertex cover for G, then for any $u, v \in V'$ there is no edge $\{u, v\}$ in G, implying that there is an edge $\{u, v\}$ in $\overline{G}$, implying that V' is a clique in $\overline{G}$. V' is an independent set for G.

Thus, if we had an algorithm for solving any one of these problems, we could easily translate it into an algorithm for the others. In particular, we have the following.

Theorem: CLIQUE is NP-complete.

CLIQUE $\in$ NP: The certificate consists of the k vertices in the clique. Given such a certificate we can easily verify in polynomial time that all pairs of vertices in the set are adjacent.

IS $\leq_P$ CLIQUE: We want to show that given an instance of the IS problem (G, k), we can produce an equivalent instance of the CLIQUE problem (G', k') in polynomial time. (Important: We do not know whether G has an independent set, and we do not have time to compute it.) Given G and k, set $G' = \overline{G}$ and $k' = k$, and output the pair (G', k'). By the above lemma, this instance is equivalent.

Theorem: VC is NP-complete.

VC $\in$ NP: The certificate consists of the k vertices in the vertex cover. Given such a certificate we can easily verify in polynomial time that every edge is incident to one of these vertices.

IS $\leq_P$ VC: We want to show that given an instance of the IS problem (G, k), we can produce an equivalent instance of the VC problem (G', k') in polynomial time. We set $G' = G$ and $k' = n - k$. By the lemma above, these instances are equivalent.

Dominating Set: As with vertex cover, dominating set is an example of a graph covering problem. Here the condition is a little different, each vertex is adjacent to the members of the dominating set, as opposed to each edge being incident to each member of the dominating set. Obviously, if G is connected and has a vertex cover of size k, then it has a dominating set of size k (the same set of vertices), but the converse is not necessarily true. However the similarity suggests that if VC in NP-complete, then DS is likely to be NP-complete as well. The main result of this section is just this.

Theorem: DS is NP-complete.

As usual the proof has two parts. First we show that DS $\in$ NP. The certificate just consists of the subset V' in the dominating set. In polynomial time we can determine whether every vertex is in V' or is adjacent to a vertex in V'.

Next we show that an existing NP-complete problem is reducible to dominating set. We choose vertex cover and show that VC $\leq_P$ DS. We want a polynomial time function, which given an instance of the vertex cover problem (G, k), produces an instance (G', k') of the dominating set problem, such that G has a vertex cover of size k if and only if G' has a dominating set of size k'.

How to we translate between these problems? The key difference is the condition. In VC: "every edge is incident to a vertex in V'". In DS: "every vertex is either in V' or is adjacent to a vertex in V'". Thus the translation must somehow map the notion of "incident" to "adjacent". Because incidence is a property of edges, and adjacency is a property of vertices, this suggests that the reduction function must somehow turn the edges of the graph into vertices such that an incident edge is mapped to an adjacent vertex.

This suggests the following idea (which does not quite work). We will insert a vertex into the middle of each edge of the graph. In other words, for each edge $\{u, v\}$, we will create a new *special vertex*, called w_{uv}, and replace the edge $\{u, v\}$ with the the two edges $\{u, w_{uv}\}$ and $\{v, w_{uv}\}$. The fact that u was incident to edge $\{u, v\}$ has now been replaced with the fact that u is adjacent to the corresponding vertex w_{uv}. We still need to dominate the neighbor v. To do this, we will leave the edge $\{u, v\}$ in the graph as well. Let G' be the resulting graph.

This is still not quite correct though. Define an *isolated vertex* to be one that is incident to no edges. If u is isolated it can only be dominated if it is included in the dominating set. Since it is not incident to any edges, it does not need to be in the vertex cover. Let V_I denote the isolated vertices in G, and let I denote the number of isolated vertices. The number of vertices to request for the dominating set will be $k' = k + I$.

Now we can give the complete reduction. Given the pair (G, k) for the VC problem, we create a graph G' as follows. Initially $G' = G$. For each edge $\{u, v\}$ in G we create a new vertex w_{uv} in G' and add edges $\{u, w_{uv}\}$ and $\{v, w_{uv}\}$ in G'. Let I denote the number of isolated vertices and set $k' = k + I$. Output (G', k'). This reduction illustrated in the following figure. Note that every step can be performed in polynomial time.

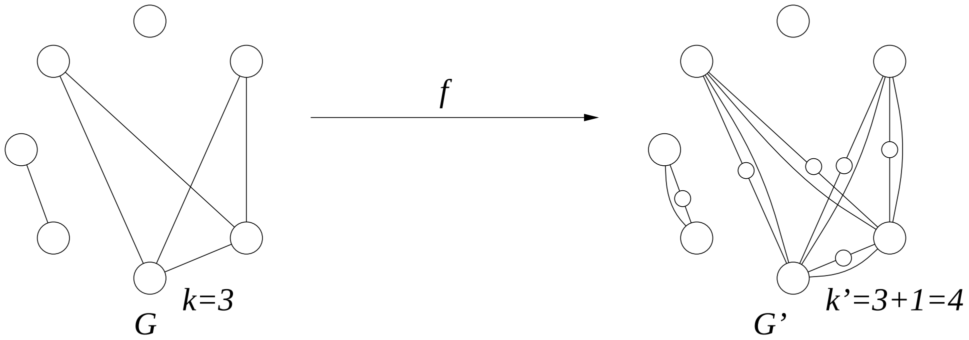


Figure 52: Dominating set reduction.

To establish the correctness of the reduction, we need to show that G has a vertex cover of size k if and only if G' has a dominating set of size k'. First we argue that if V' is a vertex cover for G, then $V'' = V' \cup V_I$ is a dominating set for G'. Observe that

$$|V''| = |V' \cup V_I| \leq k + I = k'.$$

(If this quantity is less than k', then we can add any vertices we like to make the size equal to k'.)

To see that V'' is a dominating set, first observe that all the isolated vertices are in V'' and so they are dominated. Second, each of the special vertices w_{uv} in G' corresponds to an edge $\{u, v\}$ in G implying that either u or v is in the vertex cover V'. Thus w_{uv} is dominated by the same vertex in V'' Finally, each of the nonisolated original vertices v is incident to at least one edge in G, and hence either it is in V' or else all of its neighbors are in V'. In either case, v is either in V'' or adjacent to a vertex in V''. This is shown in the top part of the following figure.

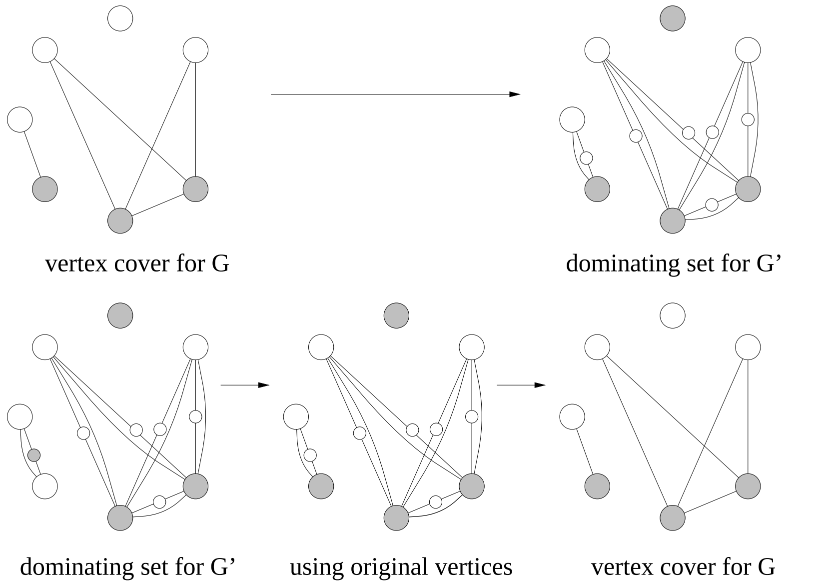


Figure 53: Correctness of the VC to DS reduction.

Conversely, we claim that if G' has a dominating set V'' of size $k' = k + k_I$ then G has a vertex cover V' of size k. Note that all k_I isolated vertices of G' must be in the dominating set. First, let $V''' = V'' - V_I$ be the remaining k vertices. We might try to claim something like: V''' is a vertex cover for G. But this will not necessarily work, because V''' may have vertices that are not part of the original graph G.

However, we claim that we never need to use any of the newly created special vertices in V'''. In particular, if some vertex $w_{uv} \in V'''$, then modify V''' by replacing w_{uv} with u. (We could have just as easily replaced it with v.) Observe that the vertex w_{uv} is adjacent to only u and v, so it dominates itself and these other two vertices. By using u instead, we still dominate u, v, and w_{uv} (because u has edges going to v and w_{uv}). Thus by replacing $w_{u,v}$ with u we dominate the same vertices (and potentially more). Let V' denote the resulting set after this modification. (This is shown in the lower middle part of the figure.)

We claim that V' is a vertex cover for G. If, to the contrary there were an edge $\{u, v\}$ of G that was not covered (neither u nor v was in V') then the special vertex w_{uv} would not be adjacent to any vertex of V'' in G', contradicting the hypothesis that V'' was a dominating set for G'.

Lecture 25: Hamiltonian Path

(Tuesday, Nov 30, 1999)

Read: The reduction we present for Hamiltonian Path is completely different from the one in Chapt 36.5.4 of CLR.

Hamiltonian Cycle: Today we consider a collection of problems related to finding paths in graphs and digraphs. Recall that given a graph (or digraph) a *Hamiltonian cycle* is a simple cycle that visits every vertex in the graph (exactly once). A *Hamiltonian path* is a simple path that visits every vertex in the graph (exactly once). The Hamiltonian cycle (HC) and Hamiltonian path (HP) problems ask whether a given graph (or digraph) has such a cycle or path, respectively. There are four variations of these problems depending on whether the graph is directed or undirected, and depending on whether you want a path or a cycle, but all of these problems are NP-complete.

An important related problem is the traveling salesman problem (TSP). Given a complete graph (or digraph) with integer edge weights, determine the cycle of minimum weight that visits all the vertices. Since the graph is complete, such a cycle will always exist. The decision problem formulation is, given a complete weighted graph G, and integer X, does there exist a Hamiltonian cycle of total weight at most X? Today we will prove that Hamiltonian Cycle is NP-complete. We will leave TSP as an easy exercise. (It is done in Section 36.5.5 in CLR.)

Component Design: Up to now, most of the reductions that we have seen (for Clique, VC, and DS in particular) are of a relatively simple variety. They are sometimes called *local replacement* reductions, because they operate by making some local change throughout the graph.

We will present a much more complex style of reduction for the Hamiltonian path problem on directed graphs. This type of reduction is called a *component design* reduction, because it involves designing special subgraphs, sometimes called *components* or *gadgets* (also called *widgets*). whose job it is to enforce a particular constraint. Very complex reductions may involve the creation of many gadgets. This one involves the construction of only one. (See CLR's presentation of HP for other examples of gadgets.)

The gadget that we will use in the directed Hamiltonian path reduction, called a *DHP-gadget*, is shown in the figure below. It consists of three incoming edges labeled i_1, i_2, i_3 and three outgoing edges, labeled o_1, o_2, o_3. It was designed so it satisfied the following property, which you can verify. Intuitively it says that if you enter the gadget on any subset of 1, 2 or 3 input edges, then there is a way to get through the gadget and hit every vertex exactly once, and in doing so each path must end on the corresponding output edge.

Claim: Given the DHP-gadget:

- For any subset of input edges, there exists a set of paths which join each input edge i_1, i_2, or i_3 to its respective output edge o_1, o_2, or o_3 such that together these paths visit every vertex in the gadget exactly once.
- Any subset of paths that start on the input edges and end on the output edges, and visit all the vertices of the gadget exactly once, must join corresponding inputs to corresponding outputs. (In other words, a path that starts on input i_1 must exit on output o_1.)

The proof is not hard, but involves a careful inspection of the gadget. It is probably easiest to see this on your own, by starting with one, two, or three input paths, and attempting to get through the gadget without skipping vertex and without visiting any vertex twice. To see whether you really understand the gadget, answer the question of why there are 6 groups of triples. Would some other number work?

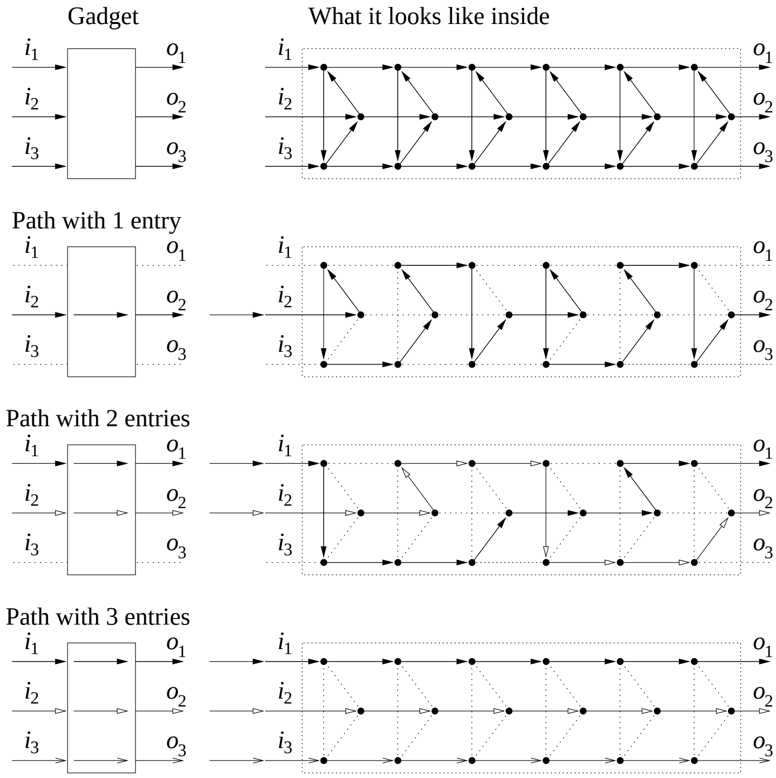


Figure 54: DHP-Gadget and examples of path traversals.

DHP is NP-complete: This gadget is an essential part of our proof that the directed Hamiltonian path problem is NP-complete.

Theorem: The directed Hamiltonian Path problem is NP-complete.

Proof: DHP $\in$ NP: The certificate consists of the sequence of vertices (or edges) in the path. It is an easy matter to check that the path visits every vertex exactly once.

3SAT $\leq_P$ DHP: This will be the subject of the rest of this section.

Let us consider the similar elements between the two problems. In 3SAT we are selecting a truth assignment for the variables of the formula. In DHP, we are deciding which edges will be a part of the path. In 3SAT there must be at least one true literal for each clause. In DHP, each vertex must be visited exactly once.

We are given a boolean formula F in 3-CNF form (three literals per clause). We will convert this formula into a digraph. Let $x_1, x_2, \ldots, x_m$ denote the variables appearing in F. We will construct one DHP-gadget for each clause in the formula. The inputs and outputs of each gadget correspond to the literals appearing in this clause. Thus, the clause $(\overline{x}_2 \vee x_5 \vee \overline{x}_8)$ would generate a clause gadget with inputs labeled $\overline{x}_2$, x_5, and $\overline{x}_8$, and the same outputs.

The general structure of the digraph will consist of a series vertices, one for each variable. Each of these vertices will have two outgoing paths, one taken if x_i is set to true and one if x_i is set to false. Each of these paths will then pass through some number of DHP-gadgets. The true path for x_i will pass through all the clause gadgets for clauses in which x_i appears, and the false path will pass through all the gadgets for clauses in which $\overline{x}_i$ appears. (The order in which the path passes through the gadgets is unimportant.) When the paths for x_i have passed through their last gadgets, then they are joined to the next variable vertex, x_{i+1}. This is illustrated in the following figure. (The figure only shows a portion of the construction. There will be paths coming into these same gadgets from other variables as well.) We add one final vertex x_e, and the last variable's paths are connected to x_e. (If we wanted to reduce to Hamiltonian cycle, rather than Hamiltonian path, we could join x_e back to x_1.)

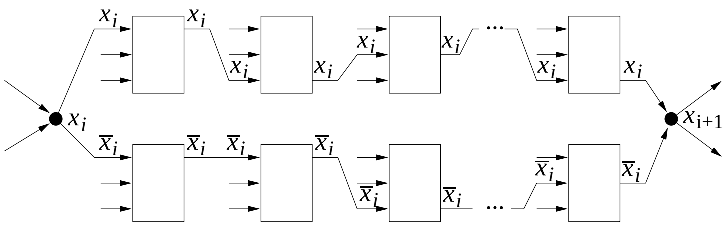


Figure 55: General structure of reduction from 3SAT to DHP.

Note that for each variable, the Hamiltonian path must either use the true path or the false path, but it cannot use both. If we choose the true path for x_i to be in the Hamiltonian path, then we will have at least one path passing through each of the gadgets whose corresponding clause contains x_i, and if we chose the false path, then we will have at least one path passing through each gadget for $\overline{x}_i$.

For example, consider the following boolean formula in 3-CNF. The construction yields the digraph shown in the following figure.

$$(\overline{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_1 \vee \overline{x}_3) \wedge (x_1 \vee x_3 \vee \overline{x}_2).$$

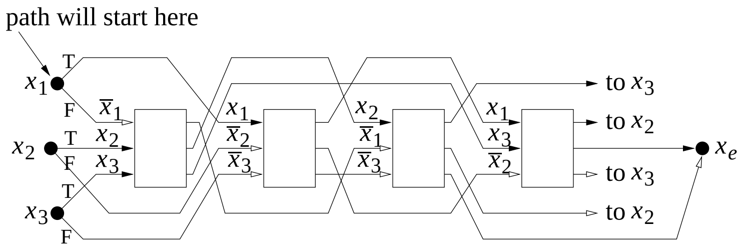


Figure 56: Example of the 3SAT to DHP reduction.

The Reduction: Let us give a more formal description of the reduction. Recall that we are given a boolean formula F in 3-CNF. We create a digraph G as follows. For each variable x_i appearing in F, we create a *variable vertex*, named x_i. We also create a vertex named x_e (the ending vertex). For each clause c, we create a DHP-gadget whose inputs and outputs are labeled with the three literals of c. (The order is unimportant, as long as each input and its corresponding output are labeled the same.)

We join these vertices with the gadgets as follows. For each variable x_i, consider all the clauses $c_1, c_2, \ldots, c_k$ in which x_i appears as a literal (uncomplemented). Join x_i by an edge to the input labeled with x_i in the gadget for c_1, and in general join the the output of gadget c_j labeled x_i with the input of gadget c_{j+1} with this same label. Finally, join the output of the last gadget c_k to the next vertex variable x_{i+1}. (If this is the last variable, then join it to x_e instead.) The resulting chain of edges is called the *true path* for variable x_i. Form a second chain in exactly the same way, but this time joining the gadgets for the clauses in which $\overline{x}_i$ appears. This is called the *false path* for x_i. The resulting digraph is the output of the reduction. Observe that the entire construction can be performed in polynomial time, by simply inspecting the formula, creating the appropriate vertices, and adding the appropriate edges to the digraph. The following lemma establishes the correctness of this reduction.

Lemma: The boolean formula F is satisfiable if and only if the digraph G produced by the above reduction has a Hamiltonian path.

Proof: We need to prove both the "only if" and the "if".

$\Rightarrow$: Suppose that F has a satisfying assignment. We claim that G has a Hamiltonian path. This path will start at the variable vertex x_1, then will travel along either the true path or false path for x_1, depending on whether it is 1 or 0, respectively, in the assignment, and then it will continue with x_2, then x_3, and so on, until reaching x_e. Such a path will visit each variable vertex exactly once.

Because this is a satisfying assignment, we know that for each clause, either 1, 2, or 3 of its literals will be true. This means that for each clause, either 1, 2, or 3, paths will attempt to travel through the corresponding gadget. However, we have argued in the above claim that in this case it is possible to visit every vertex in the gadget exactly once. Thus every vertex in the graph is visited exactly once, implying that G has a Hamiltonian path.

$\Leftarrow$: Suppose that G has a Hamiltonian path. We assert that the form of the path must be essentially the same as the one described in the previous part of this proof. In particular, the path must visit the variable vertices in increasing order from x_1 until x_e, because of the way in which these vertices are joined together.

Also observe that for each variable vertex, the path will proceed along either the true path or the false path. If it proceeds along the true path, set the corresponding variable

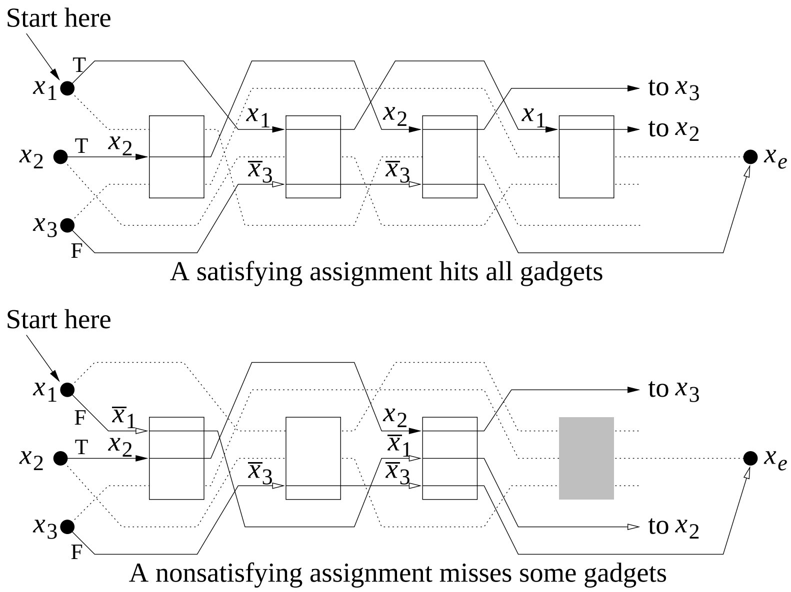


Figure 57: Correctness of the 3SAT to DHP reduction. The upper figure shows the Hamiltonian path resulting from the satisfying assignment, $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, and the lower figure shows the non-Hamiltonian path resulting from the nonsatisfying assignment $x_1 = 0$, $x_2 = 1$, $x_3 = 0$.

to 1 and otherwise set it to 0. We will show that the resulting assignment is a satisfying assignment for F.

Any Hamiltonian path must visit all the vertices in every gadget. By the above claim about DHP-gadgets, if a path visits all the vertices and enters along input edge then it must exit along the corresponding output edge. Therefore, once the Hamiltonian path starts along the true or false path for some variable, it must remain on edges with the same label. That is, if the path starts along the true path for x_i, it must travel through all the gadgets with the label x_i until arriving at the variable vertex for x_{i+1}. If it starts along the false path, then it must travel through all gadgets with the label $\overline{x}_i$.

Since all the gadgets are visited and the paths must remain true to their initial assignments, it follows that for each corresponding clause, at least one (and possibly 2 or three) of the literals must be true. Therefore, this is a satisfying assignment.

Lecture 26: Approximation Algorithms: VC and TSP

(Thursday, Dec 2, 1999)

Read: Chapt 37 (up through 37.2) in CLR.

Coping with NP-completeness: 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 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 brute-force search: Even on the fastest parallel computers this approach is viable only for the smallest instances of these problems.

Heuristics: A *heuristic* is a strategy for producing a valid solution, but there are no guarantees how close it is to optimal. This is worthwhile if all else fails, or if lack of optimality is not really an issue.

General Search Methods: There are a number of very powerful techniques for solving general combinatorial optimization problems that have been developed in the areas of AI and operations research. These go under names such as *branch-and-bound*, *A^*-search*, *simulated annealing*, and *genetic algorithms*. The performance of these approaches varies considerably from one problem to the next.

Approximation Algorithms: This is an algorithm that runs in polynomial time (ideally), and produces a solution that is within a guaranteed factor of the optimum solution.

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 define the *ratio bound* of an approximation algorithm as follows. Given an instance I of our problem, let $C(I)$ be the cost of the solution produced by our approximation algorithm, and let $C^*(I)$ be the optimal solution. We will assume that costs are strictly positive values. For a minimization problem we want $C(I)/C^*(I)$ to be small, and for a maximization problem we want $C^*(I)/C(I)$ to be small. For any input size n, we say that the approximation algorithm achieves *ratio bound* $\rho(n)$, if for all I, $|I| = n$ we have

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

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.

Some NP-complete problems can be approximated closely. Such an algorithm is given both the input, and a real value $\epsilon > 0$, and returns an answer whose ratio bound is at most $(1+\epsilon)$. Such an algorithm is called a *polynomial time approximation scheme* (or *PTAS* for short). The running time is a function of both n and ϵ. As ϵ approaches 0, the running time increases beyond polynomial time. For example, the running time might be $O(n^{\lceil 1/\epsilon \rceil})$. 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.

Although NP-complete problems are equivalent with respect to whether they can be solved exactly in polynomial time in the worst case, their approximability varies considerably.

- For some NP-complete problems, it is very unlikely that any approximation algorithm exists. For example, if the graph TSP problem had an approximation algorithm with a ratio bound of any value less than ∞, then P = NP.
- Many NP-complete can be approximated, but the ratio bound is a (slow growing) function of n. For example, the set cover problem (a generalization of the vertex cover problem), can be approximated to within a factor of $\ln n$. We will not discuss this algorithm, but it is covered in CLR.
- Some NP-complete problems can be approximated to within a fixed constant factor. We will discuss two examples below.

- Some NP-complete problems have PTAS's. One example is the subset problem (which we haven't discussed, but is described in CLR) and the Euclidean TSP problem.

In fact, much like NP-complete problems, there are collections of problems which are "believed" to be hard to approximate and are equivalent in the sense that if any one can be approximated in polynomial time then they all can be. This class is called *Max-SNP complete*. We will not discuss this further. Suffice it to say that the topic of approximation algorithms would fill another course.

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 optimum. 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.

How does one go about finding an approximation algorithm. The first approach is to try something that seems like a "reasonably" good strategy, a *heuristic*. It turns out that many simple heuristics, when not optimal, can often be proved to be close to optimal.

Here is an very simple algorithm, that guarantees an approximation within a factor of 2 for the vertex cover problem. It is based on the following observation. Consider an arbitrary edge (u, v) in the graph. One of its two vertices *must* be in the cover, but we do not know which one. The idea of this heuristic is to simply put *both* vertices into the vertex cover. (You cannot get much stupider than this!) Then we remove all edges that are incident to u and v (since they are now all covered), and recurse on the remaining edges. For every one vertex that must be in the cover, we put 2 into our cover, so it is easy to see that the cover we generate is at most twice the size of the optimum cover.

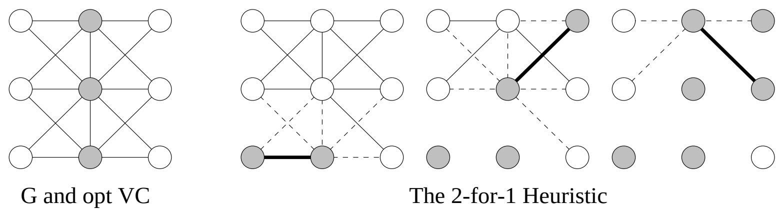


Figure 58: The 2-for-1 heuristic for vertex cover.

2-for-1 Approximation for VC

```
ApproxVC {
    C = empty-set;
    while (E is nonempty) do {
        let (u,v) be any edge of E;   (*)
        add both u and v to C;
        remove from E all edges incident to either u or v;
    }
    return C;
}
```

Claim: ApproxVC yields a factor-2 approximation for Vertex Cover.

Proof: Consider the set C output by ApproxVC. 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:

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

This proof illustrates one of the main features of the analysis of any approximation algorithm. Namely, that we need some way of finding a bound on the optimal solution. (For minimization problems we want a lower bound, for maximization problems an upper bound.) The bound should be related to something that we can compute in polynomial time. In this case, the bound is related to the set of edges A, which form a maximal independent set of edges.

Traveling Salesman Problem: In the Traveling Salesperson Problem (TSP) we are given a complete undirected graph with nonnegative edge weights, and we want to find a cycle that visits all vertices and is of minimum cost. Let $c(u, v)$ denote the weight on edge (u, v). Given a set of edges A forming a tour we define $c(A)$ to be the sum of edge weights in A. Last time we mentioned that TSP (posed as a decision problem) is NP-complete.

For many of the applications of TSP, the problem satisfies something called the *triangle inequality*. Intuitively, this says that the direct path from u to w, is never longer than an indirect path. More formally, for all $u, v, w \in V$

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

There are many examples of graphs that satisfy the triangle inequality. For example, given any weighted graph, if we define $c(u, v)$ to be the shortest path length between u and v (computed, say by Floyd's algorithm), then it will satisfy the triangle inequality. Another example is if we are given a set of points in the plane, and define a complete graph on these points, where $c(u, v)$ is defined to be the Euclidean distance between these points, then the triangle inequality is also satisfied.

When the underlying cost function satisfies the triangle inequality there is a approximation algorithm for TSP with a ratio-bound of 2. Thus, although this algorithm does not produce an optimal tour, the tour that it produces cannot be worse than twice the cost of the optimal tour.

The key insight is to observe that a TSP with one edge removed is a spanning tree. However it is not necessarily a minimum spanning tree. Therefore, the cost of the minimum TSP tour is at least as large as the cost of the MST. We can compute MST's efficiently using either Kruskal's or Prim's algorithm. If we can find some way to convert the MST into a TSP tour while increasing its cost by at most a constant factor, then we will have an approximation for TSP.

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. The figure below shows an example of this.

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. Notice that the final order in which vertices are visited using the short-cuts is exactly the same as a preorder traversal of the MST. The triangle inequality assures us that the path length will not increase when we take short-cuts.

TSP Approximation

```
ApproxTSP(G=(V,E)) {
    T = minimum spanning tree for G;
    r = any vertex;
```

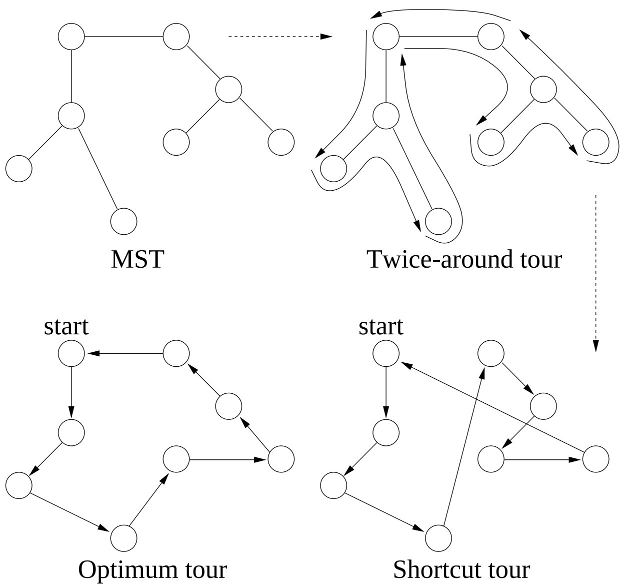


Figure 59: TSP Approximation.

```
    L = list of vertices visited by a preorder walk ot T
        starting with r;
    return 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

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

Lecture 27: More Approximations: Set Cover and Bin Packing

(Tuesday, Dec 7, 1999)

Read: Chapt 37.3. The theorem of the approximation bound proven here is a bit weaker from the one in CLR, but I think it is easier to understand. The material on the bin packing problem is presented as an exercise in CLR.

Set Cover: The set cover problem is a very important optimization problem. You are given a pair (X, F) where $X = \{x_1, x_2, \ldots, x_m\}$ is a finite set (a domain of elements) and $F = \{S_1, S_2, \ldots, S_n\}$ is a family of subsets of X, such that every element of X belongs to at least one set of F.

Consider a subset $C \subseteq F$. (This is a collection of sets over X.) We say that C *covers* the domain if every element of X is in some set of C, that is

$$X = \bigcup_{S_i \in C} S_i.$$

The problem is to find the minimum-sized subset C of F that covers X. Consider the example shown below. The optimum set cover consists of the three sets $\{S_3, S_4, S_5\}$.

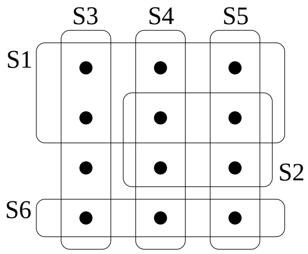


Figure 60: Set cover.

Set cover can be applied to a number of applications. For example, suppose you want to set up security cameras to cover a large art gallery. From each possible camera position, you can see a certain subset of the paintings. Each such subset of paintings is a set in your system. You want to put up the fewest cameras to see all the paintings.

Complexity of Set Cover: We have seen special cases of the set cover problems that are NP-complete. For example, vertex cover is a type of set cover problem. The domain to be covered are the edges, and each vertex covers the subset of incident edges. Therefore the set cover problem is NP-complete as well.

Unfortunately, although we have a factor-2 approximation for the vertex cover problem, it relied on the fact that each element of the domain (an edge) is in exactly 2 sets. Unfortunately, this is not true for the general set cover problem. Recently it has been proved that there is no constant factor approximation to the set cover problem (unless P = NP). This is unfortunate, because set cover is one of the most powerful NP-complete problems.

Today we will show that there is a reasonable approximation algorithm, the *greedy heuristic*, which achieves an approximation bound of $\ln m$, where m is the size of the underlying domain X. (The book proves a somewhat stronger result, that the approximation factor of $\ln m'$ where $m' \leq m$ is the size of the largest set in F. However, their proof is more complicated.)

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.

Greedy Set Cover

```
Greedy-Set-Cover(X, F) {
    U = X;                          // U are the items to be covered
    C = empty;                      // C will be the sets in the cover
    while (U is nonempty) {         // there is someone left to cover
        select S in F that covers the most elements of U;
```

```
            add S to C;
            U = U - S;
        }
        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. Thus, it would return a set cover of size 4, whereas the optimal set cover has size 3.

What is the approximation factor? The problem with the greedy set cover algorithm is that it can be "fooled" into picking the wrong set, over and over again. Consider the following example. The optimal set cover consists of sets S_5 and S_6. But (if ties are broken in the worst possible way) the greedy algorithm will select sets S_1 (size 16), then S_2 (size 8), then S_3 (size 4), then S_4 (size 2), and finally S_5 and S_6) (size 1 each). It is easy to see that this example can be generalized so that the ratio bound will be roughly $(\lg m)/2$.

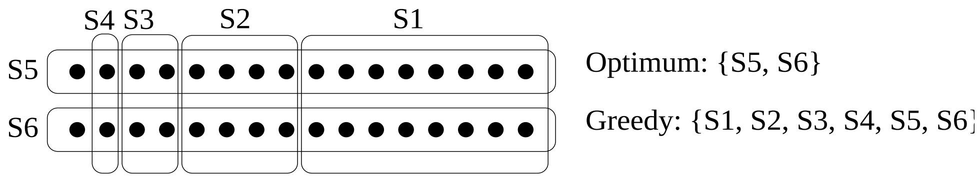


Figure 61: Greedy Set cover.

However we will show that the greedy set cover heuristic nevers performs worse than a factor of $\ln m$, where $m = |X|$. (Note that this is natural log, not base 2.)

Before giving the proof, we need one important mathematical inequality.

Lemma: For all $c > 0$,

$$\left(1 - \frac{1}{c}\right)^c \leq \frac{1}{e}.$$

where e is the base of the natural logarithm.

Proof: The book gives a proof, but here is one that I like. It requires the fact that for all x, $1 + x \leq e^x$. (The two functions are equal when $x = 0$.) Now, if we substitute $-1/c$ for x we have $(1 - 1/c) \leq e^{-1/c}$, and if we raise both sides to the cth power, we have the desired result.

Theorem: Greedy set cover has the ratio bound of at most $\ln m$ where $m = |X|$.

Proof: Let c denote the size of the optimum set cover, and let g denote the size of the greedy set cover minus 1. We will show that $g/c \leq \ln m$. (This is not quite what we wanted, but we are correct to within 1 set.)

Initially, there are $m_0 = m$ elements left to be covered. We know that there is a cover of size c (the optimal cover) and therefore by the pigeonhole principle, there must be at least one set that covers at least m_0/c elements. (Since otherwise, if every set covered less than m_0/c elements, then no collection of c sets could cover all m_0 elements.) Since the greedy algorithm selects the largest set, it will select a set that covers at least this many elements. The number of elements that remain to be covered is at most $m_1 = m_0 - m_0/c = m_0(1 - 1/c)$.

Applying the argument again, we know that we can cover these m_1 elements with a cover of size c (the optimal cover), and hence there exists a subset that covers at least m_1/c elements, leaving at most $m_2 = m_1(1 - 1/c) = m_0(1 - 1/c)^2$ elements remaining.

If we apply this argument g times, each time we succeed in covering at least a fraction of $(1-1/c)$ of the remaining elements. Then the number of elements that remain is uncovered after g sets have been chosen by the greedy algorithm is at most $m_g = m_0(1 - 1/c)^g$.

How long can this go on? Consider the largest value of g such that after removing all but the last set of the greedy cover, we still have some element remaining to be covered. Thus, we are interested in the largest value of g such that

$$1 \leq m\left(1 - \frac{1}{c}\right)^g.$$

We can rewrite this as

$$1 \leq m\left[\left(1 - \frac{1}{c}\right)^c\right]^{g/c}.$$

By the inequality above we have

$$1 \leq m\left[\frac{1}{e}\right]^{g/c}.$$

Now, if we multiply by $e^{g/c}$ and take natural logs we get that g satisfies:

$$e^{g/c} \leq m \qquad \Rightarrow \qquad \frac{g}{c} \leq \ln m.$$

This completes the proof.

Bin Packing: Bin packing is another well-known NP-complete problem, which is a variant of the knapsack problem. We are given a set of n objects, where s_i denotes the *size* of the ith object. It will simplify the presentation to assume that $0 < s_i < 1$. We want to put these objects into a set of bins. Each bin can hold a subset of objects whose total size is at most 1. The problem is to partition the objects among the bins so as to use the fewest possible bins. (Note that if your bin size is not 1, then you can reduce the problem into this form by simply dividing all sizes by the size of the bin.)

Bin packing arises in many applications. Many of these applications involve not only the size of the object but their geometric shape as well. For example, these include packing boxes into a truck, or cutting the maximum number of pieces of certain shapes out of a piece of sheet metal. However, even if we ignore the geometry, and just consider the sizes of the objects, the decision problem is still NP-complete. (The reduction is from the knapsack problem.)

Here is a simple heuristic algorithm for the bin packing problem, called the *first-fit heuristic*. We start with an unlimited number of empty bins. We take each object in turn, and find the first bin that has space to hold this object. We put this object in this bin. The algorithm is illustrated in the figure below. We claim that first-fit uses at most twice as many bins as the optimum, that is, if the optimal solution uses b^* bins, and first-fit uses b_{ff} bins, then

$$\frac{b_{ff}}{b^*} \leq 2.$$

Theorem: The first-fit heuristic achieves a ratio bound of 2.

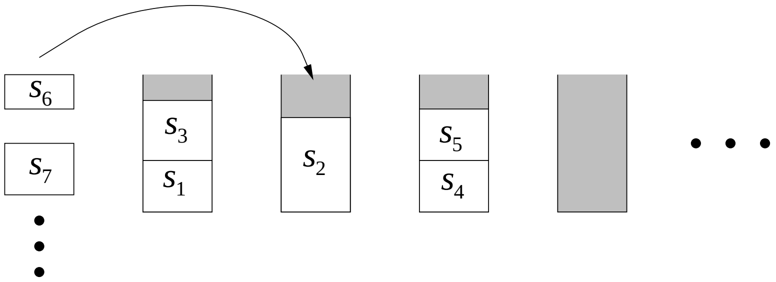


Figure 62: First-fit Heuristic.

Proof: Consider an instance $\{s_1, \ldots, s_n\}$ of the bin packing problem. Let $S = \sum_i s_i$ denote the sum of all the object sizes. Let b^* denote the optimal number of bins, and b_{ff} denote the number of bins used by first-fit.

First observe that $b^* \geq S$. This is true, since no bin can hold a total capacity of more than 1 unit, and even if we were to fill each bin exactly to its capacity, we would need at least S (in fact $\lceil S \rceil$ bins).

Next, we claim that $b_{ff} \leq 2S$. To see this, let t_i denote the total size of the objects that first-fit puts into bin i. Consider bins i and $i+1$ filled by first-fit. Assume that indexing is cyclical, so if i is the last index ($i = b_{ff}$) then $i + 1 = 1$. We claim that $t_i + t_{i+1} \geq 1$. If not, then the contents of bins i and $i+1$ could both be put into the same bin, and hence first-fit would never have started to fill the second bin, prefering to keep everything in the first bin. Thus we have:

$$\sum_{i=1}^{b_{ff}} (t_i + t_{i+1}) \geq b_{ff}.$$

But this sum adds up all the elements twice, so it has a total value of $2S$. Thus we have $2S \geq b_{ff}$. Combining this with the fact that $b^* \geq S$ we have

$$b_{ff} \leq 2S \leq 2b^*,$$

implying that $b_{ff}/b^* \leq 2$, as desired.

Lecture 28: The k-Center Problem

(Thursday, Dec 9, 1999)

Read: Today's material is not covered in our text.

Facility Location: Imagine that Blockbuster Video wants to open a 50 stores in some city. The company asks you to determine the best locations for these stores. The condition is that you are to minimize the maximum distance that any resident of the city must drive in order to arrive at the nearest store.

If we model the road network of the city as an undirected graph whose edge weights are the distances between intersections, then this is an instance of the *k-center problem*. The k-center problem is: given an undirected graph $G = (V, E)$ with nonnegative edge weights, and given an integer k, compute a subset of k vertices $C \subseteq V$, called *centers*, such that the maximum distance between any vertex in V and its nearest center in C is minimized.

More formally, let $G = (V, E)$ denote the graph, and let $w(u, v)$ denote the weight of edge (u, v). ($w(u, v) = w(v, u)$ because G is undirected.) We assume that all edge weights are nonnegative. For each pair of vertices, $u, v \in V$, let $d(u, v) = d(u, v)$ denote the *distance* between u to v, that is, the length of the shortest path from u to v. (Note that the shortest path distance satisfies the triangle inequality. This will be used in our proof.)

Consider a subset $C \subseteq V$ of vertices, the *centers*. For each vertex in V we can associate it with its nearest center in C. (This is the nearest Blockbuster store to your house). We can also associate each center with the set of vertices for which it is the closest center. (These are the houses that are closest to this center.) For $v \in V$ and $c_i \in C$, define

$$\begin{aligned} C(v) &= \{c_i \in C \mid d(v, c_i) \leq d(v, c_j) \text{ for } i \neq j\} \\ V(c_i) &= \{v \in V \mid d(v, c_i) \leq d(v, c_j) \text{ for } i \neq j\}. \end{aligned}$$

Generally $C(v)$ will consist of a single center except in a degenerate situation where v is equidistant from a number of centers. If we break ties for vertices that are equidistant from two or more centers, we can think of $V(c_1), V(c_2), \ldots$ as forming a *partition* of the vertex set of G. The *distance* associated with each center is the distance to its furthest vertex in $V(c_i)$, that is,

$$D(c_i) = \max_{v \in V(c_i)} d(v, c_i).$$

This is the furthest distance that any vertex associated with c_i is away from c_i. Finally, we define

$$D(C) = \max_{c_i \in C} D(c_i).$$

This is the maximum distance of any vertex from its nearest center. This is called the *bottleneck distance*, because it is the bottleneck (i.e. the critical element) to improving the solution. A vertex v whose distance from its nearest center equals $D(C)$ is called a *bottleneck vertex*.

Given this notation, we can now formally define the problem.

k-center problem: Given a weighted undirected graph $G = (V, E)$, and an integer $k \leq |V|$, find a subset $C \subseteq V$ of size k such that $D(C)$ is minimized.

The decision-problem formulation of the k-center problem is NP-complete (reduction from dominating set). A brute force solution to this problem would involve enumerating all k-element of subsets of V, and computing $D(C)$ for each one. However, if $n = |V|$ and k are large, the number of possible subsets is huge. It is $\binom{n}{k} = O(n^k)$. If n and k are both very large (e.g. $n = 100,000$ and $k = 20$) this number is huge. Given that the problem is NP-complete, it is highly unlikely that a significantly more efficient algorithm exists. We will show that there does exist an efficient approximation algorithm for the problem.

Approximation Algorithm: Our approximation algorithm is based on a simple greedy algorithm that produces a bottleneck distance $D(C)$ that is not more than twice the optimum bottleneck distance. We begin by letting the the first center c_1 be *any* vertex in the graph (the lower left vertex, say, in the figure below). Compute the distances between this vertex and all the other vertices in the graph (see (b)). Consider the vertex that is furthest from this center (the upper right vertex at distance 23 in the figure). This the bottleneck vertex for $\{c_1\}$. We would like to select the next center so as to reduce this distance. So let us just make it the next center, called c_2. Then we compute the distances from each vertex in the graph to the *closer* of c_1 and c_2. (See (c) where dashed lines indicate which vertices are closer to which center). Again take the bottleneck vertex to be the next center (the center vertex in (d)), and again compute the distances from each vertex to its nearest center. Repeat this until all k centers have been selected. In the example, the 3 centers are shaded, and the final bottleneck distance is 11.

Although the greedy approach has a certain intuitive appeal (because it attempts to find the vertex that gives the bottleneck distance, and then puts a center right on this vertex), it is not optimal. In the example shown in the figure, the optimum solution (shown on the right) has a bottleneck cost of 9, which beats the 11 that the greedy algorithm gave.

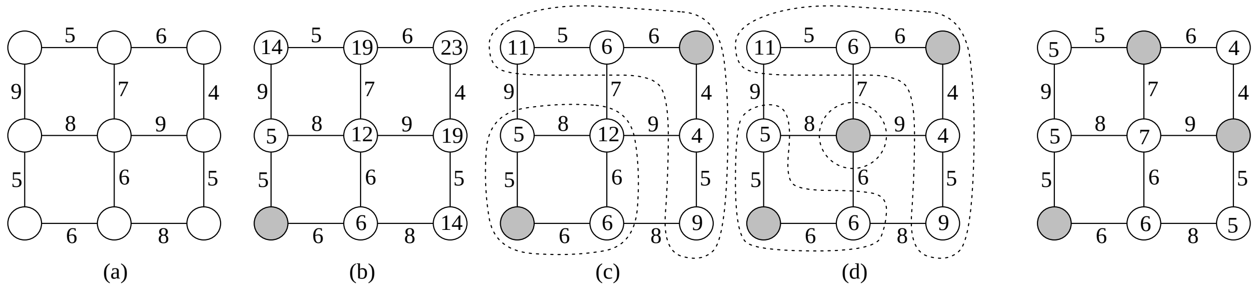


Figure 63: Greedy approximation to k-center.

Here is a summary of the algorithm. For each vertex u, let $d[u]$ denote the distance to the nearest center.

Greedy Approximation for k-center

```
K_Cent_Appx(G, k) {
    for each u in V do {              // initialize distances
        d[u] = INFINITY;
    }
    for i = 1 to k do {               // main loop
        Let c[i] = u such that d[u] is maximum;  (the bottleneck distance)
                              // update distances
        Compute d[v], the minimum distance from {c[1],c[2],...,c[i]} to
            each vertex v in V;  (the bottleneck vertex)
    }
    return c[1..k];                   // final centers
}
```

We know from Dijkstra's algorithm how to compute the shortest path from a single source to all other vertices in the graph. However, each step of this algorithm asks you to solve a multiple source shortest path problem (where the sources are the centers so far). This can be done with a simple modification to Dijkstra's algorithm. The modifications are not difficult.

The running time of the final algorithm is k times the running time of Dijkstra's algorithm, $O(k(n+e)\log n)$.

Approximation Bound: How bad could greedy be? We will argue that it has a ratio bound of 2. To see that we can get a factor of 2, consider a set of $n+1$ vertices arranged in a linear graph, in which all edges are of weight 1. The greedy algorithm can pick any initial vertex that it likes. Suppose it picks the leftmost vertex. Then the maximum (bottleneck) distance is the distance to the rightmost vertex which is n. If we had instead chosen the vertex in the middle, then the maximum distance would only be $n/2$, which is better by a factor of 2.

We want to show that this approximation algorithm always produces a final distance $D(C)$ that is within a factor of 2 of the distance of the optimal solution.

Let $O = \{o_1, o_2, \ldots, o_k\}$ denote the the centers of the optimal solution (shown as black dots in the figure below, and the lines show the partition into the neighborhoods for each of these points). Let $D_{\text{opt}} = D(O)$ be the optimal bottleneck distance.

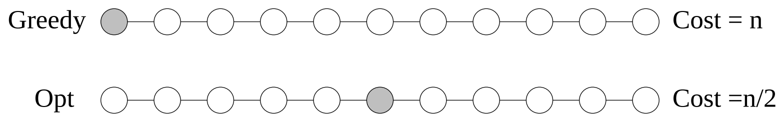


Figure 64: Worst-case for greedy.

Let $G = \{g_1, g_2, \ldots, g_k\}$ be the centers found by the greedy approximation (shown as white dots in the figure below). Also, let g_{k+1} denote the next center that *would have* been added next, that is, the bottleneck vertex for C_G. Let $D_G = D(C_G)$ denote the bottleneck distance for C_G. Notice that the distance from g_{k+1} to its nearest center is equal D_G. The proof involves a cute application of the pigeon-hold principal.

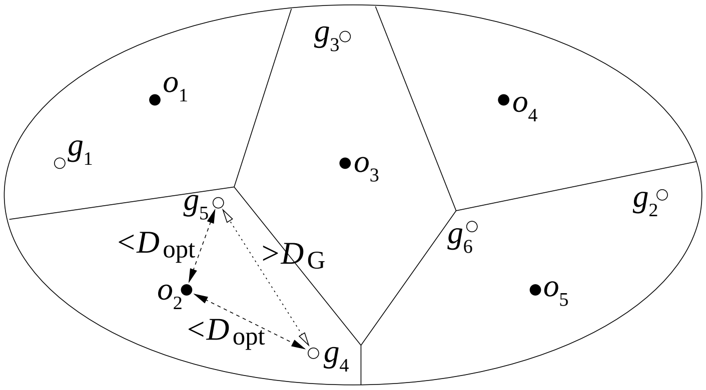


Figure 65: Analysis of the greedy heuristic for $k = 5$.

Theorem: The greedy approximation has a ratio bound of 2, that is $D_G/D_{\text{opt}} \leq 2$.

Proof: Let $G' = \{g_1, g_2, \ldots, g_k, g_{k+1}\}$ be the $(k+1)$-element set consisting of the greedy centers together with the next greedy center g_{k+1} First observe that for $i \neq j$, $d(g_i, g_j) \geq D_G$. This is a result of our greedy selection strategy. As each center is selected, it is selected to be at the maximum (bottleneck) distance from all the previous centers. As we add more centers, the maximum distance between any pair of centers decreases. Since the final bottleneck distance is D_G, all the centers are at least this far apart from one another.

Each $g_i \in G'$ is associated with its closest center in the optimal solution, that is, each belongs to $V(o_m)$ for some m. Because there are k centers in O, and $k+1$ elements in G', it follows from the pigeonhole principal, that at least two centers of G' are in the same set $V(o_m)$ for some m. (In the figure, the greedy centers g_4 and g_5 are both in $V(o_2)$). Let these be denoted g_i and g_j.

Since D_{opt} is the bottleneck distance for O, we know that there is a path of length D_{opt} from g_i to o_k and a path of length D_{opt} from o_k to g_j. By triangle inequality, this implies that there exists a path of length $2D_{\text{opt}}$ from g_i to g_j. Thus, $d(g_i, g_j) \leq 2D_{\text{opt}}$. But from the comments above we have $d(g_i, g_j) \geq D_G$. Therefore,

$$D_G \leq d(g_i, g_j) \leq 2D_{\text{opt}},$$

from which the desired ratio follows.

Lecture 29: Final Review

(Tuesday, Dec 14, 1999)

Overview: This semester we have discussed general approaches to algorithm design. The intent has been to investigate basic algorithm design paradigms: divide-and-conquer, greedy algorithms, dynamic programming, depth-first search, etc. And to consider these techniques on fairly simple, well-defined problems. We have also discussed asymptotic notation and techniques for mathematically analyzing the performance of algorithms, and we have discussed a class of very hard problems to solve, called NP-complete problems, and how to approximate them.

How to use this information: In some sense, the algorithms you have learned here are hardly ever immediately applicable to your later work (unless you go on to be an algorithm designer) because real world problems are always messier than these simple abstract problems. However, there are some important lessons to take out of this class.

Develop a clean mathematical model: Most real-world problems are messy. An important first step in solving any problem is to produce a simple and clean mathematical formulation. For example, this might involve describing the problem as an optimization problem on graphs, sets, or strings.

Create good rough designs: Before jumping in and starting coding, it is important to begin with a good rough design. If your rough design is based on a bad paradigm (e.g. exhaustive enumeration, when depth-first search could have been applied) then no amount of additional tuning and refining will save this bad design.

Prove your algorithm correct: Many times you come up with an idea that seems promising, only to find out later that it does not work. Try to prove that your algorithm is correct. If you cannot see why it is correct, chances are that it is not correct at all.

Use general paradigms in designing algorithms: Rather than designing an algorithm from scratch ask yourself whether the design paradigms you know can be used to solve your problem. Is there a divide-and-conquer solution? greedy solution? etc.

Can it be improved?: Once you have a solution, try to come up with a better one. Is there some reason why a better solution does not exist? If your solution is exponential time, then maybe your problem is NP-hard.

Prototype to generate better designs: We have attempted to analyze algorithms from an asymptotic perspective, which hides many of details of the running time (e.g. constant factors), but give a general perspective for separating good designs from bad ones. After you have isolated the good designs, then it is time to start prototyping and doing empirical tests to establish the real constant factors.

Still too slow?: If your problem has an unacceptably high execution time, you might consider an approximation algorithm. The world is full of heuristics, good and bad. You should develop a good heuristic, and if possible, prove a ratio bound for your algorithm. If you cannot prove a ratio bound, run many experiments to see how good the actual performance is.

There is still much more to be learned about algorithm design, but we have covered a great deal of the basic material. One direction is to specialize in some particular area, e.g. string pattern matching, computational geometry, parallel algorithms, randomized algorithms, or approximation algorithms. It would be easy to devote an entire semester to any one of these topics. Another direction is to gain a better understanding of average-case analysis, which we have largely ignored. Still another direction might be to study numerical algorithms (as covered in a course on numerical analysis), or to consider general search strategies such as simulated annealing. Finally, an emerging area is the study of algorithm engineering, which considers how to design algorithms that are both efficient in a practical sense, as well as a theoretical sense.

Material for the final exam:

Old Material: Know general results, but I will not ask too many detailed questions. Do not forget DFS and DP. You will likely an algorithm design problem that will involve one of these two techniques.

Shortest paths: (Chapt 25 and 26.2.)

Dijkstra's Algorithm: Single source, nonnegative edge weights. Running time $O(E \log V)$.

Bellman-Ford Algorithm: Single source, arbitrary edge weights (no negative cost cycles). Running time $O(VE)$.

Floyd-Warshall Algorithm: All-pairs shortest paths, arbitrary edge weights (no negative cost cycles). Running time $O(V^3)$.

NP-completeness: (Chapt 36.).

Basic concepts: Decision problems, polynomial time, the class P, certificates and the class NP, polynomial time reductions.

NP-completeness reductions: You are responsible for knowing the following reductions.

- 3-coloring to clique cover.
- 3SAT to independent set.
- Independent set to vertex cover and clique.
- Vertex cover to dominating set.
- 3SAT to directed Hamiltonian path.

It is also a good idea to understand all the reductions that were used in the homework solutions, since modifications of these may appear on the final.

NP-complete reductions can be challenging. If you cannot see how to solve the problem, here are some suggestions for maximizing partial credit.

All NP-complete proofs have a very specific form. Explain that you know the template, and try to fill in as many aspects as possible. Suppose that you want to prove that some problem B is NP-complete.

- $B \in$ NP. This almost always easy, so don't blow it. Remember that the certificate is almost always the thing that the problem is asking you to find.
- For some known NP-complete problem A, $A \leq_P B$. This means that you want to find a polynomial time function f that maps an instance of A to an instance of B. (Make sure to get the direction correct!)
- Show the correctness of your reduction, by showing that $x \in A$ if and only if $f(x) \in B$. First suppose that you have a solution to x and show how to map this to a solution for $f(x)$. Then suppose that you have a solution to $f(x)$ and show how to map this to a solution for x.

If you cannot figure out what f is, at least tell me what you would like f to do. Explain which elements of problem A will likely map to which elements of problem B. Remember that you are trying to translate the elements of one problem into the common elements of the other problem.

I try to make at least one reduction on the exam similar to one that you have seen before, so make sure that understand the ones that we have done either in class or on homework problems.

Approximation Algorithms: (Chapt. 37, up through 37.2).

Vertex cover: Ratio bound of 2.

TSP with triangle inequality: Ratio bound of 2.

Set Cover: Ratio bound of $\ln m$, where $m = |X|$.

Bin packing: Ratio bound of 2.

k-center: Ratio bound of 2.

Many approximation algorithms are simple. (Most are based on simple greedy heuristics.) The key to proving many ratio bounds is first coming up with a lower bound on the optimal solution (e.g., $\text{TSP}_{\text{opt}} \le \text{MST}$). Next, provide an upper bound on the cost of your heuristic relative to this same quantity (e.g., the shortcut twice-around tour for the MST is at most twice the MST cost).

Lecture X01: Recurrences and Generating Functions

(Supplementary)

Read: This material is not covered in CLR. There a good description of generating functions in D. E. Knuth, *The Art of Computer Programming, Vol 1.*

Induction and Constructive Induction: Last time we introduced recurrences as a method for analyzing recursive algorithms, where the sizes of the recursive subproblems is predictable. Another technique for solving recurrences (and this works for summations as well) is to guess the solution, or the general form of the solution, and then attempt to verify its correctness through induction. Sometimes there are parameters whose values you do not know. This is fine. In the course of the induction proof, you will usually find out what these values must be. We will consider a famous example, that of the *Fibonacci numbers.*

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{for } n \ge 2. \end{aligned}$$

The Fibonacci numbers arise in data structure design. If you study AVL, or height balanced, trees in CMSC 420, you will learn that the minimum-sized AVL trees are produced by the recursive construction given below. Let $L(i)$ denote the number of leaves in the minimum-sized AVL tree of height i. To construct a minimum-sized AVL tree of height i, you create a root node whose children consist of a minimum-sized AVL tree of heights $i-1$ and $i-2$. Thus the number of leaves obeys $L(0) = L(1) = 1$, $L(i) = L(i-1) + L(i-2)$. It is easy to see that $L(i) = F_{i+1}$.

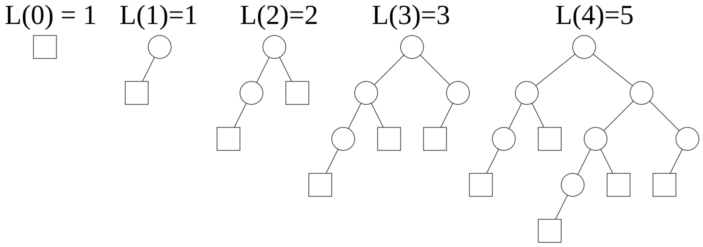


Figure 66: Minimum-sized AVL trees.

If you expand the Fibonacci series for a number of terms, you will observe that F_n appears to grow exponentially, but not as fast as 2^n. It is tempting to conjecture that $F_n \le \phi^{n-1}$, for some real parameter ϕ, where $1 < \phi < 2$. We can use induction to prove this and derive a bound on ϕ.

Lemma: For all integers $n \geq 1$, $F_n \leq \phi^{n-1}$ for some constant ϕ, $1 < \phi < 2$.

Proof: We will try to derive the tightest bound we can on the value of ϕ.

Basis: For the basis cases we consider $n = 1$. Observe that $F_1 = 1 \leq \phi^0$, as desired.

Induction step: For the induction step, let us assume that $F_m \leq \phi^{m-1}$ whenever $1 \leq m < n$. Using this *induction hypothesis* we will show that the lemma holds for n itself, whenever $n \geq 2$.

Since $n \geq 2$, we have $F_n = F_{n-1} + F_{n-2}$. Now, since $n-1$ and $n-2$ are both strictly less than n, we can apply the induction hypothesis, from which we have

$$F_n \leq \phi^{n-2} + \phi^{n-3} \;=\; \phi^{n-3}(1+\phi).$$

We want to show that this is at most ϕ^{n-1} (for a suitable choice of ϕ). Clearly this will be true if and only if $(1+\phi) \leq \phi^2$. This is not true for all values of ϕ (for example it is not true when $\phi = 1$ but it is true when $\phi = 2$.)

At the critical value of ϕ this inequality will be an equality, implying that we want to find the roots of the equation

$$\phi^2 - \phi - 1 = 0.$$

By the quadratic formula we have

$$\phi \;=\; \frac{1 \pm \sqrt{1+4}}{2} \;=\; \frac{1 \pm \sqrt{5}}{2}.$$

Since $\sqrt{5} \approx 2.24$, observe that one of the roots is negative, and hence would not be a possible candidate for ϕ. The positive root is

$$\phi \;=\; \frac{1 + \sqrt{5}}{2} \approx 1.618.$$

There is a very subtle bug in the preceding proof. Can you spot it? The error occurs in the case $n = 2$. Here we claim that $F_2 = F_1 + F_0$ and then we apply the induction hypothesis to both F_1 and F_0. But the induction hypothesis only applies for $m \geq 1$, and hence cannot be applied to F_0! To fix it we could include F_2 as part of the basis case as well.

Notice not only did we prove the lemma by induction, but we actually determined the value of ϕ which makes the lemma true. This is why this method is called *constructive induction*.

By the way, the value $\phi = \frac{1}{2}(1+\sqrt{5})$ is a famous constant in mathematics, architecture and art. It is the *golden ratio*. Two numbers A and B satisfy the golden ratio if

$$\frac{A}{B} = \frac{A+B}{A}.$$

It is easy to verify that $A = \phi$ and $B = 1$ satisfies this condition. This proportion occurs throughout the world of art and architecture.

Generating Functions: The method of constructive induction provided a way to get a bound on F_n, but we did not get an exact answer, and we had to generate a good guess before we were even able to start.

Let us consider an approach to determine an exact representation of F_n, which requires no guesswork. This method is based on a very elegant concept, called a *generating function*. Consider any infinite sequence:

$$a_0, a_1, a_2, a_3, \ldots$$

If we would like to "encode" this sequence succinctly, we could define a polynomial function such that these are the coefficients of the function:

$$G(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \ldots$$

This is called the *generating function* of the sequence. What is z? It is just a symbolic variable. We will (almost) never assign it a specific value. Thus, every infinite sequence of numbers has a corresponding generating function, and vice versa. What is the advantage of this representation? It turns out that we can perform arithmetic transformations on these functions (e.g., adding them, multiplying them, differentiating them) and this has a corresponding effect on the underlying transformations. It turns out that some nicely-structured sequences (like the Fibonacci numbers, and many sequences arising from linear recurrences) have generating functions that are easy to write down and manipulate.

Let's consider the generating function for the Fibonacci numbers:

$$\begin{aligned} G(z) &= F_0 + F_1 z + F_2 z^2 + F_3 z^3 + \ldots \\ &= z + z^2 + 2z^3 + 3z^4 + 5z^5 + \ldots \end{aligned}$$

The trick in dealing with generating functions is to figure out how various manipulations of the generating function to generate algebraically equivalent forms. For example, notice that if we multiply the generating function by a factor of z, this has the effect of shifting the sequence to the right:

$$\begin{array}{rcccccccccccc} G(z) & = & F_0 & + & F_1 z & + & F_2 z^2 & + & F_3 z^3 & + & F_4 z^4 & + & \ldots \\ zG(z) & = & & & F_0 z & + & F_1 z^2 & + & F_2 z^3 & + & F_3 z^4 & + & \ldots \\ z^2 G(z) & = & & & & & F_0 z^2 & + & F_1 z^3 & + & F_2 z^4 & + & \ldots \end{array}$$

Now, let's try the following manipulation. Compute $G(z) - zG(z) - z^2G(z)$, and see what we get

$$\begin{aligned} (1 - z - z^2)G(z) &= F_0 + (F_1 - F_0)z + (F_2 - F_1 - F_0)z^2 + (F_3 - F_2 - F_1)z^3 \\ &\qquad + \ldots + (F_i - F_{i-1} - F_{i-2})z^i + \ldots \\ &= z. \end{aligned}$$

Observe that every term except the second is equal to zero by the definition of F_i. (The particular manipulation we picked was chosen to cause this cancellation to occur.) From this we may conclude that

$$G(z) = \frac{z}{1 - z - z^2}.$$

So, now we have an alternative representation for the Fibonacci numbers, as the coefficients of this function if expanded as a power series. So what good is this? The main goal is to get at the coefficients of its power series expansion. There are certain common tricks that people use to manipulate generating functions.

The first is to observe that there are some functions for which it is very easy to get an power series expansion. For example, the following is a simple consequence of the formula for the geometric series. If $0 < c < 1$ then

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1 - c}.$$

Setting $z = c$, we have

$$\frac{1}{1 - z} = 1 + z + z^2 + z^3 + \ldots$$

(In other words, $1/(1-z)$ is the generating function for the sequence $(1, 1, 1, \ldots)$. In general, given an constant a we have

$$\frac{1}{1-az} = 1 + az + a^2z^2 + a^3z^3 + \ldots$$

is the generating function for $(1, a, a^2, a^3, \ldots)$. It would be great if we could modify our generating function to be in the form of $1/(1-az)$ for some constant a, since then we could then extract the coefficients of the power series easily.

In order to do this, we would like to rewrite the generating function in the following form:

$$G(z) \;=\; \frac{z}{1-z-z^2} \;=\; \frac{A}{1-az} + \frac{B}{1-bz},$$

for some A, B, a, b. We will skip the steps in doing this, but it is not hard to verify the roots of $(1-az)(1-bz)$ (which are $1/a$ and $1/b$) must be equal to the roots of $1-z-z^2$. We can then solve for a and b by taking the reciprocals of the roots of this quadratic. Then by some simple algebra we can plug these values in and solve for A and B yielding:

$$G(z) \;=\; \frac{z}{1-z-z^2} \;=\; \left(\frac{1/\sqrt{5}}{1-\phi z} + \frac{-1/\sqrt{5}}{1-\hat{\phi}}\right) \;=\; \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}}\right),$$

where $\phi = (1+\sqrt{5})/2$ and $\hat{\phi} = (1-\sqrt{5})/2$. (In particular, to determine A, multiply the equation by $1-\phi z$, and then consider what happens when $z = 1/\phi$. A similar trick can be applied to get B. In general, this is called the method of *partial fractions*.)

Now we are in good shape, because we can extract the coefficients for these two fractions from the above function. From this we have the following:

$$\begin{array}{rcllllllll} G(z) & = & \frac{1}{\sqrt{5}}\,(& 1 & + & \phi z & + & \phi^2 z^2 & + & \ldots \\ & & & -1 & + & -\hat{\phi} z & + & -\hat{\phi}^2 z^2 & + & \ldots\;) \end{array}$$

Combining terms we have

$$G(z) = \frac{1}{\sqrt{5}} \sum_{i=0}^{\infty} (\phi^i - \hat{\phi}^i) z^i.$$

We can now read off the coefficients easily. In particular it follows that

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n).$$

This is an exact result, and no guesswork was needed.

Lecture X02: Articulation Points and Biconnected Components

(Supplementary)

Read: This material is not covered in the text (except as Problem 23–2).

Articulation Points and Biconnected Graphs: Today we discuss another application of DFS, this time to a problem on undirected graphs. 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.)

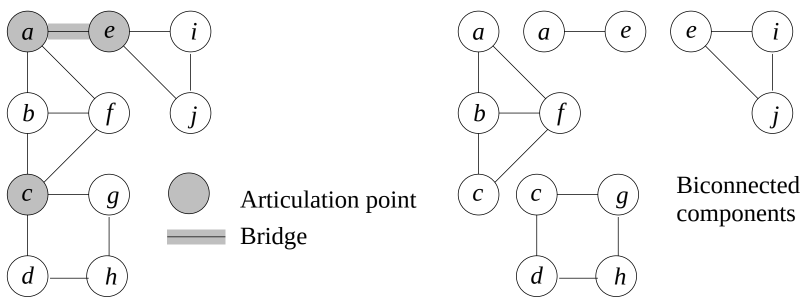


Figure 67: Articulation Points and Bridges

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.

Last time we observed that the notion of mutual reachability partitioned the vertices of a digraph into equivalence classes. We would like to do the same thing here. We say that two edges e_1 and e_2 are *cocyclic* if either $e_1 = e_2$ or if there is a simple cycle that contains both edges. It is not too hard to verify that this defines an equivalence relation on the edges of a graph. Notice that if two edges are cocyclic, then there are essentially two different ways of getting from one edge to the other (by going around the the cycle each way).

Biconnected components: The biconnected components of a graph are the equivalence classes of the cocylicity relation.

Notice that unlike strongly connected components of a digraph (which form a partition of the vertex set) the biconnected components of a graph form a partition of the edge set. You might think for a while why this is so.

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 u 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 we assume is only one tree in the DFS forest. Because G is undirected, the DFS tree has a simpler structure. First off, we cannot distinguish between forward edges and back edges, and we just call them back edges. Also, there are no cross edges. (You should take a moment to convince yourself why this is true.)

For now, let us consider the typical case of a vertex u, where u is not a leaf and u is not the root. Let's let $v_1, v_2, \ldots, v_k$ be the children of u. 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

u, then if we were to remove u, this subtree would become disconnected from the rest of the graph, and hence u is an articulation point. On the other hand, if every one of the subtrees rooted at the children of u have back edges to proper ancestors of u, then if u is removed, the graph remains connected (the backedges hold everything together). This leads to the following.

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

Please check this condition carefully to see that you understand it. In particular, notice that the condition for whether u is an articulation point depends on a test applied to its children. This is the most common source of confusion for this algorithm.

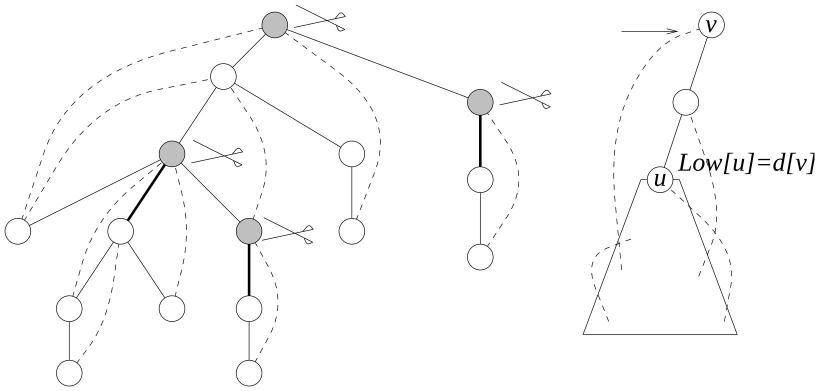


Figure 68: Articulation Points and DFS

What about the leaves? If u 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 edges. 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 u, this one will.

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

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

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

The term "descendent" is used in the nonstrict sense, that is, v may be equal to u. Intuitively, $Low[u]$ is the highest (closest to the root) that you can get in the tree by taking any one backedge from either u or any of its descendents. (Beware of this notation: "Low" means low discovery time, not low in the tree. In fact $Low[u]$ tends to be "high" in the tree, in the sense of being close to the root.)

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

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

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

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

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

The Final Algorithm: There is one subtlety that we must watch for in designing the algorithm (in particular this is true for any DFS on undirected graphs). When processing a vertex u, we need to know when a given edge (u, v) is a back edge. How do we do this? An almost correct answer is to test whether v is colored gray (since all gray vertices are ancestors of the current vertex). This is not quite correct because v may be the parent of v in the DFS tree and we are just seeing the "other side" of the tree edge between v and u (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 v is not the parent of u in the DFS tree.

The complete algorithm for computing articulation points is given below. The main procedure for DFS is the same as before, except that it calls the following routine rather than `DFSvisit()`.

Articulation Points

```
ArtPt(u) {
    color[u] = gray
    Low[u] = d[u] = ++time
    for each (v in Adj(u)) {
        if (color[v] == white) {          // (u,v) is a tree edge
```

```
                pred[v] = u
                ArtPt(v)
                Low[u] = min(Low[u], Low[v])  // update Low[u]
                if (pred[u] == NULL) {        // root: apply Observation 3
                    if (this is u's second child)
                        Add u to set of articulation points
                }
                else if (Low[v] >= d[u]) {    // internal node: apply Observation 1
                    Add u to set of articulation points
                }
            }
            else if (v != pred[u]) {          // (u,v) is a back edge
                Low[u] = min(Low[u], d[v])    // update L[u]
            }
        }
    }
```

An example is shown in the following figure. As with all DFS-based algorithms, the running time is $\Theta(n+e)$. There are some interesting problems that we still have not discussed. We did not discuss how to compute the bridges of a graph. This can be done by a small modification of the algorithm above. We'll leave it as an exercise. (Notice that if $\{u, v\}$ is a bridge then it does not follow that u and v are both articulation points.) Another question is how to determine which edges are in the biconnected components. A hint here is to store the edges in a stack as you go through the DFS search. When you come to an articulation point, you can show that all the edges in the biconnected component will be consecutive in the stack.

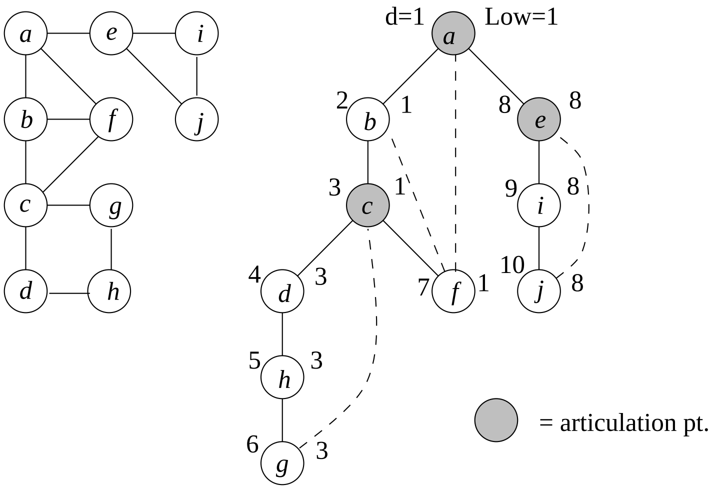


Figure 69: Articulation Points.

Lecture X03: Network Flows and Matching

(Supplementary)

Read: This lecture was presented by Dr. Khuller. See Chapt 27 in CLR.

Maximum Flow: The Max Flow problem is one of the basic problems of algorithm design. Intuitively we can think of a flow network as a directed graph in which fluid is flowing along the edges of the graph. Each edge has certain maximum capacity that it can carry. The idea is to find out how much flow we can push from one point to another.

The max flow problem has applications in areas like transportation, routing in networks. It is the simplest problem in a line of many important problems having to do with the movement of commodities through a network. These are often studied in business schools, and operations research.

Flow Networks: A *flow network* $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative *capacity* $c(u, v) \geq 0$. If $(u, v) \notin E$ we model this by setting $c(u, v) = 0$. There are two special vertices: a *source* s, and a *sink* t. We assume that every vertex lies on some path from the source to the sink (for otherwise the vertex is of no use to us). (This implies that the digraph is connected, and hence $e \geq n - 1$.)

A *flow* is a real valued function on pairs of vertices, $f : V \times V \to R$ which satisfies the following three properties:

Capacity Constraint: For all $u, v \in V$, $f(u, v) \leq c(u, v)$.

Skew Symmetry: For all $u, v \in V$, $f(u, v) = -f(v, u)$. (In other words, we can think of backwards flow as negative flow. This is primarily for making algebraic analysis easier.)

Flow conservation: For all $u \in V - \{s, t\}$, we have

$$\sum_{v \in V} f(u, v) = 0.$$

(Given skew symmetry, this is equivalent to saying, flow-in = flow-out.) Note that flow conservation does NOT apply to the source and sink, since we think of ourselves as pumping flow from s to t. Flow conservation means that no flow is lost anywhere else in the network, thus the flow out of s will equal the flow into t.

The quantity $f(u, v)$ is called the *net flow* from u to v. The total *value* of the flow f is defined as

$$|f| = \sum_{v \in V} f(s, v)$$

i.e. the flow out of s. It turns out that this is also equal to $\sum_{v \in V} f(v, t)$, the flow into t. We will show this later.

The *maximum-flow problem* is, given a flow network, and source and sink vertices s and t, find the flow of maximum value from s to t.

Example: Page 581 of CLR.

Multi-source, multi-sink flow problems: It may seem overly restrictive to require that there is only a single source and a single sink vertex. Many flow problems have situations in which many source vertices $s_1, s_2, \ldots, s_k$ and many sink vertices $t_1, t_2, \ldots, t_l$. This can easily be modelled by just adding a special *supersource* s' and a *supersink* t', and attaching s' to all the s_i and attach all the t_j to t'. We let these edges have infinite capacity. Now by pushing the maximum flow from s' to t' we are effectively producing the maximum flow from all the s'_i to all the t_j's.

Note that we don't care which flow from one source goes to another sink. If you require that the flow from source i goes ONLY to sink i, then you have a tougher problem called the *multi-commodity flow problem.*

Set Notation: Sometimes rather than talking about the flow from a vertex u to a vertex v, we want to talk about the flow from a SET of vertices X to another SET of vertices Y. To do this we extend the definition of f to sets by defining

$$f(X, Y) = \sum_{x \in X} \sum y \in Y f(x, y).$$

Using this notation we can define flow balance for a vertex u more succintly by just writing $f(u, V) = 0$. One important special case of this concept is when X and Y define a *cut* (i.e. a partition of the vertex set into two disjoint subsets $X \subseteq V$ and $Y = V - X$). In this case $f(X, Y)$ can be thought of as the net amount of flow crossing over the cut.

From simple manipulations of the definition of flow we can prove the following facts.

Lemma:

(i) $f(X, X) = 0$.

(ii) $f(X, Y) = -f(Y, X)$.

(iii) If $X \cap Y = \emptyset$ then $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ and $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.

Ford-Fulkerson Method: The most basic concept on which all network-flow algorithms work is the notion of *augmenting flows*. The idea is to start with a flow of size zero, and then incrementally make the flow larger and larger by finding a path along which we can push more flow. A path in the network from s to t along which more flow can be pushed is called an *augmenting path*. This idea is given by the most simple method for computing network flows, called the Ford-Fulkerson method.

Ford-Fulkerson Network Flow

```
Ford_Fulkerson(G, s, t) {
    initialize flow f to 0;
    while (there exists an augmenting path p) {
        augment the flow along p;
    }
    output the final flow f;
}
```

Almost all network flow algorithms are based on this simple idea. They only differ in how they decide which path or paths along which to push flow. We will prove that when it is impossible to "push" any more flow through the network, we have reached the maximum possible flow (i.e. a locally maximum flow is globally maximum).

Residual Network: To define the notion of an augmenting path, we first define the notion of a residual network. Given a flow network G and a flow f, define the *residual capacity* of a pair $u, v \in V$ to be $c_f(u, v) = c(u, v) - f(u, v)$. Because of the capacity constraint, $c_f(u, v) \geq 0$. Observe that if $c_f(u, v) > 0$ then it is possible to push more flow through the edge (u, v). Otherwise we say that the edge is *saturated*.

The *residual network* is the directed graph G_f with the same vertex set as G but whose edges are the pairs (u, v) such that $c_f(u, v) > 0$. Each edge in the residual network is weighted with its residual capacity.

Example: Page 589 of CLR.

Lemma: Let f be a flow in G and let f' be a flow in G_f. Then $(f+f')$ (defined $(f+f')(u, v) = f(u, v) + f'(u, v)$) is a flow in G. The value of the flow is $|f| + |f'|$.

Proof: Basically the residual network tells us how much additional flow we can push through G. This implies that $f + f'$ never exceeds the overall edge capacities of G. The other rules for flows are easy to verify.

Augmenting Paths: An *augmenting path* is a simple path from s to t in G_f. The *residual capacity* of the path is the MINIMUM capacity of any edge on the path. It is denoted $c_f(p)$. Observe that by pushing $c_f(p)$ units of flow along each edge of the path, we get a flow in G_f, and hence we can use this to augment the flow in G. (Remember that when defining this flow that whenever we push $c_f(p)$ units of flow along any edge (u, v) of p, we have to push $-c_f(p)$ units of flow along the reverse edge (v, u) to maintain skew-symmetry. Since every edge of the residual network has a strictly positive weight, the resulting flow is strictly larger than the current flow for G.

In order to determine whether there exists an augmenting path from s to t is an easy problem. First we construct the residual network, and then we run DFS or BFS on the residual network starting at s. If the search reaches t then we know that a path exists (and can follow the predecessor pointers backwards to reconstruct it). Since DFS and BFS take $\Theta(n + e)$ time, and it can be shown that the residual network has $\Theta(n + e)$ size, the running time of Ford-Fulkerson is basically

$$\Theta((n + e)(\text{number of augmenting stages})).$$

Later we will analyze the latter quantity.

Correctness: To establish the correctness of the Ford-Fulkerson algorithm we need to delve more deeply into the theory of flows and cuts in networks. A *cut*, (S, T), in a flow network is a partition of the vertex set into two disjoint subsets S and T such that $s \in S$ and $t \in T$. We define the flow across the cut as $f(S, T)$, and we define the capcity of the cut as $c(S, T)$. Note that in computing $f(S, T)$ flows from T to S are counted negatively (by skew-symmetry), and in computing $c(S, T)$ we ONLY count constraints on edges leading from S to T ignoring those from T to S).

Lemma: The amount of flow across any cut in the network is equal to $|f|$.

Proof:

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) \\ &= f(s, V) + f(S - s, V) \\ &= f(s, V) \\ &= |f| \end{aligned}$$

(The fact that $f(S - s, V) = 0$ comes from flow conservation. $f(u, V) = 0$ for all u other than s and t, and since $S - s$ is formed of such vertices the sum of their flows will be zero also.)

Corollary: The value of any flow is bounded from above by the capacity of any cut. (i.e. Maximum flow $\leq$ Minimum cut).

Proof: You cannot push any more flow through a cut than its capacity.

The correctness of the Ford-Fulkerson method is based on the following theorem, called the Max-Flow, Min-Cut Theorem. It basically states that in any flow network the minimum capacity cut acts like a bottleneck to limit the maximum amount of flow. Ford-Fulkerson algorithm terminates when it finds this bottleneck, and hence it finds the minimum cut and maximum flow.

Max-Flow Min-Cut Theorem: The following three conditions are equivalent.

(i) f is a maximum flow in G,

(ii) The residual network G_f contains no augmenting paths,

(iii) $|f| = c(S, T)$ for some cut (S, T) of G.

Proof: (i) $\Rightarrow$ (ii): If f is a max flow and there were an augmenting path in G_f, then by pushing flow along this path we would have a larger flow, a contradiction.

(ii) $\Rightarrow$ (iii): If there are no augmenting paths then s and t are not connected in the residual network. Let S be those vertices reachable from s in the residual network and let T be the rest. (S, T) forms a cut. Because each edge crossing the cut must be saturated with flow, it follows that the flow across the cut equals the capacity of the cut, thus $|f| = c(S, T)$.

(iii) $\Rightarrow$ (i): Since the flow is never bigger than the capacity of any cut, if the flow equals the capacity of some cut, then it must be maximum (and this cut must be minimum).

Analysis of the Ford-Fulkerson method: The problem with the Ford-Fulkerson algorithm is that depending on how it picks augmenting paths, it may spend an inordinate amount of time arriving a the final maximum flow. Consider the following example (from page 596 in CLR). If the algorithm were smart enough to send flow along the edges of weight 1,000,000, the algorithm would terminate in two augmenting steps. However, if the algorithm were to try to augment using the middle edge, it will continuously improve the flow by only a single unit. 2,000,000 augmenting will be needed before we get the final flow. In general, Ford-Fulkerson can take time $\Theta((n + e)|f^*|)$ where f^* is the maximum flow.

An Improvement: We have shown that if the augmenting path was chosen in a bad way the algorithm could run for a very long time before converging on the final flow. It seems (from the example we showed) that a more logical way to push flow is to select the augmenting path which holds the maximum amount of flow. Computing this path is equivalent to determining the path of maximum capacity from s to t in the residual network. (This is exactly the same as the beer transport problem given on the last exam.) It is not known how fast this method works in the worst case, but there is another simple strategy that is guaranteed to give good bounds (in terms of n and e).

Edmonds-Karp Algorithm: The Edmonds-Karp algorithm is Ford-Fulkerson, with one little change. When finding the augmenting path, we use Breadth-First search in the residual network, starting at the source s, and thus we find the shortest augmenting path (where the length of the path is the number of edges on the path). We claim that this choice is particularly nice in that, if we do so, the number of flow augmentations needed will be at most $O(e \cdot n)$. Since each augmentation takes $O(n + e)$ time to compute using BFS, the overall running time will be $O((n+e)e \cdot n) = O(n^2 e + e^2 n) \in O(e^2 n)$ (under the reasonable assumption that $e \geq n$). (The best known algorithm is essentially $O(e \cdot n \log n)$.

The fact that Edmonds-Karp uses $O(en)$ augmentations is based on the following observations.

Observation: If the edge (u, v) is an edge on the minimum length augmenting path from s to t in G_f, then $\delta_f(s, v) = \delta_f(s, u) + 1$.

Proof: This is a simple property of shortest paths. Since there is an edge from u to v, $\delta_f(s, v) \leq \delta_f(s, u) + 1$, and if $\delta_f(s, v) < \delta_f(s, u) + 1$ then u would not be on the shortest path from s to v, and hence (u, v) is not on any shortest path.

Lemma: For each vertex $u \in V - \{s,t\}$, let $\delta_f(s,u)$ be the distance function from s to u in the residual network G_f. Then as we peform augmentations by the Edmonds-Karp algorithm the value of $\delta_f(s,u)$ increases monotonically with each flow augmentation.

Proof: (Messy, but not too complicated. See the text.)

Theorem: The Edmonds-Karp algorithm makes at most $O(n \cdot e)$ augmentations.

Proof: An edge in the augmenting path is *critical* if the residual capacity of the path equals the residual capacity of this edge. In other words, after augmentation the critical edge becomes saturated, and disappears from the residual graph.

How many times can an edge become critical before the algorithm terminates? Observe that when the edge (u,v) is critical it lies on the shortest augmenting path, implying that $\delta_f(s,v) = \delta_f(s,u) + 1$. After this it disappears from the residual graph. In order to reappear, it must be that we reduce flow on this edge, i.e. we push flow along the reverse edge (v,u). For this to be the case we have (at some later flow f') $\delta_{f'}(s,u) = \delta_{f'}(s,v)+1$. Thus we have:

$$\begin{aligned} \delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 \\ &\geq \delta_f(s,v) + 1 \qquad \text{since dists increase with time} \\ &= (\delta_f(s,u) + 1) + 1 \\ &= \delta_f(s,u) + 2. \end{aligned}$$

Thus, between the time that an edge becomes critical, its tail vertex increases in distance from the source by two. This can only happen $n/2$ times, since no vertex can be further than n from the source. Thus, each edge can become critical at most $O(n)$ times, there are $O(e)$ edges, hence after $O(ne)$ augmentations, the algorithm must terminate.

In summary, the Edmonds-Karp algorithm makes at most $O(ne)$ augmentations and runs in $O(ne^2)$ time.

Maximum Matching: One of the important elements of network flow is that it is a very general algorithm which is capable of solving many problems. (An example is problem 3 in the homework.) We will give another example here.

Consider the following problem, you are running a dating service and there are a set of men L and a set of women R. Using a questionaire you establish which men are compatible which which women. Your task is to pair up as many compatible pairs of men and women as possible, subject to the constraint that each man is paired with at most one woman, and vice versa. (It may be that some men are not paired with any woman.)

This problem is modelled by giving an undirected graph whose vertex set is $V = L \cup R$ and whose edge set consists of pairs (u,v), $u \in L$, $v \in R$ such that u and v are compatible. The problem is to find a *matching*, that is a subset of edges M such that for each $v \in V$, there is at most one edge of M incident to v. The desired matching is the one that has the maximum number of edges, and is called a *maximum matching.*

Example: See page 601 in CLR.

The resulting undirected graph has the property that its vertex set can be divided into two groups such that all its edges go from one group to the other (never within a group, unless the dating service is located on Dupont Circle). This problem is called the *maximum bipartite matching problem.*

Reduction to Network Flow: We claim that if you have an algorithm for solving the network flow problem, then you can use this algorithm to solve the maximum bipartite matching problem. (Note that this idea does not work for general undirected graphs.)

Construct a flow network $G' = (V', E')$ as follows. Let s and t be two new vertices and let $V' = V \cup \{s, t\}$.

$$E' = \{(s, u) | u \in L\} \cup \{(v, t) | v \in R\} \cup \{(u, v) | (u, v) \in E\}.$$

Set the capacity of all edges in this network to 1.

Example: See page 602 in CLR.

Now, compute the maximum flow in G'. Although in general it can be that flows are real numbers, observe that the Ford-Fulkerson algorithm will only assign integer value flows to the edges (and this is true of all existing network flow algorithms).

Since each vertex in L has exactly 1 incoming edge, it can have flow along at most 1 outgoing edge, and since each vertex in R has exactly 1 outgoing edge, it can have flow along at most 1 incoming edge. Thus letting f denote the maximum flow, we can define a matching

$$M = \{(u, v) | u \in L,\ v \in R,\ f(u, v) > 0\}.$$

We claim that this matching is maximum because for every matching there is a corresponding flow of equal value, and for every (integer) flow there is a matching of equal value. Thus by maximizing one we maximize the other.

Lecture X04: The 0–1 Knapsack Problem

(Supplementary)

Read: The introduction to Chapter 16 in CLR. The material on the Knapsack Problem is not presented in our text, but is briefly discussed in Section 17.2.

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 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

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

Let us assume that the v_i's, w_i's and W are all positive integers. It turns out that this problem is NP-complete, and so we cannot really hope to find an efficient solution. However if we make the same sort of assumption that we made in counting sort, we can come up with an efficient solution.

We assume that the w_i's are small integers, and that W itself is a small integer. We 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 $0 \leq j \leq W$, the entry $V[i, j]$ we will store the maximum value of any subset of objects $\{1, 2, \ldots, i\}$ that can fit into a knapsack of weight j. 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 $V[0, j] = 0$ for $0 \leq j \leq W$ since if we have no items then we have no value. 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 j with the remaining objects $\{1, 2, \ldots, i-1\}$. This is just $V[i-1, j]$.

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 $j - w_i$ capacity in the knapsack, we can fill it in the best possible way with objects $\{1, 2, \ldots, i-1\}$. This is $v_i + V[i-1, j-w_i]$. This is only possible if $w_i \leq j$.

Since these are the only two possibilities, we can see that we have the following rule for constructing the array V. The ranges on i and j are $i \in [0..n]$ and $j \in [0..W]$.

$$\begin{aligned} V[0, j] &= 0 \\ V[i, j] &= \begin{cases} V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], v_i + V[i-1, j-w_i]) & \text{if } w_i \leq j \end{cases} \end{aligned}$$

The first line states that if there are no objects, then there is no value, irrespective of j. The second line implements the rule above.

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 $O((n+1)(W+1)) = O(nW)$. The algorithm is given below.

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

An example is shown in the figure below. The final output is $V[n, W] = V[4, 10] = 90$. This reflects the selection of items 2 and 4, of values \$40 and \$50, respectively and weights $4+3 \leq 10$.

Values of the objects are $\langle 10, 40, 30, 50 \rangle$.
Weights of the objects are $\langle 5, 4, 6, 3 \rangle$.

		Capacity $\rightarrow$	$j=0$	1	2	3	4	5	6	7	8	9	10
Item	Value	Weight	0	0	0	0	0	0	0	0	0	0	0
1	10	5	0	0	0	0	0	10	10	10	10	10	10
2	40	4	0	0	0	0	40	40	40	40	40	50	50
3	30	6	0	0	0	0	40	40	40	40	40	50	70
4	50	3	0	0	0	50	50	50	50	90	90	90	90

Final result is $V[4, 10] = 90$ (for taking items 2 and 4).

Figure 70: 0–1 Knapsack Example.

The only missing detail is what items should we select to achieve the maximum. We will leave this as an exercise. They key is to record for each entry $V[i, j]$ in the matrix whether we got this entry by taking the ith item or leaving it. With this information, it is possible to reconstruct the optimum knapsack contents.

Lecture X05: Subset Sum

(Supplementary)
Read: Chapt 36.5.3 and 36.5.5.

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).

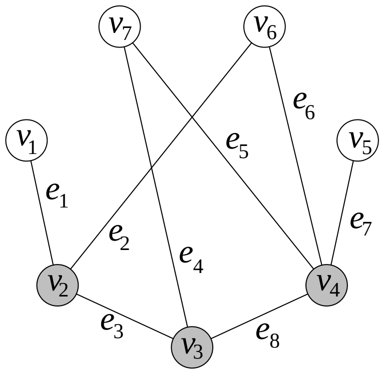
Claim: Subset sum (SS) is NP-complete.

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 $\leq_P$ 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	
	1	1	1	1	1	1	1	1	$v_2 \vee v_3 \vee v_4$

Figure 71: 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. $1211\ldots112$. 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 $2222\ldots222$ (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.

		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 72: Vertex cover to subset sum reduction.

Lecture X06: Subset Sum Approximation

(Supplementary)
Read: Section 37.4.

Polynomial Approximation Schemes: Last time we saw that for some NP-complete problems, it is possible to approximate the problem to within a fixed constant ratio bound. For example, the approximation algorithm produces an answer that is within a factor of 2 of the optimal solution. However, in practice, people would like to the control the precision of the approximation. This is done by specifying a parameter $\epsilon > 0$ as part of the input to the approximation algorithm, and requiring that the algorithm produce an answer that is within a *relative error* of ϵ of the optimal solution. It is understood that as ϵ tends to 0, the running time of the algorithm will increase. Such an algorithm is called a *polynomial approximation scheme*.

For example, the running time of the algorithm might be $O(2^{(1/\epsilon)}n^2)$. It is easy to see that in such cases the user pays a big penalty in running time as a function of ϵ. (For example, to produce a 1% error, the "constant" factor would be 2^{100} which would be around 4 quadrillion centuries on your 100 Mhz Pentium.) A *fully polynomial approximation scheme* is one in which the running time is polynomial in both n and $1/\epsilon$. For example, a running time of $O((n/\epsilon)^2)$ would satisfy this condition. In such cases, reasonably accurate approximations are computationally feasible.

Unfortunately, there are very few NP-complete problems with fully polynomial approximation schemes. In fact, recently there has been strong evidence that many NP-complete problems do not have polynomial approximation schemes (fully or otherwise). Today we will study one that does.

Subset Sum: Recall that in the subset sum problem we are given a set S of positive integers $\{x_1, x_2, \ldots, x_n\}$ and a target value t, and we are asked whether there exists a subset $S' \subseteq S$ that sums exactly to t. The optimization problem is to determine the subset whose sum is as large as possible but not larger than t.

This problem is basic to many packing problems, and is indirectly related to processor scheduling problems that arise in operating systems as well. Suppose we are also given $0 < \epsilon < 1$. Let $z^* \leq t$ denote the optimum sum. The approximation problem is to return a value $z \leq t$ such that

$$z \geq z^*(1 - \epsilon).$$

If we think of this as a knapsack problem, we want our knapsack to be within a factor of $(1-\epsilon)$ of being as full as possible. So, if $\epsilon = 0.1$, then the knapsack should be at least 90% as full as the best possible.

What do we mean by polynomial time here? Recall that the running time should be polynomial in the size of the input length. Obviously n is part of the input length. But t and the numbers x_i could also be huge binary numbers. Normally we just assume that a binary number can fit into a word of our computer, and do not count their length. In this case we will to be on the safe side. Clearly t requires $O(\log t)$ digits to be store in the input. We will take the input size to be $n + \log t$.

Intuitively it is not hard to believe that it should be possible to determine whether we can fill the knapsack to within 90% of optimal. After all, we are used to solving similar sorts of packing problems all the time in real life. But the mental heuristics that we apply to these problems are not necessarily easy to convert into efficient algorithms. Our intuition tells us that we can afford to be a little "sloppy" in keeping track of exactly full the knapsack is at any point. The value of ϵ tells us just how sloppy we can be. Our approximation will do something similar. First we consider an exponential time algorithm, and then convert it into an approximation algorithm.

Exponential Time Algorithm: This algorithm is a variation of the dynamic programming solution we gave for the knapsack problem. Recall that there we used an 2-dimensional array to keep track of whether we could fill a knapsack of a given capacity with the first i objects. We will do something similar here. As before, we will concentrate on the question of which sums are possible, but determining the subsets that give these sums will not be hard.

Let L_i denote a list of integers that contains the sums of all 2^i subsets of $\{x_1, x_2, \ldots, x_i\}$ (including the empty set whose sum is 0). For example, for the set $\{1, 4, 6\}$ the corresponding list of sums contains $\langle 0, 1, 4, 5(= 1+4), 6, 7(= 1+6), 10(= 4+6), 11(= 1+4+6)\rangle$. Note that L_i can have as many as 2^i elements, but may have fewer, since some subsets may have the same sum.

There are two things we will want to do for efficiency. (1) Remove any duplicates from L_i, and (2) only keep sums that are less than or equal to t. Let us suppose that we a procedure `MergeLists(L1, L2)` which merges two sorted lists, and returns a sorted lists with all duplicates removed. This is essentially the procedure used in MergeSort but with the added duplicate element test. As a bit of notation, let $L + x$ denote the list resulting by adding the number x to every element of list L. Thus $\langle 1, 4, 6\rangle + 3 = \langle 4, 7, 9\rangle$. This gives the following procedure for the subset sum problem.

Exact Subset Sum

```
Exact_SS(x[1..n], t) {
    L = <0>;
    for i = 1 to n do {
        L = MergeLists(L, L+x[i]);
        remove for L all elements greater than t;
    }
    return largest element in L;
}
```

For example, if $S = \{1, 4, 6\}$ and $t = 8$ then the successive lists would be

$$
\begin{aligned}
L_0 &= \langle 0\rangle \\
L_1 &= \langle 0\rangle \cup \langle 0+1\rangle = \langle 0, 1\rangle \\
L_2 &= \langle 0, 1\rangle \cup \langle 0+4, 1+4\rangle = \langle 0, 1, 4, 5\rangle \\
L_3 &= \langle 0, 1, 4, 5\rangle \cup \langle 0+6, 1+6, 4+6, 5+6\rangle = \langle 0, 1, 4, 5, 6, 7, 10, 11\rangle.
\end{aligned}
$$

The last list would have the elements 10 and 11 removed, and the final answer would be 7. The algorithm runs in $\Omega(2^n)$ time in the worst case, because this is the number of sums that are generated if there are no duplicates, and no items are removed.

Approximation Algorithm: To convert this into an approximation algorithm, we will introduce a "trim" the lists to decrease their sizes. The idea is that if the list L contains two numbers that are very close to one another, e.g. $91,048$ and $91,050$, then we should not need to keep both of these numbers in the list. One of them is good enough for future approximations. This will reduce the size of the lists that the algorithm needs to maintain. But, how much trimming can we allow and still keep our approximation bound? Furthermore, will we be able to reduce the list sizes from exponential to polynomial?

The answer to both these questions is yes, provided you apply a proper way of trimming the lists. We will trim elements whose values are sufficiently close to each other. But we should define close in manner that is relative to the sizes of the numbers involved. The trimming must also depend on ϵ. We select $\delta = \epsilon/n$. (Why? We will see later that this is the value that makes everything work out in the end.) Note that $0 < \delta < 1$. Assume that the elements of L are sorted. We walk through the list. Let z denote the last untrimmed element in L, and let $y \geq z$ be the next element to be considered. If

$$\frac{y-z}{y} \leq \delta$$

then we trim y from the list. Equivalently, this means that the final trimmed list cannot contain two value y and z such that

$$(1-\delta)y \leq z \leq y.$$

We can think of z as *representing* y in the list.

For example, given $\delta = 0.1$ and given the list

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle,$$

the trimmed list L' will consist of

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle.$$

Another way to visualize trimming is to break the interval from $[1, t]$ into a set of *buckets* of exponentially increasing size. Let $d = 1/(1-\delta)$. Note that $d > 1$. Consider the intervals $[1, d], [d, d^2], [d^2, d^3], \ldots, [d^{k-1}, d^k]$ where $d^k \geq t$. If $z \leq y$ are in the same interval $[d^{i-1}, d^i]$ then

$$\frac{y-z}{y} \leq \frac{d^i - d^{i-1}}{d^i} = 1 - \frac{1}{d} = \delta.$$

Thus, we cannot have more than one item within each bucket. We can think of trimming as a way of enforcing the condition that items in our lists are not relatively too close to one another, by enforcing the condition that no bucket has more than one item.

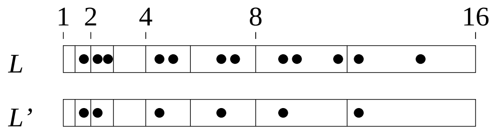


Figure 73: Trimming Lists for Approximate Subset Sum.

Claim: The number of distinct items in a trimmed list is $O((n \log t)/\epsilon)$, which is polynomial in input size and $1/\epsilon$.

Proof: We know that each pair of consecutive elements in a trimmed list differ by a ratio of at least $d = 1/(1-\delta) > 1$. Let k denote the number of elements in the trimmed list, ignoring the element of value 0. Thus, the smallest nonzero value and maximum value in the the trimmed list differ by a ratio of at least d^{k-1}. Since the smallest (nonzero) element is at least as large as 1, and the largest is no larger than t, then it follows that $d^{k-1} \leq t/1 = t$. Taking the natural log of both sides we have $(k-1)\ln d \leq \ln t$. Using the facts that $\delta = \epsilon/n$ and the log identity that $\ln(1+x) \leq x$, we have

$$\begin{aligned} k - 1 &\leq \frac{\ln t}{\ln d} = \frac{\ln t}{-\ln(1-\delta)} \\ &\leq \frac{\ln t}{\delta} = \frac{n \ln t}{\epsilon} \\ k &= O\left(\frac{n \log t}{\epsilon}\right). \end{aligned}$$

Observe that the input size is at least as large as n (since there are n numbers) and at least as large as $\log t$ (since it takes $\log t$ digits to write down t on the input). Thus, this function is polynomial in the input size and $1/\epsilon$.

The approximation algorithm operates as before, but in addition we call the procedure `Trim` given below.

Approximate Subset Sum

```
Trim(L, delta) {
    let the elements of L be denoted y[1..m];
    L' = <y[1]>;                        // start with first item
    last = y[1];                        // last item to be added
    for i = 2 to m do {
        if (last < (1-delta) y[i]) {     // different enough?
            append y[i] to end of L';
            last = y[i];
        }
    }
}

Approx_SS(x[1..n], t, eps) {
    delta = eps/n;                      // approx factor
    L = <0>;                            // empty sum = 0
    for i = 1 to n do {
        L = MergeLists(L, L+x[i]);      // add in next item
        L = Trim(L, delta);             // trim away "near" duplicates
        remove for L all elements greater than t;
    }
    return largest element in L;
}
```

For example, consider the set $S = \{104, 102, 201, 101\}$ and $t = 308$ and $\epsilon = 0.20$. We have $\delta = \epsilon/4 = 0.05$. Here is a summary of the algorithm's execution.

$$
\begin{array}{rrcl}
\text{init:} & L_0 & = & \langle 0 \rangle \\
\\
\text{merge:} & L_1 & = & \langle 0, 104 \rangle \\
\text{trim:} & L_1 & = & \langle 0, 104 \rangle \\
\text{remove:} & L_1 & = & \langle 0, 104 \rangle \\
\\
\text{merge:} & L_2 & = & \langle 0, 102, 104, 206 \rangle \\
\text{trim:} & L_2 & = & \langle 0, 102, 206 \rangle \\
\text{remove:} & L_2 & = & \langle 0, 102, 206 \rangle \\
\\
\text{merge:} & L_3 & = & \langle 0, 102, 201, 206, 303, 407 \rangle \\
\text{trim:} & L_3 & = & \langle 0, 102, 201, 303, 407 \rangle \\
\text{remove:} & L_3 & = & \langle 0, 102, 201, 303 \rangle
\end{array}
$$

$$\begin{array}{rrcl} \text{merge:} & L_4 & = & \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle \\ \text{trim:} & L_4 & = & \langle 0, 101, 201, 302, 404 \rangle \\ \text{remove:} & L_4 & = & \langle 0, 101, 201, 302 \rangle \end{array}$$

The final output is 302. The optimum is $307 = 104 + 102 + 101$. So our actual relative error in this case is within 2%.

The running time of the procedure is $O(n|L|)$ which is $O(n^2 \ln t/\epsilon)$ by the earlier claim.

Approximation Analysis: The final question is why the algorithm achieves an relative error of at most ϵ over the optimum solution. Let Y^* denote the optimum (largest) subset sum and let Y denote the value returned by the algorithm. We want to show that Y is not too much smaller than Y^*, that is,

$$Y \geq Y^*(1 - \epsilon).$$

Our proof will make use of an important inequality from real analysis.

Lemma: For $n > 0$ and a real numbers,

$$(1 + a) \leq \left(1 + \frac{a}{n}\right)^n \leq e^a.$$

Recall that our intuition was that we would allow a relative error of ϵ/n at each stage of the algorithm. Since the algorithm has n stages, then the total relative error should be (obviously?) $n(\epsilon/n) = \epsilon$. The catch is that these are relative, not absolute errors. These errors to not accumulate additively, but rather by multiplication. So we need to be more careful.

Let L_i^* denote the i-th list in the exponential time (optimal) solution and let L_i denote the i-th list in the approximate algorithm. We claim that for each $y \in L_i^*$ there exists a representative item $z \in L_i$ whose relative error from y that satisfies

$$(1 - \epsilon/n)^i y \leq z \leq y.$$

The proof of the claim is by induction on i. Initially $L_0 = L_0^* = \langle 0 \rangle$, and so there is no error. Suppose by induction that the above equation holds for each item in L_{i-1}^*. Consider an element $y \in L_{i-1}^*$. We know that y will generate two elements in L_i^*: y and $y + x_i$. We want to argue that there will be a representative that is "close" to each of these items.

By our induction hypothesis, there is a representative element z in L_{i-1} such that

$$(1 - \epsilon/n)^{i-1} y \leq z \leq y.$$

When we apply our algorithm, we will form two new items to add (initially) to L_i: z and $z + x_i$. Observe that by adding x_i to the inequality above and a little simplification we get

$$(1 - \epsilon/n)^{i-1}(y + x_i) \leq z + x_i \leq y + x_i.$$

The items z and $z + x_i$ might not appear in L_i because they may be trimmed. Let z' and z'' be their respective representatives. Thus, z' and z'' are elements of L_i. We have

$$\begin{array}{rcl} (1 - \epsilon/n)z & \leq z' \leq & z \\ (1 - \epsilon/n)(z + x_i) & \leq z'' \leq & z + x_i. \end{array}$$

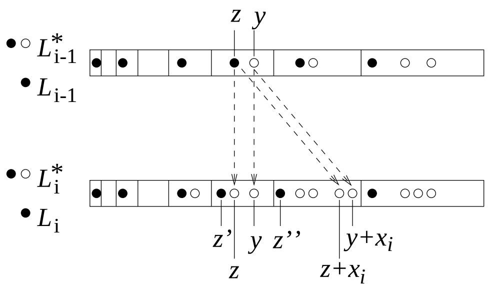


Figure 74: Subset sum approximation analysis.

Combining these with the inequalities above we have

$$\begin{array}{rcll} (1-\epsilon/n)^{i-1}(1-\epsilon/n)y & \leq & (1-\epsilon/n)^i y & \leq z' \leq \quad y \\ (1-\epsilon/n)^{i-1}(1-\epsilon/n)(y+x_i) & \leq & (1-\epsilon/n)^i (y+x_i) & \leq z'' \leq \quad z+y_i. \end{array}$$

Since z and z'' are in L_i this is the desired result. This ends the proof of the claim.

Using our claim, and the fact that Y^* (the optimum answer) is the largest element of L_n^* and Y (the approximate answer) is the largest element of L_n we have

$$(1-\epsilon/n)^n Y^* \leq Y \leq Y^*.$$

This is not quite what we wanted. We wanted to show that $(1-\epsilon)Y^* \leq Y$. To complete the proof, we observe from the lemma above (setting $a = -\epsilon$) that

$$(1-\epsilon) \leq \left(1 - \frac{\epsilon}{n}\right)^n.$$

This completes the approximate analysis.