CS170: Efficient Algorithms and Intractable Problems
Fall 2001

Luca Trevisan

These is the complete set of lectures notes for the Fall 2001 offering of CS170. The notes are, mostly, a revised version of notes used in past semesters. I believe there may be several errors and imprecisions.

Berkeley, November 30, 2001.

Luca Trevisan

Contents

Chapter 1

Introduction

1.1 Topics to be covered

1. Data structures and graph algorithms—Quick review of prerequisites.
2. *Divide-and-conquer algorithms* work by dividing problem into two smaller parts and combining their solutions. They are natural to implement by recursion, or a stack (e.g. binary search; Fast Fourier Transform in signal processing, speech recognition)
3. *Graph algorithms*: depth-first search (DFS), breadth-first search (BFS) on graphs, with properties and applications (e.g. finding shortest paths, as in trip planning)
4. *Randomization*: how to use probability theory to come up with fast and elegant algorithms and data structures.
5. *Data Compression*: the theory and the algorithms.
6. *Dynamic programming*: works by solving subproblems, like in divide-and-conquer, but differently.
7. *Linear Programming* (LP) solves systems of linear equations and inequalities. We will concentrate on using LP to solve other problems, (e.g. production scheduling, computing capacities of networks)
8. *NP-completeness*: many algorithms we will study are quite efficient, costing $O(n)$ or $O(n \log n)$ or $O(n^2)$ on input size n. But some problems are much harder, with algorithms costing at least $\Omega(2^n)$ (or some other *exponential*, rather than *polynomial* function of n). It is believed that we will *never* be able to solve such problem completely for large n, e.g. TSP. So we are forced to approximate them. We will learn to recognize when a problem is so hard (NP-complete), and how to devise algorithms that provide approximate if not necessarily optimal solutions.
9. The Fast Fourier Transform (or FFT) is widely used in signal processing (including speech recognition and image compression) as well as in scientific and engineering applications.

1.2 On Algorithms

An algorithm is a recipe or a well-defined procedure for transforming some input into a desired output. Perhaps the most familiar algorithms are those those for adding and multiplying integers. Here is a multiplication algorithm that is different the algorithm you learned in school: write the multiplier and multiplicand side by side, say 13x15=195. Repeat the following operations—divide the first number by 2 (throw out any fractions) and multiply the second by 2, until the first number is 1. This results in two columns of numbers. Now cross out all rows in which the first entry is even, and add all entries of the second column that haven't been crossed out. The result is the product of the two numbers.

In this course we will ask a number of basic questions about algorithms:

- Does it halt?

  The answer for the algorithm given above is clearly yes, *provided* we are multiplying positive integers. The reason is that for any integer greater than 1, when we divide it by 2 and throw out the fractional part, we always get a smaller integer which is greater than or equal to 1.

- Is it correct?

  To see that the algorithm correctly computes the product of the integers, observe that if we write a 0 for each crossed out row, and 1 for each row that is not crossed out, then reading from bottom to top just gives us the first number in binary. Therefore, the algorithm is just doing the standard multiplication in binary, and is in fact essentially the algorithm used in computers, which represent numbers in binary.

- How much time does it take?

  It turns out that the algorithm is as fast as the standard algorithm. (How do we implement the division step, which seems harder than multiplication, in a computer?) In particular, if the input numbers are n digits long, the algorithm takes $O(n^2)$ operations. Later in the course, we will study a faster algorithm for multiplying integers.

- How much memory does it use? (When we study cache-aware algorithms, we will ask more details about what kinds of memory are used, eg cache, main memory, etc.)

The history of algorithms for simple arithmetic is quite fascinating. Although we take them for granted, their widespread use is surprisingly recent. The key to good algorithms for arithmetic was the positional number system (such as the decimal system). Roman numerals (I, II, III, IV, V, VI, etc) were just the wrong data structure for performing arithmetic efficiently. The positional number system was first invented by the Mayan Indians in Central America about 2000 years ago. They used a base 20 system, and it is unknown whether they had invented algorithms for performing arithmetic, since the Spanish conquerors destroyed most of the Mayan books on science and astronomy.

The decimal system that we use today was invented in India in roughly 600 AD. This positional number system, together with algorithms for performing arithmetic were transmitted to Persia around 750 AD, when several important Indian works were translated into arabic. In particular, it was around this time that the Persian mathematician Al-Khwarizmi wrote his arabic textbook on the subject. The word "algorithm" comes from Al-Khwarizmi's name. Al-Khwarizmi's work was translated into Latin around 1200 AD, and the positional number system was propagated throughout Europe from 1200 to 1600 AD.

The decimal point was not invented until the 10^{th} century AD, by a Syrian mathematician al-Uqlidisi from Damascus. His work was soon forgotten, and five centuries passed before decimal fractions were re-invented by the Persian mathematician al-Kashi.

With the invention of computers in this century, the field of algorithms has seen explosive growth. There are a number of major successes in this field: not all of which we can discuss in CS 170:

- Parsing algorithms—these form the basis of the field of programming languages (CS 164)
- Fast Fourier transform—the field of digital signal processing is built upon this algorithm. (CS 170, EE)
- Linear programming—this algorithm is extensively used in resource scheduling. (CS 170, IEOR)
- Sorting algorithms - until recently, sorting used up the bulk of computer cycles. (CS 61B, CS 170)
- String matching algorithms—these are extensively used in computational biology. (CS 170)
- Number theoretic algorithms—these algorithms make it possible to implement cryptosystems such as the RSA public key cryptosystem. (CS 276)
- Numerical algorithms, for evaluating models of physical systems, replacing the need for expensive or impossible physical experiments (e.g. climate modeling, earthquake modeling, building the Boeing 777) (Ma 128ab, and many engineering and science courses)
- Geometric algorithms, for computing and answering questions about physical shapes that appear in computer graphics and models of physical systems.

The algorithms we discuss will be in "pseudo-code", so we will not worry about certain details of implementation. So we might say "choose the i-th element of a list" without worrying about exactly how the list is represented. But the efficiency of an algorithm can depend critically on the right choice of data structure (e.g. $O(1)$ for an array versus $O(i)$ for a linked list). So we will have a few programming assignments where success will depend critically on the correct choice of data structure.

1.3 Computing the nth Fibonacci Number

Remember the famous sequence of numbers invented in the 15th century by the Italian mathematician Leonardo Fibonacci? The sequence is represented as $F_0, F_1, F_2, \ldots$, where $F_0 = F_1 = 1$, and for all $n \geq 2$ F_n is defined as $F_{n-1} + F_{n-2}$. The first few numbers are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ... F_{30} is greater than a million! The Fibonacci numbers grow exponentially. Not quite as fast as 2^n, but close: F_n is about $2^{.694...n}$.

Suppose we want to compute the number F_n, for some given large integer n. Our first algorithm is the one that slavishly implements the definition:

function $F(n : \text{integer})$: integer
if $n \leq 1$ then return 1
else return $F(n-1) + F(n-2)$

It is obviously correct, and always halts. The problem is, it is too slow. Since it is a recursive algorithm, we can express its running time on input n, $T(n)$, by a *recurrence equation*, in terms of smaller values of T (we shall talk a lot about such recurrences later in this class). So, what is $T(n)$? We shall be interested in the *order of growth* of $T(n)$, ignoring constants. If $n \leq 1$, then we do constant amount of work to obtain F_n; since we are suppressing constants, we can write $T(n) = 1$. Otherwise, if $n \geq 2$, we have:

$$T(n) = T(n-1) + T(n-2) + 1,$$

because in this case the running time of F on input n is just the running time of F on input $n-1$—which is $T(n-1)$—plus $T(n-2)$, plus the time for the addition. This is nearly the Fibonacci equation. We can change it to be exactly the Fibonacci equation by noting that the new variable $T'(n) \equiv T(n) + 1$ satisfies $T'(n) = T'(n-1) + T'(n-2)$, $T'(1) = T(1) + 1 = 2$ and $T'(0) = T(0) + 1 = 2$. Comparing this recurrence for $T'(n)$ to the Fibonacci recurrence, we see $T'(n) = 2F_n$ and so $T(n) = 2F_n - 1$. In other words, the running time of our Fibonacci program *grows as the Fibonacci numbers*—that is to say, way too fast.

Can we do better? This is the question we shall always ask of our algorithms. The trouble with the naive algorithm is wasteful use of recursion: The function F is called with the same argument over an over again, exponentially many times (try to see how many times $F(1)$ is called in the computation of $F(5)$). A simple trick for improving this performance is to *memoize* the recursive algorithm, by remembering the results from previous calls. Specifically, we can maintain an array $A[0 \ldots n]$, initially all zero, except that $A[0] = A[1] = 1$. This array is updated before the return step of the F function, which now becomes $A[n] := A[n-1] + A[n-2]$; return $A[n]$. More importantly, *it is consulted in the beginning* of the F function, and, if its value is found to be non-zero—that is to say, defined— it is immediately returned. But then of course, there would be little point in keeping the recursive structure of the algorithm, we could just write:

function $F(n : \text{integer})$: integer
array $A[1 \ldots n]$ of integers, initially all 0
$A[0] := A[1] := 1$
for $i = 2$ to n do:
$A[n] := A[n-1] + A[n-2]$
return $A[n]$

This algorithm is correct, because it is just another way of implementing the definition of Fibonacci numbers. The point is that its running time is now much better. We have a single "for" loop, executed $n-1$ times. And the body of the loop takes only constant time (one addition, one assignment). Hence, the algorithm takes only $O(n)$ operations.

This is usually an important moment in the study of a problem: We have come from the naive but prohibitively exponential algorithm to a polynomial one.

Now, let us be more precise in our accounting of the time requirements for all these methods, we have made a grave and common error: We have been too liberal about *what constitutes an elementary step of the algorithm.* In general, in analyzing our algorithms we shall assume that each arithmetic step takes unit time, because the numbers involved will be typically small enough—about n, the size of the problem— that we can reasonably expect them to fit within a computer's word. But in the present case, we are doing arithmetic on huge numbers, with about n bits—remember, a Fibonacci number has about $.694\ldots n$ bits—and of course we are interested in the case n is truly large. When dealing with such huge numbers, and if exact computation is required, we have to use sophisticated *long integer packages.* Such algorithms take $O(n)$ time to add two n-bit numbers—hence the complexity of the two methods was not really $O(F_n)$ and $O(n)$, but instead $O(nF_n)$ and $O(n^2)$, respectively; notice that the latter is still exponentially faster.

1.4 Algorithm Design Paradigms

Every algorithmic situation (every computational problem) is different, and there are no hard and fast algorithm design rules that work all the time. But the vast majority of algorithms can be categorized with respect to the *concept of progress* they utilize.

In an algorithm you want to make sure that, after each execution of a loop, or after each recursive call, some kind of *progress* has been made. In the absence of such assurance, you cannot be sure that your algorithm will even terminate. For example, in depth-first search, after each recursive call you know that another node has been visited. And in Dijkstra's algorithm you know that, after each execution of the main loop, the correct distance to another node has been found. And so on.

Divide-and-Conquer Algorithms: These algorithms have the following outline: To solve a problem, divide it into subproblems. Recursively solve the subproblems. Finally glue the resulting solutions together to obtain the solution to the original problem. Progress here is measured by how much smaller the subproblems are compared to the original problem.

You have already seen an example of a divide and conquer algorithm in cs61B: mergesort. The idea behind mergesort is to take a list, *divide* it into two smaller sublists, *conquer* each sublist by sorting it, and then *combine* the two solutions for the subproblems into a single solution. These three basic steps—divide, conquer, and combine—lie behind most divide and conquer algorithms.

With mergesort, we kept dividing the list into halves until there was just one element left. In general, we may divide the problem into smaller problems in any convenient fashion. Also, in practice it may not be best to keep dividing until the instances are completely trivial. Instead, it may be wise to divide until the instances are reasonably small, and then apply an algorithm that is faster on small instances. For example, with mergesort, it might be best to divide lists until there are only four elements, and then sort these small lists quickly by other means. We will consider these issues in some of the applications we consider.

1.5 Maximum/minimum

Suppose we wish to find the minimum *and* maximum items in a list of numbers. How many comparisons does it take? The obvious loop takes $2n - 2$:

```
m = L[1]; M = L[1];
for i=2, length(L)
   m = min(m, L[i])
   M = max(M, L[i])
```

Can we do it in fewer? A natural approach is to try a divide and conquer algorithm. Split the list into two sublists of equal size. (Assume that the initial list size is a power of two.) Find the maxima and minimum of the sublists. Two more comparisons then suffice to find the maximum and minimum of the list.

```
function [m,M] = MinMax(L)
  if (length(L) < 3)
     if (L[1] < L[length(L))
        return [ L[1], L[length(L)] ]
     else
        return [ L[length(L)], L[1] ]
  else
     [ m1, M1 ] = MinMax( L[ 1 : n/2 ] )    ... find min, max of first half of L
     [ m2, M2 ] = MinMax( L[ n/2+1 : n ] )  ... find min, max of last  half of L
     return [ min(m1,m2) , max(M1,M2) ]
```

Hence, if $T(n)$ is the number of comparisons, then $T(n) = 2T(n/2) + 2$ and $T(2) = 1$.

How do we solve such a recursion? Suppose that we "guess" that the solution is of the form $T(n) = an + b$. Then a and b have to satisfy the system of equations

$$\begin{cases} 2a + b = 1 \\ an + b = 2\left(a\frac{n}{2} + b\right) + 2 \end{cases}$$

which solves to $b = -2$ and $a = 3/2$, so that that (for n a power of 2), $T(n) = 3n/2 - 2$, or 75% of the obvious algorithm. One can in fact show that $\lceil 3n/2 \rceil - 2$ comparisons are *necessary* to find both the minimum and maximum (CLR Problem 10.1-2).

A similar idea can be applied to a sequential algorithm. Suppose for simplicity that the array has an odd number of entries, then the algorithm is as follows:

```
function [m,M] = MinMax(L)
  curr_min = min(L[1],L[2])
  curr_max = max(L[1],L[2])
  for (i=3,i< length(L),i=i+2)
     m = min(L[i],L[i+1])
     M = max(L[i],L[i+1])
     if m < curr_min
```

```
            curr_min = m
         if M > curr_max
            curr_max = M
      return [ curr_min , curr_max ]
```

Each iteration of the `for` loop requires three comparisons, and the initialization of the variables requires one comparison. Since the loop is repeated $(n-2)/2$ times, we have a total of $3n/2-2$ comparisons, for even n.

1.6 Integer Multiplication

The standard multiplication algorithm takes time $\Theta(n^2)$ to multiply together two n digit numbers. This algorithm is so natural that we may think that no algorithm could be better. Here, we will show that better algorithms exist (at least in terms of asymptotic behavior).

Imagine splitting each number x and y into two parts: $x = 2^{n/2}a+b, y = 2^{n/2}c+d$. Then:

$$xy = 2^n ac + 2^{n/2}(ad+bc) + bd.$$

The additions and the multiplications by powers of 2 (which are just shifts!) can all be done in linear time. So the algorithm is

```
function Mul(x,y)
   n = common bit length of x and y
   if n small enough
      return x*y
   else
      write x = 2^(n/2)a+b and y = 2^(n/2)*c+d       ... no cost for this
      p1 = Mul(a,c)
      p2 = Mul(a,d) + Mul(b,c)
      p3 = Mul(b,d)
      return 2^n*p1 + 2^(n/2)*p2 + p3
   end
```

We have therefore reduced our multiplication into four smaller multiplications problems, so the recurrence for the time $T(n)$ to multiply two n-digit numbers becomes

$$T(n) = 4T(n/2) + \Theta(n).$$

Unfortunately, when we solve this recurrence, the running time is still $\Theta(n^2)$, so it seems that we have not gained anything. (Again, we explain the solution of this recurrence later.)

The key thing to notice here is that four multiplications is too many. Can we somehow reduce it to three? It may not look like it is possible, but it is using a simple trick. The trick is that *we do not need to compute ad and bc separately; we only need their sum $ad+bc$.* Now note that

$$(a+b)(c+d) = (ad+bc) + (ac+bd).$$

So if we calculate ac,bd, and $(a+b)(c+d)$, we can compute $ad+bc$ by the subtracting the first two terms from the third! Of course, we have to perform more additions, but since the bottleneck to speeding up the algorithm was the number of multiplications, that does not matter. This means the 3 lines computing `p1`, `p2`, and `p3` above should be replaced by

```
p1 = Mul(a,c)
p3 = Mul(b,d)
p2 = Mul(a+b,c+d) - p1 - p3
```

The recurrence for $T(n)$ is now

$$T(n) = 3T(n/2) + \Theta(n),$$

and we find that $T(n)$ is $\Theta(n^{\log_2 3})$ or approximately $\Theta(n^{1.59})$, improving on the quadratic algorithm (later we will show how we solved this recurrence).

If one were to implement this algorithm, it would probably be best not to divide the numbers down to one digit. The conventional algorithm, because it uses fewer additions, is more efficient for small values of n. Moreover, on a computer, there would be no reason to continue dividing once the length n is so small that the multiplication can be done in one standard machine multiplication operation!

It also turns out that using a more complicated algorithm (based on a similar idea, but with each integer divided into many more parts) the asymptotic time for multiplication can be made very close to linear – $O(n \cdot \log n \cdot \log\log n)$ (Schönhage and Strassen, 1971). Note that this can also be written $O(n^{1+\epsilon})$ for any $\epsilon > 0$.

1.7 Solving Divide-and Conquer Recurrences

A typical divide-and-conquer recurrence is of the following form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n),$$

where we assume that $T(1) = O(1)$. This corresponds to a recursive algorithm that in order to solve an instance of size n, generates and recursively solves a instances of size n/b, and then takes $f(n)$ time to recover a solution for the original instance from the solutions for the smaller instances. For simplicity in the analysis, we further assume that n is a power of b: $n = b^m$, where of course m is $\log_b n$. This allows us to sweep under the rug the question "what happens if n is not exactly divisible by b" either in the first or subsequent recursive calls. We can always "round n up" to the next power of b to make sure this is the case (but, of course, in a practical implementation we would have to take care of the case in which the problem would have to be split in slightly unbalanced subproblems).

Certainly, we have $T(n) \geq f(n)$, since $f(n)$ is just the time that it takes to "combine" the solutions at the top level of the recursion.

Let us now consider the number of distinct computations that are recursively generated. At each level that we go down the recursion tree, the size of instance is divided by b, so the recursion tree has m levels, where $m = \log_b n$. For each instance in the recursion tree, a

instances are generated at the lower level, which means that $a^m = a^{\log_b n} = n^{\log_b a}$ instances of size 1 are generated while solving an instance of size n. Clearly, $n^{\log_b a}$ is also a lower bound for the running time $T(n)$.

Indeed, in most cases, $n^{\log_b a}$ or $f(n)$ can also be tight upper bound (up to a multiplicative constant) for $T(n)$.

Specifically:

- If there is an $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a})$.
- If there is an $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, then $T(n) = \Theta(f(n))$.

CLR calls this the "Master Theorem" (section 4.3 of CLR). The Master theorem applies if $f(n)$ are exactly of the same order of magnitude, or if their ratio grows at least like n^ϵ, for some $\epsilon > 0$, but there are possible values for a, b and $f(n)$ such that neither case applies.

Chapter 2

Graph Algorithms

2.1 Introduction

Sections 5.3 and 5.4 of CLR, which you should read, introduce a lot of standard notation for talking about graphs and trees. For example a graph $G = G(V, E)$ is defined by the finite set V of *vertices* and the finite set E of *edges* (u, v) (which connect pairs of vertices u and v). The edges may be *directed or undirected* (depending on whether the edge points from one vertex to the other or not). We will draw graphs with the vertices as dots and the edges as arcs connecting them (with arrows if the graph is directed). We will use terms like *adjacency*, *neighbors*, *paths* (directed or undirected), *cycles*, etc. to mean the obvious things. We will define other notation as we introduce it.

Graphs are useful for modeling a diverse number of situations. For example, the vertices of a graph might represent cities, and edges might represent highways that connect them. In this case, each edge might also have an associated length. Alternatively, an edge might represent a flight from one city to another, and each edge might have a weight which represents the cost of the flight. The typical problem in this context is computing shortest paths: given that you wish to travel from city X to city Y, what is the shortest path (or the cheapest flight schedule). There are very efficient algorithms for solving these problems. A different kind of problem—the traveling salesman problem—is very hard to solve. Suppose a traveling salesman wishes to visit each city exactly once and return to his starting point, in which order should he visit the cities to minimize the total distance traveled? This is an example of an NP-complete problem, and one we will study later in this course.

A different context in which graphs play a critical modeling role is in networks of pipes or communication links. These can, in general, be modeled by directed graphs with capacities on the edges. A directed edge from u to v with capacity c might represent a pipeline that can carry a flow of at most c units of oil per unit time from u to v. A typical problem in this context is the max-flow problem: given a network of pipes modeled by a directed graph with capacities on the edges, and two special vertices—a source s and a sink t—what is the maximum rate at which oil can be transported from s to t over this network of pipes? There are ingenious techniques for solving these types of flow problems, and we shall see some of them later in this course.

In all the cases mentioned above, the vertices and edges of the graph represented something quite concrete such as cities and highways. Often, graphs will be used to represent more abstract relationships. For example, the vertices of a graph might represent tasks, and the edges might represent precedence constraints: a directed edge from u to v says that task u must be completed before v can be started. An important problem in this context is scheduling: in what order should the tasks be scheduled so that all the precedence constraints are satisfied. There is a very fast algorithm for solving this problem that we will see shortly.

We usually denote the number of nodes of a graph by $|V| = n$, and its number of edges by $|E| = e$. e is always less than or equal to n^2, since this is the number of all ordered pairs of n nodes—for undirected graphs, this upper bound is $\frac{n(n-1)}{2}$. On the other hand, it is reasonable to assume that e is not much smaller than n; for example, if the graph has no *isolated nodes*, that is, if each node has at least one edge into or out of it, then $e \geq \frac{n}{2}$. Hence, e ranges roughly between n and n^2. Graphs that have about n^2 edges—that is, nearly as many as they could possibly have—are called *dense*. Graphs with far fewer than n^2 edges are called *sparse*. Planar graphs (graphs that can be drawn on a sheet of paper without crossings of edges) are always sparse, having $O(n)$ edges.

2.2 Reasoning about Graphs

We will do a few problems to show how to reason about graphs.

Handshaking Lemma: Let $G = (V, E)$ be an undirected graph. Let degree(v) be the degree of vertex v, i.e. the number of edges incident on v, Then $\sum_{v \in V} \text{degree}(v) = 2e$.

Proof: Let E_v be the list of edges adjacent to vertex v, so degree(v) $= |E_v|$. Then $\sum_{v \in V} \text{degree}(v) = \sum_{v \in V} |E_v|$. Since every edge (u, v) appears in both E_u and E_v, each edge is counted exactly twice in $\sum_{v \in V} |E_v|$, which must then equal $2e$.

A *free tree* is a connected, acyclic undirected graph. A *rooted tree* is a free tree where one vertex is identified as the *root*.

Corollary: In a tree (free or rooted), $e = n - 1$.

Proof: If the tree is free, pick any vertex to be the root, so we can identify the *parent* and *children* of any vertex. Every vertex in the tree except the root is a child of another vertex, and is connected by an edge to its parent. Thus e is the number of children in the tree, or $n - 1$, since all vertices except the root are children.

In fact one can show that if $G = (V, E)$ is undirected, then it is a free tree if and only if it is connected (or acyclic) and $e = n - 1$ (see Theorem 5.2 in CLR).

2.3 Data Structures for Graphs

One common representation for a graph $G(V, E)$ is the *adjacency matrix*. Suppose $V = \{1, \cdots, n\}$. The adjacency matrix for $G(V, E)$ is an $n \times n$ matrix A, where $a_{i,j} = 1$ if $(i, j) \in E$ and $a_{i,j} = 0$ otherwise. The advantage of the adjacency matrix representation is that it takes constant time (just one memory access) to determine whether or not there is an edge between any two given vertices. In the case that each edge has an associated length or a weight, the adjacency matrix representation can be appropriately modified so entry $a_{i,j}$ contains that length or weight instead of just a 1. The most important disadvantage of the adjacency matrix representation is that it requires n^2 storage, even if the graph is very sparse, having as few as $O(n)$ edges. Moreover, just examining all the entries of the matrix would require n^2 steps, thus precluding the possibility of algorithms for sparse graphs that run in linear ($O(e)$) time.

The *adjacency list* representation avoids these disadvantages. The adjacency list for a vertex i is just a list of all the vertices adjacent to i (in any order). In the adjacency list representation, we have an array of size n to represent the vertices of the graph, and the i^{th} element of the array points to the adjacency list of the i^{th} vertex. The total storage used by an adjacency list representation of a graph with n vertices and e edges is $O(n+e)$. We will use this representation for all our graph algorithms that take linear or near linear time. Using the adjacency lists representation, it is easy to iterate through all edges going out of a particular vertex v—and this is a very useful kind of iteration in graph algorithms (see for example the depth-first search procedure in a later section); with an adjacency matrix, this would take n steps, even if there were only a few edges going out of v. One potential disadvantage of adjacency lists is that determining whether there is an edge from vertex i to vertex j may take as many as n steps, since there is no systematic shortcut to scanning the adjacency list of vertex i.

So far we have discussed how to represent directed graphs on the computer. To represent undirected graphs, just remember that an undirected graph is just a special case of directed graph: one that is symmetric and has no loops. The adjacency matrix of an undirected graph has the additional property that $a_{ij} = a_{ji}$ for all vertices i and j. This means the adjacency matrix is symmetric.

What data structure(s) would you use if you wanted the advantages of both adjacency lists and adjacency matrices? That is, you wanted to loop though all $k = \text{degree}(v)$ edges adjacent to v in $O(k)$ time, determine if edge (u, v) exists in $O(1)$ time on the average for any pair of vertices u and v, and use $O(n+e)$ total storage?

Chapter 3

Depth First Search

3.1 Depth-First Search

We will start by studying two fundamental algorithms for searching a graph: *depth-first search* and *breadth-first search.* To better understand the need for these algorithms, let us imagine the computer's view of a graph that has been input into it: it can examine each of the edges adjacent to a vertex in turn, by traversing its adjacency list; it can also mark vertices as "visited." This corresponds to exploring a dark maze with a flashlight and a piece of chalk. You are allowed to illuminate any corridor of the maze emanating from the current room, and you are also allowed to use the chalk to mark the current room as having been "visited." Clearly, it would not be easy to find your way around without the use of any additional data structures.

Mythology tells us that the right data structure for exploring a maze is a ball of string. Depth-first search is a technique for exploring a graph using as a basic data structure a *stack*—the cyberanalog of a ball of string. It is not hard to visualize why a stack is the right way to implement a ball of string in a computer. A ball of string allows two primitive steps: *unwind* to get into a new room (the stack analog is *push the new room*) and *rewind* to return to the previous room—the stack analog is *pop*.

Actually, we shall present depth-first search not as an algorithm explicitly manipulating a stack, but as a *recursive* procedure, using the stack of activation records provided by the programming language. We start by defining a recursive procedure `explore`. When `explore` is invoked on a vertex v, it explores all previously unexplored vertices that are reachable from v.

```
procedure explore(v:  vertex)
visited(v) := true
previsit(v)
for each edge (v,w) out of v do
  {if not visited(w) then explore(w)}
postvisit(v)

algorithm dfs(G = (V,E): graph)
for each v in V do {visited(v) := false}
for each v in V do
```

```
10.     {if not visited(v) then explore(v)}}
```

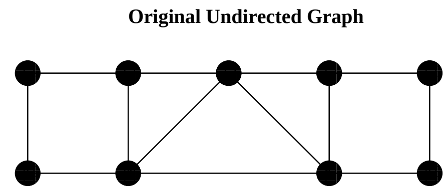
`previsit(v)` and `postvisit(v)` are two routines that perform at most some constant number of operations on the vertex v. As we shall see, by varying these routines depth-first search can be tuned to accomplish a wide range of important and sophisticated tasks.

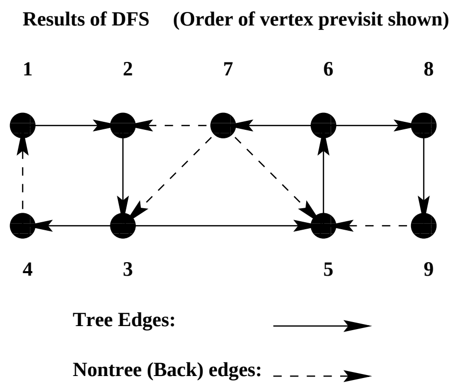
Depth-first search takes $O(n+e)$ steps on a graph with n vertices and e edges, because the algorithm performs *at most some constant number of steps per each vertex and edge of the graph.* To see this clearly, let us go through the exercise of "charging" each operation of the algorithm to a particular vertex or edge of the graph. The procedure `explore` (lines 1 through 6) is called at most once—in fact, *exactly* once—for each vertex; the boolean variable `visited` makes sure each vertex is visited only once. The finite amount of work required to perform the call `explore(v)` (such as adding a new activation record on the stack and popping it in the end) is charged to the vertex v. Similarly, the finite amount of work in the routines `previsit(v)` and `postvisit(v)` (lines 3 and 6) is charged to the vertex v. Finally, the work done for each outgoing edge (v,w) (looking it up in the adjacency list of v and checking whether v has been visited, lines 4 and 5) is charged to that edge. Each edge (v,w) is processed only once, because its tail (also called source) v is visited only once—in the case of an undirected graph, each edge will be visited twice, once in each direction. Hence, every element (vertex or edge) of the graph will end up being charged at most some constant number of elementary operations. Since this accounts for all the work performed by the algorithm, it follows that depth-first search takes $O(n+e)$ work—it is a *linear-time algorithm.*

This is the fastest possible running time of any algorithm solving a nontrivial problem, since any decent problem will require that each data item be inspected at least once.[1] It means that a graph can be searched in time comparable to the time it takes to read it into the memory of a computer!

Here is an example of DFS on an undirected graph, with vertices labeled by the order in which they are first visited by the DFS algorithm. DFS defines a tree in a natural way: each time a new vertex, say w, is discovered, we can incorporate w into the tree by connecting w to the vertex v it was discovered from via the edge (v,w). If the graph is disconnected, and thus depth-first search has to be restarted, a separate tree is created for each restart. The edges of the depth-first search tree(s) are called *tree edges;* they are the edges through which the control of the algorithm flows. The remaining edges, shown as broken lines, go from a vertex of the tree to an *ancestor* of the vertex, and are therefore called *back edges.* If there is more than one tree, the collection of all of them is called a *forest.*

[1] There is an exception to this, namely *database queries*, which can often be answered without examining the whole database (for example, by binary search). Such procedures, however, require data that have been preprocessed and structured in a particular way, and so cannot be fairly compared with algorithms, such as depth-first search, which must work on unpreprocessed inputs.





By modifying the procedures `previsit` and `postvisit`, we can use depth-first search to compute and store useful information and solve a number of important problems. The simplest such modification is to record the "time" each vertex is first seen by the algorithm, i.e. produce the labels on the vertices in the above picture. We do this by keeping a counter (or *clock*), and assigning to each vertex the "time" `previsit` was executed (`postvisit` does nothing). This would correspond to the following code:

```
procedure previsit(v:  vertex)
pre(v) := clock++
```

Naturally, `clock` will have to be initialized to zero in the beginning of the main algorithm. If we think of depth-first search as using an explicit stack, pushing the vertex being visited next on top of the stack, then `pre(v)` is the order in which vertex v is first pushed on the stack. The contents of the stack at any time yield a path from the root to some vertex in the depth first search tree. Thus, the tree summarizes the history of the stack during the algorithm.

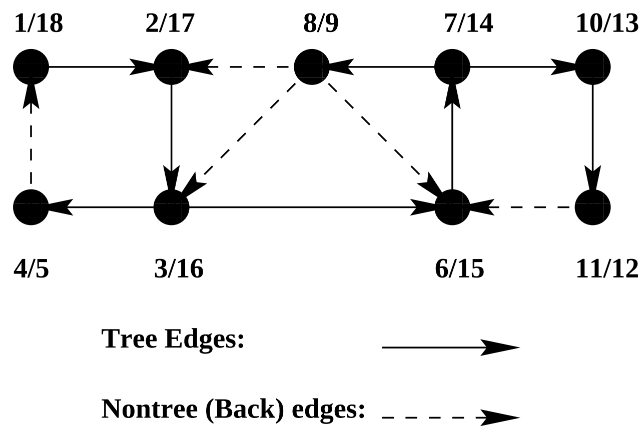
More generally, we can modify both `previsit` and `postvisit` to increment `clock` and record the time vertex v was first visited, or pushed on the stack (`pre(v)`), and the time vertex v was last visited, or popped from the stack (`post(v)`):

```
procedure previsit(v:  vertex)
pre(v) := clock++

procedure postvisit(v:  vertex)
post(v) := clock++
```

Let us rerun depth-first search on the undirected graph shown above. Next to each vertex v we showing the numbers `pre(v)/post(v)` computed as in the last code segment, next to each vertex v.

Results of DFS (pre(v)/post(v) shown next to each vertex x)



How would the nodes be numbered if `previsit()` did not do anything?

Notice the following important property of the numbers `pre(v)` and `post(v)` (when both are computed by `previsit` and `postvisit`: Since `[pre[v],post[v]]` is essentially the time interval during which v stayed on the stack, it is always the case that

- *two intervals* `[pre[u],post[u]]` *and* `[pre[v],post[v]]` *are either disjoint, or one contains another.*

  In other words, any two such intervals cannot partially overlap. We shall see many more useful properties of these numbers.

One of the simplest feats that can accomplished by appropriately modifying depth-first search is to subdivide an undirected graph to its *connected components*, defined next. A *path* in a graph $G = (V, E)$ is a sequence $(v_0, v_1, \ldots, v_n)$ of vertices, such that $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, n$. An undirected graph is said to be *connected* if there is a path between any two vertices.[2] If an undirected graph is disconnected, then its vertices can be partitioned into *connected components.* We can use depth-first search to tell if a graph is connected, and, if not, to assign to each vertex v an integer, `ccnum[v]`, identifying the connected component of the graph to which v belongs. This can be accomplished by adding a line to `previsit`.

```
procedure previsit(v:  vertex)
pre(v) := clock++
ccnum(v) := cc
```

Here `cc` is an integer used to identify the various connected components of the graph; it is initialized to zero just after line 7 of dfs, and increased by one just before each call of `explore` at line 10 of dfs.

[2]As we shall soon see, connectivity in directed graphs is a much more subtle concept.

3.2 Applications of Depth-First Searching

We can represent the tasks in a business or other organization by a *directed graph*: each vertex u represents a task, like manufacturing a widget, and each directed edge (u, v) means that task u must be completed before task v can be done (such as shipping a widget to a customer). The simplest question one can ask about such a graph is: *in what orders can I complete the tasks, satisfying the precedence order defined by the edges, in order to run the business?* We say "orders" rather than "order" because there is not necessarily a unique order (can you give an example?) Furthermore it may be that no order is legal, in which case you have to rethink the tasks in your business (can you give an example?). In the language of graph theory, answering these questions involves *computing strongly connected components* and *topological sorting*.

Here is a slightly more difficult question that we will also answer eventually: suppose each vertex is labeled by a positive number, which we interpret as the time to execute the task it represents. Then what is the minimum time to complete all tasks, assuming that independent tasks can be executed simultaneously?

The tool to answer these questions is DFS.

3.3 Depth-first Search in Directed Graphs

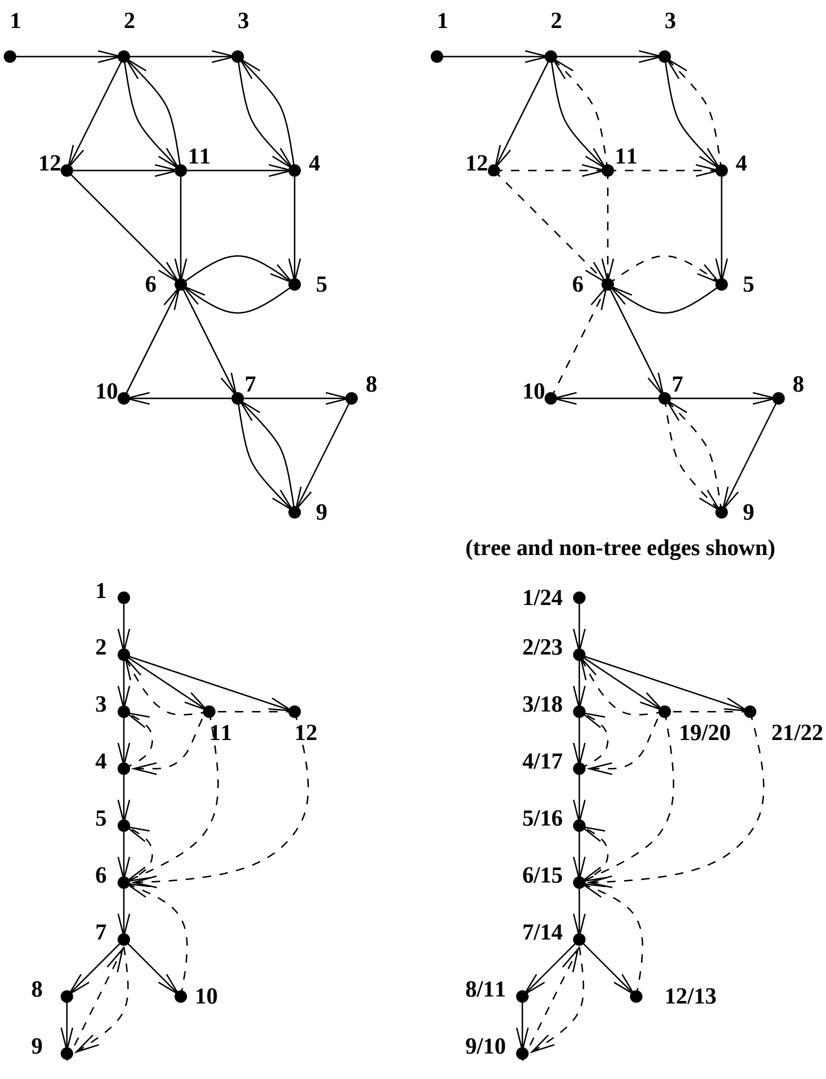
We can also run depth-first search on a directed graph, such as the one shown below.

The resulting `pre[v]`/`post[v]` numbers and depth-first search tree are shown. Notice, however, that not all non-tree edges of the graph are now back edges, going from a vertex in the tree to an ancestor. The non-tree edges of the graph can be classified into three types:

- Forward edges: these go from a vertex to a descendant (other than child) in the depth-first search tree. You can tell such an edge (v, w) because `pre[v]` $<$ `pre[w]`.
- Back edges: these go from a vertex to an ancestor in the depth-first search tree. You can tell such an edge (v, w) because, at the time it is traversed, `pre[v]` $>$ `pre[w]`, and `post[w]` is undefined.
- Cross edges: these go from "right to left," from a newly discovered vertex to a vertex that lies in a part of the tree whose processing has been concluded. You can tell such an edge (v, w), because, at the time it is traversed, `pre[v]` $>$ `pre[w]`, and `post[w]` is defined. Can there be cross edges in an undirected graph?

Why do all edges fall into one of these four categories (tree, forward, back, cross)?

DFS of a Directed Graph



(tree and non-tree edges shown)

(tree and non-tree edges shown) (pre/post numbers shown)

3.4 Directed Acyclic Graphs

A *cycle* in a directed graph is a path $(v_0, v_1, \ldots, v_n)$ such that (v_n, v_0) is also an edge. A directed graph is *acyclic*, or a *dag*, if it has no cycles. See CLR page 88 for a more careful definition.

CLAIM 1
A directed graph is acyclic if and only if depth-first search on it discovers no backedges.

PROOF: If (u, v) is a backedge, then (u, v) together with the path from v to u in the depth-first search tree form a cycle.

Conversely, suppose that the graph has a cycle, and consider the vertex v on the cycle assigned the lowest `pre[v]` number, i.e. v is the first vertex on the cycle that is visited.

Then all the other vertices on the cycle must be descendants of v, because there is path from v to each of them. Let (w, v) be the edge on the cycle pointing to v; it must be a backedge. $\Box$

It is therefore very easy to use depth-first search to see if a graph is acyclic: Just check that no backedges are discovered. But we often want more information about a dag: We may want to *topologically sort it.* This means to order the vertices of the graph from top to bottom so that all edges point downward. (*Note:* This is possible if and only if the graph is acyclic. Can you prove it? One direction is easy; and the topological sorting algorithm described next provides a proof of the other.) This is interesting when the vertices of the dag are tasks that must be scheduled, and an edge from u to v says that task u must be completed before v can be started. The problem of topological sorting asks: in what order should the tasks be scheduled so that all the precedence constraints are satisfied. The reason having a cycle means that topological sorting is impossible is that having u and v on a cycle means that task u must be completed before task v, and task v must be completed before task u, which is impossible.

To topologically sort a dag, we simply do a depth-first search, and then arrange the vertices of the dag in decreasing `post[v]`. That this simple method correctly topologically sorts the dag is a consequence of the following simple property of depth-first search:

LEMMA 2
For each edge (u, v) of G, `post(u)` $<$ `post(v)` *if and only if (u, v) is a back edge.*
PROOF: Edge (u, v) must either be a tree edge (in which case descendent v is popped before ancestor u, and so `post(u)`$>$`post(v)`), or a forward edge (same story), or a cross edge (in which case u is in a subtree explored after the subtree containing v is explored, so v is both pushed and popped before u is pushed and popped, i.e. `pre(v)` and `post(v)` are both less `pre(u)` and `post(u)`), or a back edge (u is pushed and popped after v is pushed and before v is popped, i.e. `pre(v)`$<$`pre(u)`$<$`post(u)`$<$`post(v)`). $\Box$

This property proves that our topological sorting method correct. Because take any edge (u, v) of the dag; since this is a dag, it is not a back edge; hence `post[u]` $>$ `post[v]`. Therefore, our method will list u before v, as it should. We conclude that, using depth-first search we can determine in linear time whether a directed graph is acyclic, and, if it is, to topologically sort its vertices, *also in linear time.*

Dags are an important subclass of directed graphs, useful for modeling hierarchies and causality. Dags are more general than rooted trees and more specialized than directed graphs. It is easy to see that every dag has a *sink* (a vertex with no outgoing edges). Here is why: Suppose that a dag has no sink; pick any vertex v_1 in this dag. Since it is not a sink, there is an outgoing edge, say (v_1, v_2). Consider now v_2, it has an outgoing edge (v_2, v_3). And so on. Since the vertices of the dag are finite, this cannot go on forever, vertices must somehow repeat—and we have discovered a cycle! Symmetrically, every dag also has a *source*, a vertex with no incoming edges. (But the existence of a source and a sink does not of course guarantee the graph is a dag!)

The existence of a source suggests another algorithm for outputting the vertices of a dag in topological order:

- `Find a source, output it, and delete it from the graph. Repeat until the graph is empty.`

Can you see why this correctly topologically sorts any dag? What will happen if we run it on a graph that has cycles? How would you implement it in linear time?

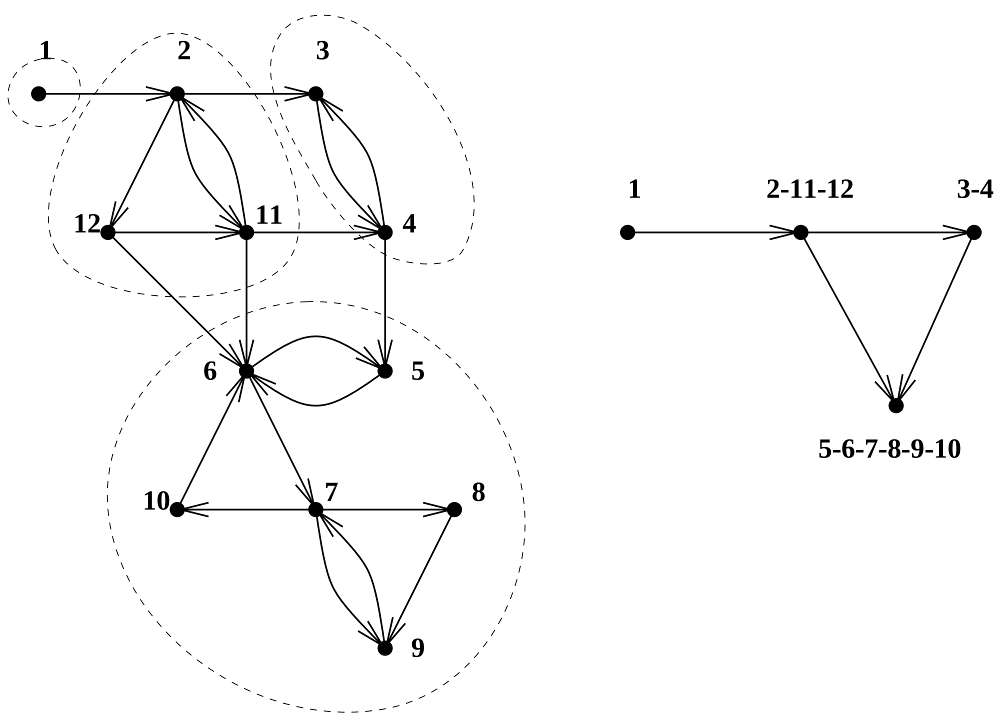
3.5 Strongly Connected Components

Connectivity in undirected graphs is rather straightforward: A graph that is not connected is naturally and obviously decomposed in several *connected components*. As we have seen, depth-first search does this handily: Each call of `explore` by `dfs` (which we also call a "restart" of `explore`) marks a new connected component.

In *directed graphs*, however, connectivity is more subtle. In some primitive sense, the directed graph above is "connected" (no part of it can be "pulled apart," so to speak, without "breaking" any edges). But this notion does not really capture the notion of connectedness, because there is no path from vertex 6 to 12, or from anywhere to vertex 1. The only meaningful way to define connectivity in directed graphs is this:

Call two vertices u and v of a directed graph $G = (V, E)$ *connected* if there is a path from u to v, *and* one from v to u. This relation between vertices is reflexive, symmetric, and transitive (check!), so it is an equivalence relation on the vertices. As such, it partitions V into disjoint sets, called the *strongly connected components* of the graph. In the figure below is a directed graph whose four strongly connected components are circled.

Strongly Connected Components



If we shrink each of these strongly connected components down to a single vertex, and draw an edge between two of them if there is an edge from some vertex in the first to some vertex in the second, the resulting directed graph has to be a *directed acyclic graph (dag)*—that is to say, it can have no cycles (see above). The reason is simple: A cycle containing several strongly connected components would merge them all to a single strongly connected component. We can restate this observation as follows:

Claim 3
Every directed graph is a dag of its strongly connected components.

This important decomposition theorem allows one to fathom the subtle connectivity information of a directed graph in a two-level manner: At the top level we have a dag, a rather simple structure. For example, we know that a dag is guaranteed to have at least one *source* (a vertex without incoming edges) and at least one *sink* (a vertex without outgoing edges), and can be topologically sorted. If we want more details, we could look inside a vertex of the dag to see the full-fledged strongly connected component (a tightly connected graph) that lies there.

This decomposition is extremely useful and informative; it is thus very fortunate that we have a very efficient algorithm, based on depth-first search, that finds it *in linear time!* We motivate this algorithm next. It is based on several interesting and slightly subtle properties of depth-first search:

Lemma 4
If depth-first search of a graph is started at a vertex u, then it will get stuck and restarted precisely when all vertices that are reachable from u have been visited. Therefore, if depth-first search is started at a vertex of a sink strongly connected component (a strongly connected component that has no edges leaving it in the dag of strongly connected components), then it will get stuck after it visits precisely the vertices of this strongly connected component.

For example, if depth-first search is started at vertex 5 above (a vertex in the only sink strongly connected component in this graph), then it will visit vertices 5, 6, 7, 8, 9, and 10—the six vertices form this sink strongly connected component—and then get stuck. Lemma 4 suggests a way of starting the sought decomposition algorithm, by finding our first strongly connected component: Start from any vertex in a sink strongly connected component, and, when stuck, output the vertices that have been visited: They form a strongly connected component!

Of course, this leaves us with two problems: (A) How to guess a vertex in a sink strongly connected component, and (B) how to continue our algorithm after outputting the first strongly connected component, by continuing with the second strongly connected component, etc.

Let us first face Problem (A). It is hard to solve it directly. There is no easy, direct way to obtain a vertex in a sink strongly connected component. *But* there is a fairly easy way to obtain a vertex in a *source* strongly connected component. In particular:

Lemma 5
The vertex with the highest `post` *number in depth-first search (that is, the vertex where the depth-first search ends) belongs to a source strongly connected component.*

Lemma 5 follows from a more basic fact.

Lemma 6

Let C and C' be two strongly connected components of a graph, and suppose that there is an edge from a vertex of C to a vertex of C'. Then the vertex of C visited first by depth-first search has higher `post` *than any vertex of C'.*

PROOF: There are two cases: Either C is visited before C' by depth-first search, or the other way around. In the first case, depth-first search, started at C, visits all vertices of C and C' before getting stuck, and thus the vertex of C visited first was the last among the vertices of C and C' to finish. If C' was visited before C by depth-first search, then depth-first search from it was stuck before visiting any vertex of C (the vertices of C are not reachable from C'), and thus again the property is true. □

Lemma 6 can be rephrased in the following suggestive way: *Arranging the strongly connected components of a directed graph in decreasing order of the highest* `post` *number topologically sorts the strongly connected components of the graph!* This is a generalization of our topological sorting algorithm for dags: After all, a dag is just a directed graph with singleton strongly connected components.

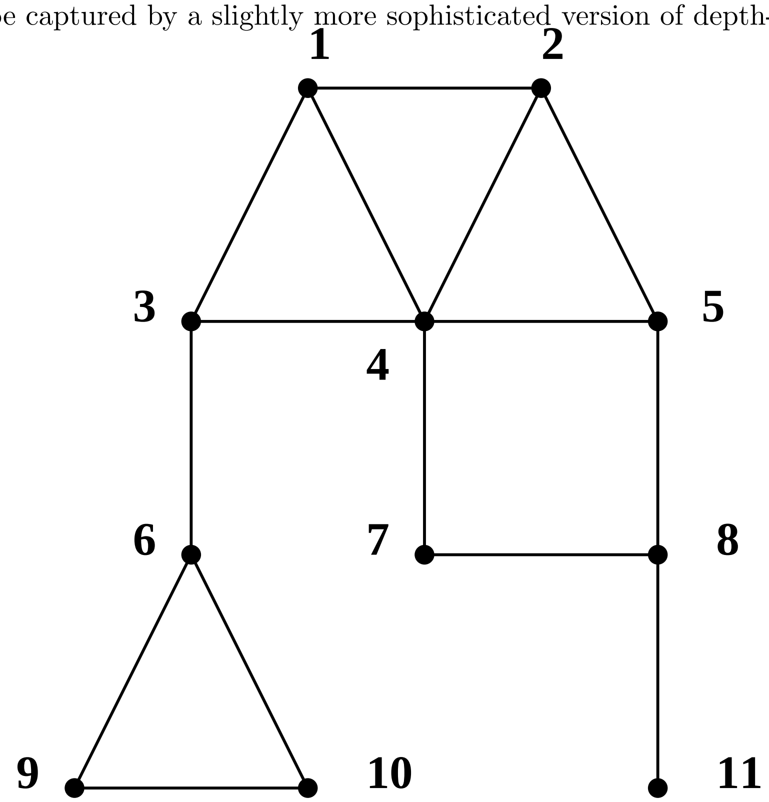
Lemma 5 provides an indirect solution to Problem (A): Consider the *reverse* graph of $G = (V, E)$, $G^R = (V, E^R)$—G with the directions of all edges reversed. G_R has precisely the same strongly connected components as G (why?). So, if we make a depth-first search in G^R, then the vertex where we end (the one with the highest `post` number) belongs to a source strongly connected component of G^R—that is to say, a sink strongly connected component of G. We have solved Problem (A)!

Onwards to Problem (B): How to continue after the first sink component is output? The solution is also provided by Lemma 6: After we output the first strongly connected componentand delete it from the graph, the vertex with the highest `post` from the depth-first search of G^R among the remaining vertices belongs to a sink strongly connected component of the remaining graph. Therefore, we can use the same depth-first search information from G^R to output the second strongly connected component, the third strongly connected component, and so on. The full algorithm is this:

1. Perform DFS on G^R.
2. Run DFS, labeling connected components, processing the vertices of G in order of decreasing `post` found in step 1.

Needless to say, this algorithm is as linear-time as depth-first search, only the constant of the linear term is about twice that of straight depth-first search. (*Question:* How does one construct G^R—that is, the adjacency lists of the reverse graph—from G in linear time? And how does one order the vertices of G in decreasing `post[v]` also in linear time?) If we run this algorithm on the directed graph above, Step 1 yields the following order on the vertices (decreasing post-order in G^R's depth-first search—watch the three "negatives" here): 5, 6, 10, 7, 9, 8, 3, 4, 2, 11, 12, 1, (assuming the are visited taking the lowest numbered vertex first). Step 2 now discovers the following strongly connected components: component 1: 5, 6, 10, 7, 9, 8; component 2: 3, 4; component 3: 2, 11, 12; and component 4: 1.

Incidentally, there *is* more sophisticated connectivity information that one can derive from *undirected* graphs. An *articulation point* is a vertex whose deletion increases the number of connected components in the undirected graph (in the graph below there are four articulation points: 3, 6, and 8. Articulation points divide the graph into *biconnected components* (the pieces of the graph between articulation points); this is not quite a partition, because neighboring biconnected components share an articulation point. For example, the graph below has four biconnected components: 1-2-3-4-5-7-8, 3-6, 6-9-10, and 8-11. Intuitively, biconnected components are the parts of the graph such that any two vertices have not just one path between them but two vertex-disjoint paths (unless the biconnected component happens to be a single edge). Just like any directed graph is a dag of its strongly connected components, any undirected graph can be considered *a tree of its biconnected components*. Not coincidentally, this more sophisticated and subtle connectivity information can also be captured by a slightly more sophisticated version of depth-first search.



Chapter 4

Breadth First Search and Shortest Paths

4.1 Breadth-First Search

Breadth-first search (BFS) is the variant of search that is guided by a *queue*, instead of the stack that is implicitly used in DFS's recursion. In preparation for the presentation of BFS, let us first see what an iterative implementation of DFS looks like.

```
procedure i-DFS(u:  vertex)
initialize empty stack S
push(u,S)
while not empty(S)
  v=pop(S)
  visited(v)=true
  for each edge (v,w) out of v do
    if not visited(w) then push(w)

algorithm dfs(G = (V,E): graph)
for each v in V do visited(v) := false
for each v in V do
  if not visited(v) then i-DFS(v)
```

There is one stylistic difference between DFS and BFS: One does not restart BFS, because BFS only makes sense in the context of exploring the part of the graph that is reachable from a particular node (s in the algorithm below). Also, although BFS does not have the wonderful and subtle properties of DFS, it does provide useful information: Because it tries to be "fair" in its choice of the next node, it visits nodes in order of increasing distance from s. In fact, our BFS algorithm below labels each node with the shortest distance from s, that is, the number of edges in the shortest path from s to the node. The algorithm is this:

```
Algorithm BFS(G=(V,E): graph, s:  node);
initialize empty queue Q
for all v ∈ V do dist[v]=∞
```

```
insert(s,Q)
dist[s]:=0
while Q is not empty do
  v:= remove(Q),
  for all edges (v,w) out of v do
    if dist[w] = ∞ then
      insert(w,Q)
      dist[w]:=dist[v]+1
```

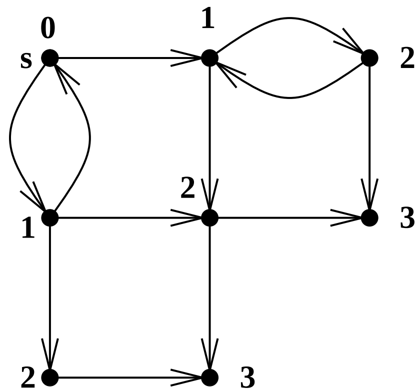


Figure 4.1: BFS of a directed graph

For example, applied to the graph in Figure 4.1, this algorithm labels the nodes (by the array `dist`) as shown. We would like to show that the values of *dist* are exactly the distances of each vertex from s. While this may be intuitively clear, it is a bit complicated to prove it formally (although it does not have to be as complicated as in CLR/CLRS). We first need to observe the following fact.

Lemma 7
In a BFS, the order in which vertices are removed from the queue is always such that if u is removed before v, then $dist[u] \leq dist[v]$.
Proof: Let us first argue that, at any given time in the algorithm, the following invariant remains true:

> if $v_1, \ldots, v_r$ are the vertices in the queue then $dist[v_1] \leq \ldots \leq dist[v_r] \leq dist[v_1] + 1$.

At the first step, the condition is trivially true because there is only one element in the queue. Let now the queue be $(v_1, \ldots, v_r)$ at some step, and let us see what happens at the following step. The element v_1 is removed from the queue, and its non-visited neighbors $w_1, \ldots, w_i$ (possibly, $i = 0$) are added to queue, and the vector *dist* is updated so that $dist[w_1] = dist[w_2] = \ldots = dist[w_i] = dist[v_1] + 1$, while the new queue is $(v_2, \ldots, v_r, w_1, \ldots, w_i)$ and we can see that the invariant is satisfied.

Let us now prove that if u is removed from the queue in the step before v is removed from the queue, then $dist[u] \leq dist[v]$. There are two cases: either u is removed from the queue at a time when v is immediatly after u in the queue, and then we can use the invariant to say that $dist[u] \leq dist[v]$, or u was removed at a time when it was the only element in the queue. Then, if v is removed at the following step, it must be the case that v has been added to queue while processing u, which means $dist[v] = dist[u] + 1$.

The lemma now follows by observing that if u is removed before v, we can call $w_1, \ldots, w_i$ the vertices removed between u and v, and see that $dist[u] \leq dist[w_1] \leq \ldots \leq dist[w_i] \leq dist[v]$. □

We are now ready to prove that the *dist* values are indeed the lengths of the shortest paths from s to the other vertices.

Lemma 8
At the end of BFS, for each vertex v reachable from s, the value dist[v] equals the length of the shortest path from s to v.
Proof: By induction on the value of $dist[v]$. The only vertex for which *dist* is zero is s, and zero is the correct value for s.

Suppose by inductive hypothesis that for all vertices u such that $dist[u] \leq k$ then $dist[u]$ is the true distance from s to u, and let us consider a vertex w for which $dist[w] = k+1$. By the way the algorithm works, if $dist[w] = k + 1$ then w was first discovered from a vertex v such that the edge (v, w) exists and such that $dist[v] = k$. Then, there is a path of length k from s to v, and so there is a path of length $k + 1$ from s to w. It remains to prove that this is the shortest path. Suppose by contradiction that there is a path $(s, \ldots, v', w)$ of length $\leq k$. Then the vertex v' is reachable from s via a path of length $\leq k - 1$, and so $dist[v'] \leq k - 1$. But this implies that v' was removed from the queue before v (because of Lemma 7), and when processing v' we would have discovered w, and assigned to $dist[w]$ the smaller value $dist[v'] + 1$. We reached a contradiction, so indeed $k + 1$ is the length of the shortest path from s to w, and this completes the inductive step and the proof of the lemma. □

Breadth-first search runs, of course, in linear time $O(|V| + |E|)$. The reason is the same as with DFS: BFS visits each edge exactly once, and does a constant amount of work per edge.

4.2 Dijkstra's Algorithm

Suppose each edge (v, w) of our graph has a *weight*, a positive integer denoted weight(v, w), and we wish to find the shortest from s to all vertices reachable from it.[1]

We will still use BFS, but instead of choosing which vertices to visit by a queue, which pays no attention to how far they are from s, we will use a *heap*, or *priority queue*, of vertices. The priority depends on our current best estimate of how far away a vertex is from s: we will visit the closest vertices first.

These distance estimates will always *overestimate* the actual shortest path length from s to each vertex, but we are guaranteed that the *shortest* distance estimate in the queue is actually the true shortest distance to that vertex, so we can correctly mark it as finished.

[1] What if we are interested only in the shortest path from s to a specific vertex t? As it turns out, all algorithms known for this problem also give us, as a free byproduct, the shortest path from s to all vertices reachable from it.

```
algorithm Dijkstra(G=(V, E, weight), s)
  variables:
    v,w:  vertices (initially all unmarked)
    dist:  array[V] of integer
    prev:  array[V] of vertices
    heap of vertices prioritized by dist
  for all v ∈ V do { dist[v] := ∞, prev[v] :=nil}
  H:={s} , dist[s] :=0 , mark(s)
  while H is not empty do
  {
    v := deletemin(H) , mark(v)
    for each edge (v,w) out of E do
      {
        if w unmarked and dist[w] > dist[v] + weight[v,w] then
          {
            dist[w] := dist[v] + weight[v,w]
            prev[w] := v
            insert(w,H)
          }
      }
  }
```

Figure 4.2: Dijkstra's algorithm.

As in all shortest path algorithms we shall see, we maintain two arrays indexed by V. The first array, `dist[v]`, contains our overestimated distances for each vertex v, and will contain the true distance of v from s when the algorithm terminates. Initially, `dist[s]`$=0$ and the other `dist[v]`$=\infty$, which are sure-fire overestimates. The algorithm will repeatedly decrease each `dist[v]` until it equals the true distance. The other array, `prev[v]`, will contain the last vertex before v in the shortest path from s to v. The pseudo-code for the algorithm is in Figure 4.2

The procedure `insert(w,H)` must be implemented carefully: if w is already in H, we do not have to actually insert w, but since w's priority `dist[w]` is updated, the position of w in H must be updated.

The algorithm, run on the graph in Figure 4.3, will yield the following heap contents (vertex: dist/priority pairs) at the beginning of the while loop: $\{s\}$, $\{a : 2, b : 6\}$, $\{b : 5, c : 3\}$, $\{b : 4, e : 7, f : 5\}$, $\{e : 7, f : 5, d : 6\}$, $\{e : 6, d : 6\}$, $\{e : 6\}$, $\{\}$. The distances from s are shown in the figure, together with the *shortest path tree from s*, the rooted tree defined by the pointers prev.

4.2.1 What is the complexity of Dijkstra's algorithm?

The algorithm involves $|E|$ insert operations (in the following, we will call m the number $|E|$) on H and $|V|$ deletemin operations on H (in the following, we will call n the number

Shortest Paths

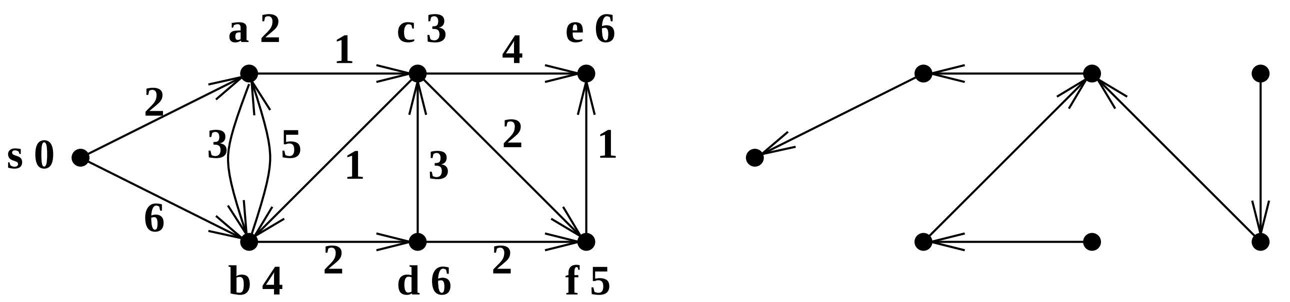


Figure 4.3: An example of a shortest paths tree, as represented with the `prev[]` vector.

$|V|$), and so the running time depends on the implementation of the heap H, so let us discuss this implementation. There are many ways to implement a heap.[2] Even the most unsophisticated one (an amorphous set, say a linked list of vertex/priority pairs) yields an interesting time bound, $O(n^2)$ (see first line of the table below). A binary heap gives $O(m \log n)$.

Which of the two should we use? The answer depends on how *dense* or *sparse* our graphs are. In all graphs, m is between n and n^2. If it is $\Omega(n^2)$, then we should use the linked list version. If it is anywhere below $\frac{n^2}{\log n}$, we should use binary heaps.

heap implementation	deletemin	insert	$n\times$deletemin$+m\times$insert
linked list	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$

4.2.2 Why does Dijkstra's algorithm work?

Here is a sketch. Recall that the inner loop of the algorithm (the one that examines edge (v, w) and updates `dist`) examines each edge in the graph exactly once, so at any point of the algorithm we can speak about the subgraph $G' = (V, E')$ examined *so far*: it consists of all the nodes, and the edges that have been processed. Each pass through the inner loop adds one edge to E'. We will show that the following property of `dist[]` is an invariant of the inner loop of the algorithm:

> `dist[w]` is the minimum distance from s to w in G', and
> if $\mathtt{v_k}$ is the k-th vertex marked, then $\mathtt{v_1}$ through $\mathtt{v_k}$ are the k closest vertices to `s` in G, and the algorithm has found the shortest paths to them.

We prove this by induction on the number of edges $|E'|$ in E'. At the very beginning, before examining *any* edges, $E' = \emptyset$ and $|E'| = 0$, so the correct minimum distances are `dist[s]`

[2] In all heap implementations we assume that we have an array of pointers that give, for each vertex, its position in the heap, if any. This allows us to always have at most one copy of each vertex in the heap. Each time `dist[w]` is decreased, the `insert(w,H)` operation finds w in the heap, changes its priority, and possibly moves it up in the heap.

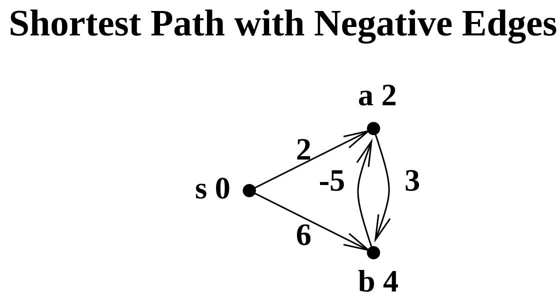


Figure 4.4: A problematic instance for Dijkstra's algorithm.

`= 0` and `dist[w] =` ∞, as initialized. And `s` is marked first, with the distance to it equal to zero as desired.

Now consider the next edge (v, w) to get $E'' = E' \cup (v, w)$. Adding this edge means there is a new way to get to `w` from `s` in E'': from `s` to `v` to `w`. The shortest way to do this is `dist[v] + weight(v,w)` by induction. Also by induction the shortest path to get to `w` not using edge (`v`, `w`) is `dist[w]`. The algorithm then replaces `dist[w]` by the minimum of its old value and possible new value. Thus `dist[w]` is still the shortest distance to `w` in E''. We still have to show that all the other `dist[u]` values are correct; for this we need to use the heap property, which guarantees that (`v`, `w`) is an edge coming out of the node `v` in the heap *closest* to the source `s`. To complete the induction we have to show that adding the edge (v, w) to the graph G' does not change the values of the distance of *any* other vertex `u` from `s`. `dist[u]` could only change if the shortest path from `s` to `u` had previously gone through `w`. But this is impossible, since we just decided that `v` was closer to `s` than `w` (since the shortest path to `w` is via `v`), so the heap would not yet have marked `w` and examined edges out of it.

4.3 Negative Weights—Bellman-Ford Algorithm

Our argument of correctness of our shortest path algorithm was based on the fact that adding edges can only make a path longer. This however would not work if we had *negative edges:* If the weight of edge (b, a) in figure 4.4 were -5, instead of 5 as in the last lecture, the first event (the arrival of BFS at vertex a, with `dist[a]=2`) would not be suggesting the correct value of the shortest path from s to a. Obviously, with negative weights we need more involved algorithms, which repeatedly update the values of dist.

The basic information updated by the negative edge algorithms is the same, however. They rely on arrays `dist` and `prev`, of which `dist` is always a conservative overestimate of the true distance from s (and is initialized to ∞ for all vertices, except for s for which it is 0). The algorithms maintain `dist` so that it is always such a conservative overestimate. This is done by the same scheme as in our previous algorithm: Whenever "tension" is discovered between vertices v and w in that `dist(w)>dist(v)+weight(v,w)`—that is, when it is discovered that `dist(w)` is a more conservative overestimate than it has to be—then this "tension" is "relieved" by this code:

```
procedure update(v,w:  edge)
```

```
if dist[w] > dist[v] + weight[v,w] then
dist[w] := dist[v] + weight[v,w], prev[w] := v
```

One crucial observation is that this procedure is *safe*, it never invalidates our "invariant" that `dist` is a conservative overestimate. Our algorithm for negative edges will consist of many updates of the edges.

A second crucial observation is the following: Let $a \neq s$ be a vertex, and consider the shortest path from s to a, say $s, v_1, v_2, \ldots, v_k = a$ for some k between 1 and $n-1$. If we perform update first on (s, v_1), later on (v_1, v_2), and so on, and finally on (v_{k-1}, a), then we are sure that $\mathrm{dist}(a)$ contains the true distance from s to a—and that the true shortest path is encoded in prev. We must thus find a sequence of updates that guarantee that these edges are updated in this order. We don't care if these or other edges are updated several times in between, all we need is to have a sequence of updates that contains this particular subsequence.

And there is a very easy way to guarantee this: *Update all edges $n-1$ times in a row!* Here is the algorithm:

```
algorithm BellmanFord(G=(V, E, weight): graph with weights; s: vertex)
dist: array[V] of integer; prev: array[V] of vertices
for all v ∈ V do { dist[v] := ∞, prev[v] := nil}
for i := 1,...,n − 1 do
  { for each edge (v,w) ∈ E do update[v,w] }
```

This algorithm solves the general single-source shortest path problem in $O(n \cdot m)$ time.

4.4 Negative Cycles

In fact, if the weight of edge (b, a) in the figure above were indeed changed to -5, then there would be a bigger problem with the graph of this figure: It would have a *negative cycle* (from a to b and back). On such graphs, it does not make sense to even *ask* the shortest path question. What is the shortest path from s to a in the modified graph? The one that goes directly from s to b to a (cost: 1), or the one that goes from s to b to a to b to a (cost: -1), or the one that takes the cycle twice (cost: -3)? And so on.

The shortest path problem is ill-posed in graphs with negative cycles. It is still an interesting question to ask however. If the vertices represent currencies (dollars, marks, francs, etc.) and the edge weights represent (logarithms of) exchange rates, then a negative cycle means that it is possible to start with \$100, say, buy and sell a sequence of other currencies, coming back to dollars, and ending up with more than \$100. This process may be repeated, and is called *arbitrage*. In the real world, the exchange rates keep changing, so the problem is more challenging. See Question 25-3 in CLR.

Our algorithm in the previous section works only in the absence of negative cycles. (Where did we assume no negative cycles in our correctness argument? Answer: When we asserted that a shortest path from s to a exists ...) But it would be useful if our algorithm were able to *detect* whether there is a negative cycle in the graph, and thus to report reliably on the meaningfulness of the shortest path answers it provides.

This is easily done as follows: After the $n-1$ rounds of updates of all edges, do a last round. If anything is changed during this last round of updates—if, that is, there is still "tension" in some edges—this means that there is no well-defined shortest path (because, if there were, $n-1$ rounds would be enough to relieve all tension along it), and thus there is a negative cycle reachable from s.

Chapter 5

Minimum Spanning Trees

5.1 Minimum Spanning Trees

A tree is an undirected graph that is connected and acyclic. It is easy to show (see Theorem 5.2 in CLR) that if graph $G(V,E)$ that satisfies any two of the following properties also satisfies the third, and is therefore a tree:

- $G(V,E)$ is connected
- $G(V,E)$ is acyclic
- $|E| = |V| - 1$

A spanning tree in a graph $G(V,E)$ is a subset of edges $T \subseteq E$ that are acyclic and connect all the vertices in V. It follows from the above conditions that a spanning tree must consist of exactly $n-1$ edges. Now suppose that each edge has a weight associated with it: $w : E \to Z$. Say that the weight of a tree T is the sum of the weights of its edges; $w(T) = \sum_{e \in T} w(e)$. The minimum spanning tree in a weighted graph $G(V,E)$ is one which has the smallest weight among all spanning trees in $G(V,E)$.

In general, the number of spanning trees in $G(V,E)$ grows exponentially in the number of vertices in $G(V,E)$. Therefore it is infeasible to search through all possible spanning trees to find the lightest one. Luckily it is not necessary to examine all possible spanning trees; minimum spanning trees satisfy a very important property which makes it possible to efficiently zoom in on the answer.

We shall construct the minimum spanning tree by successively selecting edges to include in the tree. We will guarantee after the inclusion of each new edge that the selected edges, X, form a subset of some minimum spanning tree, T. How can we guarantee this if we don't yet know any minimum spanning tree in the graph? The following property provides this guarantee:

Lemma 9 (Cut Lemma)
Let $X \subseteq T$ where T is a MST in $G(V,E)$. Let $S \subset V$ such that no edge in X crosses between S and $V-S$; i.e. no edge in X has one endpoint in S and one endpoint in $V-S$. Among edges crossing between S and $V-S$, let e be an edge of minimum weight. Then $X \cup \{e\} \subseteq T'$ where T' is a MST in $G(V,E)$.

Before proving the Cut Lemma, we need to make some observations about trees. Let $G = (V, E)$ be a tree, let v, w be vertices such that $(v, w) \notin E$, and consider the graph obtained from G by adding the edge (v, w), that is, $G' = (V, E')$ where $E' = E \cup \{(u, v)\}$. Then

1. G' has exactly one simple cycle

2. If we remove from G' any of the edges on the unique cycle, we obtain a tree.

To prove part (1), notice that, since G is connected, there is a path from v to w, and this path, together with the edge (v, w) gives a cycle. Now, every simple cycle in G' must contain the edge (v, w), and then a simple path in G from v to w, but in a tree there is only one simple path between two fixed vertices (if there were two simple paths with the same endpoints, their union would contain a cycle).

To prove part (2), let $v, u_1, \ldots, u_k, w, v$ be the unique cicle in G', and let us call $v = v_0$ and $w = v_{k+1}$. Suppose we remove from G' the edge (v_i, v_{i+1}), for some $i \in \{0, \ldots, k\}$, and let us call G'' the resulting graph. Then G'' is acyclic, because we have broken the unique cycle in G', and it is connected, since G' is connected, and any path in G' that used the edge (v_i, v_{i+1}) can be re-routed via the path from v_i to v_{i+1} that still exists.

We can now prove the Cut Lemma.

PROOF: [Of Lemma 9] If $e \in T$, then we can set $T' = T$ and we are done.

Let us now consider the case $e \notin T$. Adding e into T creates a unique cycle. We will remove a single edge e' from this unique cycle, thus getting $T' = (T \cup \{e\}) - \{e'\}$ which, by the above analysis is again a tree.

We will now show that it is always possible to select an edge e' in the cycle such that it crosses between S and $V - S$. Now, since e is a minimum weight edge crossing between S and $V - S$, $w(e') \geq w(e)$. Therefore $w(T') = w(T) + w(e) - w(e') \leq w(T)$. However since T is a MST, it follows that T' is also a MST and $w(e) = w(e')$. Furthermore, since X has no edge crossing between S and $V - S$, it follows that $X \subseteq T'$ and thus $X \cup \{e\} \subseteq T'$.

How do we know that there is an edge $e' \neq e$ in the unique cycle created by adding e into T, such that e' crosses between S and $V - S$? This is easy to see, because as we trace the cycle, e crosses between S and $V - S$, and we must cross back along some other edge to return to the starting point. $\Box$

In light of this, the basic outline of our minimum spanning tree algorithms is going to be the following:

$X := \{\ \}$ ... X contains the edges of the MST
Repeat until $|X| = n - 1$
    Pick a set $S \subseteq V$ such that no edge in X crosses between S and $V - S$
    Let e be a lightest edge in $G(V, E)$ that crosses between S and $V - S$
    $X := X \cup \{e\}$

We will now describe two implementations of the above general procedure.

5.2 Prim's algorithm:

In the case of Prim's algorithm, X consists of a single tree, and the set S is the set of vertices of that tree. In order to find the lightest edge crossing between S and $V - S$, Prim's algorithm maintains a heap containing all those vertices in $V - S$ which are adjacent to some vertex in S. The key of a vertex v, according to which the heap is ordered, is the weight of its lightest edge to a vertex in S. This is reminiscent of Dijkstra's algorithm. As in Dijkstra's algorithm, each vertex v will also have a parent pointer `prev[v]` which is the other endpoint of the lightest edge from v to a vertex in S. Notice that the pseudocode for Prim's algorithm is identical to that for Dijkstra's algorithm, except for the definition of the key under which the heap is ordered:

```
algorithm Prim(weighted graph G=(V, E))
initialize empty priority queue H
for all v ∈ V do
  key[v] = ∞; prev[v] =nil
pick an arbitrary vertex s
H={s}; key[s] =0; mark(s)
while H is not empty do
  v := deletemin(H)
  mark(v)
  for each edge (v,w) out of E do
    if w unmarked and key[w] > weight[v,w] then
      key[w] := weight[v,w]; prev[w] = v; insert(w,H)
```

As in Dijkstra's algorithm, `insert(w,H)` really means to insert only if $w \notin H$, and to update w's priority `key` in H otherwise.

The complexity analysis of Prim's algorithm is identical to Dijkstra: each vertex and each edge is processed once, so the cost is $|V|\cdot\text{Cost(deletemin)} + |E|\cdot\text{Cost(insert)}$.

The vertices that are removed from the heap form the set S in the cut property stated above. The set X of edges chosen to be included in the MST are given by the parent pointers `prev` of the vertices in the set S. Since the smallest key in the heap at any time gives the lightest edge crossing between S and $V - S$, Prim's algorithm follows the generic outline for a MST algorithm presented above, and therefore its correctness follows from the cut property.

5.3 Kruskal's algorithm

Kruskal's algorithm starts with the edges sorted in increasing order by weight. Initially $X = \{\ \}$, and each vertex in the graph regarded as a trivial tree (with no edges). Each edge in the sorted list is examined in order, and if its endpoints are in the same tree, then the edge is discarded; otherwise it is included in X and this causes the two trees containing the endpoints of this edge to merge into a single tree. Thus, X consists of a forest of trees, and edges are added until it consists of exactly one tree, a MST. At each step S consists of the endpoints of vertices of one tree in X, the tree which contains one endpoint of the chosen edge.

To implement Kruskal's algorithm, given a forest of trees, we must decide given two vertices whether they belong to the same tree. For the purposes of this test, each tree in the forest can be represented by a set consisting of the vertices in that tree. We also need to be able to update our data structure to reflect the merging of two trees into a single tree. Thus our data structure will maintain a collection of disjoint sets (disjoint since each vertex is in exactly one tree), and support the following two operations:

- `find(x)`: Given an element x, which set does it belong to?
- `union(x,y)`: replace the set containing x and the set containing y by their union.

The data structure is constructed with the operation `makeset(x)`, that adds to the data structure a set that contains the only element x.

We will discuss the implementation of `find` and `union` later. The pseudocode for Kruskal's algorithm follows:

```
algorithm Kruskal(weighted graph G(V,E))
  X = { }
  sort E by weight
  for u ∈ V
      makeset(u)
  for (u,v) ∈ E in increasing order by weight
      if find(u) ≠ find(v) do
          X = X ∪ {(u,v)}
          union(u,v)
  return(X)
end
```

The correctness of Kruskal's algorithm follows from the following argument: Kruskal's algorithm adds an edge e into X only if it connects two trees; let S be the set of vertices in one of these two trees. Then e must be the first edge in the sorted edge list that has one endpoint in S and the other endpoint in $V - S$, and is therefore the lightest edge that crosses between S and $V - S$. Thus the cut property of MST implies the correctness of the algorithm.

The running time of the algorithm is dominated by the set operations `union` and `find` and by the time to sort the edge weights. There are $n - 1$ `union` operations (one corresponding to each edge in the spanning tree), and $2m$ `find` operations (2 for each edge). Thus the total time of Kruskal's algorithm is $O(m \times FIND + n \times UNION + m \log m)$. This will be seen to be $O(m \log n)$.

5.4 Exchange Property

Actually spanning trees satisfy an even stronger property than the cut property—the exchange property. The exchange property is quite remarkable since it implies that we can "walk" from any spanning tree T to a minimum spanning tree $\hat{T}$ by a sequence of exchange moves—each such move consists of throwing an edge out of the current tree that is not in

$\hat{T}$, and adding a new edge into the current tree that is in $\hat{T}$. Moreover, each successive tree in the "walk" is guaranteed to weigh no more than its predecessor.

Lemma 10 (Exchange Lemma)
Let T and T' be spanning trees in $G(V, E)$. Given any $e' \in T' - T$, there exists an edge $e \in T - T'$ such that $(T - \{e\}) \cup \{e'\}$ is also a spanning tree.
Proof: [Sketch] The proof is quite similar to that of the Cut Lemma. Adding e' into T results in a unique cycle. There must be some edge in this cycle that is not in T' (since otherwise T' must have a cycle). Call this edge e. Then deleting e restores a spanning tree, since connectivity is not affected, and the number of edges is restored to $n - 1$. $\Box$

To see how one may use this exchange property to "walk" from any spanning tree to a MST: let T be any spanning tree and let $\hat{T}$ be a MST in $G(V, E)$. Let e' be the lightest edge that is not in both trees. Perform an exchange using this edge. Since the exchange was done with the lightest such edge, the new tree must be at least as light as the old one. Since $\hat{T}$ is already a MST, it follows that the exchange must have been performed upon T and results in a lighter spanning tree which has more edges in common with $\hat{T}$ (if there are several edges of the same weight, then the new tree might not be lighter, but it still has more edges in common with $\hat{T}$).

Chapter 6

Universal Hashing

6.1 Hashing

We assume that all the basics about hash tables have been covered in 61B.

We will make the simplifying assumption that the keys that we want to hash have been encoded as integers, and that such integers are in the range $1, \ldots, M$. We also assume that collisions are handled using linked lists.

Suppose that we are using a table of size m, that we have selected a hash function $h : \{1, \ldots, M\} \to \{0, \ldots, m-1\}$ and that, at some point, the keys $y_1, \ldots, y_n$ have been inserted in the data structure, and that we want to find, or insert, or delete, the key x. The running time of such operation will be a big-Oh of the number of elements y_i such that $h(y_i) = h(x)$.

No matter what h does, if $M > m(n+1)$, there will always be a worst-case situation where $y_1, \ldots, y_n, x$ are all mapped by h into the same bucket, and then the running time of find, insert and delete will be $\Theta(n)$. However, in practice, hash tables work well. In order to explain the real behavior of hash tables we need to go beyond a worst-case analysis, and do a probabilistic analysis.

A simple analysis can be done assuming that the keys to be inserted in the table come from the uniform distribution over $\{1, \ldots, M\}$. It is not hard to come up with a function $h : \{1, \ldots, M\} \to \{0, \ldots, m-1\}$ with the property that if $y_1, \ldots, y_n, x$ are uniformly and independently distributed over $\{1, \ldots, M\}$, then $h(y_1), \ldots, h(y_n), h(x)$ are uniformly (or almost uniformly) distributed over $\{0, \ldots, m-1\}$, and then argue that, on average, the number of y_i such that $h(x) = h(y_i)$ is about n/m, and so an operation on x can be performed in $O(1)$ time assuming that, say, $m = 2n$.

In practice, however, inputs do not come from a uniform distribution, and choices of h that make the above proof work may or may not work well if the inputs come from different distributions.

A much more sound approach is to consider the behavior of the data structure on arbitrary (worst-case) inputs, but to let randomness come in the definition of the hash function h. Then we will look at the average running time of find, insert and delete operations, but this time the average will be over the randomness used by the algorithm, and there will be no assumption on the way the input is distributed. This kind of analysis is the object of today's lecture.

6.2 Universal Hashing

We want to consider hash functions whose definition involves random choices. Equivalently, we consider *families* of functions, and consider the randomized process of selecting at random a function from the family.

A collection H of hash functions $h : \{1, \ldots, M\} \to \{0, \ldots, m-1\}$ is said to be *2-universal* if for every two different $x, y \in \{1, \ldots, M\}$ we have

$$\mathbf{Pr}_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}$$

(Note that the CLR/CLRS definition has '=' instead of '$\leq$')

Consider the following construction. Fix a prime $p > M$. It is known that there is at least one (in fact, there are several) prime between M and $2M$. A procedure for finding such a p could be to generate at random a number between M and $2M$ and then test for primality. (There is an efficient randomized algorithms for testing primality, but we will probably not see it in this course.)

Then define, for every $a \in \{1, \ldots, p-1\}$ and every $b \in \{0, \ldots, p-1\}$ the function

$$g_{a,b}(x) = ax + b \pmod{p}$$

For each such g we define a hash function

$$h_{a,b}(x) = g_{a,b}(x) \pmod{m} \ ,$$

we will prove that this construction is 2-universal

Lemma 11

For every fixed two distinct values $x, y \in \{0, \ldots, m-1\}$, the number of pairs a, b such that $h_{a,b}(x) = h_{a,b}(y)$ is at most $p(p-1)/m$.

Proof: We will consider all possible $s, t \in \{0, \ldots, p-1\}$ such that $s = t \pmod{m}$, and then for each such pair (s, t), we will count how many pairs (a, b) are there such that $h_{a,b}(x) = s \pmod{m}$ and $h_{a,b}(y) = t \pmod{m}$.

The number of choices for the pair (s, t) is $p(\lceil p/m \rceil - 1) < p(p-1)/m$.

For each pair s, t we are now asking how many values $a, b \in \{0, \ldots, p-1\}$ are there that satisfy the following system

$$\begin{cases} ax + b = s \pmod{p} \\ ay + b = t \pmod{p} \end{cases}$$

Algebra tells us that, if p is prime, the solution is unique. So the number of pairs (a, b) for which $h_{a,b}(x) = h_{a,b}(y)$ is at most $p(p-1)/m$. $\square$

Since there are $p(p-1)$ functions in our family, the probability that $h_{a,b}(x) = h_{a,b}(y)$ is at most $1/m$, and so our family is indeed 2-universal.

6.3 Hashing with 2-Universal Families

Suppose now that we pick at random h from a family of 2-universal hash functions, and we build a hash table by inserting elements $y_1, \ldots, y_n$. Then we are given a key x that we want to find, insert or delete from the table. What will the expected number of collisions between x and $y_1, \ldots, y_n$ be?

LEMMA 12
Let H be a 2-universal family of hash functions mapping $\{1, \ldots, M\}$ into $\{0, \ldots, m-1\}$, and let $x, y_1, \ldots, y_n$ be elements of $\{1, \ldots, M\}$.

If we pick h at random from H, then the average number of elements y_i such that $h(x) = h(y_i)$ is at most n/m; in symbols

$$\mathbf{E}[|\{i : h(x) = h(y_i)\}|] \leq n/m$$

PROOF: Call C the random variable (depending on the choice of h) that counts the number of collisions between $h(x)$ and $h(y_1), \ldots, h(y_n)$. I.e. $C = |\{j : h(y_j) = h(x)\}|$.

Call C_y the random variable (depending on the choice of h) that is 1 if $h(x) = h(y)$, and 0 otherwise.

Then for every y

$$\begin{aligned} \mathbf{E}[C_y] &= 0 \cdot \mathbf{Pr}[h(x) \neq h(y)] + 1 \cdot \mathbf{Pr}[h(x) = h(y)] \\ &= \mathbf{Pr}[h(x) = h(y)] \leq \frac{1}{m} \end{aligned}$$

Since $C = \sum_{i=1}^{n} C_{y_i}$, we have $\mathbf{E}[C] = \mathbf{E}[\sum_{i=1}^{n} C_{y_i}] = \sum_{i=1}^{n} \mathbf{E}[C_{y_i}] \leq n/m$ $\Box$

So, if we choose $m = \Theta(n)$, each operation has $O(1)$ average running time.

Chapter 7

A Randomized Min Cut Algorithm

7.1 Min Cut

Given an undirected graph $G = (V, E)$, a *cut* is a partition of the set of vertices into two non-empty subsets S and $V - S$. The *cost* of a cut $(S, V - S)$ is the number of edges of E that are incident on one vertex in S and one vertex in $V - S$. The *global min-cut* problem (or, in short, the min-cut problem) is the problem of finding a cut $(S, V - S)$ of minimum cost.

Another way to look at the problem is that we are looking at the smallest set $C \subseteq E$ of edges such that removing those edges from the graph disconnects it. If the cost of a min-cut in a graph is k, then removing any $k-1$ edges from the graph leaves it connected, but there is a way of removing k edges that disconnects it. A graph whose min-cut has cost k is also said to be k-connected. In a k connected graph, there are always at least k different simple paths between any two edges. Computing the min-cut of the graph of a network is then, for example, a way to estimate the worst-case reliability of the network.

More generally, we may be also given a weight function that associates a positive value $weight(u, v)$ to each edge $(u, v) \in E$. In this case, the cost of a cut $(S, V - S)$ is the sum of the weights of the edges that are incident on a vertex in S and a vertex in $V - S$.

Algorithms to compute the min-cut of a graph can be derived from network flow algorithms that we will see later in the course. Such flow-based algorithms are not very efficient, and they are complicated to implement and to analyze. Today, we will see a randomized algorithms that seems just too simple to possibly work, and then we will see that it also has a fairly simple analysis. A straightforward implementation of the algorithm would be quite slow, but it is possible to optimize the implementation somewhat, and come up with a very efficient, and still simple, randomized algorithm.

7.2 The Contract Algorithms

We introduce the following graph operation (that was implicit in one the minimum spanning tree algorithms that we have seen): given an undirected graph $G = (V, E)$, a weight function $weight()$ and an edge $(u, v) \in E$, `contract ((u,v),G)` returns a graph G' that is identical to G except that the vertices u and v have been replaced by a new vertex uv, and that all neighbors of u and v in G are now neighbors of uv in G'; the weight of the edge (uv, z) in G' is the sum of the weights of the edges (u, z) and (v, z) in G (if any).

7.2.1 Algorithm

The algorithm is as follows.

```
algorithm Contract(G=(V,E): graph)
    while |V| > 2 do
       pick at random an edge (u,v) ∈ E
          // with probability proportional to weight(u,v)
       (G,w) = contract(G,w,(u,v))
    return names of two remaining vertices in G
```

Basically, the algorithm keeps contracting random edges, so that, after a few rounds, the graph just contains a few vertices that corresponds to sets of vertices in the original graph. When only two "macro-vertices" are left, they represent a cut, and such a cut is returned by the algorithm.

7.2.2 Analysis

Our strategy will be the following. We fix a min-cut $(S, V - S)$ in the input graph G, and we argue that there is a reasonable probability that the algorithm converges to that particular min-cut.

In order for the algorithm to converge to $(S, V - S)$, it must always contract edges that do not cross the final cut: in fact, this is a necessary and sufficient condition.

Suppose the cost of a min-cut in G is k. Then it follows that, for every vertex v in G, the sum of the weight of the edges incident on v is at least k. (Otherwise, the cut $(\{v,\}, V - \{v\})$ would be better than the optimum.) Consider now the expression $\sum_{v \in V} \sum_{(v,z) \in E} weight(v, z)$. On the one hand, by the above argument, we have

$$\sum_{v \in V} \sum_{(v,z) \in E} weight(v, z) \geq \sum_{v \in V} k = n \cdot k$$

where $n = |V|$; on the other hand, the expression is counting twice the weight of every edge, so we have

$$\sum_{v \in V} \sum_{(v,z) \in E} weight(v, z) = 2 \sum_{(v,z) \in E} weight(v, z)$$

Now, the probability of picking an edge is proportional to its weight, the total weight of the cut $(S, V - S)$ is k, the total weight of all edges is at least $kn/2$, it follows that there is at least a probability $1 - 2/n$ of doing well in the first step of the algorithm.

If we think about it, we quickly realize that, if the algorithm does well in the first pick, at the second stage it has a graph with $n - 1$ nodes where the cut $(S, V - S)$ still is optimum and has cost k. A reasoning similar to the one above will give us that there is a probability at least $1 - 2/(n - 1)$ that we do well at the second step.

Overall, we can see that the probability of converging to the cut $(S, V - S)$ is at least

$$\left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-2}\right)\cdots\left(1-\frac{2}{4}\right)\left(1-\frac{2}{3}\right) \tag{7.1}$$

We can write Expression 7.1 slightly differently as

$$\frac{(n-2)}{n}\cdot\frac{(n-3)}{n-1}\cdot\frac{(n-4)}{n-2}\cdot\frac{(n-5)}{n-3}\cdots\frac{3}{5}\cdot\frac{2}{4}\cdot\frac{1}{3}$$

which is clearly equal to $2/n(n-1) > 2/n^2$.

Now, if we repeat the algorithm $n^2/2$ times, the probability of finding the cut $(S, V-S)$ at least once is at least

$$\left(1-\frac{2}{n^2}\right)^{n^2/2} \approx 1/e$$

and if we repeat the algorithm $100n^2$ times, the probability of finding the right cut at least once becomes very close to 1. What we will do, then, is to repeat the algorithm $100n^2$ times, and then take the solution of minimum cost. At least one solution will be optimal, and so we are done.

Notice that, in one run of the algorithm, we have a probability at least $2/n^2$ to converge to each fixed optimum cut: this means that there are never more than $n^2/2$ distinct optimum cuts. (This is in contrast to the case of the minimum spanning tree problem, where, if the weights are not all different, there can be an exponential number of minimal trees.)

The `contract` operator can be implemented in $O(n)$ time, the operator is invoked n times in each iteration of the basic algorithm, and we have $O(n^2)$ iterations of the basic algorithm. In total, the running time is $O(n^4)$. The best known min-cut algorithm based on the `contract` operator and on the above ideas runs in $O(n^2(\log n)^{O(1)})$ time.

Chapter 8

Union-Find Data Structures

8.1 Disjoint Set Union-Find

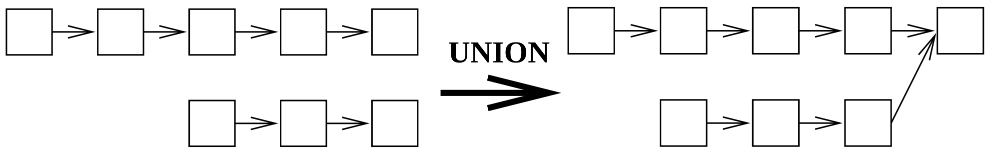
Kruskal's algorithm for finding a minimum spanning tree used a structure for maintaining a collection of disjoint sets. Here, we examine efficient implementations of this structure. It supports the following three operations:

- MAKESET(x) - create a new set containing the single element x.
- UNION(x,y) - replace the two sets containing x and y by their union.
- FIND(x) - return the name of the set containing the element x. For our purposes this will be a *canonical element* in the set containing x.

We will consider how to implement this efficiently, where we measure the cost of doing an *arbitrary* sequence of m UNION and FIND operations on n initial sets created by MAKESET. The minimum possible cost would be $O(m+n)$, i.e. cost $O(1)$ for each call to MAKESET, UNION, or FIND. Our ultimate implementation will be nearly this cheap, and indeed be this cheap for all practical values of m and n.

The simplest implementation one could imagine is to represent each set as a linked list, where we keep track of both the head and the tail. The canonical element is the tail of the list (the final element reached by following the pointers in the other list elements), and UNION simply concatenates lists. In this case FIND has maximal cost proportional to the length of the list, since following each pointer costs $O(1)$, and UNION has cost $O(1)$, to point the tail of one set to the head of the other. The worst case cost is attained by doing n UNIONs, to get a single set, and then m FINDs on the head of the list, for a total cost of $O(mn)$, much larger than our target $O(m+n)$.

To do a better job, we need a more clever data structure. Let us think about how to improve the above simple one. First, instead of taking the union by concatenating lists, we simply make the tail of one list point to the tail of the other, as illustrated below. That way the maximum cost of FIND on any element of the union will have cost proportional to the maximum of the two list lengths (plus one, if both have the same length), rather than the sum.



More generally, we see that a sequence of UNIONs will result in a tree representing each set, with the root of the tree as the canonical element. To simplify coding, we will mark the root by setting the pointer in the root to point to itself. This leads to the following *initial* implementations of MAKESET and FIND:

```
procedure MAKESET(x) ...  initial implementation
  p(x) := x

function FIND(x) ...  initial implementation
  if x ≠ p(x) then return FIND(p(x))
  else return x
```

It is convenient to add a fourth operation LINK(x,y) where x and y are required to be two roots. LINK changes the parent pointer of one of roots, say x, and makes it point to y. It returns the root of the composite tree y. Then UNION(x,y) = LINK(FIND(x), FIND(y)).

But this by itself is not enough to reduce the cost; if we are so unlucky as to make the root of the bigger tree point to the root of the smaller tree, n UNION operations can still lead to a single chain of length n, and the same cost as above.

This motivates the first of our two heuristics: UNION BY RANK. This simply means that we keep track of the depth (or RANK) of each tree, and make the shorter tree point to the root of the taller tree; code is shown below. Note that if we take the UNION of two trees of the same RANK, the RANK of the UNION is one larger than the common RANK, and otherwise equal to the max of the two RANKs. This will keep the RANK of tree of n nodes from growing past $O(\log n)$, but m UNIONs and FINDs can then still cost $O(mlogn)$.

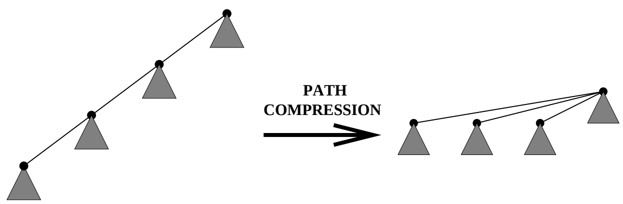
```
procedure MAKESET(x) ...  final implementation
  p(x) := x
  RANK(x) := 0

function LINK(x,y)
  if RANK(x) > RANK(y) then swap x and y
  if RANK(x) = RANK(y) then RANK(y) = RANK(y) + 1
  p(x) := y
  return(y)
```

The second heuristic, PATH COMPRESSION, is motivated by observing that since each FIND operation traverses a linked list of vertices on the way to the root, one could make *later* FIND operations cheaper by making each of these vertices point directly to the root:

```
function FIND(x) ...  final implementation
  if x ≠ p(x) then
    p(x) := FIND(p(x))
  return(p(x))
```

```
else return(x)
```



We will prove below that any sequence of m UNION and FIND operations on n elements take at most $O((m+n)\log^* n)$ steps, where $\log^* n$ is the number of times you must iterate the log function on n before you get a number less than or equal to 1. Recall that $\log^* n <= 5$ for all $n \leq 2^{2^{16}} = 2^{65536} \approx 10^{19728}$. Since the number of atoms in the universe is estimated to be at most 10^{80}, which is a conservative upper bound on the size of any computer memory (as long each bit is at least the size of an atom), it is unlikely that you will ever have a graph with this many vertices, so $\log^* n \leq 5$ in practice.

8.2 Analysis of Union-Find

Suppose we initialize the data structure with n `makeset` operations, so that we have n elements each forming a different set of size 1, and let us suppose we do a sequence of k operations of the type `union` or `find`. We want to get a bound on the total running time to perform the k operations. Each `union` performs two `find` and then does a constant amount of extra work. So it will be enough to get a bound on the running time needed to perform $m \leq 2k$ `find` operations.

Let us consider at how the data structure looks at the end of all the operations, and let us see what is the rank of each of the n elements. First, we have the following result.

Lemma 13
If an element has rank k, then it is the root of a subtree of size at least 2^k.
Proof: An element of rank 0 is the root of a subtree that contains at least itself (and so is of size at least 1). An element u can have rank $k+1$ only if, at some point, it had rank k and it was the root of a tree that was joined with another tree whose root had rank k. Then u became the root of the union of the two trees. Each tree, by inductive hypothesis was of size at least 2^k, and so now u is the root of a tree of size at least 2^{k+1}. □

Let us now group our n elements according to their final rank. We will have a group 0 that contains elements of rank 0 and 1, group 1 contains elements of rank 2, group 2 contains elements of rank in the range $\{3,4\}$, group 3 contains elements of rank between 5 and 16, group 4 contains elements of rank between 17 and 2^{16} and so on. (In practice, of course, no element will belong to group 5 or higher.) Formally, each group contains elements of rank in the range $(k, 2^k]$, where k itself is a power of a power ... of a power of 2. We can see that these groups become sparser and sparser.

Lemma 14
No more than $n/2^k$ elements have rank in the range $(k, 2^k]$.
Proof: We have seen that if an element has rank r, then it is the root of a subtree of size at least 2^r. It follows that there cannot be more than $n/2^r$ elements of rank r. The total

number of elements of rank between $k+1$ and 2^k is then at most

$$n \sum_{r=k+1}^{2^k} \frac{1}{2^r} < n \sum_{r=k+1}^{\infty} \frac{1}{2^r} = \frac{n}{2^k} \sum_{i=1}^{\infty} \frac{1}{2^i} = \frac{n}{2^k}$$

□

By definition, there are no more than $\log^* n$ groups.

To compute the running time of our m operations, we will use the following trick. We will assign to each element u a certain number of "tokens," where each token is worth $O(1)$ running time. We will give out a total of $n \log^* n$ tokens.

We will show that each `find` operation takes $O(\log^* n)$ time, plus some additional time that is paid for using the tokens of the vertices that are visited during the `find` operation. In the end, we will have used at most $O((m+n)\log^* n)$ time.

Let us define the token distribution. If an element u has (at the end of the m operations) rank in the range $(k, 2^k]$ then we will give (at the beginning) 2^k tokens to it.

Lemma 15

We are distributing a total of at most $n \log^* n$ tokens.

Proof: Consider the group of elements of rank in the range $(k, 2^k]$: we are giving 2^k tokens to them, and there are at most $n/2^k$ elements in the group, so we are giving a total of n tokens to that group. In total we have at most $\log^* n$ groups, and the lemma follows. □

We need one more observation to keep in mind.

Lemma 16

At any time, for every u that is not a root, $rank[u] < rank[p[u]]$.

Proof: After the initial series of `makeset`, this is an invariant that is maintained by each `find` and each `union` operation. □

We can now prove our main result

Theorem 17

Any sequence of operations involving m `find` operations can be completed in $O((m+n)\log^* n)$ time.

Proof: Apart from the work needed to perform `find`, each operation only requires constant time (for a total of $O(m)$ time). We now claim that each `find` takes $O(\log^* n)$ time, plus time that is paid for using tokens (and we also want to prove that we do not run out of tokens).

The accounting is done as follows: the running time of a `find` operation is a constant times the number of pointers that are followed until we get to the root. When we follow a pointer from u to v (where $v = p[u]$) we charge the cost to `find` if u and v belong to different groups, or if u is a root, or if u is a child of a root; and we charge the cost to u if u and v are in the same group (charging the cost to u means removing a token from u's allowance) . Since there are at most $\log^* n$ groups, we are charging only $O(\log^* n)$ work to `find`. How can we make sure we do not run out of coins? When `find` arrives at a node u and charges u, it will also happen that u will move up in the tree, and become a child of the root (while previously it was a grand-child or a farther descendent); in particular, u now points to a vertex whose rank is larger than the rank of the vertex it was pointing to before. Let k be such that u belongs to the range group $(k, 2^k]$, then u has 2^k coins at the beginning. At any time, u either points to itself (while it is a root) or to a vertex of higher rank. Each time u is charged by a `find` operation, u gets to point to a parent node of higher and higher rank. Then u cannot be charged more than 2^k time, because after that the parent of u will move to another group. $\square$

Chapter 9

Dynamic Programming

9.1 Introduction to Dynamic Programming

Recall our first algorithm for computing the n-th Fibonacci number F_n; it just recursively applied the definition $F_n = F_{n-1} + F_{n-2}$, so that a function call to compute F_n resulted in two functions calls to compute F_{n-1} and F_{n-2}, and so on. The problem with this approach was that it was very expensive, because it ended up calling a function to compute F_j for each $j < n$ possibly very many times, even after F_j had already been computed. We improved this algorithm by building a table of values of Fibonacci numbers, computing F_n by looking up F_{n-1} and F_{n-2} in the table and simply adding them. This lowered the cost of computing F_n from exponential in n to just linear in n.

This worked because we could sort the problems of computing F_n simply by increasing n, and compute and store the Fibonacci numbers with small n before computing those with large n.

Dynamic programming uses exactly the same idea:

1. Express the solution to a problem in terms of solutions to smaller problems.

2. Solve all the smallest problems first and put their solutions in a table, then solve the next larger problems, putting their solutions into the table, solve and store the next larger problems, and so on, up to the problem one originally wanted to solve. Each problem should be easily solvable by looking up and combining solutions of smaller problems in the table.

For Fibonacci numbers, how to compute F_n in terms of smaller problems F_{n-1} and F_{n-2} was obvious. For more interesting problems, figuring out how to break big problems into smaller ones is the tricky part. Once this is done, the the rest of algorithm is usually straightforward to produce. We will illustrate by a sequence of examples, starting with "one-dimensional" problems that are most analogous to Fibonacci.

9.2 String Reconstruction

Suppose that all blanks and punctuation marks have been inadvertently removed from a text file, and its beginning was polluted with a few extraneous characters, so the file looks something like "lionceuponatimeinafarfarawayland..." You want to reconstruct the file using a dictionary.

This is a typical problem solved by dynamic programming. We must define what is an appropriate notion of *subproblem*. Subproblems must be ordered by *size*, and each subproblem must be easily solvable, once we have the solutions to all smaller subproblems. Once we have the right notion of a subproblem, we write the appropriate recursive equation expressing how a subproblem is solved based on solutions to smaller subproblems, and the program is then trivial to write. The complexity of the dynamic programming algorithm is precisely the total number of subproblems times the number of smaller subproblems we must examine in order to solve a subproblem.

In this and the next few examples, we do dynamic programming on a one-dimensional object—in this case a string, next a sequence of matrices, then a set of strings alphabetically ordered, etc. The basic observation is this: *A one-dimensional object of length n has about n^2 sub-objects* (substrings, etc.), where a sub-object is defined to span the range from i to j, where $i, j \leq n$. In the present case a subproblem is to tell whether the substring of the file from character i to j is the concatenation of words from the dictionary. Concretely, let the file be $f[1 \ldots n]$, and consider a 2-D array of Boolean variables $T(i,j)$, where $T(i,j)$ is true if and only if the string $f[i \ldots j]$ is the concatenation of words from the dictionary. The recursive equation is this:

$$T(i,j) = dict(x[i \ldots j]) \vee \bigvee_{i \leq k < j} [T(i,k) \wedge T(k+1,j)]$$

In principle, we could write this equation verbatim as a recursive function and execute it. The problem is that there would be *exponentially many* recursive calls for each short string, and 3^n calls overall.

Dynamic programming can be seen as a technique of implementing such recursive programs, that have heavy overlap between the recursion trees of the two recursive calls, so that the recursive function is called once for each distinct argument; indeed the recursion is usually "unwound" and disappears altogether. This is done by modifying the recursive program so that, in place of each recursive call a table is consulted. To make sure the needed answer is in the table, we note that the lengths of the strings on the right hand side of the equation above are $k-i+1$ and $j-k$, a both of which are shorter than the string on the left (of length $j-i+1$). This means we can fill the table in *increasing order of string length.*

for $d := 0$ to $n-1$ do ... $d+1$ is the size (string length) of the subproblem being solved
  for $i := 1$ to $n-d$ do ... the start of the subproblem being solved
  $j = i + d$
  if $dict(x[i \ldots j])$ then $T(i,j)$:=true else
    for $k := i$ to $j-1$ do
    if $T(i,k)$ =true and $T(k+1,j)$ =true then do $\{T(i,j)$:=true$\}$

The complexity of this program is $O(n^3)$: three nested loops, ranging each roughly over n values.

Unfortunately, this program just returns a meaningless Boolean, and does not tell us how to reconstruct the text. Here is how to reconstruct the text. Just expand the innermost loop (the last assignment statement) to

{$T[i,j]$:=true, first$[i,j]$:= k, exit for}

where first is an array of pointers initialized to *nil*. Then if $T[i,j]$ is true, so that the substring from i to j is indeed a concatenation of dictionary words, then first$[i,j]$ points to a breaking point in the interval i,j. Notice that this improves the running time, by exiting the for loop after the first match; more optimizations are possible. This is typical of dynamic programming algorithms: Once the basic algorithm has been derived using dynamic programming, clever modifications that exploit the structure of the problem speed up its running time.

9.3 Edit Distance

9.3.1 Definition

When you run a spell checker on a text, and it finds a word not in the dictionary, it normally proposes a choice of possible corrections.

If it finds `stell` it might suggest `tell`, `swell`, `stull`, `still`, `steel`, `steal`, `stall`, `spell`, `smell`, `shell`, and `sell`.

As part of the heuristic used to propose alternatives, words that are "close" to the misspelled word are proposed. We will now see a formal definition of "distance" between strings, and a simple but efficient algorithm to compute such distance.

The distance between two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is the minimum number of "errors" (edit operations) needed to transform x into y, where possible operations are:

- insert a character.
  $insert(x, i, a) = x_1 x_2 \cdots x_i a x_{i+1} \cdots x_n$.
- delete a character.
  $delete(x, i) = x_i x_2 \cdots x_{i-1} x_{i+1} \cdots x_n$.
- modify a character.
  $modify(x, i, a) = x_1 x_2 \cdots x_{i-1} a x_{i+1} \cdots x_n$.

For example, if $x = aabab$ and $y = babb$, then one 3-steps way to go from x to y is

a	a	b	a	b		x
b	a	a	b	a	b	x' = insert(x,0,b)
b	a	b	a	b		x" = delete (x',2)
b	a	b	b			y = delete (x",4)

another sequence (still in three steps) is

a	a	b	a	b	x
a	b	a	b		x' = delete (x,1)
b	a	b			x" = delete(x',1)
b	a	b	b		y = insert (x",3,b)

Can you do better?

9.3.2 Computing Edit Distance

To transform $x_1 \cdots x_n$ into $y_1 \cdots y_m$ we have three choices:

- put y_m at the end: $x \to x_1 \cdots x_n y_m$ and then transform $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.
- delete x_n: $x \to x_1 \cdots x_{n-1}$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.
- change x_n into y_m (if they are different): $x \to x_1 \cdots x_{n-1} y_m$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$.

This suggests a recursive scheme where the sub-problems are of the form "how many operations do we need to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

Our dynamic programming solution will be to define a $(n+1) \times (m+1)$ matrix $M[\cdot,\cdot]$, that we will fill so that for every $0 \leq i \leq n$ and $0 \leq j \leq m$, $M[i,j]$ is the minimum number of operations to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

The content of our matrix M can be formalized recursively as follows:

- $M[0,j] = j$ because the only way to transform the empty string into $y_1 \cdots y_j$ is to add the j characters $y_1, \ldots, y_j$.
- $M[i,0] = i$ for similar reasons.
- For $i,j \geq 1$,

$$
\begin{aligned}
M[i,j] \quad = \min\{ \quad & M[i-1,j]+1, \\
& M[i,j-1]+1, \\
& M[i-1,j-1]+\mathit{change}(x_i,y_j)\}
\end{aligned}
$$

  where $\mathit{change}(x_i,y_j) = 1$ if $x_i \neq y_j$ and $\mathit{change}(x_i,y_j) = 0$ otherwise.

As an example, consider again $x = aabab$ and $y = babb$

	λ	b	a	b	b
λ	0	1	2	3	4
a	1	1	1	2	3
a	2	2	1	2	3
b	3	2	2	1	2
a	4	3	2	2	2
b	5	4	3	2	2

What is, then, the edit distance between x and y?

The table has $\Theta(nm)$ entries, each one computable in constant time. One can construct an auxiliary table $Op[\cdot,\cdot]$ such that $Op[\cdot,\cdot]$ specifies what is the first operation to do in order to optimally transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$. The full algorithm that fills the matrices can be specified in a few lines

```
algorithm EdDist(x,y)
    n = length(x)
    m = length(y)
    for i = 0 to n
```

```
  M[i,0] = i
for j = 0 to m
  M[0,j] = j
for i = 1 to n
  for j = 1 to m
    if x_i == y_j then change = 0 else change = 1
    M[i,j] = M[i-1,j] + 1; Op[i,j] = delete(x,i)
    if M[i,j-1] + 1 < M[i,j] then
      M[i,j] = M[i,j-1] + 1; Op[i,j] = insert(x,i,y_j)
    if M[i-1,j-1] + change < M[i,j] then
      M[i,j] = M[i-1,j-1] + change
      if (change == 0) then Op[i,j] = none
      else Op[i,j] = change(x,i,y_j)
```

9.4 Longest Common Subsequence

A *subsequence* of a string is obtained by taking a string and possibly deleting elements.

If $x_1 \cdots x_n$ is a string and $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ is a strictly increasing sequence of indices, then $x_{i_1} x_{i_2} \cdots x_{i_k}$ is a subsequence of x. For example, **art** is a subsequence of **a**lgo**r**i**t**hm.

In the *longest common subsequence problem*, given strings x and y we want to find the longest string that is a subsequence of both.

For example, **art** is the longest common subsequence of **a**lgo**r**i**t**hm and p**ar**achu**t**e.

As usual, we need to find a recursive solution to our problem, and see how the problem on strings of a certain length can be reduced to the same problem on smaller strings.

The length of the l.c.s. of $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is either

- The length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_m$ or;
- The length of the l.c.s. of $x_1 \cdots x_n$ and $y_1 \cdots y_{m-1}$ or;
- 1 + the length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_{m-1}$, if $x_n = y_m$.

The above observation shows that the computation of the length of the l.c.s. of x and y reduces to problems of the form "what is the length of the l.c.s. between $x_1 \cdots x_i$ and $y_1 \cdots y_i$?"

Our dynamic programming solution uses an $(n+1) \times (m+1)$ matrix M such that for every $0 \leq i \leq n$ and $0 \leq j \leq m$, $M[i,j]$ contains the length of the l.c.s. between $x_1 \cdots x_i$ and $y_1 \cdots y_j$. The matrix has the following formal recursive definition

- $M[i,0] = 0$
- $M[0,j] = 0$

•

$$
\begin{aligned}
M[i,j] \quad = \max\{ \quad & M[i-1,j] \\
& M[i,j-1] \\
& M[i-1,j-1] + eq(x_i,y_j)\}
\end{aligned}
$$

where $eq(x_i, y_j) = 1$ if $x_i = y_j$, $eq(x_i, y_j) = 0$ otherwise.

The following is the content of the matrix for the words `algorithm` and `parachute`.

	λ	p	a	r	a	c	h	u	t	e
λ	0	0	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1	1	1
l	0	0	1	1	1	1	1	1	1	1
g	0	0	1	1	1	1	1	1	1	1
o	0	0	1	1	1	1	1	1	1	1
r	0	0	1	2	2	2	2	2	2	2
i	0	0	1	2	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	3	3
h	0	0	1	2	2	2	3	3	3	3
m	0	0	1	2	2	2	3	3	3	3

The matrix can be filled in $O(nm)$ time. How do you reconstruct the longest common substring given the matrix?

9.5 Chain Matrix Multiplication

Suppose that you want to multiply four matrices $A \times B \times C \times D$ of dimensions 40×20, 20×300, 300×10, and 10×100, respectively. Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications (a good enough estimate of the running time).

To multiply these matrices as $(((A \times B) \times C) \times D)$ takes $40 \cdot 20 \cdot 300 + 40 \cdot 300 \cdot 10 + 40 \cdot 10 \cdot 100 = 380,000$. A more clever way would be to multiply them as $(A \times ((B \times C) \times D))$, with total cost $20 \cdot 300 \cdot 10 + 20 \cdot 10 \cdot 100 + 40 \cdot 20 \cdot 100 = 160,000$. An even better order would be $((A \times (B \times C)) \times D)$ with total cost $20 \cdot 300 \cdot 10 + 40 \cdot 20 \cdot 10 + 40 \cdot 10 \cdot 100 = 108,000$. Among the five possible orders (the five possible binary trees with four leaves) this latter method is the best.

How can we automatically pick the best among all possible orders for multiplying n given matrices? Exhaustively examining all binary trees is impractical: There are $C(n) = \frac{1}{n}\binom{2n-2}{n-1} \approx \frac{4^n}{n\sqrt{n}}$ such trees ($C(n)$ is called the *Catalan* number of n). Naturally enough, dynamic programming is the answer.

Suppose that the matrices are $A_1 \times A_2 \times \cdots \times A_n$, with dimensions, respectively, $m_0 \times m_1, m_1 \times m_2, \ldots m_{n-1} \times m_n$. Define a *subproblem* (remember, this is the most crucial and nontrivial step in the design of a dynamic programming algorithm; the rest is usually automatic) to be to multiply the matrices $A_i \times \cdots \times A_j$, and let $M(i,j)$ be the optimum number of multiplications for doing so. Naturally, $M(i,i) = 0$, since it takes no effort to multiply a chain consisting just of the i-th matrix. The recursive equation is

$$M(i,j) = \min_{i \le k < j} [M(i,k) + M(k+1,j) + m_{i-1} \cdot m_k \cdot m_j].$$

This equation defines the program and its complexity—$O(n^3)$.

for $i := 1$ to n do $M(i,i) := 0$
    for $d := 1$ to $n-1$ do
      for $i := 1$ to $n-d$ do
        $j = i + d, M(i,j) = \infty$, best$(i,j)$:=nil
        for $k := i$ to $j-1$ do
          if $M(i,j) > M(i,k) + M(k+1,j) + m_{i-1} \cdot m_k \cdot m_j$ then
            $M(i,j) := M(i,k) + M(k+1,j) + m_{i-1} \cdot m_k \cdot m_j$, best$(i,j) := k$

As usual, improvements are possible (in this case, down to $O(n \log n)$).

Run this algorithm in the simple example of four matrices given to verify that the claimed order is the best!

9.6 Knapsack

A burglar breaks into a house, and finds n objects that he may want to steal. Object i has weight v_i and costs about c_i dollars, and the burglar knows he cannot carry a weight larger than B. What is the set of objects of larger total value, subject to the constraint that their total weight is less than B? (In an alternative, legal, formulation, you are packing your knapsack for a hike, the knapsack has volume B, and there are i items that you may want to pack, item i has volume v_i and its "usefulness" is c_i; what is the set of items of larger total usefulness that will fit into the knapsack?)

Formally, the KNAPSACK problem is as follows:

Given: n items of "cost" $c_1, c_2, \ldots, c_n$ (positive integers), and of "volume" $v_1, v_2, \ldots, v_n$ (positive integers); a volume value B (for *b*ound).

Find a subset $S \subseteq \{1, \ldots, n\}$ of the items such that

$$\sum_{i \in S} v_i \le B \tag{9.1}$$

and such that the total cost $\sum_{i \in S} c_i$ is maximized.

For reasons that will be clear later in the course, there is probably no algorithm for this problem that runs in time polynomial in the length of the input (which is about $n \log B$), however there is an algorithm that runs in time polynomial in n and B.

We want to solve this problem using dynamic programming, so we should think of how to reduce it to a smaller problem. Let us think of whether item n should or should not be in the optimal solution. If item n is not in the optimal solution, then the optimal solution is the optimal solution for the same problem but only with items $1, \ldots, n-1$; if item n is in the optimal solution, then it leaves $B - v_n$ units of volume for the other items, which means that the optimal solution contains item n, and then contains the optimal solution of the same problem on items $1, \ldots, n-1$ and volume bound $B - v_n$.

This recursion generates subproblems of the form "what is the optimal solution that uses only items $1, \ldots, i$ and volume B'"?

Our dynamic programming algorithm constructs a $n \times (B+1)$ table $M[\cdot, \cdot]$, where $M[k, B']$ contains the cost of an optimum solution for the instance that uses only a subset of the first k elements (of volume $v_1, \ldots, v_k$ and cost $c_1, \ldots, c_k$) and with volume B'. The recursive definition of the matrix is

- $M[1, B'] = c_1$ if $B' \geq v_1$; $M[1, B'] = 0$ otherwise.
- for every $k, B' \geq 1$,

$$M[k, B'] = \max\{M[k-1, B']\ ,\ M[k-1, B' - v_k] + c_k\}$$

The matrix can be filled in $O(Bn)$ time (constant time per entry), and, at the end, the cost of the optimal solution for our input is reported in $M[n, B]$.

To *construct* an optimum solution, we also build a Boolean matrix $C[\cdot, \cdot]$ of the same size.

For every i and B', $C[i, B'] = True$ if and only if there is an optimum solution that packs a subset of the first i items in volume B' so that item i *is* included in the solution.

Let us see an example. Consider an instance with 9 items and a bag of size 15. The costs and the volumes of the items are as follows:

Item	1	2	3	4	5	6	7	8	9
Cost	2	3	3	4	4	5	7	8	8
Volume	3	5	7	4	3	9	2	11	5

This is table M

B'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$k=9$	0	0	7	7	7	11	11	15	15	15	19	19	19	21	23	23
$k=8$	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
$k=7$	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
$k=6$	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
$k=5$	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
$k=4$	0	0	0	2	4	4	4	6	6	7	7	7	9	9	9	9
$k=3$	0	0	0	2	2	3	3	3	5	5	5	5	6	6	6	8
$k=2$	0	0	0	2	2	3	3	3	5	5	5	5	5	5	5	5
$k=1$	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2

And this is table C.

B'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$k=9$	0	0	0	0	0	0	0	1	1	0	1	1	1	1	1	1
$k=8$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$k=7$	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$k=6$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$k=5$	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1
$k=4$	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
$k=3$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
$k=2$	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
$k=1$	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

The solution is

Optimum: 23
Item 9 Cost 8 Volume 5
Item 7 Cost 7 Volume 2
Item 5 Cost 4 Volume 3
Item 4 Cost 4 Volume 4

The algorithm that fills the matrices is as follows.

`algorithm Knapsack(B,n,c[],v[])`
    `for` $b = 0$ `to` B
        `if` $(v[1] \leq b)$ `then`
            $M[1,b] = c[1]$; $C[1,b] = true$;
        `else`
            $M[1,b] = 0$; $C[1,b] = false$;
    `for` $i = 2$ `to` n
        `for` $b = 0$ `to` B
            `if` $b \geq v[i]$ `and` $M[i-1, b-v[i]] + c[i] > M[i-1,b]$
                $M[i,b] = M[i-1, b-v[i]] + c[i]$; $C[i,b] = true$
            `else`
                $M[i,b] = M[i-1,b]; C[i,b] = false$

9.7 All-Pairs-Shortest-Paths

Now we move on to a different type of dynamic programming algorithm, that is not on a sequence but a general graph: We are given a directed graph with edge weights, and wish to compute the *shortest path from every vertex to every other vertex*. This is also called the *all-pairs-shortest-paths* problem. Recall that the question only makes sense if there are no negative cycles, which we shall henceforth assume.

We already have an algorithm that we can apply to this problem: Bellman-Ford. Recall that Bellman-Ford computes the shortest paths from *one* vertex to all other vertices in $O(nm)$ time, where n is the number of vertices and m is the number of edges. Then we could simply run this algorithm n times, for each vertex in the graph, for a total cost of $O(n^2m)$. Since m varies from $n-1$ (for a connected graph) to n^2-n (for a fully connected graph), the cost varies from $O(n^3)$ to $O(n^4)$.

In this section we will present an algorithm based on dynamic programming that always runs in $O(n^3)$ time, independent of m. To keep the presentation simple, we will only show how to compute the lengths of the shortest paths, and leave their explicit construction as an exercise.

Here is how to construct a sequence of subproblems that can be solved easily using dynamic programming. We start by letting $T^0(i,j)$ be the length of the shortest *direct* path from i to j; this is the length of edge (i,j) if one exists, and ∞ otherwise. We set $T^0(i,i)=0$ for all i.

Now we define problem $T^k(i,j)$ as the shortest path from i to j that is only allowed to use intermediate vertices numbered from 1 to k. For example, $T^1(i,j)$ is the shortest path among the following: $i \to j$ and $i \to 1 \to j$, and $T^2(i,j)$ is the shortest path among the following: $i \to j$, $i \to 1 \to j$, $i \to 2 \to j$, $i \to 1 \to 2 \to j$, and $i \to 2 \to 1 \to j$.

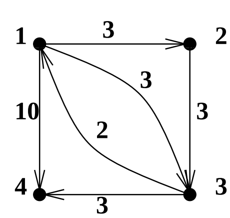
Next, we have to define $T^k(i,j)$ in terms of $T^{k-1}(.,.)$ in order to use dynamic programming. Consider a shortest path from i to j that uses vertices up to k. There are two cases: Either the path contains k, or it does not. If it does not, then $T^k(i,j) = T^{k-1}(i,j)$. If it does contain k, then it will only pass through k once (since it is shortest), and so the shortest path must go from i to k *via vertices* 1 *through* $k-1$, and then from k to j *via vertices* 1 *through* $k-1$. In other words $T^k(i,j) = T^{k-1}(i,k) + T^{k-1}(k,j)$. Putting these two cases together yields

$$T^k(i,j) = \min(T^{k-1}(i,j), T^{k-1}(i,k) + T^{k-1}(k,j)) \ .$$

The algorithm, called the *Floyd-Warshall algorithm*, is now simple:

for $i,j = 1$ to n, $T^0(i,j) =$ length of edge (i,j) if it exists, 0 if $i=j$, and ∞ otherwise
for $k = 1$ to n
    for $i = 1$ to n
        for $j = 1$ to n
            $T^k(i,j) = \min(T^{k-1}(i,j), T^{k-1}(i,k) + T^{k-1}(k,j))$

Here is an example:



$$T^0 = \begin{matrix} 0 & 3 & 3 & 10 \\ \infty & 0 & 3 & \infty \\ 2 & \infty & 0 & 3 \\ \infty & \infty & \infty & 0 \end{matrix} \ , \ T^1 = \begin{matrix} 0 & 3 & 3 & 10 \\ \infty & 0 & 3 & \infty \\ 2 & 5 & 0 & 3 \\ \infty & \infty & \infty & 0 \end{matrix} \ , \ T^2 = T^1 \ , \ T^3 = \begin{matrix} 0 & 3 & 3 & 6 \\ 5 & 0 & 3 & 6 \\ 2 & 5 & 0 & 3 \\ \infty & \infty & \infty & 0 \end{matrix} \ , \ T^4 = T^3$$

Here is a simple variation on this problem. Suppose instead of computing the length of the shortest path between any pair of vertices, we simply wish to know whether a path exists. Given G, the graph $T(G)$ that has an edge from u to v if and only if G has a path from u to v is called the *transitive closure* of G. Here is an example:

$$G = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad T(G) = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

How do we compute $T(G)$? The solution is very similar to Floyd-Warshall. Indeed, we could just run Floyd-Warshall, since the length of the shortest path from u to v is less than ∞ if and only if there is some path from u to v. But here is a simpler solution: Let $\tilde{T}^k(i,j)$ be true if and only if there is a path from i to j using intermediate vertices from among $1, 2, ..., k$ only. The recursive equation defining $T^k(i,j)$ is

$$\tilde{T}^k(i,j) = \tilde{T}^{k-1}(i,j) \vee [\tilde{T}^{k-1}(i,k) \wedge \tilde{T}^{k-1}(k,j)]$$

This is nearly the same as the definition of $T^k(i,j)$ for Floyd-Warshall, with "$\vee$" replacing "min" and "$\wedge$" replacing "+". The dynamic programming algorithm is also gotten from Floyd-Warshall just by changing "min" to "$\vee$" and "+" to "$\wedge$", and initializing $\tilde{T}^0(i,j)$ to true if there is an edge (i,j) in G or $i = j$, and false otherwise.

Chapter 10

Data Compression

10.1 Data Compression via Huffman Coding

Huffman codes are used for data compression. The motivations for data compression are obvious: reducing time to transmit large files, and reducing the space required to store them on disk or tape.

Suppose that you have a file of 100K characters. To keep the example simple, suppose that each character is one of the 8 letters from a through h. Since we have just 8 characters, we need just 3 bits to represent a character, so the file requires 300K bits to store. Can we do better?

Suppose that we have more information about the file: the *frequency* which which each character appears. The idea is that we will use a *variable length code* instead of a *fixed length code* (3 bits for each character), with fewer bits to store the common characters, and more bits to store the rare characters. At one obvious extreme, if only 2 characters actually appeared in the file, we could represent each one with just one bit, and reduce the storage from 300K bits to 100K bits (plus a short header explaining the encoding). It turns out that all characters can appear, but that as long as each one does not appear nearly equally often (100K/8 times in our case), then we can probably save space by encoding.

For example, suppose that the characters appear with the following frequencies, and following codes:

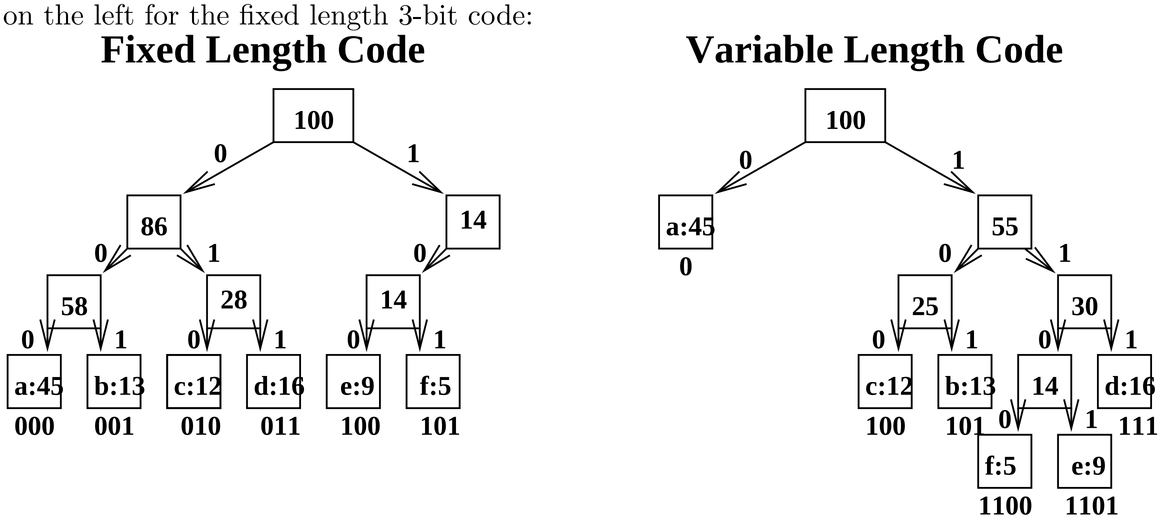
	a	b	c	d	e	f	g	h
Frequency	45K	13K	12K	16K	9K	5K	0K	0K
Fixed-length code	000	001	010	011	100	101	110	111
Variable-length code	0	101	100	111	1101	1100	—	—

Then the variable-length coded version will take not 300K bits but $45\text{K} \cdot 1 + 13\text{K} \cdot 3 + 12\text{K} \cdot 3 + 16\text{K} \cdot 3 + 9\text{K} \cdot 4 + 5\text{K} \cdot 4 = 224\text{K}$ bits to store, a 25% saving. In fact this is the optimal way to encode the 6 characters present, as we shall see.

We will consider only codes in which no code is a prefix of any other code; such codes are called *prefix codes* (though perhaps they should be called prefix-free codes). The attraction of such codes is that it is easy to encode and decode data. To encode, we need only concatenate the codes of consecutive characters in the message. So for example "face" is encoded as "110001001101". To decode, we have to decide where each code begins and ends, since they are no longer all the same length. But this is easy, since, no codes share a prefix. This means we need only scan the input string from left to right, and as soon as we recognize a code, we can print the corresponding character and start looking for the next code. In the above case, the only code that begins with "1100..." or a prefix is "f", so we can print "f" and start decoding "0100...", get "a", etc.

To see why the no-common prefix property is essential, suppose that we tried to encode "e" with the shorter code "110", and tried to decode "1100"; we could not tell whether this represented "ea" or "f". (Furthermore, one can show that one cannot compress any better with a non-prefix code, although we will not show this here.)

We can represent the decoding algorithm by a binary tree, where each edge represents either 0 or 1, and each leaf corresponds to the sequence of 0s and 1s traversed to reach it, ie a particular code. Since no prefix is shared, all legal codes are at the leaves, and decoding a string means following edges, according to the sequence of 0s and 1s in the string, until a leaf is reached. The tree for the above code is shown on the right below, along with a tree on the left for the fixed length 3-bit code:



Each leaf is labeled by the character it represents (before the colon), as well as the frequency with which it appears in the text (after the colon, in 1000s). Each internal node is labeled by the frequency with which all leaf nodes under it appear in the text (ie the sum of their frequencies). The bit string representing each character is also shown beneath each leaf.

We will denote the set of all characters by X, an arbitrary character by $x \in X$, the frequency with with x appears by $f(x)$, and its depth in the tree by $d(x)$. Note that $d(x)$ is the number of bits in the code for x.

Given a tree T representing a prefix code, it is easy to compute the number of bits needed to represent a file with the given frequencies $f(x)$: $B(T) = \sum_{x \in X} f(x)d(x)$, which we call the *cost of* T.

The greedy algorithm for computing the optimal Huffman coding tree T given the character frequencies $f(x)$ is as follows. It starts with a forest of one-node trees representing each $x \in X$, and merges them in a greedy style reminiscent of Prim's MST algorithm, using a priority queue Q, sorted by the *smallest* frequency:

procedure Huffman(X, $f(.)$)
$n = |X|$, the number of characters
for all $x \in X$, enqueue$((x, f(x)), Q)$
for $i = 1$ to n
    allocate a new tree node z
    $left\_child$ =deletemin(Q)
    $right\_child$ =deletemin(Q)
    $f(z) = f(left\_child) + f(right\_child)$
    Make $left\_child$ and $right\_child$ the children of z
    enqueue$((z, f(z)), Q)$

The cost of this algorithm is clearly the cost of $2n$ deletemins and n enqueues onto the priority queue Q. Assuming Q is implemented with a binary heap, so that these operations cost $O(\log n)$, the whole algorithm costs $O(n \log n)$, or as much as sorting.

Here is why it works, i.e. produces the tree T minimizing $B(T)$ over all possible trees. We will use induction on the number of characters in X. When $|X| = 2$, the optimal tree clearly has two leaves, corresponding to strings 0 and 1, which is what the algorithm constructs. Now suppose $|X| > 2$. The first thing the algorithm does is make the two lowest-frequency characters (call them x and y) into leaf nodes, create a new node z with frequency $f(z)$ equal to the sum of their frequencies, and apply the algorithm to the new set $\bar{X} = X - \{x, y\} \cup \{z\}$, which has $|\bar{X}| = |X| - 1$ characters, so we can apply induction. Thus, we can assume that the algorithm builds an optimal tree $\bar{T}$ for $\bar{X}$.

The trick is to show that T, gotten by adding x and y as left and right children to z in $\bar{T}$, is also optimal. We do this by contradiction: suppose there is a better tree T' for X, i.e. with a cost $B(T') < B(T)$. Then we will show there is a better tree $\bar{T}'$ for $\bar{X}$, ie with $B(\bar{T}') < B(\bar{T})$, contradicting the optimality of $\bar{T}$.

First note that

$$\begin{aligned} B(T) &= B(\bar{T}) - f(z)d(z) + f(x)d(x) + f(y)d(y) \\ &= B(\bar{T}) - (f(x) + f(y))(d(x) - 1) + f(x)d(x) + f(y)d(x) \\ &= B(\bar{T}) + f(x) + f(y) \quad . \end{aligned}$$

Similarly, if x and y are siblings in T', we can build the tree $\bar{T}'$ by removing x and y from T' and making their parent a leaf. The same argument shows that $B(T') = B(\bar{T}') + f(x) + f(y)$. Since this is less than $B(T) = B(\bar{T}) + f(x) + f(y)$, we get $B(\bar{T}') < B(\bar{T})$, the contradiction we wanted.

When x and y are *not* siblings in T', we have to work harder: we will show that there is *another* optimal tree T'' where they *are* siblings, so we can apply the above argument with T'' in place of T'. To construct T'' from T', let b and c be the two siblings of maximum depth $d(b) = d(c)$ in T'. Since all leaf nodes have siblings by construction, we have $d(b) = d(c) \geq d(z)$ for any other leaf z. Suppose without loss of generality that $f(x) \leq f(y)$ and $f(b) \leq f(c)$ (if this is not true, swap the labels x and y, or b and c). Then $f(x) \leq f(b)$ and $f(y) \leq f(c)$. Then let T'' be the tree with b and x swapped, and c and y swapped. Clearly x and y are siblings in T''. We need to show that $B(T'') \leq B(T')$; since we know that $B(T')$ is minimal, this will mean that $B(T'') = B(T')$ is minimal as well. We compute as follows (the depths $d(.)$ refer to T'):

$$\begin{aligned}
B(T'') &= B(T') + [-f(b)d(b) - f(x)d(x) + f(b)d(x) + f(x)d(b)] \\
&\quad +[-[f(c)d(c) - f(y)d(y) + f(c)d(y) + f(y)d(c)] \\
&= B(T') - [(f(b) - f(x)) \cdot (d(b) - d(x))] - [(f(c) - f(y)) \cdot (d(c) - d(y))] \\
&\leq B(T')
\end{aligned}$$

since all the quantities in parenthesis are nonnegative. This completes the construction of T'' and the proof.

10.2 The Lempel-Ziv algorithm

There is a sense in which the Huffman coding was "optimal", but this is under several assumptions:

1. The compression is lossless, i.e. uncompressing the compressed file yields exactly the original file. When lossy compression is permitted, as for video, other algorithms can achieve much greater compression, and this is a very active area of research because people want to be able to send video and audio over the Web.

2. We know all the frequencies $f(i)$ with which each character appears. How do we get this information? We could make two passes over the data, the first to compute the $f(i)$, and the second to encode the file. But this can be much more expensive than passing over the data once for large files residing on disk or tape. One way to do just one pass over the data is to assume that the fractions $f(i)/n$ of each character in the file are similar to files you've compressed before. For example you could assume all Java programs (or English text, or PowerPoint files, or ...) have about the same fractions of characters appearing. A second cleverer way is to estimate the fractions $f(i)/n$ on the fly as you process the file. One can make Huffman coding adaptive this way.

3. We know the set of characters (the alphabet) appearing in the file. This may seem obvious, but there is a lot of freedom of choice. For example, the alphabet could be the characters on a keyboard, or they could be the key words and variables names appearing in a program. To see what difference this can make, suppose we have a file consisting of n strings *aaaa* and n strings *bbbb* concatenated in some order. If we choose the alphabet $\{a, b\}$ then $8n$ bits are needed to encode the file. But if we choose the alphabet $\{aaaa, bbbb\}$ then only $2n$ bits are needed.

Picking the correct alphabet turns out to be crucial in practical compression algorithms. Both the UNIX compress and GNU gzip algorithms use a greedy algorithm due to Lempel and Ziv to compute a good alphabet in one pass while compressing. Here is how it works.

If s and t are two bit strings, we will use the notation $s \circ t$ to mean the bit string gotten by concatenating s and t.

We let F be the file we want to compress, and think of it just as a string of bits, that is 0's and 1's. We will build an alphabet A of common bit strings encountered in F, and use

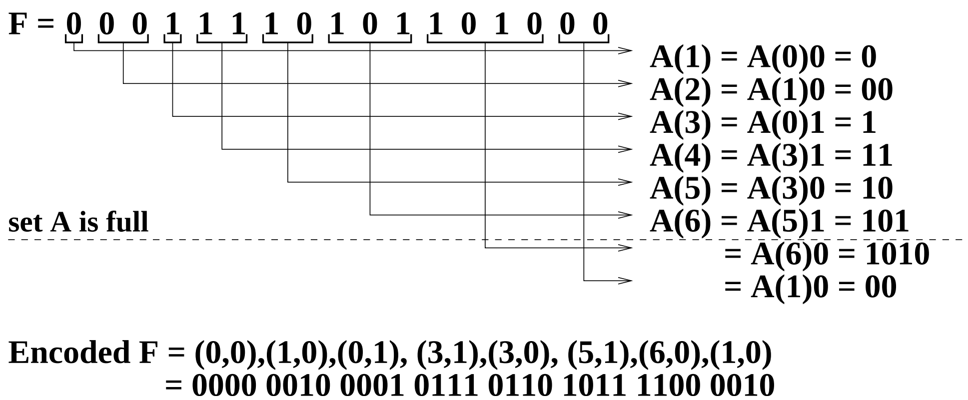


Figure 10.1: An example of the Lempel-Ziv algorithm.

it to compress F. Given A, we will break F into shorter bit strings like

$$F = A(1) \circ 0 \circ A(2) \circ 1 \circ \cdots \circ A(7) \circ 0 \circ \cdots \circ A(5) \circ 1 \circ \cdots \circ A(i) \circ j \circ \cdots$$

and encode this by

$$1 \circ 0 \circ 2 \circ 1 \circ \cdots \circ 7 \circ 0 \circ \cdots \circ 5 \circ 1 \circ \cdots \circ i \circ j \circ \cdots$$

The indices i of $A(i)$ are in turn encoded as fixed length binary integers, and the bits j are just bits. Given the fixed length (say r) of the binary integers, we decode by taking every group of $r+1$ bits of a compressed file, using the first r bits to look up a string in A, and concatenating the last bit. So when storing (or sending) an encoded file, a header containing A is also stored (or sent).

Notice that while Huffman's algorithm encodes blocks of fixed size into binary sequences of variable length, Lempel-Ziv encodes blocks of varying length into blocks of fixed size.

Here is the algorithm for encoding, including building A. Typically a fixed size is available for A, and once it fills up, the algorithm stops looking for new characters.

$A = \{\emptyset\}$ ... start with an alphabet containing only an empty string
$i = 0$ ... points to next place in file f to start encoding
repeat
    find $A(k)$ in the current alphabet that matches as many leading bits $f_i F_{i+1} f_{i+2} \cdots$ as possible
        ... initially only $A(0) =$ empty string matches
        ... Let b be the number of bits in $A(k)$
    if A is not full, add $A(k) \circ f_{i+b}$ to A
        ... f_{i+b} is the first bit unmatched by $A(k)$
    output $k \circ F_{i+b}$
    $i = i + b + 1$
until $i > length(F)$

Note that A is built "greedily", based on the beginning of the file. Thus there are no optimality guarantees for this algorithm. It can perform badly if the nature of the file changes substantially after A is filled up, however the algorithm makes only one pass through the file (there are other possible implementations: A may be unbounded, and the index k would be encoded with a variable-length code itself).

In Figure 10.1 there is an example of the algorithm running, where the alphabet A fills up after 6 characters are inserted. In this small example no compression is obtained, but if A were large, and the same long bit strings appeared frequently, compression would be substantial. The gzip manpage claims that source code and English text is typically compressed 60%-70%.

To observe an example, we took a latex file of $74,892$ bytes. Running Huffman's algorithm, with bytes used as blocks, we could have compressed the file to $36,757$ bytes, plus the space needed to specify the code. The Unix program `compress` produced an encoding of size $34,385$, while `gzip` produced an encoding of size $22,815$.

10.3 Lower bounds on data compression

10.3.1 Simple Results

How much can we compress a file without loss? We present some results that give lower bounds for *any* compression algorithm. Let us start from a "worst case" analysis.

THEOREM 18
Let $C : \{0,1\}^n \to \{0,1\}^*$ be an encoding algorithm that allows lossless decoding (i.e. let C be an injective function mapping n bits into a sequence of bits). Then there is a file $f \in \{0,1\}^n$ such that $|C(f)| \geq n$.

In words, for any lossless compression algorithm there is always a file that the algorithm is unable to compress.

PROOF: Suppose, by contradiction, that there is a compression algorithm C such that, for all $f \in \{0,1\}^n$, $|C(f)| \leq n-1$. Then the set $\{C(f) : f \in \{0,1\}^n\}$ has 2^n elements because C is injective, but it is also a set of strings of length $\leq n-1$, and so it has at most $\sum_{l=1}^{n-1} 2^l \leq 2^n - 2$ elements, which gives a contradiction. $\square$

While the previous analysis showed the existence of incompressible files, the next theorem shows that random files are hard to compress, thus giving an "average case" analysis.

THEOREM 19
Let $C : \{0,1\}^n \to \{0,1\}^*$ be an encoding algorithm that allows lossless decoding (i.e. let C be an injective function mapping n bits into a sequence of bits). Let $f \in \{0,1\}^n$ be a sequence of n randomly and uniformly selected bits. Then, for every t,

$$\Pr[|C(f)| \leq n-t] \leq \frac{1}{2^{t-1}}$$

For example, there is less than a chance in a million of compression an input file of n bits into an output file of length $n-21$, and less than a chance in eight millions that the output will be 3 bytes shorter than the input or less.

PROOF: We can write

$$\Pr[|C(f)| \leq n-t] = \frac{|\{f : |C(f)| \leq n-t\}|}{2^n}$$

Regarding the numerator, it is the size of a set that contains only strings of length $n-t$ or less, so it is no more than $\sum_{l=1}^{n-t} 2^l$, which is at most $2^{n-t+1} - 2 < 2^{n-t+1} = 2^n/2^{t-1}$. $\square$

The following result is harder to prove, and we will just state it.

THEOREM 20
Let $C : \{0,1\}^n \to \{0,1\}^*$ be a prefix-free encoding, and let f be a random file of n bits. Then $\mathbf{E}[|C(f)|] \geq n$.

This means that, from the average point of view, the optimum prefix-free encoding of a random file is just to leave the file as it is.

In practice, however, files are not completely random. Once we formalize the notion of a not-completely-random file, we can show that some compression is possible, but not below a certain limit.

First, we observe that even if not all n-bits strings are possible files, we still have lower bounds.

THEOREM 21
Let $F \subseteq \{0,1\}^n$ be a set of possible files, and let $C\{0,1\}^n \to \{0,1\}^*$ be an injective function. Then

1. *There is a file $f \in F$ such that $|C(f)| \geq \log_2 |F|$.*

2. *If we pick a file f uniformly at random from F, then for every t we have*

$$\Pr[|C(f)| \leq (\log_2 |F|) - t] \leq \frac{1}{2^{t-1}}$$

3. *If C is prefix-free, then when we pick a file f uniformly at random from F we have $\mathbf{E}[|C(f)|] \geq \log_2 |F|$.*

PROOF: Part 1 and 2 is proved with the same ideas as in Theorem 19 and Theorem 20. Part 3 has a more complicated proof that we omit. $\square$

10.3.2 Introduction to Entropy

Suppose now that we are in the following setting:

the file contains n characters
there are c different characters possible
character i has probability $p(i)$ of appearing in the file

What can we say about probable and expected length of the output of an encoding algorithm?

Let us first do a very rough approximate calculation. When we pick a file according to the above distribution, very likely there will be about $p(i) \cdot n$ characters equal to i. Each file with these "typical" frequencies has a probability about $p = \prod_i p(i)^{p(i) \cdot n}$ of being generated. Since files with typical frequencies make up almost all the probability mass, there must be about $1/p = \prod_i (1/p(i))^{p(i) \cdot n}$ files of typical frequencies. Now, we are in a setting which is similar to the one of parts 2 and 3 of Theorem 21, where F is the set of files with typical frequencies. We then expect the encoding to be of length at least $\log_2 \prod_i p(i)^{p(i)\cdot n} = n \cdot \sum_i p(i) \log_2(1/p(i))$. The quantity $\sum_i p(i) \log_2 1/(p(i))$ is the expected number of bits that it takes to encode each character, and is called the *entropy* of the distribution over the characters. The notion of entropy, the discovery of several of its properties, (a formal version of) the calculation above, as well as a (inefficient) optimal compression algorithm, and much, much more, are due to Shannon, and appeared in the late 40s in one of the most influential research papers ever written.

10.3.3 A Calculation

Making the above calculation precise would be long, and involve a lot of ϵs. Instead, we will formalize a slightly different setting. Consider the set F of files such that

the file contains n characters
there are c different characters possible
character i occurs $n \cdot p(i)$ times in the file

We will show that F contains roughly $2^{n \sum_i p(i) \log_2 1/p(i)}$ files, and so a random element of F cannot be compressed to less than $n \sum_i p(i) \log_2 1/p(i)$ bits. Picking a random element of F is almost but not quite the setting that we described before, but it is close enough, and interesting in its own. Let us call $f(i) = n \cdot p(i)$ the number of occurrences of character i in the file.

We need two results, both from Math 55. The first gives a formula for $|F|$:

$$|F| = \frac{n!}{f(1)! \cdot f(2)! \cdots f(c)!}$$

Here is a sketch of the proof of this formula. There are $n!$ permutations of n characters, but many are the same because there are only c different characters. In particular, the $f(1)$ appearances of character 1 are the same, so all $f(1)!$ orderings of these locations are identical. Thus we need to divide $n!$ by $f(1)!$. The same argument leads us to divide by all other $f(i)!$.

Now we have an exact formula for $|F|$, but it is hard to interpret, so we replace it by a simpler approximation. We need a second result from Math 55, namely Stirling's formula for approximating $n!$:

$$n! \approx \sqrt{2\pi} n^{n+.5} e^{-n}$$

This is a good approximation in the sense that the ratio $n!/[\sqrt{2\pi} n^{n+.5} e^{-n}]$ approaches 1 quickly as n grows. (In Math 55 we motivated this formula by the approximation $\log n! = \sum_{i=2}^{n} \log i \approx \int_1^n \log x dx$.) We will use Stirling's formula in the form

$$\log_2 n! \approx \log_2 \sqrt{2\pi} + (n + .5) \log_2 n - n \log_2 e$$

Stirling's formula is accurate for large arguments, so we will be interested in approximating $\log_2 |F|$ for large n. Furthermore, we will actually estimate $\frac{\log_2 |F|}{n}$, which can be interpreted as the *average number of bits per character* to send a long file. Here goes:

$$
\begin{aligned}
\frac{\log_2 |F|}{n} &= \frac{\log_2(n!/(f(1)! \cdots f(c)!))}{n} \\
&= \frac{\log_2 n! - \sum_{i=1}^{c} \log_2 f(i)!}{n} \\
&\approx \frac{1}{n} \cdot [\log_2 \sqrt{2\pi} + (n + .5) \log_2 n - n \log_2 e \\
&\quad - \sum_{i=1}^{c} (\log_2 \sqrt{2\pi} + (f(i) + .5) \log_2 f(i) - f(i) \log_2 e)] \\
&= \frac{1}{n} \cdot [n \log_2 n - \sum_{i=1}^{c} f(i) \log_2 f(i) \\
&\quad + (1 - c) \log_2 \sqrt{2\pi} + .5 \log_2 n - .5 \sum_{i=1}^{c} \log_2 f(i)] \\
&= \log_2 n - \sum_{i=1}^{c} \frac{f(i)}{n} \log_2 f(i) \\
&\quad + \frac{(1 - c) \log_2 \sqrt{2\pi}}{n} + \frac{.5 \log_2 n}{n} - \frac{.5 \sum_{i=1}^{c} \log_2 f(i)}{n}
\end{aligned}
$$

As n gets large, the three fractions on the last line above all go to zero: the first term looks like $O(1/n)$, and the last two terms look like $O(\frac{\log_2 n}{n})$. This lets us simplify to get

$$
\begin{aligned}
\frac{\log_2 |F|}{n} &\approx \log_2 n - \sum_{i=1}^{c} \frac{f(i)}{n} \log_2 f(i) \\
&= \log_2 n - \sum i = 1^c p(i) \log_2 np(i) \\
&= \sum_{i=1}^{c} (\log_2 n) p(i) - \sum i = 1^c p(i) \log_2 np(i) \\
&= \sum_{i=1}^{c} p(i) \log_2 n/np(i) \\
&= \sum_{i=1}^{c} p(1) \log_2 1/p(i)
\end{aligned}
$$

Normally, the quantity $\sum_i p(i) \log_2 1/p(i)$ is denoted by H.

How much more space can Huffman coding take to encode a file than Shannon's lower bound Hn? A theorem of Gallagher (1978) shows that at worst Huffman will take $n \cdot (p_{max} + .086)$ bits more than Hn, where p_{max} is the largest of any p_i. But it often does much better.

Furthermore, if we take blocks of k characters, and encode them using Huffman's algorithm, then, for large k and for n tending to infinity, the average length of the encoding tends to the entropy.

Chapter 11

Linear Programming

11.1 Linear Programming

It turns out that a great many problems can be formulated as *linear programs*, i.e. maximizing (or minimizing) a linear function of some variables, subject to *constraints* on the variables; these constraints are either *linear equations* or *linear inequalities*, i.e. linear functions of the variables either set equal to a constant, or $\leq$ a constant, or $\geq$ a constant. Most of this lecture will concentrate on recognizing how to reformulate (or *reduce*) a given problem to a linear program, even though it is not originally given this way. The advantage of this is that there are several good algorithms for solving linear programs that are available. We will only say a few words about these algorithms, and instead concentrate on formulating problems as linear programs.

11.2 Introductory example in 2D

Suppose that a company produces two products, and wishes to decide the level of production of each product so as to maximize profits. Let x_1 be the amount of Product 1 produced in a month, and x_2 that of Product 2. Each unit of Product 1 brings to the company a profit of 120, and each unit of Product 2 a profit of 500. At this point it seems that the company should only produce Product 2, but there are some constraints on x_1 and x_2 that the company must satisfy (besides the obvious one, $x_1, x_2 \geq 0$). First, x_1 cannot be more than 200, and x_2 more than 300—because of raw material limitations, say. Also, the sum of x_1 and x_2 must be at most 400, because of labor constraints. What are the best levels of production to maximize profits?

We represent the situation by a *linear program*, as follows (where we have numbered the constraints for later reference):

$$\max 120x_1 + 500x_2$$

$$\begin{aligned}
&(1) \qquad && x_1 \leq 200 \\
&(2) \qquad && x_2 \leq 300 \\
&(3) \qquad && x_1 + x_2 \leq 400 \\
&(4) \qquad && x_1 \geq 0
\end{aligned}$$

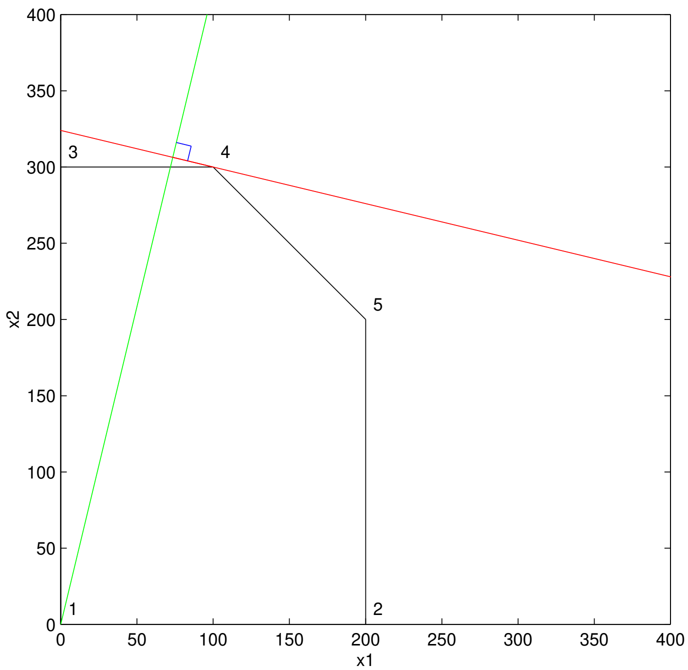


Figure 11.1: The feasible region (polygon), solution vertex (#4), and line of constant profit

$$(5) \qquad x_2 \geq 0$$

The set of all *feasible* solutions of this linear program (that is, all vectors (x_1, x_2) in 2D space that satisfy all constraints) is precisely the (black) polygon shown in Figure 1 below, with vertices numbered 1 through 5.

The vertices are given in the following table, and labelled in Figure 11.1 (we explain the meaning of "active constraint" below):

Vertex	x1-coord	x2-coord	Active constraints
1	0	0	4,5
2	200	0	1,5
3	0	300	2,4
4	100	300	2,3
5	200	200	1,3

The reason all these constraints yield the polygon shown is as follows. Recall that a linear equation like $ax_1 + bx_2 = p$ defines a line in the plane. The *inequality* $ax_1 + bx_2 \leq p$ defines all points on one side of that line, i.e. a *half-plane*, which you can think of as an (infinite) polygon with just one side. If we have two such constraints, the points have to lie in the intersection of two half-planes, i.e. a polygon with 2 sides. Each constraint adds (at most) one more side to the polygon. For example, the 5 constraints above yield 5 sides in the polyhedron: constraint (1), $x_2 \leq 200$, yields the side with vertices #2 and #5, constraint (2), $x_3 \leq 300$, yields the side with vertices #3 and #4, constraint (3), $x_1 + x_2 \leq 400$, yields the side with vertices #4 and #5, constraint (4), $x_1 \geq 0$, yields the side with vertices #1 and #3, and constraint (5), $x_2 \geq 0$, yields the side with vertices #1 and #2. We also say that constraint (1) is *active* at vertices #2 and #5 since it is just barely satisfied at those vertices (at other vertices x_2 is strictly less than 200).

We wish to maximize the linear function profit$=120x_1 + 500x_2$ over all points of this polygon. We think of this geometrically as follows. The set of all points satisfying $p = 120x_1 + 500x_2$ for a fixed p is a line. As we vary p, we get different lines, all parallel to one another, and all perpendicular to the vector $(120, 500)$. (We review this basic geometrical fact below).

Geometrically, we want to increase p so that the line is just barely touching the polygon at one point, and increasing p would make the plane miss the polygon entirely. It should be clear geometrically that this point will usually be a *vertex* of the polygon. This point is the *optimal solution* of the linear program. This is shown in the figure above, where the green line (going from the origin to the top of the graph) is parallel to the vector $(120, 500)$, the red line (going all the way from left to right across the graph) is perpendicular to the green line and connects to the solution vertex #4 $(100, 300)$, which occurs for $p = 120 * 100 + 500 * 300 = 162000$ in profit. (The blue "L" connecting the green and red lines indicates that they are perpendicular.)

(Now we review why the equation $y_1 \cdot x_1 + y_2 \cdot x_2 = p$ defines a line perpendicular to the vector $y = (y_1, y_2)$. You may skip this if this is familiar material. Write the equation as a dot product of y and $x = (x_1, x_2)$: $y * x = p$. First consider the case $p = 0$, so $y * x = 0$. Recall that the if the dot product of two vectors is 0, then the vectors are perpendicular. So when $p = 0$, $y * x = 0$ defines the set of all vectors (points) x perpendicular to y, which is a line through the origin. When $p \neq 0$, we argue as follows. Note that $y * y = y_1^2 + y_2^2$. Then define the vector $\bar{y} = (p/(y * y))y$, a multiple of y. Then we can easily confirm that $\bar{y}$ satisfies the equation because $y * \bar{y} = (p/(y * y))(y * y) = p$. Now think of every point x as the sum of two vectors $x = \bar{x} + \bar{y}$. Substituting in the equation for x we get $p = y * x = y * (\bar{x} + \bar{y}) = y * \bar{x} + y * \bar{y} = y * \bar{x} + p$, or $y * \bar{x} = 0$. In other words, the points $\bar{x}$ lie in a plane through the origin perpendicular to y, and the points $x = \bar{x} + \bar{y}$ are gotten just by adding the vector $\bar{y}$ to each vector in this plane. This just shifts the plane in the direction $\bar{y}$, but leaves it perpendicular to y.)

There are three other geometric possibilities that could occur:

- If the planes for each p are parallel to an edge or face touching the solution vertex, then all points in that edge or face will also be solutions. This just means that the solution is not unique, but we can still solve the linear program. This would occur in the above example if we changed the profits from $(120, 500)$ to $(100, 100)$; we would get equally large profits of $p = 40000$ either at vertex #5 $(200, 200)$, vertex #4 $(100, 300)$, or anywhere on the edge between them.

- It may be that the polygon is *infinite*, and that p can be made arbitrarily large. For example, removing the constraints $x_1 + x_2 \leq 400$ and $x_1 \leq 200$ means that x_1 could become arbitrarily large. Thus $(x_1, 0)$ is in the polygon for all $x_1 > 0$, yielding an arbitrarily large profit $120x_1$. If this happens, it probably means you forgot a constraint and so formulated your linear program incorrectly.

- It may be that the polygon is *empty*, which is also called *infeasible*. This means that *no* points (x_1, x_2) satisfy the constraints. This would be the case if we added the constraint, say, that $x_1 + 2x_2 \geq 800$; since the largest value of $x_1 + 2x_2$ occurs at vertex #4, with $x_1 + 2x_2 = 100 + 2 * 300 = 700$, this extra constraint cannot be satisfied. When this happens it means that your problem is overconstrained, and you have to weaken or eliminate one or more constraints.

11.3 Introductory Example in 3D

Now we take the same company as in the last section, add Product 3 to its product line, along with some constraints, and ask how the problem changes. Each unit of Product 3 brings a profit of 200, and the sum of x_2 and three times x_3 must be at most 600, because Products 2 and 3 share the same piece of equipment ($x_2 + 3x_3 \leq 600$).

This changes the linear program to

$$
\begin{array}{lll}
\max 120x_1 + 500x_2 + 200x_3 & & \\
 & (1) & x_1 \leq 200 \\
 & (2) & x_2 \leq 300 \\
 & (3) & x_1 + x_2 \leq 400 \\
 & (4) & x_1 \geq 0 \\
 & (5) & x_2 \geq 0 \\
 & (6) & x_3 \geq 0 \\
 & (7) & x_2 + 3x_3 \leq 600
\end{array}
$$

Each constraint correspond to being on one side of a plane in (x_1, x_2, x_3) space, a *half-space*. The 7 constraints result in a 7-sided polyhedron shown in Figure 11.2. The polyhedron has vertices and active constraints show here:

Vertex	x1-coord	x2-coord	x3-coord	Active constraints
1	0	0	0	4,5,6
2	200	0	0	1,5,6
3	0	300	0	2,4,6
4	100	300	0	2,3,6
5	200	200	0	1,3,6
6	0	0	200	4,5,7
7	100	300	100	2,3,7
8	200	0	200	1,5,7

Note that a vertex now has 3 active constraints, because it takes the intersection of at least 3 planes to make a corner in 3D, whereas it only took the intersection of 2 lines to make a corner in 2D.

Again the (green) line is in the direction (120,500,200) of increasing profit, the maximum of which occurs at vertex #7. There is a (red) line connecting vertex #7 to the green line, to which it is perpendicular.

In general m constraints on n variables can yield an m-sided polyhedron in n-dimensional space. Such a polyhedron can be seen to have as many as $\begin{pmatrix} m \\ n \end{pmatrix}$ vertices, since n constraints

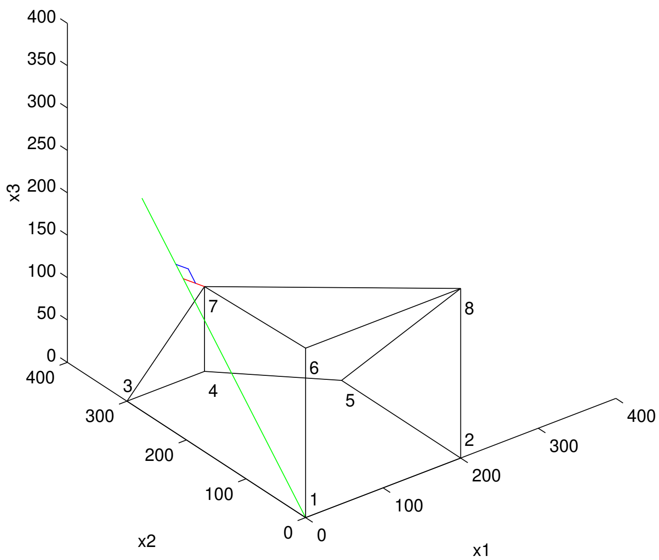


Figure 11.2: The feasible region (polyhedron).

are active at a corner, and there are $\begin{pmatrix} m \\ n \end{pmatrix}$ ways to choose n constraints. Each of these very many vertices is a candidate solution. So when m and n are large, we must rely on a systematic algorithm rather than geometric intuition in order to find the solution.

11.4 Algorithms for Linear Programming

Linear programming was first solved by the *simplex method* devised by George Dantzig in 1947.

We describe the algorithm with reference to last lecture's example. The linear program was

$$\begin{aligned}
&\max 120x_1 + 500x_2 + 200x_3 \\
&\qquad (1) \qquad x_1 \leq 200 \\
&\qquad (2) \qquad x_2 \leq 300 \\
&\qquad (3) \qquad x_1 + x_2 \leq 400 \\
&\qquad (4) \qquad x_1 \geq 0 \\
&\qquad (5) \qquad x_2 \geq 0 \\
&\qquad (6) \qquad x_3 \geq 0 \\
&\qquad (7) \qquad x_2 + 3x_3 \leq 600
\end{aligned}$$

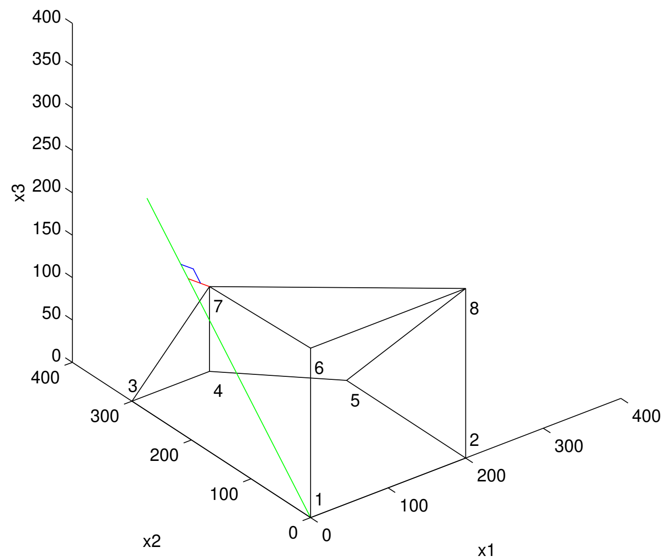


Figure 11.3: The feasible region (polyhedron).

and the polyhedron of feasible solutions is shown in Figure 11.3 and its set of vertices is

Vertex	x1-coord	x2-coord	x3-coord	Active constraints
1	0	0	0	4,5,6
2	200	0	0	1,5,6
3	0	300	0	2,4,6
4	100	300	0	2,3,6
5	200	200	0	1,3,6
6	0	0	200	4,5,7
7	100	300	100	2,3,7
8	200	0	200	1,5,7

The simplex method starts from a vertex (in this case the vertex $(0, 0, 0)$) and repeatedly looks for a vertex that is adjacent, and has better objective value. That is, it is a kind of *hill-climbing* in the vertices of the polytope. When a vertex is found that has no better neighbor, simplex stops and declares this vertex to be the optimum. For example, in Figure 2, if we start at vertex #1 $(0, 0, 0)$, then the adjacent vertices are #2, #3, and #4 with profits 24000, 150000 and 40000, respectively. If the algorithm chooses to go to #3, it then examines vertices #6 and #7, and discovers the optimum #7. There are now implementations of simplex that solve routinely linear programs with *many* thousands of variables and constraints.

The simplex algorithm will also discover and report the other two possibilities: that the solution is infinite, or that the polyhedron is empty. In the worst case, the simplex algorithm takes exponential time in n, but this is very rare, so simplex is widely used in practice. There are other algorithms (by Khachian in 1979 and Karmarkar in 1984) that are guaranteed to run in polynomial time, and are sometimes faster in practice.

11.5 Different Ways to Formulate a Linear Programming Problem

There are a number of equivalent ways to write down the constraints in a linear programming problem. Some formulations of the simplex method use one and some use another, so it is important to see how to transform among them.

One standard formulation of the simplex method is with a matrix A of constraint coefficients, a vector b of constraints, and a vector f defining the linear function $f * x = \sum_i f_i \cdot x_i$ (the dot product of f and x) to be maximized. The constraints are written as the single inequality $A \cdot x \leq b$, which means that every component $(A \cdot x)_i$ of the vector $A \cdot x$ is less than or equal to the corresponding component b_i: $(A \cdot x)_i \leq b_i$. Thus, A has as many rows as there are constraints, and as many columns as there are variables.

In the example above, $f = [120, 500, 200]$,

$$A = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 3 \end{matrix} \quad \text{and} \quad b = \begin{matrix} 200 \\ 300 \\ 400 \\ 0 \\ 0 \\ 0 \\ 600 \end{matrix}$$

Note that the constraints 4, 5 and 6 are $-x_1 \leq 0$, $-x_2 \leq 0$ and $-x_3 \leq 0$, or $x_1 \geq 0$, $x_2 \geq 0$ and $x_3 \geq 0$, respectively. In other words, constraints with $\geq$ can be changed into $\leq$ just by multiplying by -1.

Note that by changing f to $-f$, and maximizing $-f * x$, we are actually *minimizing* $f * x$. So linear programming handles both maximization and minimization equally easily.

Matlab 5.3, which is available on UNIX machines across campus, has a function linprog($-f, A, b$) for solving linear programs in this format. (This implementation *minimizes* the linear function instead of maximizing it, but since minimizing $-f * x$ is the same as maximizing $f * x$, we only have to negate the f input argument to get Matlab to maximize $f * x$). In earlier Matlab versions this program is called LP.

A Java applet with a nice GUI for solving linear programming problems is available at URL riot.ieor.berkeley.edu/riot (click on "Linear Program Solver with Simplex").

Now suppose that in addition to inequalities, we have equalities, such as $x_1 + x_3 = 10$. How do we express this in terms of inequalities? This is simple: write each equality as *two* inequalities: $x_1 + x_3 \leq 10$ and $x_1 + x_3 \geq 10$ (or $-x_1 - x_2 \leq -10$).

Similarly, one can turn any linear program into one just with equalities, and all inequalities of the form $x_i \geq 0$; some versions of simplex require this form. To turn an inequality like $x_1 + x_2 \leq 400$ into an equation, we introduce a new variable s (the *slack variable* for this inequality), and rewrite this inequality as $x_1 + x_2 + s = 400, s \geq 0$. Similarly, any inequality like $x_1 + x_3 \geq 20$ is rewritten as $x_1 + x_3 - s = 20, s \geq 0$; s is now called a *surplus* variable.

We handle an unrestricted variable x as follows: We introduce two nonnegative variables, x^+ and x^-, and replace x by $x^+ - x^-$. This way, x can take on any value.

11.6 A Production Scheduling Example

We have the demand estimates for our product for all months of 1997, $d_i : i = 1, \ldots, 12$, and they are very uneven, ranging from 440 to 920. We currently have 60 employees, each of which produce 20 units of the product each month at a salary of 2,000; we have no stock of the product. How can we handle such fluctuations in demand? Three ways:

- overtime—but this is expensive since it costs 80% more than regular production, and has limitations, as workers can only work 30% overtime.
- hire and fire workers—but hiring costs 320, and firing costs 400.
- store the surplus production—but this costs 8 per item per month

This rather involved problem can be formulated and solved as a linear program. As in all such reductions, a crucial first step is defining the variables:

- Let w_i be the number of workers we have in the ith month—we start with $w_0 = 60$.
- Let x_i be the production for month i.
- o_i is the number of items produced by overtime in month i.
- h_i and f_i are the numbers of workers hired/fired in the beginning of month i.
- s_i is the amount of product stored after the end of month i.

We now must write the constraints:

- $x_i = 20w_i + o_i$—the amount produced is the one produced by regular production, plus overtime.
- $w_i = w_{i-1} + h_i - f_i, w_i \geq 0$—the changing number of workers.
- $s_i = s_{i-1} + x_i - d_i \geq 0$—the amount stored in the end of this month is what we started with, plus the production, minus the demand.
- $o_i \leq 6w_i$—only 30% overtime.

Finally, what is the objective function? It is

$$\min 2000 \sum w_i + 400 \sum f_i + 320 \sum h_i + 8 \sum s_i + 180 \sum o_i.$$

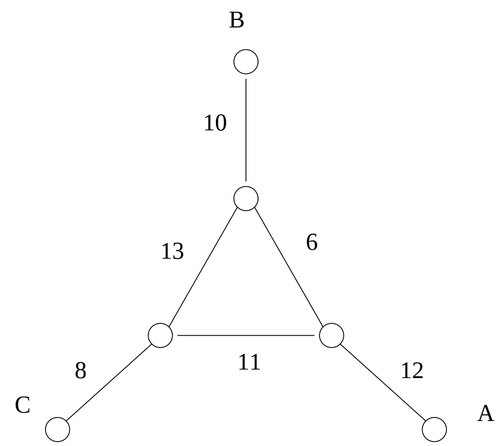


Figure 11.4: A communication network

11.7 A Communication Network Problem

We have a network whose lines have the bandwidth shown in Figure 11.4. We wish to establish three calls: One between A and B (call 1), one between B and C (call 2), and one between A and C (call 3). We must give each call at least 2 units of bandwidth, but possibly more. The link from A to B pays 3 per unit of bandwidth, from B to C pays 2, and from A to C pays 4. Notice that each call can be routed in two ways (the long and the short path), or by a combination (for example, two units of bandwidth via the short route, and three via the long route). How do we route these calls to maximize the network's income?

This is also a linear program. We have variables for each call and each path (long or short); for example x_1 is the short path for call 1, and x_2' the long path for call 2. We demand that (1) no edge bandwidth is exceeded, and (2) each call gets a bandwidth of 2.

$$
\begin{array}{rcl}
\max 3x_1 + 3x_1' + 2x_2 + 2x_2' + 4x_3 + 4x_3' & & \\
x_1 + x_1' + x_2 + x_2' & \leq & 10 \\
x_1 + x_1' + x_3 + x_3' & \leq & 12 \\
x_2 + x_2' + x_3 + x_3' & \leq & 8 \\
x_1 + x_2' + x_3' & \leq & 6 \\
x_1' + x_2 + x_3' & \leq & 13 \\
x_1' + x_2' + x_3 & \leq & 11 \\
x_1 + x_1' & \geq & 2 \\
x_2 + x_2' & \geq & 2 \\
x_3 + x_3' & \geq & 2 \\
x_1, x_1' \ldots, x_3' & \geq & 0
\end{array}
$$

The solution, obtained via simplex in a few milliseconds, is the following: $x_1 = 0, x_1' = 7, x_2 = x_2' = 1.5, x_3 = .5, x_3' = 4.5$.

Question: Suppose that we removed the constraints stating that each call should receive at least two units. Would the optimum change?

Chapter 12

Flows and Matchings

12.1 Network Flows

12.1.1 The problem

Suppose that we are given the network of Figure 12.1 (top), where the numbers indicate capacities, that is, the amount of flow that can go through the edge in unit time. We wish to find the maximum amount of flow that can go through this network, from s to t.

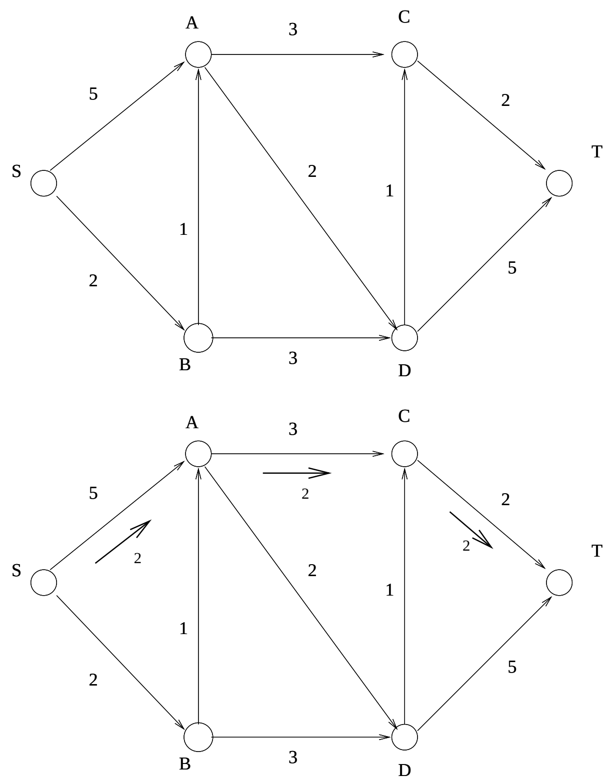


Figure 12.1: Max flow.

This problem can also be reduced to linear programming. We have a nonnegative variable for each edge, representing the flow through this edge. These variables are denoted $f_{s,a}, f_{s,b}, \ldots$ We have two kinds of constraints:

- *capacity* constraints such as $f_{s,a} \leq 5$ (a total of 9 such constraints, one for each edge), and
- *flow conservation* constraints (one for each node except s and t), such as $f_{a,d} + f_{b,d} = f_{d,c} + f_{d,t}$ (a total of 4 such constraints).

We wish to maximize $f_{s,a} + f_{s,b}$, the amount of flow that leaves s, subject to these constraints. It is easy to see that this linear program is equivalent to the max-flow problem. The simplex method would correctly solve it.

In general, we are given a "network" that is just a directed graph $G = (V, E)$ with two special vertices s, t, such that s has only outgoing edges and t has only incoming edges, and a capacity $c_{u,v}$ associated to each edge $(u, v) \in E$. Then the maximum flow in the network is the solution of the following linear program, having a variable $f_{u,v}$ for every edge.

$$
\begin{array}{lrcll}
\text{maximize} & \sum_{v:(s,v)\in E} f_{u,v} & & & \\
\text{subject to} & & & & \\
& f_{u,v} & \leq & c_{u,v} & \text{for every } (u,v) \in E \\
& \sum_{u:(u,v)\in E} f_{u,v} - \sum_{w:(v,w)\in E} f_{v,w} & = & 0 & \text{for every } v \in V - \{s,t\} \\
& f_{u,v} & \geq & 0 & \text{for every } (u,v) \in E
\end{array}
$$

12.1.2 The Ford-Fulkerson Algorithm

If the simplex algorithm is applied to the linear program for the maximum flow problem, it will find the optimal flow in the following way: start from a vertex of the polytope, such as the vertex with $f_{u,v} = 0$ for all edges $(u, v) \in E$, and then proceed to improve the solution (by moving along edges of the polytope from one polytope vertex to another) until we reach the polytope vertex corresponding to the optimal flow.

Let us now try to come up directly with an efficient algorithm for max flow based on the idea of starting from an empty flow and progressively improving it.

How can we find a small improvement in the flow? We can find a path from s to t (say, by depth-first search), and move flow along this path of total value equal to the *minimum* capacity of an edge on the path (we can obviously do no better). This is the first iteration of our algorithm (see the bottom of Figure 12.1).

How to continue? We can look for another path from s to t. Since this time we already partially (or totally) use some of the edges, we should do depth-first search on the edges that have some *residual capacity*, above and beyond the flow they already carry. Thus, the edge (c, t) would be ignored, as if it were not there. The depth-first search would now find the path $s - a - d - t$, and augment the flow by two more units, as shown in the top of Figure 12.2.

Next, we would again try to find a path from s to t. The path is now $s - b - d - t$ (the edges $c - t$ and $a - d$ are full are are therefore ignored), and we augment the flow as shown in the bottom of Figure 12.2.

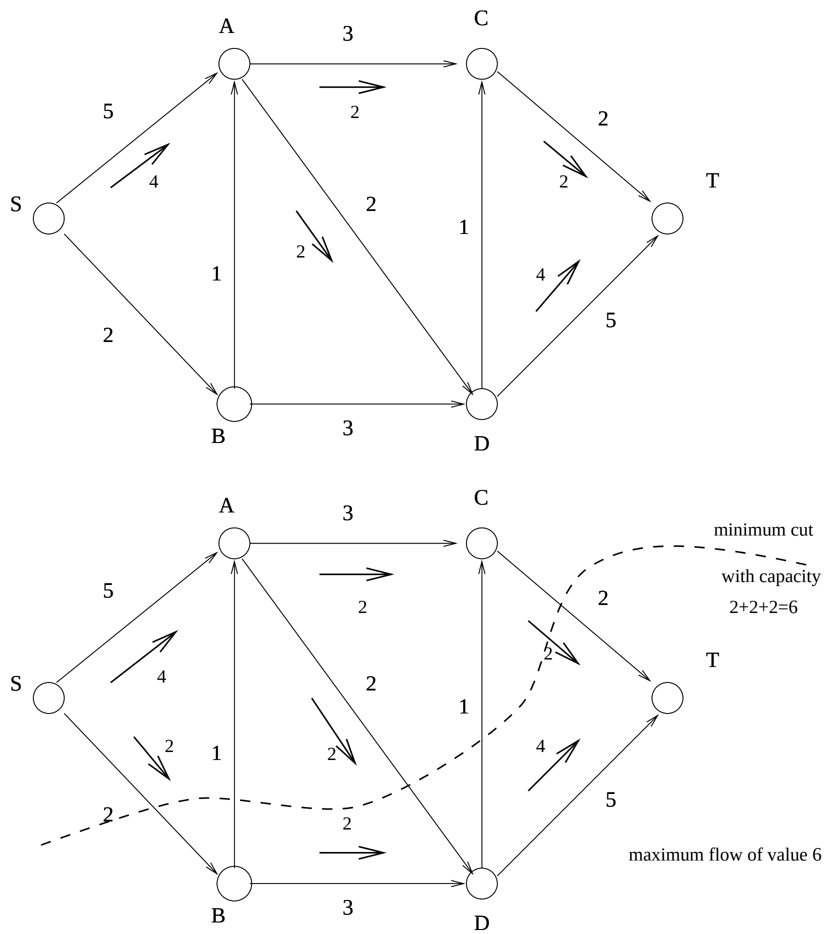
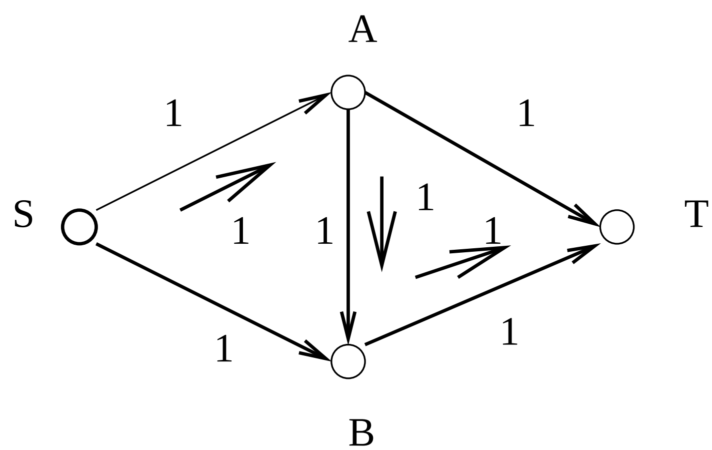


Figure 12.2: Max flow (continued).

Next we would again try to find a path. But since edges $a-d$, $c-t$, and $s-b$ are full, they must be ignored, and therefore depth-first search would fail to find a path, after marking the nodes s, a, c as reachable from S. *We then return the flow shown, of value 6, as maximum.*

There is a complication that we have swept under the rug so far: When we do depth-first search looking for a path, we use not only the edges that are not completely full, but we must also traverse *in the opposite direction* all edges that already have some non-zero flow. This would have the effect of cancelling some flow; cancelling may be necessary to achieve optimality, see Figure 12.1.2. In this figure the only way to augment the current flow is via the path $s-b-a-t$, which traverses the edge $a-b$ in the reverse direction (a legal traversal, since $a-b$ is carrying non-zero flow).



Flows may have to be cancelled.

The algorithm that we just outlined is the Ford-Fulkerson algorithm for the maximum flow problem. The pseudocode for Ford-Fulkerson is as follows.

1. Given in input $G = (V, E)$, vertices s, t, and capacities $c_{u,v}$. Initialize $f_{u,v}$ to zero for all edges.

2. Use depth-first search to find a path from s to t. If no path is found, return the current flow.

3. Let c be the smallest capacity along the path.

4. For each edge (u, v) in the path, decrease $c_{u,v}$ by c, increase $f_{u,v}$ by c, and increase $c_{v,u}$ by c (if the edge (v, u) does not exist in G, create it). Delete edges with capacity zero.

5. Go to step 2

12.1.3 Analysis of the Ford-Fulkerson Algorithm

How can we be sure that the flow found by the Ford-Fulkerson algorithm is optimal?

Let us say that a set of vertices $S \subseteq V$ is a *cut* in a network if $s \in S$ and $t \notin S$. Let us define the *capacity* of a cut to be $\sum_{u \in S, v \notin S, (u,v) \in E} c_{u,v}$.

We first note that if we fix a flow f, and consider a cut S in the network, then the amount of flow that passes "through" the cut, is independent of S, and it is, indeed, the cost of the flow.

LEMMA 22
Fix a flow f. For every cut S the quantity

$$\sum_{u \in S, v \notin S, (u,v) \in E} f(u,v) - \sum_{u \in S, v \notin S, (v,u) \in E} f(v,u)$$

is always the same, independently of S.

As a special case, we have that $\sum_u f(s,u) = \sum_v f(v,t)$, so that the flow coming out of s is exactly the flow getting into t.
PROOF: Assume $f_{u,v}$ is defined for every pair (u,v) and $f_{u,v} = 0$ if $(u,v) \notin E$.

$$\begin{aligned}
& \sum_{u \in S, v \notin S} f_{u,v} - \sum_{u \notin S, v \in S} f_{u,v} \\
= & \sum_{u \in S, v \in V} f_{u,v} - \sum_{u \in S, v \in S} f_{u,v} - \sum_{u \notin S, v \in S} f_{u,v} \\
= & \sum_{u \in S, v \in V} f_{u,v} - \sum_{u \in V, v \in S} f_{u,v} \\
= & \sum_{v \in V} f(s,v) + \sum_{u \in S - \{s\}, v \in V} f_{u,v} - \sum_{u \in V, v \in S - \{s\}} f_{u,v} \\
= & \sum_{v \in V} f_{s,v}
\end{aligned}$$

and the last term is independent of S. □

A consequence of the previous result is that for every cut S and every flow f, the cost of the flow has to be smaller than or equal to the capacity of S. This is true because the cost of f is

$$\sum_{u \in S, v \notin S} f_{u,v} - \sum_{v \notin S, u \in S} f_{v,u} \leq \sum_{u \in S, v \notin S} f_{u,v} \leq \sum_{u \in S, v \notin S} c_{u,v}$$

and the right-hand side is exactly the capacity of S. Notice how this implies that if we find a cut S and a flow f such that the cost of f equals the capacity of S, then it must be the case that f is optimal. This is indeed the way in which we are going to prove the optimality of Ford-Fulkerson.

LEMMA 23
Let $G = (V,E)$ be a network with source s, sink t, and capacities $c_{u,v}$, and let f be the flow found by the Ford-Fulkerson algorithm. Let S be the set of vertices that are reachable from s in the residual network. *Then the capacity of S equals the cost of f, and so f is optimal.*

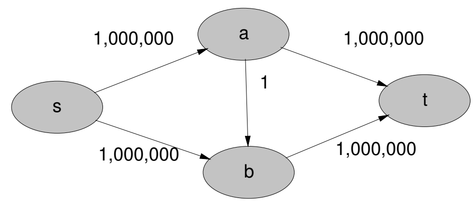


Figure 12.3: A bad instance for Ford-Fulkerson.

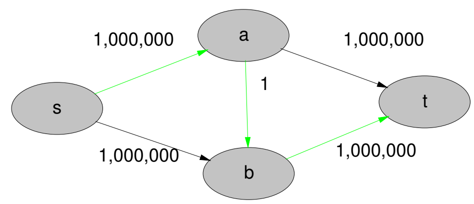


Figure 12.4: A bad instance for Ford-Fulkerson – Choice of path.

PROOF: First of all, S is a cut: s belongs to S by definition, and it is impossible that $t \in S$, because otherwise t would be reachable from s in the residual network of f, and f could not be the output of Ford-Fulkerson.

Now, the cost of the flow is $\sum_{u \in S, v \notin S} f_{u,v} - \sum_{v \notin S, u \in S} f_{v,u}$, while the capacity of the cut is $\sum_{u \in S, v \notin S} c_{u,v}$.

For every $u \in S$ and $v \notin S$, we must have $f_{u,v} = c_{u,v}$, otherwise v would be reachable from s in the residual network (which would contradict the definition of S). Also, for every $v \notin S$ and ever $u \in S$, we must have $f_{v,u} = 0$, otherwise the residual network would have an edge with non-zero capacity going from u to v, and then v would be reachable from s which is impossible. So the cost of the flow is the same as the capacity of the cut. $\Box$

Notice that, along the way, we have proved the following important theorem.

THEOREM 24 (MAX FLOW / MIN CUT THEOREM)
In every network, the cost of the maximum flow equals the capacity of the minimum cut.

We have established the correctness of the Ford-Fulkerson algorithm. What about the running time? Each step of finding an augmenting path and updating the residual network can be implemented in $O(|V| + |E|)$ time. How many steps there can be in the worst case? This may depend on the values of the capacities, and the actual number of steps may be exponential in the size of the input. Consider the network in Figure 12.3.

If we choose the augmenting path with the edge of capacity 1, as shown in Figure 12.4, then after the first step we are left with the residual network of Figure 12.5. Now you see where this is going.

If step 2 of Ford-Fulkerson is implemented with breadth-first search, then each time we use an augmenting path with a minimal number of edges. This implementation of Ford-Fulkerson is called the Edmonds-Karp algorithm, and Edmond and Karp showed that the number of steps is always at most $O(|V||E|)$, regardless of the values of the capacities, for a total running time of $O(|V||E|(|V| + |E|))$. The expression can be simplified by observing that we are only going to consider the portion of the network that is reachable from s, and

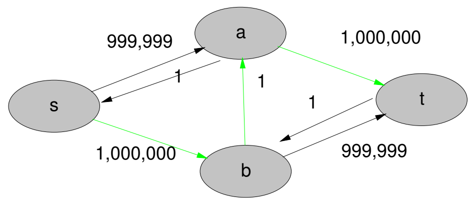


Figure 12.5: A bad instance for Ford-Fulkerson – Residual network.

that contains $O(E)$ vertices, so that an augmenting path can indeed be found in $O(E)$ time, and the total running time can be written as $O(|V||E|^2)$.

12.2 Duality

As it turns out, the max-flow min-cut theorem is a special case of a more general phenomenon called *duality*. Basically, duality means that a maximization and a minimization problem have the property that any feasible solution of the min problem is greater than or equal any feasible solution of the max problem (see Figure 12.6). Furthermore, and more importantly, *they have the same optimum*.

Consider the network shown in Figure 12.6 below, and the corresponding max-flow problem. We know that it can be written as a linear program as follows ($f_{xy} \geq 0$ in the last line is shorthand for all 5 inequalities like $f_{s,a} \geq 0$, etc.):

$$P: \left\{ \begin{array}{cccccccc} \max f_{sa} & +f_{sb} & & & & & & \\ f_{sa} & & & & & \leq & 3 \\ & f_{sb} & & & & \leq & 2 \\ & & f_{ab} & & & \leq & 1 \\ & & & f_{at} & & \leq & 1 \\ & & & & f_{bt} & \leq & 3 \\ f_{sa} & & -f_{ab} & -f_{at} & & = & 0 \\ & f_{sb} & +f_{ab} & & -f_{bt} & = & 0 \\ & & & & f_{xy} & \geq & 0 \end{array} \right.$$

Consider now the following linear program (where again $y_{xy} \geq 0$ is shorthand for all inequalities of that form):

$$D: \left\{ \begin{array}{ccccccccc} \min 3y_{sa} & +2y_{sb} & +y_{ab} & +y_{at} & +3y_{bt} & & & & \\ y_{sa} & & & & & +u_a & & \geq & 1 \\ & y_{sb} & & & & & +u_b & \geq & 1 \\ & & y_{ab} & & & -u_a & +u_b & \geq & 0 \\ & & & y_{at} & & -u_a & & \geq & 0 \\ & & & & y_{bt} & & -u_b & \geq & 0 \\ & & & & & & y_{xy} & \geq & 0 \end{array} \right.$$

This LP describes the min-cut problem! To see why, suppose that the u_A variable is meant to be 1 if A is in the cut with S, and 0 otherwise, and similarly for u_B (naturally,

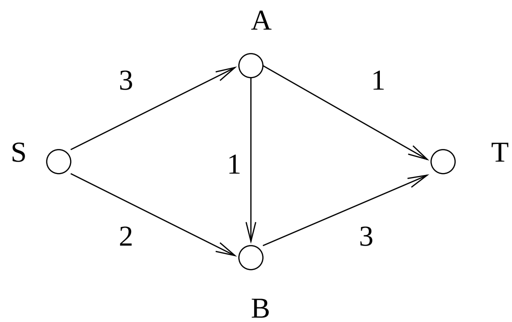


Figure 12.6: A simple max-flow problem.

by the definition of a cut, S will always be with S in the cut, and T will never be with S). Each of the y variables is to be 1 if the corresponding edge contributes to the cut capacity, and 0 otherwise. Then the constraints make sure that these variables behave exactly as they should. For example, the first constraint states that *if A is not with S, then SA must be added to the cut.* The third one states that *if A is with S and B is not* (this is the only case in which the sum $-u_A + u_B$ becomes -1), *then AB must contribute to the cut.* And so on. Although the y and u's are free to take values larger than one, they will be "slammed" by the minimization down to 1 or 0 (we will not prove this here).

Let us now make a remarkable observation: These two programs have strikingly symmetric, *dual*, structure. Each variable of P corresponds to a constraint of D, and vice-versa. Equality constraints correspond to unrestricted variables (the u's), and inequality constraints to restricted variables. Minimization becomes maximization. The matrices are transposes of one another, and the roles of right-hand side and objective function are interchanged. Such LP's are called *dual* to each other.

By the max-flow min-cut theorem, the two LP's P and D above have the same optimum. *In fact, this is true for general dual LP's!* This is the *duality theorem*, which can be stated as follows.

Theorem 25 (Duality Theorem)
Consider a primal *LP problem written in standard form as* "maximize $c * x$ subject to $A \cdot x \leq b$ and $x \geq 0$". *We define the* dual *problem to be* "minimize $b * y$ subject to $A^T y \geq c$ and $y \geq 0$". *Suppose the primal LP has a finite solution. Then so does the dual problem, and the two optimal solutions have the same cost.*

The theorem has an easy part and a hard part. It is easy to prove that for every feasible x for the primal and every feasible y for the dual we have $c * x \leq b * y$. This implies that the optimum of the primal is at most the optimum of the dual (this property is called *weak duality*. To prove that the costs are equal for the optimum is the hard part.

Let us see the proof of weak duality. Let x be feasible for the primal, and y be feasible for the dual. The constraint on the primal impose that $a_1 * x \leq b_1$, $a_2 * x \leq b_2$, $\ldots$, $a_m * x \leq b_m$, where $a_1, \ldots, a_m$ are the rows of A and $b_1, \ldots, b_m$ the entries of b.

Now, the cost of y is $y_1 b_1 + \cdots + y_m b_n$, which, by the above observations, is at least

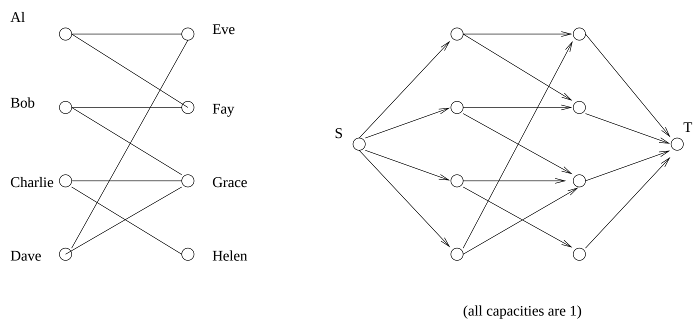


Figure 12.7: Reduction from matching to max-flow.

$y_1(a_1 * x) + \cdot + y_m(a_m * x)$, that we can rewrite as

$$\sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j} y_i x_j \ . \tag{12.1}$$

Each term $\sum_{i=1}^{m} a_{i,j} y_j$ is at least c_j, because this is one of the constraints of the dual, and so we have that Expression (12.1) is at least $\sum_i x_i c_i$, that is the cost of the primal solution.

12.3 Matching

12.3.1 Definitions

A *bipartite graph* is a (typically undirected) graph $G = (V, E)$ where the set of vertices can be partitioned into subsets V_1 and V_2 such that each edge has an endpoint in V_1 and an endpoint in V_2.

Often bipartite graphs represent relationships between different entities: clients/servers, people/projects, printers/files to print, senders/receivers ...

Often we will be given bipartite graphs in the form $G = (L, R, E)$, where L, R is already a partition of the vertices such that all edges go between L and R.

A *matching* in a graph is a set of edges that do not share any endpoint. In a bipartite graph a matching associates to some elements of L precisely one element of R (and vice versa). (So the matching can be seen as an *assignment*.)

We want to consider the following optimization problem: given a bipartite graph, find the matching with the largest number of edges. We will see how to solve this problem by reducing it to the maximum flow problem, and how to solve it directly.

12.3.2 Reduction to Maximum Flow

Let us first see the reduction to maximum flow on an example. Suppose that the *bipartite* graph shown in Figure 12.7 records the compatibility relation between four straight boys and four straight girls. We seek a maximum matching, that is, a set of edges that is as large as possible, and in which no two edges share a node. For example, in the figure below there is a *perfect* matching (a matching that involves all nodes).

To reduce this problem to max-flow we do this: We create a new source and a new sink, connect the source with all boys and all girls with the sinks, and direct all edges of the original bipartite graph from the boys to the girls. All edges have capacity one. It is easy to see that the maximum flow in this network corresponds to the maximum matching.

Well, the situation is slightly more complicated than was stated above: What is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching. We would be at a loss interpreting as a matching a flow that ships .7 units along the edge Al-Eve! Fortunately, what the algorithm in the previous section establishes is that *if the capacities are integers, then the maximum flow is integer.* This is because we only deal with integers throughout the algorithm. Hence *integrality comes for free in the max-flow problem.*

In general, given a bipartite graph $G = (L, R, E)$, we will create a network $G' = (V, E')$ where $V' = L \cup R \cup \{s, t\}$, and E' contains all directed edges of the form (s, u), for $u \in L$, (v, t), for $v \in R$, and (u, v), for $\{u, v\} \in E$. All edges have capacity one.

12.3.3 Direct Algorithm

Let us now describe a direct algorithm, that is essentially the composition of Ford-Fulkerson with the above reduction.

The algorithm proceeds in phases, starting with an empty matching. At each phase, it either finds a way to get a bigger matching, or it gets convinced that it has constructed the largest possible matching.

We first need some definitions. Let $G = (L, R, E)$ be a bipartite graph, $M \subseteq E$ be a matching. A vertex is *covered* by M if it is the endpoint of one of the edges in E.

An *alternating path* (in G, with respect to M) is a path of odd length that starts with a non-covered vertex, ends with a non-covered vertex, and alternates between using edges not in M and edges in M.

If M is our current matching, and we find an alternating path with respect to M, then we can increase the size of M by discarding all the edges in of M which are in the path, and taking all the others.

Starting with the empty matching, in each phase, the algorithm looks for an alternating path with respect to the current matching. If it finds an alternating path, it updates, and improves, the current matching as described above. If no alternating path is found, the current matching is output.

It remains to prove the following:

1. Given a bipartite graph G and a matching M, an alternating path can be found, if it exists, in $O(|V| + |E|)$ time using a variation of BFS.

2. If there is no alternating path, then M is a maximum matching.

We leave part (1) as an exercise and prove part(2).

Suppose M is not an optimal matching, that is, some other matching M^* has more edges. We prove that M must have an alternating path.

Let $G = (L, R, E')$ be the graph where E' contains the edges that are either in M or in M^* *but not in both* (i.e. $E' = M \oplus M^*$).

Every vertex of G has degree at most two. Furthermore if the degree is two, then one of the two incident edges is coming from M and the other from M^*.

Since the maximum degree is 2, G is made out of paths and cycles. Furthermore the cycles are of even length and contain each one an equal number of edges from M and form M^*.

But since $|M^*| > |M|$, M^* must contribute more edges than M to G, and so there must be some path in G where there are more edges of M^* than of M. This is an augmenting path for M.

This completes the description and proof of correctness of the algorithm. Regarding the running time, notice that no matching can contain more than $|V|/2$ edges, and this is an upper bound to the number of phases of the algorithm. Each phase takes $O(|E| + |V|) = O(|E|)$ time (we can ignore vertices that have degree zero, and so we can assume that $|V| = O(|E|)$), so the total running time is $O(|V||E|)$.

Chapter 13

NP-Completeness and Approximation

13.1 Tractable and Intractable Problems

So far, almost all of the problems that we have studied have had complexities that are *polynomial*, i.e. whose running time $T(n)$ has been $O(n^k)$ for some fixed value of k. Typically k has been small, 3 or less. We will let **P** denote the class of all problems whose solution can be computed in polynomial time, i.e. $O(n^k)$ for some fixed k, whether it is 3, 100, or something else. We consider all such problems efficiently solvable, or *tractable*. Notice that this is a very relaxed definition of tractability, but our goal in this lecture and the next few ones is to understand which problems are *intractable*, a notion that we formalize as *not being solvable in polynomial time*. Notice how *not being in* **P** is certainly a strong way of being intractable.

We will focus on a class of problems, called the *NP-complete problems*, which is a class of very diverse problems, that share the following properties: we only know how to solve those problems in time much larger than polynomial, namely *exponential time*, that is $2^{O(n^k)}$ for some k; and if we could solve one NP-complete problem in polynomial time, then there is a way to solve *every* **NP**-complete problem in polynomial time.

There are two reasons to study NP-complete problems. The practical one is that if you recognize that your problem is NP-complete, then you have three choices:

1. you can use a known algorithm for it, and accept that it will take a long time to solve if n is large;

2. you can settle for *approximating* the solution, e.g. finding a nearly best solution rather than the optimum; or

3. you can change your problem formulation so that it is in **P** rather than being NP-complete, for example by restricting to work only on a subset of simpler problems.

Most of this material will concentrate on recognizing NP-complete problems (of which there are a large number, and which are often only slightly different from other, familiar, problems in **P**) and on some basic techniques that allow to solve some NP-complete problems in an *approximate* way in polynomial time (whereas an exact solution seems to require exponential time).

The other reason to study NP-completeness is that one of the most famous open problem in computer science concerns it. We stated above that "we *only know* how to solve NP-complete problems in time much larger than polynomial" not that we *have proven* that NP-complete problems require exponential time. Indeed, this is the million dollar question,[1] one of the most famous open problems in computer science, the question whether "**P** = **NP**?", or whether the class of NP-complete problems have polynomial time solutions. After decades of research, everyone believes that **P**≠**NP**, i.e. that no polynomial-time solutions for these very hard problems exist. But no one has proven it. If you do, you will be very famous, and moderately wealthy.

So far we have not actually defined what NP-complete problems are. This will take some time to do carefully, but we can sketch it here. First we define the larger class of problems called **NP**: these are the problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time. For example, suppose the problem is to answer the question "Does a graph have a simple path of length $|V|$?". If someone hands you a path, i.e. a sequence of vertices, and you can *check* whether this sequence of vertices is indeed a path and that it contains all vertices in polynomial time, then the problem is in **NP**. It should be intuitive that any problem in **P** is also in **NP**, because are all familiar with the fact that checking the validity of a solution is easier than coming up with a solution. For example, it is easier to get jokes than to be a comedian, it is easier to have average taste in books than to write a best-seller, it is easier to read a textbook in a math or theory course than to come up with the proofs of all the theorems by yourself. For all this reasons (and more technical ones) people believe that **P**≠**NP**, although nobody has any clue how to prove it. (But once it will be proved, it will probably not be too hard to understand the proof.)

The NP-complete problems have the interesting property that if you can solve any one of them in polynomial time, then you can solve *every* problem in **NP** in polynomial time. In other words, they are at least as hard as any other problem in **NP**; this is why they are called *complete*. Thus, if you could show that *any one* of the NP-complete problems that we will study *cannot* be solved in polynomial time, then you will have not only shown that **P**≠**NP**, but also that none of the **NP**-compete problems can be solved in polynomial time. Conversely, if you find a polynomial-time algorithm for just one NP-complete problem, you will have shown that **P**=**NP**.[2]

13.2 Decision Problems

To simplify the discussion, we will consider only problems with Yes-No answers, rather than more complicated answers. For example, consider the *Traveling Salesman Problem* (TSP) on a graph with nonnegative integer edge weights. There are two similar ways to state it:

[1]This is not a figure of speech. See `http://www.claymath.org/prizeproblems`.

[2]Which still entitles you to the million dollars, although the sweeping ability to break every cryptographic protocol and to hold the world banking and trading systems by ransom might end up being even more profitable.

1. Given a weighted graph, what is the minimum length cycle that visits each node exactly once? (If no such cycle exists, the minimum length is defined to be ∞.)

2. Given a weighted graph and an integer K, is there a cycle that visits each node exactly once, with weight at most K?

Question 1 above seems more general than Question 2, because if you could answer Question 1 and find the minimum length cycle, you could just compare its length to K to answer Question 2. But Question 2 has a Yes/No answer, and so will be easier for us to consider. In particular, if we show that Question 2 is NP-complete (it is), then that means that Question 1 is at least as hard, which will be good enough for us.[3]

Another example of a problem with a Yes-No answer is *circuit satisfiability* (which we abbreviate CSAT). Suppose we are given a Boolean circuit with n Boolean inputs $x_1, ..., x_n$ connected by AND, OR and NOT gates to one output x_{out}. Then we can ask whether there is a set of inputs (a way to assign True or False to each x_i) such that x_{out} =True. In particular, we will not ask what the values of x_i are that make x_{out} True.

If A is a Yes-No problem (also called a *decision* problem), then for an input x we denote by $A(x)$ the right Yes-No answer.

13.3 Reductions

Let A and B be two problems whose instances require Yes/No answers, such as TSP and CSAT. A *reduction* from A to B is a polynomial-time algorithm R which transforms inputs of A to equivalent inputs of B. That is, given an input x to problem A, R will produce an input $R(x)$ to problem B, such that x is a "yes" input of A if and only if $R(x)$ is a "yes" input of B. In a compact notation, if R is a reduction from A to B, then for every input x we have $A(x) = B(R(x))$.

A reduction from A to B, together with a polynomial time algorithm for B, constitute a polynomial algorithm for A (see Figure13.1). For any input x of A of size n, the reduction R takes time $p(n)$—a polynomial—to produce an equivalent input $R(x)$ of B. Now, this input $R(x)$ can have size at most $p(n)$—since this is the largest input R can conceivably construct in $p(n)$ time. If we now submit this input to the assumed algorithm for B, running in time $q(m)$ on inputs of size m, where q is another polynomial, then we get the right answer of x, within a total number of steps at most $p(n) + q(p(n))$—also a polynomial!

We have seen many reductions so far, establishing that problems are easy (e.g., from matching to max-flow). In this part of the class we shall use reductions in a more sophisticated and counterintuitive context, in order to prove that certain problems are hard. If we reduce A to B, we are essentially establishing that, *give or take a polynomial, A is no harder than B*. We could write this as

$$A \leq B$$

an inequality between the complexities of the two problems. If we know B is easy, this establishes that A is easy. If we know A is hard, this establishes B is hard. It is this latter implication that we shall be using soon.

[3]It is, in fact, possible to prove that Questions 1 and 2 are equally hard.

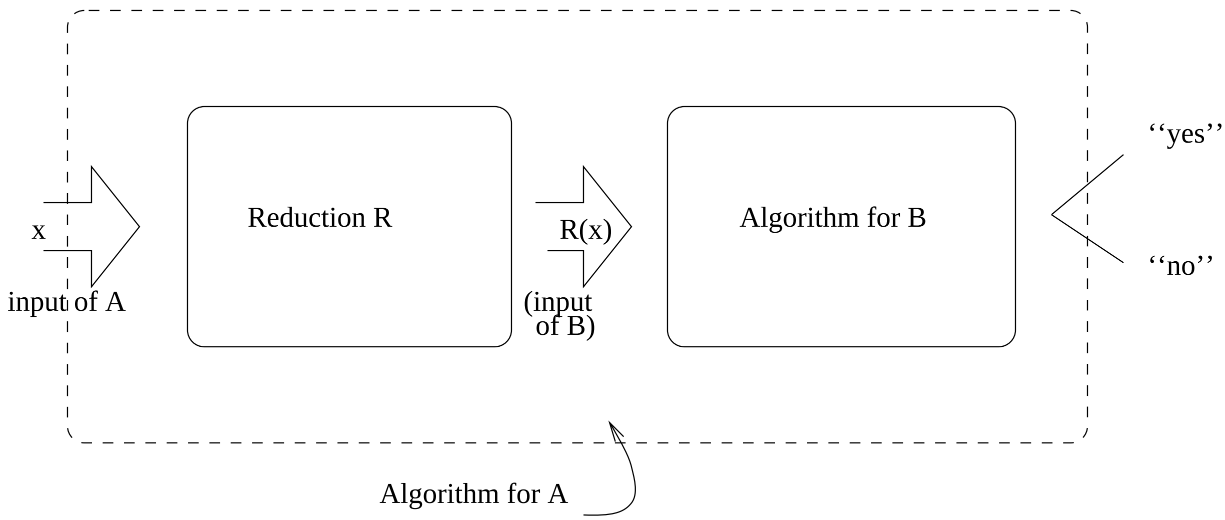


Figure 13.1: A reduction.

13.4 Definition of Some Problems

Before giving the formal definition of **NP** and of **NP**-complete problem, we define some problems that are **NP**-complete, to get a sense of their diversity, and of their similarity to some polynomial time solvable problems.

In fact, we will look at pairs of very similar problems, where in each pair a problem is solvable in polynomial time, and the other is presumably not.

- **minimum spanning tree:** Given a weighted graph and an integer K, is there a tree that connects all nodes of the graph whose total weight is K or less?
- **travelling salesman problem:** Given a weighted graph and an integer K, is there a cycle that visits all nodes of the graph whose total weight is K or less?

Notice that we have converted each one of these familiar problems into a decision problem, a "yes-no" question, by supplying a goal K and asking if the goal can be met. Any optimization problem can be so converted

If we can solve the optimization problem, we can certainly solve the decision version (actually, the converse is in general also true). Therefore, proving a negative complexity result about the decision problem (for example, proving that it cannot be solved in polynomial time) immediately implies the same negative result for the optimization problem.

By considering the decision versions, we can study optimization problems side-by-side with decision problems (see the next examples). This is a great convenience in the theory of complexity which we are about to develop.

- **Eulerian graph:** Given a directed graph, is there a closed path that visits each edge of the graph exactly once?
- **Hamilitonian graph:** Given a directed graph, is there a closed path that visits each *node* of the graph exactly once?

A graph is Eulerian if and only if it is strongly connected and each node has equal in-degree and out-degree; so the problem is squarely in **P** There is no known such characterization—or algorithm—for the Hamilton problem (and notice its similarity with the TSP).

- **circuit value:** Given a Boolean circuit, and its inputs, is the output T?

- **circuit SAT:** Given a Boolean circuit, is there a way to set the inputs so that the output is T? (Equivalently: If we are given *some* of its inputs, is there a way to set the remaining inputs so that the output is T.)

We know that **circuit value** is in **P**: also, the naïve algorithm for that evaluates all gates bottom-up is polynomial. How about **circuit SAT?** There is no obvious way to solve this problem, sort of trying all input combinations for the unset inputs—and this is an exponential algorithm.

General circuits connected in arbitrary ways are hard to reason about, so we will consider them in a certain standard form, called *conjunctive normal form (CNF)*: Let $x_1, ..., x_n$ be the input Boolean variables, and x_{out} be the output Boolean variable. Then a Boolean expression for x_{out} in terms of $x_1, ..., x_n$ is in CNF if it is the AND of a set of *clauses*, each of which is the OR of some subset of the set of *literals* $\{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$. (Recall that "conjunction" means "and", whence the name CNF.) For example,

$$x_{out} = (x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee \neg x_1) \wedge (x_1 \vee x_2) \wedge (x_3)$$

is in CNF. This can be translated into a circuit straightforwardly, with one gate per logical operation. Furthermore, we say that an expression is in **2-CNF** if each clause has two distinct literals. Thus the above expression is not 2-CNF but the following one is:

$$(x_1 \vee \neg x_1) \wedge (x_3 \vee x_2) \wedge (x_1 \vee x_2)$$

3-CNF is defined similarly, but with 3 distinct literals per clause:

$$(x_1 \vee \neg x_1 \vee x_4) \wedge (x_3 \vee x_2 \vee x_1) \wedge (x_1 \vee x_2 \vee \neg x_3)$$

- **2SAT:** Given a Boolean formula in **2-CNF**, is there a satisfying truth assignment to the input variables?

- **3SAT:** Given a Boolean formula in **3-CNF** is there a satisfying truth assignment to the input variables?

2SAT can be solved by graph-theoretic techniques in polynomial time. For **3SAT**, no such techniques are available, and the best algorithms known for this problems are exponential in the worst case, and they run in time roughly $(1.4)^n$, where n is the number of variables. (Already a non-trivial improvement over 2^n, which is the time needed to check all possible assignments of values to the variables.)

- **matching:** Given a boys-girls compatibility graph, is there a complete matching?

- **3D matching:** Given a boys-girls-homes compatibility relation (that is, a set of boy-girl-home "triangles"), is there a complete matching (a set of disjoint triangles that covers all boys, all girls, and all homes)?

We know that matching can be solved by a reduction to max-flow. For **3D matching** there is a reduction too. Unfortunately, the reduction is *from 3SAT to* **3D matching**—and this is bad news for **3D matching**...

- **unary knapsack:** Given integers $a_1, \ldots, a_n$, and another integer K *in unary*, is there a subset of these integers that sum exactly to K?

- **knapsack:** Given integers $a_1, \ldots, a_n$, and another integer K *in binary*, is there a subset of these integers that sum exactly to K?

unary knapsack is in **P**—simply because the input is represented so wastefully, with about $n + K$ bits, so that a $O(n^2 K)$ dynamic programming algorithm, which would be exponential *in the length of the input* if K were represented in binary, is bounded by a polynomial in the length of the input. There is no polynomial algorithm known for the real **knapsack** problem. This illustrates that you have to represent your input in a sensible way, binary instead of unary, to draw meaningful conclusions.

13.5 NP, NP-completeness

Intuitively, a problem is in **NP** if it can be formulated as the problem of whether there is a solution

- They are *small*. In each case the certificate would never have to be longer than a polynomial in the length of the input.

- They are *easily checkable*. In each case there is a polynomial algorithm which takes as inputs the input of the problem and the alleged certificate, and checks whether the certificate is a valid one for this input. In the case of **3SAT**, the algorithm would just check that the truth assignment indeed satisfies all clauses. In the case of *Hamilton cycle* whether the given closed path indeed visits every node once. And so on.

- Every "yes" input to the problem has at least one certificate (possibly many), and each "no" input has none.

Not all decision problems have such certificates. Consider, for example, the problem **non-Hamiltonian graph:** Given a graph G, is it true that there is no Hamilton cycle in G? How would you prove to a suspicious person that a given large, dense, complex graph has *no* Hamilton cycle? Short of listing all cycles and pointing out that none visits all nodes once (a certificate that is certainly not succinct)?

These are examples of problems in NP:

- Given a graph G and an integer k, is there a simple path of length at least k in G?

- Given a set of integers $a_1, \ldots, a_n$, is there a subset S of them such that $\sum_{a \in S} a = \sum_{a \notin S} a$?

We now come to the formal definition.

DEFINITION 1 *A problem A is* **NP** *if there exist a polynomial p and a polynomial-time algorithm $V()$ such that $x \in A$ iff there exists a y, with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.*

We also call **P** the set of decision problems that are solvable in polynomial time. Observe every problem in **P** is also in **NP**.

We say that a problem A is **NP**-hard if for every N in **NP**, N is reducible to A, and that a problem A is **NP**-complete if it is **NP**-hard *and* it is contained in **NP**. As an exercise to understand the formal definitions, you can try to prove the following simple fact, that is one of the fundamental reasons why **NP**-completeness is interesting.

LEMMA 26
If A is **NP***-complete, then A is in* **P** *if and only if* **P** =**NP**.

So now, if we are dealing with some problem A that we can prove to be **NP**-complete, there are only two possibilities:

- A has no efficient algorithm.
- All the infinitely many problems in **NP**, including factoring and all conceivable optimization problems are in **P**.

If **P** =**NP**, then, given the statement of a theorem, we can find a proof in time polynomial in the number of pages that it takes to write the proof down.

If it was so easy to find proof, why do papers in mathematics journal have theorems *and* proofs, instead of just having theorems. And why theorems that had reasonably short proofs have been open questions for centuries? Why do newspapers publish solutions for crossword puzzles? If **P** =**NP**, whatever exists can be found efficiently. It is too bizarre to be true.

In conclusion, it is safe to assume **P** $\neq$ **NP**, or at least that the contrary will not be proved by anybody in the next decade, and it is *really* safe to assume that the contrary will not be proved by us in the next month. So, if our short-term plan involves finding an efficient algorithm for a certain problem, and the problem turns out to be **NP**-hard, then we should change the plan.

13.6 NP-completeness of Circuit-SAT

We will prove that the circuit satisfiability problem CSAT described in the previous notes is **NP**-complete.

Proving that it is in **NP** is easy enough: The algorithm $V()$ takes in input the description of a circuit C and a sequence of n Boolean values $x_1, \ldots x_n$, and $V(C, x_1, \ldots, x_n) = C(x_1, \ldots, x_n)$. I.e. V *simulates* or *evaluates* the circuit.

Now we have to prove that for every decision problem A in **NP**, we can find a reduction from A to CSAT. This is a difficult result to prove, and it is impossible to prove it really formally without introducing the *Turing machine* model of computation. We will prove the result based on the following fact, of which we only give an informal proof.

THEOREM 27

Suppose A is a decision problem that is solvable in $p(n)$ time by some program P, where n is the length of the input. Also assume that the input is represented as a sequence of bits.

Then, for every fixed n, there is a circuit C_n of size about $O((p(n)^2) \cdot (\log p(n))^{O(1)})$ such that for every input $x = (x_1, \ldots, x_n)$ of length n, we have

$$A(x) = C_n(x_1, \ldots, x_n)$$

That is, circuit C_n solves problem A on all the inputs of length n.

Furthermore, there exists an efficient algorithm (running in time polynomial in $p(n)$) that on input n and the description of P produces C_n.

The algorithm in the "furthermore" part of the theorem can be seen as the ultimate CAD tool, that on input, say, a C++ program that computes a boolean function, returns the description of a circuit that computes the same boolean function. Of course the generality is paid in terms of inefficiency, and the resulting circuits are fairly big.

PROOF: [Sketch] Without loss of generality, we can assume that the language in which P is written is some very low-level machine language (as otherwise we can compile it).

Let us restrict ourselves to inputs of length n. Then P runs in at most $p(n)$ steps. It then accesses at most $p(n)$ cells of memory.

At any step, the "global state" of the program is given by the content of such $p(n)$ cells plus $O(1)$ registers such as program counter etc. No register/memory cell needs to contain numbers bigger than $\log p(n) = O(\log n)$. Let $q(n) = (p(n) + O(1))O(\log n)$ denote the size of the whole global state.

We maintain a $q(n) \times p(n)$ "tableau" that describes the computation. The row i of the tableau is the global state at time i. Each row of the tableau can be computed starting from the previous one by means of a small circuit (of size about $O(q(n))$). In fact the microprocessor that executes our machine language is such a circuit (this is not totally accurate). □

Now we can argue about the **NP**-completeness of CSAT. Let us first think of how the proof would go if, say, we want to reduce the Hamiltonian cycle problem to CSAT. Then, given a graph G with n vertices and m edges we would construct a circuit that, given in input a sequence of n vertices of G, outputs 1 if and only if the sequence of vertices is a Hamiltonian cycle in G. How can we construct such a circuit? There is a computer program that given G and the sequence checks if the sequence is a Hamiltonian cycle, so there is also a circuit that given G and the sequence does the same check. Then we "hard-wire" G into the circuit and we are done. Now it remains to observe that the circuit is a Yes-instance of CSAT if and only if the graph is Hamiltonian.

The example should give an idea of how the general proof goes. Take an arbitrary problem A in **NP**. We show how to reduce A to Circuit Satisfiability.

Since A is in **NP**, there is some polynomial-time computable algorithm V_A and a polynomial p_A such that $A(x) =$ YES if and only if there exists a y, with $length(y) \leq p_A(length(x))$, such that $V(x, y)$ outputs YES.

Consider now the following reduction. On input x of length n, we construct a circuit C that on input y of length $p(n)$ decides whether $V(x, y)$ outputs YES or NOT.

Since V runs in time polynomial in $n+p(n)$, the construction can be done in polynomial time. Now we have that the circuit is satisfiable if and only if $x \in A$.

13.7 Proving More NP-completeness Results

Now that we have one **NP**-complete problem, we do not need to start "from scratch" in order to prove more **NP**-completeness results. Indeed, the following result clearly holds:

LEMMA 28
If A reduces to B, and B reduces to C, then A reduces to C.

PROOF: If A reduces to B, there is a polynomial time computable function f such that $A(x) = B(f(x))$; if B reduces to C it means that there is a polynomial time computable function g such that $B(y) = C(g(y))$. Then we can conclude that we have $A(x) = C(g(f(x))$, where $g(f())$ is computable in polynomial time. So A does indeed reduce to C. $\Box$

Suppose that we have some problem A in **NP** that we are studying, and that we are able to prove that CSAT reduces to A. Then we have that every problem N in **NP** reduces to CSAT, which we have just proved, and CSAT reduces to A, so it is also true that every problem in **NP** reduces to A, that is, A is **NP**-hard. This is very convenient: a single reduction from CSAT to A shows the existence of all the infinitely many reductions needed to establish the **NP**-hardness of A. This is a general method:

LEMMA 29
Let C be an **NP***-complete problem and A be a problem in* **NP***. If we can prove that C reduces to A, then it follows that A is* **NP***-complete.*

Right now, literally thousands of problems are known to be **NP**-complete, and each one (except for a few "root" problems like CSAT) has been proved **NP**-complete by way of a single reduction from another problem previously proved to be **NP**-complete. By the definition, all **NP**-complete problems reduce to each other, so the body of work that lead to the proof of the currently known thousands of **NP**-complete problems, actually implies *millions* of pairwise reductions between such problems.

13.8 NP-completeness of SAT

We defined the CNF Satisfiability Problem (abbreviated SAT) above. SAT is clearly in **NP**. In fact it is a special case of Circuit Satisfiability. (Can you see why?) We want to prove that SAT it is **NP**-hard, and we will do so by reducing from Circuit Satisfiability.

First of all, let us see how *not* to do the reduction. We might be tempted to use the following approach: given a circuit, transform it into a Boolean CNF formula that computes the same Boolean function. Unfortunately, this approach cannot lead to a polynomial time reduction. Consider the Boolean function that is 1 iff an odd number of inputs is 1. There is a circuit of size $O(n)$ that computes this function for inputs of length n. But the smallest CNF for this function has size more than 2^n.

This means we cannot translate a circuit into a CNF formula of comparable size that computes the same function, but we may still be able to transform a circuit into a CNF formula such that the circuit is satisfiable iff the formula is satifiable (although the circuit and the formula do compute somewhat different Boolean functions).

We now show how to implement the above idea. We will need to add new variables. Suppose the circuit C has m gates, including input gates, then we introduce new variables $g_1, \ldots, g_m$, with the intended meaning that variable g_j corresponds to the output of gate j.

We make a formula F which is the AND of $m + 1$ sub-expression. There is a sub-expression for every gate j, saying that the value of the variable for that gate is set in accordance to the value of the variables corresponding to inputs for gate j.

We also have a $(m + 1)$-th term that says that the output gate outputs 1. There is no sub-expression for the input gates.

For a gate j, which is a NOT applied to the output of gate i, we have the sub-expression

$$(g_i \vee g_j) \wedge (\bar{g}_i \vee \bar{g}_j)$$

For a gate j, which is a AND applied to the output of gates i and l, we have the sub-expression

$$(\bar{g}_j \vee g_i) \wedge (\bar{g}_j \vee g_l) \wedge (g_j \vee \bar{g}_i \vee \bar{g}_l)$$

Similarly for OR.

This completes the description of the reduction. We now have to show that it works. Suppose C is satisfiable, then consider setting g_j being equal to the output of the j-th gate of C when a satisfying set of values is given in input. Such a setting for $g_1, \ldots, g_m$ satisfies F.

Suppose F is satisfiable, and give in input to C the part of the assignment to F corresponding to input gates of C. We can prove by induction that the output of gate j in C is also equal to g_j, and therefore the output gate of C outputs 1.

So C is satisfiable if and only if F is satisfiable.

13.9 NP-completeness of 3SAT

SAT is a much simpler problem than Circuit Satisfiability, if we want to use it as a starting point of **NP**-completeness proofs. We can use an even simpler starting point: 3-CNF Formula Satisfiability, abbreviated 3SAT. The 3SAT problem is the same as SAT, except that each OR is on precisely 3 (possibly negates) variables. For example, the following is an instance of 3SAT:

$$(x_2 \vee \bar{x}_4 \vee x_5) \wedge (x_1 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee x_3 \vee x_5) \tag{13.1}$$

Certainly, 3SAT is in **NP**, just because it's a special case of SAT.

In the following we will need some terminology. Each little OR in a SAT formula is called a *clause*. Each occurrence of a variable, complemented or not, is called a *literal*.

We now prove that 3SAT is **NP**-complete, by reduction from SAT. Take a formula F of SAT. We transform it into a formula F' of 3SAT such that F' is satisfiable if and only if F is satisfiable.

Each clause of F is transformed into a sub-expression of F'. Clauses of length 3 are left unchanged.

A clause of length 1, such as (x) is changed as follows

$$(x \vee y_1 \vee y_2) \wedge (x \vee y_1 \vee \bar{y}_2)(x \vee \bar{y}_1 \vee y_2) \wedge (x \vee \bar{y}_1 \vee \bar{y}_2)$$

where y_1 and y_2 are two new variables added specifically for the transformation of that clause.

A clause of length 2, such as $x_1 \vee x_2$ is changed as follows

$$(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \bar{y})$$

where y is a new variable added specifically for the transformation of that clause.

For a clause of length $k \geq 4$, such as $(x_1 \vee \cdots \vee x_k)$, we change it as follows

$$(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_4) \wedge \cdots \wedge (\bar{y}_{k-3} \vee x_{k-1} \vee x_k)$$

where $y_1, \cdots, y_{k-3}$ are new variables added specifically for the transformation of that clause.

We now have to prove the correctness of the reduction.

- We first argue that if F is satisfiable, then there is an assignment that satisfies F'.

  For the shorter clauses, we just set the y-variables arbitrarily. For the longer clause it is slightly more tricky.

- We then argue that if F is not satisfiable, then F' is not satisfiable.

  Fix an assignment to the x variables. Then there is a clause in F that is not satisfied. We argue that one of the derived clauses in F' is not satisfied.

13.10 Some NP-complete Graph Problems

13.10.1 Independent Set

Given an undirected non-weighted graph $G = (V, E)$, an *independent set* is a subset $I \subseteq V$ of the vertices such that no two vertices of I are adjacent. (This is similar to the notion of a *matching*, except that it involves vertices and not edges.)

We will be interested in the following optimization problem: given a graph, find a largest independent set. We have seen that this problem is easily solvable in forests. In the general case, unfortunately, it is much harder.

The problem models the execution of conflicting tasks, it is related to the construction of error-correcting codes, and it is a special case of more interesting problems. We are going to prove that it is not solvable in polynomial time unless $\mathbf{P} = \mathbf{NP}$.

First of all, we need to formulate it as a decision problem:

- Given a graph G and an integer k, does there exist an independent set in G with at least k vertices?

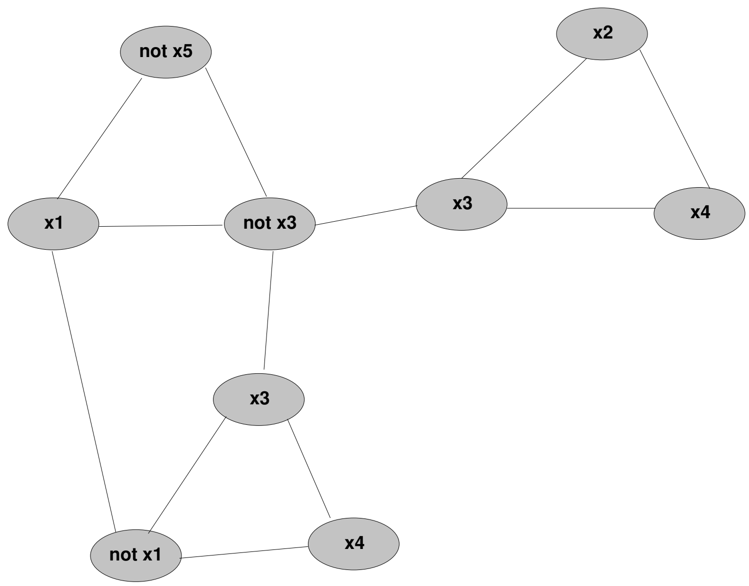


Figure 13.2: The reduction from 3SAT to Independent Set.

It is easy to see that the problem is in NP. We have to see that it is NP-hard. We will reduce 3SAT to Maximum Independent Set.

Starting from a formula ϕ with n variables $x_1, \ldots, x_n$ and m clauses, we generate a graph G_ϕ with $3m$ vertices, and we show that the graph has an independent set with at least m vertices if and only if the formula is satisfiable. (In fact we show that the size of the largest independent set in G_ϕ is equal to the maximum number of clauses of ϕ that can be simultaneously satisfied. — This is more than what is required to prove the NP-completeness of the problem)

The graph G_ϕ has a triangle for every clause in ϕ. The vertices in the triangle correspond to the three literals of the clause.

Vertices in different triangles are joined by an edge iff they correspond to two literals that are one the complement of the other. In Figure 13.2 we see the graph resulting by applying the reduction to the following formula:

$$(x_1 \vee \neg x_5 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_3 \vee x_2 \vee x_4)$$

To prove the correctness of the reduction, we need to show that:

- If ϕ is satisfiable, then there is an independent set in G_ϕ with at least m vertices.
- If there is an independent set in G with at least m vertices, then ϕ is satisfiable.

From Satisfaction to Independence. Suppose we have an assignment of Boolean values to the variables $x_1, \ldots, x_n$ of ϕ such that all the clauses of ϕ are satisfied. This means that for every clause, at least on of its literals is satisfied by the assignment. We construct an independent set as follows: for every triangle we pick a node that corresponds to a satisfied literal (we break ties arbitrarily). It is impossible that two such nodes are adjacent, since only nodes that corresponds to a literal and its negation are adjacent; and they cannot be both satisfied by the assignment.

From Independence to Satisfaction. Suppose we have an independent set I with m vertices. We better have exactly one vertex in I for every triangle. (Two vertices in the same triangle are always adjacent.) Let us fix an assignment that satisfies all the literals that correspond to vertices of I. (Assign values to the other variables arbitrarily.) This is a consistent rule to generate an assignment, because we cannot have a literal and its negation in the independent set). Finally, we note how every clause is satisfied by this assignment.

Wrapping up:

- We showed a reduction $\phi \to (G_\phi, m)$ that given an instance of 3SAT produces an instance of the decision version of Maximum Independent Set.
- We have the property that ϕ is satisfiable (answer YES for the 3SAT problem) if and only if G_ϕ has an independent set of size at least m.
- We knew 3SAT is NP-hard.
- Then also Max Independent Set is NP-hard; and so also NP-complete.

13.10.2 Maximum Clique

Given a (undirected non-weighted) graph $G = (V, E)$, a *clique* K is a set of vertices $K \subseteq V$ such that *any two* vertices in K are adjacent. In the Maximum Clique problem, given a graph G we want to find a largest clique.

In the decision version, given G and a parameter k, we want to know whether or not G contains a clique of size at least k. It should be clear that the problem is in NP.

We can prove that Maximum Clique is NP-hard by reduction from Maximum Independent Set. Take a graph G and a parameter k, and consider the graph G', such that two vertices in G' are connected by an edge if and only if they are not connected by an edge in G. We can observe that every independent set in G is a clique in G', and every clique in G' is an independent set in G. Therefore, G has an independent set of size at least k if and only if G' has a clique of size at least k.

13.10.3 Minimum Vertex Cover

Given a (undirected non-weighted) graph $G = (V, E)$, a *vertex cover* C is a set of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$, either $u \in C$ or $v \in C$ (or, possibly, both). In the Minimum Vertex Cover problem, given a graph G we want to find a smallest vertex cover.

In the decision version, given G and a parameter k, we want to know whether or not G contains a vertex cover of size at most k. It should be clear that the problem is in NP.

We can prove that Minimum Vertex Cover is NP-hard by reduction from Maximum Independent Set. The reduction is based on the following observation:

Lemma 30

If I is an independent set in a graph $G = (V, E)$, then the set of vertices $C = V - I$ that are not *in I is a vertex cover in G. Furthermore, if C is a vertex cover in G, then $I = V - C$ is an independent set in G.*

Proof: Suppose C is not a vertex cover: then there is some edge (u, v) neither of whose endpoints is in C. This means both the endpoints are in I and so I is not an independent set, which is a contradiction. For the "furthermore" part, suppose I is not an independent set: then there is some edge $(u, v) \in E$ such that $u \in I$ and $v \in I$, but then we have an edge in E neither of whose endpoints are in C, and so C is not a vertex cover, which is a contradiction. □

Now the reduction is very easy: starting from an instance (G, k) of Maximum Independent set we produce an instance $(G, n - k)$ of Minimum Vertex Cover.

13.11 Some NP-complete Numerical Problems

13.11.1 Subset Sum

The **Subset Sum** problem is defined as follows:

- Given a sequence of integers $a_1, \ldots, a_n$ and a parameter k,
- Decide whether there is a subset of the integers whose sum is exactly k. Formally, decide whether there is a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = k$.

Subset Sum is a true *decision problem*, not an optimization problem forced to become a decision problem. It is easy to see that Subset Sum is in NP.

We prove that Subset Sum is NP-complete by reduction from Vertex Cover. We have to proceed as follows:

- Start from a graph G and a parameter k.
- Create a sequence of integers and a parameter k'.
- Prove that the graph has vertex cover with k vertices iff there is a subset of the integers that sum to k'.

Let then $G = (V, E)$ be our input graph with n vertices, and let us assume for simplicity that $V = \{1, \ldots, n\}$, and let k be the parameter of the vertex cover problem.

We define integers $a_1, \ldots, a_n$, one for every vertex; and also integers $b_{(i,j)}$, one for every edge $(i, j) \in E$; and finally a parameter k'. We will define the integers a_i and $b_{(i,j)}$ so that if we have a subset of the a_i and the $b_{(i,j)}$ that sums to k', then: the subset of the a_i corresponds to a vertex cover C in the graph; and the subset of the $b_{(i,j)}$ corresponds to the edges in the graph such that exactly one of their endpoints is in C. Furthermore the construction will force C to be of size k.

How do we define the integers in the subset sum instance so that the above properties hold? We represent the integers in a matrix. Each integer is a row, and the row should be seen as the base-4 representation of the integer, with $|E| + 1$ digits.

The first column of the matrix (the "most significant digit" of each integer) is a special one. It contains 1 for the a_is and 0 for the $b_{(i,j)}$s.

Then there is a column (or digit) for every edge. The column (i, j) has a 1 in a_i, a_j and $b_{(i,j)}$, and all 0s elsewhere.

The parameter k' is defined as

$$k' := k \cdot 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^i$$

This completes the description of the reduction. Let us now proceed to analyze it.

From Covers to Subsets Suppose there is a vertex cover C of size k in G. Then we choose all the integers a_i such that $i \in C$ and all the integers $b_{(i,j)}$ such that exactly one of i and j is in C. Then, when we sum these integers, doing the operation in base 4, we have a 2 in all digits except for the most significant one. In the most significant digit, we are summing one $|C| = k$ times. The sum of the integers is thus k'.

From Subsets to Covers Suppose we find a subset $C \subseteq V$ and $E' \subseteq E$ such that

$$\sum_{i \in C} a_i + \sum_{(i,j) \in E'} b_{(i,j)} = k'$$

First note that we never have a carry in the $|E|$ less significant digits: operations are in base 4 and there are at most 3 ones in every column. Since the $b_{(i,j)}$ can contribute at most one 1 in every column, and k' has a 2 in all the $|E|$ less significant digits, it means that for every edge (i, j) C must contain either i or j. So C is a cover. Every a_i is at least $4^{|E|}$, and k' gives a quotient of k when divided by $4^{|E|}$. So C cannot contain more than k elements.

13.11.2 Partition

The **Partition** problem is defined as follows:

- Given a sequence of integers $a_1, \ldots, a_n$.
- Determine whether there is a partition of the integers into two subsets such the sum of the elements in one subset is equal to the sum of the elements in the other.

  Formally, determine whether there exists $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = (\sum_{i=1}^{n} a_i)/2$.

Clearly, Partition is a special case of Subset Sum. We will prove that Partition is is NP-hard by reduction from Subset Sum.[4]

Given an instance of Subset Sum we have to construct an instance of Partition. Let the instance of Subset Sum have items of size $a_1, \ldots, a_n$ and a parameter k, and let $A = \sum_{i=1}^{n} a_i$.

Consider the instance of Partition $a_1, \ldots, a_n, b, c$ where $b = 2A - k$ and $c = A + k$.

Then the total size of the items of the Partition instance is $4A$ and we are looking for the existence of a subset of $a_1, \ldots, a_n, b, c$ that sums to $2A$.

It is easy to prove that the partition exists if and only if there exists $I \subseteq \{1, \ldots, n\}$ such that $\sum_i a_i = k$.

[4]The reduction goes in the non-trivial direction!

13.11.3 Bin Packing

The **Bin Packing** problem is one of the most studied optimization problems in Computer Science and Operation Research, possibly the second most studied after TSP. It is defined as follows:

- Given items of size $a_1, \ldots, a_n$, and given unlimited supply of bins of size B, we want to pack the items into the bins so as to use the minimum possible number of bins.

You can think of bins/items as being CDs and MP3 files; breaks and commercials; bandwidth and packets, and so on.

The decision version of the problem is:

- Given items of size $a_1, \ldots, a_n$, given bin size B, and parameter k,
- Determine whether it is possible to pack all the items in k bins of size B.

Clearly the problem is in NP. We prove that it is NP-hard by reduction from Partition.

Given items of size $a_1, \ldots, a_n$, make an instance of Bin Packing with items of the same size and bins of size $(\sum_i a_i)/2$. Let $k = 2$.

There is a solution for Bin Packing that uses 2 bins if and only if there is a solution for the Partition problem.

13.12 Approximating NP-Complete Problems

What does it mean to approximate the solution of an NP-complete problem, when we have so far been considering only questions with yes/no answers? Actually, many of the problems we have looked at are most naturally stated as *optimization problems*, which we can approximate. For example, in **TSP** the yes/no question is whether there is a cycle visiting each node once of length at most K; the optimization problem is to find the *shortest* cycle; and the approximation problem is to find a *short* cycle, i.e. one whose length is as short as one can afford to compute. Similarly, for **Vertex Cover** the yes/no question is whether there is subset of vertices touching each edge, of cardinality at most k; the optimization problem is to find the vertex cover with the *fewest* vertices; and the approximation problem is to find a vertex cover with *few* vertices, i.e. one whose cardinality is as small as one can afford to compute.

To measure how good an approximation we compute, we need a measure of error. Suppose $c > 00$ is an approximation to the exact value $c^* > 0$. (For Vertex Cover, c is the size of a vertex cover found by an approximate algorithm, and c^* is the true minimum vertex cover size.) Then the *ratio bound* (or *performance ratio* or *approximation ratio*) of c is ρ where

$$\max\left\{\frac{c}{c^*}, \frac{c^*}{c}\right\} \le \rho \ .$$

Note that $\rho \ge 1$, and $\rho = 1$ exactly when $c = c^*$. If the size of the problem whose answer is c^* is n, we may also write $\rho(n)$ to indicate that the error may depend on n.

For example, $\rho(n) = 2$, means that the computed result c never differs from c^* by more than a factor of 2.

We will characterize our approximation algorithms by the value of ρ that they guarantee. It turns out that some NP-complete problems let us compute approximations for small ρ in polynomial time, but for other NP-complete problems computing any approximation with bounded ρ is still NP-complete.

13.12.1 Approximating Vertex Cover

Recall that a *vertex cover* C is a subset $C \subset V$ of the set of vertices of a graph $G = (V, E)$ with the property that every edge in E has a vertex in C as an end point. The vertex cover optimization problem is to find the vertex cover of smallest cardinality c^*. It turns out that the following simple greedy algorithm never finds a cover more than twice too large, i.e. $c \leq 2c^*$, or $\rho = 2$.

$C = \emptyset$ // initialize an empty vertex cover
for all $(u, v) \in E$
    if $u \notin C$ and $v \notin C$ then $C := C \cup \{u, v\}$
return C

The set C can be represented using a Boolean vector of size $|V|$, so that checking membership in C can be done in constant time. The running time is obviously $O(|E|+|V|)$, since we use $O(|V|)$ time to initialize the representation of C and then we do a constant amount of work for every edge.

THEOREM 31
The above algorithm satisfies $\rho \leq 2$, i.e. if C is the cover computed by the algorithm and C^* is an optimal cover, then $|C| \leq 2|C^*|$.

PROOF: First of all, it should be clear that the algorithm returns a vertex cover. Every time an edge is found that is not covered by the current set C, then both endpoints are added to C, thus guaranteeing that the edge is covered.

For the approximation, let M be the set of edges (u, v) such that when (u, v) is considered in the `for` loop, the vertices u, v are added to C. By construction, $|C| = 2|M|$. Furthermore, M is a matching, and so each edge of M must be covered by a distinct vertex; even if C^* is an optimum cover, we must have $|C^*| \geq |M|$, and so $|C| \leq 2|C^*|$. $\square$

Let us now consider another way of achieving an approximation within a factor of 2. Given a graph $G = (V, E)$, write a linear program having a variable x_v for each vertex v, and structured as follows:

$$
\begin{array}{lll}
\min & \sum_v x_v & \\
\text{s.t.} & & \\
& x_u + x_v \geq 1 & \text{(for each } (u,v) \in E) \\
& x_v \geq 0 & \text{(for each } v \in V)
\end{array}
$$

Now, let C be a vertex cover, and consider the solution defined by setting $x_v = 0$ if $v \notin C$, and $x_v = 1$ if $v \in C$. Then the solution is feasible, because all values are ≥ 0, and each (u, v) is covered by C, so that $x_u + x_v$ is equal to either one or two (and so is at least one). Furthermore, the cost of the solution is equal to the cardinality of C. This implies that the cost of the optimum of the linear program is at most the cost of an optimum solution for the vertex cover problem.

Let's now solve optimally the linear program, and get the optimal values x_v^*. By the above reasoning, $\sum_x x_v^* \leq c^*$. Let us then define a vertex cover C, as follows: a vertex v belongs to C if and only if $x_v^* \geq 1/2$. We observe that:

- C is a valid vertex cover, because for each edge (u, v) we have $x_u^* + x_v^* \geq 1$, so at least one of x_u^* or x_v^* must be at least $1/2$, and so at least one of u or v belongs to C.
- $\sum_v x_v^* \geq |C|/2$, because every vertex in C contributes at least $1/2$ to the sum, and the vertices not in C contribute at least 0. Equivalently, $|C| \leq \sum_v x_v^* \leq 2c^*$ where c^* is the cardinality of an optimum vertex cover.

So, again, we have a 2-approximate algorithm.

No algorithm achieving an approximation better than 2 is known. If $\mathbf{P}\neq\mathbf{NP}$, then there is no polynomial time algorithm that achieves an approximation better than 7/6 (proved by Håstad in 1997). A few months ago, a proof has been announced that, if $\mathbf{P}\neq\mathbf{NP}$, then there is no polynomial time algorithm that achieves an approximation better than 4/3 (the result is still unpublished).

The greedy algorithm might have been independently discovered by several people and was never published. It appears to have been first mentioned in a 1974 manuscript. The linear programming method is from the late 1970s and it also applies to the *weighted* version of the problem, where every vertex has a positive weight and we want to find the vertex cover of minimum total weight.

The LP-based algorithm was one of the first applications of the following general methodology: formulate a linear program that "includes" all solutions to an NP-complete problem, plus fractional solutions; solve the linear program optimally; "round" the, possibly fractional, optimum LP solution into a solution for the NP-complete problem; argue that the rounding process creates a solution whose cost is not too far away from the cost of the LP optimum, and thus not too far away from the cost of the optimum solution of the NP-complete problem. The same methodology has been applied to several other problems, with considerable success.

13.12.2 Approximating TSP

In the Travelling Salesman Problem (TSP), the input is an undirected graph $G = (V, E)$ and weights, or distances, $d(u, v)$ for each edge (u, v). (Often, the definition of the problem requires the graph to be complete, i.e., E to contain all possible edges.) We want to find an order in which to visit all vertices exactly once, so that the total distance that we have to travel is as small as possible. The vertices of the graph are also called "cities." The idea is that there is a salesman that has to go through a set of cities, he knows the distances between cities, and he wants to visit every city and travel the smallest total distance. This problem turns out to be extremely hard to approximate in this general formulation, mostly because the formulation is too general.

A more interesting formulation requires the graph to be complete and the distances to satisfy the "triangle inequality" that says that for every three cities u, v, w we have $d(u,w) \leq d(u,v) + d(v,w)$. This version of the problem is called "metric" TSP, or ΔTSP, where the letter Δ represents the "triangle" inequality.[5]

Consider the following algorithm for metric TSP: find a minimum spanning tree T in G, then define a cycle that starts at the root of the tree, and then moves along the tree in the order in which a DFS would visit the tree. In this way, we start and end at the root, we visit each vertex at least once (possibly, several times) and we pass through each edge of the minimum spanning tree exactly twice, so that the total length of our (non-simple) cycle is $2 \cdot cost(T)$.

What about the fact that we are passing through some vertices multiple times? We can straighten our tour in the following way: suppose a section of the tour is $u \to v \to w$, and v is a vertex visited by the tour some other time. Then we can change that section to $u \to w$, and by the triangle inequality we have only improved the tour. This action reduced the number of repetitions of vertices, so eventually we get a tour with no repetitions, and whose cost is only smaller than $2 \cdot cost(T)$.

Finally, take an optimal tour C, and remove an edge from it. Then what we get is a spanning tree, whose cost must be at least $cost(T)$. Therefore, the cost the optimal tour is at least the cost of the minimum spanning tree; our algorithm finds a tour of cost at most twice the cost of the minimum spanning tree and so the solution is 2-approximate.

This algorithm has never been published and is of uncertain attribution. An algorithm achieving a 3/2-approximation was published in 1976, based on similar principles. There has been no improvement since then, but there are reasons to believe that a 4/3-approximation is possible.

In 2000 it has been shown that, if **P**$\neq$**NP**, then there is no polynomial time algorithm achieving an approximation of about 1.01 for metric TSP.

If points are in the plane, and the distance is the standard geometric distance, then any approximation ratio bigger than one can be achieved in polynomial time (the algorithm is by Arora, 1996). This remains true even in geometries with more than two dimensions, but not if the number of dimensions grows at least logarithmically with the number of cities.

13.12.3 Approximating Max Clique

Håstad proved in 1996 that there can be no algorithm for Max Clique achieving an approximation ratio $\rho(n) = n^{.99}$, unless **NP** has polynomial time probabilistic algorithms (which is almost but not quite the same as **P**=**NP**). In fact, any constant smaller than one can be replaced in the exponent, and the result is still true.

Notice that the algorithm that returns a single vertex is an n-approximation: a vertex is certainly a clique of size 1, and no clique can be bigger than n.

[5] The metric TSP is equivalent to the problem where we are given a graph $G = (V, E)$, not necessarily complete, and weights, or distances, $d(u, v)$ for each edge (u, v), and we want to find an order in which to visit each vertex *at least* once, so that the total distance that we have to travel is as small as possible. (Notice that we may come back to the same vertex more than once.) You could try to prove the equivalence as an exercise.

We will see a $(n/\log n)$-approximation. In light of Håstad's result we cannot hope for much more.[6]

Given a graph $G = (V, E)$, the algorithm divides the set of vertices in $k = n/logn$ blocks $B_1, \ldots, B_k$, each block containing $\log n$ vertices. (It's not important how the blocks are chosen, as long as they all have size $\log n$.) Then, for each block B_i, the algorithm finds the largest subset $K_i \subseteq B_i$ that is a clique in G. This can be done in time $O(n(\log n)^2)$ time, because there are n possible subset, and checking that a set is a clique can be done in quadratic time (in the size of the set). Finally, the algorithm returns the largest of the cliques found in this way.

Let K^* be the largest clique in the graph. Clearly, there must be one block B_i that contains at least $|K^*|/k$ vertices of K^*, and when the algorithm considers B_i it will find a clique with at least $|K^*|/k$ vertices. Possibly, the algorithm might find a bigger clique in other blocks. In any case, the size of the clique given in output by the algorithm is at least $1/k = \log n/n$ times the size of K^*.

[6]There is reason to believe that the right result is that a $n/2^{\sqrt{\log n}}$-approximation is possible in polynomial time, but not better. So far, the best known approximation algorithm has roughly $\rho(n) = n/(\log n)^2$, while a result by Khot shows that, under strong assumption, $\rho(n) = n/2^{(\log n)^{1-o(1)}}$ is not achievable in polynomial time.