

# CMSC 451

## Design and Analysis of Computer Algorithms<sup>1</sup>

David M. Mount  
Department of Computer Science  
University of Maryland  
Spring 2008

---

<sup>1</sup>Copyright, David M. Mount, 2008 Dept. of Computer Science, University of Maryland, College Park, MD, 20742. These lecture notes were prepared by David Mount for the course CMSC 451, Design and Analysis of Computer Algorithms, at the University of Maryland. Permission to use, copy, modify, and distribute these notes for educational purposes and without fee is hereby granted, provided that this copyright notice appear in all copies.

# Introduction to Algorithm Design

**What is an algorithm?** A common definition of an *algorithm* is:

Any well-defined computational procedure that takes some values as *input* and produces some values as *output*.

Like a cooking recipe, an algorithm provides a step-by-step method for solving a computational problem. Implicit in this definition is the constraint that procedure defined by the algorithm must eventually terminate.

Algorithms should be distinguished from *software*. Often, software can be thought of as a concrete implementation of an algorithm, written in a particular programming language, and is designed to be compiled and run on certain computer systems. In contrast, algorithms are mathematical entities, which can be thought of as existing in the absence of any real computer. Perhaps more intuitively an algorithm can be thought of running on some sort of *idealized computer*. At an abstract level, we can think of computing as involving of three elements.

$$\text{computing} = \text{data} + \text{program} + \text{control},$$

where the *data* is the information that is being manipulated, the *program* is an appropriate encoding of some algorithm, and *control* is the mechanism (hardware) that carries out the steps of the program. Although no one can deny that hardware technology has made monumental advancements since the dawn of computers in the middle part of the 20th century, all computers that have ever existed follow this general structure.

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

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

The focus of this course is on how to design good algorithms, and how to analyze their efficiency. This is among the most basic aspects of good programming.

**Our Guiding Philosophy:** First develop a good (and ideally clean and simple) algorithm design, and then implement it and fine tune its performance.

**Course Overview:** This course will consist of a number of major sections. The first will be a short review of some preliminary material, including asymptotics, summations, and recurrences and sorting. These have been covered in earlier courses, and so we will breeze through them pretty quickly. Next we will consider a number of common graph algorithms. You will have seen some of these in prior courses. We will then cover three different methods of designing efficient algorithms: greedy algorithms, divide and conquer, and dynamic programming. Following this we will discuss network flows, an important computational problem which is often used for solving other (seemingly unrelated) problems.

Most of the emphasis of the first portion of the course will be on problems that can be solved efficiently, in the latter portion we will discuss intractability and NP-hard problems. These are problems for which no efficient

solution is known. Finally, we will discuss methods to approximate NP-hard problems, and how to prove how close these approximations are to the optimal solutions.

This year we are using a new text book (at least it is new to me). Consequently, our presentation of some of the material will not be the same as in the text, and it may appear in different order.

**Issues in Algorithm Design:** Algorithms are mathematical objects (in contrast to the much more concrete notion of a computer program implemented in some programming language and executing on some machine). As such, we can reason about the properties of algorithms mathematically. When designing an algorithm there are two fundamental issues to be considered: *correctness* and *efficiency*.

**Correctness:** It is important to justify an algorithm's correctness mathematically. For very complex algorithms, this typically requires a careful mathematical proof, which may require the proof of many lemmas and properties of the solution, upon which the algorithm relies. For simple algorithms (BubbleSort, for example) a short intuitive explanation of the algorithm's basic invariants is sufficient. (For example, in BubbleSort, the principal invariant is that on completion of the  $i$ th iteration, the last  $i$  elements are in their proper sorted positions.)

**Efficiency:** Establishing efficiency is a much more complex endeavor. Intuitively, an algorithm's efficiency is a function of the amount of computational resources it requires, measured typically as execution time and the amount of space, or memory, that the algorithm uses. The amount of computational resources can be a complex function of the size and structure of the input set. In order to reduce matters to their simplest form, it is common to consider efficiency as a function of input size. Among all inputs of the same size, we consider the maximum possible running time. This is called *worst-case analysis*. It is also possible, and often more meaningful, to measure *average-case analysis*. Average-case analyses tend to be more complex, and may require that some probability distribution be defined on the set of inputs.

To keep matters simple, we will usually focus on worst-case analysis in this course. You should be mindful, however, that worst-case analysis is not always the best way to analyze an algorithm's performance. There are some algorithms that perform well for almost all inputs, but may perform abysmally on a very tiny fraction of inputs. Luckily, none of the algorithms that we will see this semester have this undesirable property.

**Describing Algorithms:** Throughout out this course, when you are asked to present an algorithm, this means that you need to do three things:

**Present the Algorithm:** Present a clear, simple and unambiguous description of the algorithm (in pseudo-code, for example). A guiding principal here is to *keep it simple*. Uninteresting details should be kept to a minimum, so that the key computational issues stand out. For example, it is not necessary to declare variables whose purpose is obvious, and it is often simpler and clearer to simply say, "Add element  $X$  to the end of list  $L$ " than to present code to do this or use some arcane syntax, such as "*theList.insertAtEnd(theCurrentElement)*." Although this more verbose style is good when writing large complex programs, where you may have hundreds of procedures and many different variables, algorithms are typically short, and conciseness and clarity are valued.

**Prove its Correctness:** Present a justification or proof of the algorithm's correctness. Your justification should assume that the reader is someone of similar background as yourself, say another student in this class, and should be convincing enough make a skeptic believe that your algorithm does indeed solve the problem correctly. Avoid rambling about obvious or trivial elements. A good proof provides an overview of what the algorithm does, and then focuses on any tricky elements that may not be obvious.

**Analyze its Efficiency:** Present a worst-case analysis of the algorithms efficiency, typically its running time (but also its space, if space is an issue). Sometimes this is straightforward, but if not, concentrate on the parts of the analysis that are not obvious.

Note that the presentation does not need to be in this order. Often it is good to begin with an explanation of how you derived the algorithm, emphasizing particular elements of the design that establish its correctness and efficiency. Then, once this groundwork has been laid down, present the algorithm itself. If this seems to be a bit abstract now, don't worry. We will see many examples of this process throughout the semester.

## Algorithm Design Review: Stable Marriage Problem

**Read:** Review Chaps. 1–2 in KT. (Unless otherwise specified all readings are from *Algorithm Design* by Kleinberg and Tardos.)

**Algorithm Analysis:** In the next couple of lectures we will review some of the basic elements of algorithm analysis, which were covered in previous courses. These include basic algorithm design, proofs of correctness, analysis of running times, and mathematical basics, such as asymptotics, summations, and recurrences.

**Stable Marriage:** As an introduction to algorithm design, we will consider a well known discrete computational problem, called the *stable marriage problem*. In spite of the name, the problem's original formulation had nothing to do with the institution of marriage, but it was motivated by a number of practical applications where it was desired to set up pairings between entities, e.g., assigning medical school graduates to hospitals for resident training, assigning student applications to universities, or assigning students to fraternities or sororities. In all these applications we may view two groups of entities (e.g., students and university admission slots) where we wish to make an assignment from one to the other and where each side has some notion of preference. For example, each student has a ranking of the universities he/she wishes to attend and each university has a ranking of students it wants to admit. The goal is to produce a pairing that is in some sense stable.

We will couch this problem abstract in terms of a group of  $n$  men and  $n$  women that wish to marry, and we will place the algorithm in the role of a matchmaker. First, as suggested by the term “marriage,” the outcome of our process will be a pairing, one man to one woman and vice versa. Second, we assume that there is some notion of preference involved. This will be modeled by assuming that each man provides a rank ordering of the women according to decreasing preference level and vice versa. Consider the following example. There are three men in our system: Bill (B), Ted (T), and a robot named (X). There are three women: Mary-Kate (MK), Ashley (A), and a fembot named Y (Y). Here are their rank orderings (from most to least desired).

Men			Women		
Bill (B)	Ted (T)	Robot X	Mary-Kate (MK)	Ashley (A)	Fembot Y
MK	MK	Y	X	X	B
A	Y	MK	B	T	T
Y	A	A	T	B	X

**Stability:** There are many ways in which we might define the notion of stable pairing of men to women. Clearly, we cannot guarantee that everyone will get their first preference. (Both Bill and Ted list Mary-Kate first.) Intuitively, it should not be the case that there is a single unmarried pair would find it in their simultaneous best interest to ignore the pairing set up by the matchmaker and elope. That is, there should be no man who says to another woman, “I love you more than my fiancée—let's run away together!” where the woman agrees, because she loves this man more than her fiancé. If no such *instability* exists, the marriage is said to be stable.

For example, among the following, can you spot which are stable and which are unstable?

Assignment I	Assignment II	Assignment III
B ↔ A	B ↔ Y	B ↔ Y
T ↔ Y	T ↔ MK	T ↔ A
X ↔ MK	X ↔ A	X ↔ MK

It is actually not that easy to see. To make it easier, after each person I have listed in brackets the people that they would have preferred to their assigned choice. Now can you spot the unstable one?

Assignment I	Assignment II	Assignment III
B [MK] $\leftrightarrow$ A [X, T]	B [MK, A] $\leftrightarrow$ Y [ ]	B [MK, A] $\leftrightarrow$ Y [ ]
T [MK] $\leftrightarrow$ Y [B]	T [ ] $\leftrightarrow$ MK [X, B]	T [MK, Y] $\leftrightarrow$ A [X]
X [Y] $\leftrightarrow$ MK [ ]	X [Y, MK] $\leftrightarrow$ A [ ]	X [Y] $\leftrightarrow$ MK [ ]

The only unstable one is II. Observe that Bill would have preferred Mary-Kate over his assigned fiancée Y, and Mary-Kate would have preferred Bill to her fiancé Ted. Thus, the pair of engagements ( $B \leftrightarrow Y$ ) and ( $T \leftrightarrow MK$ ) is an example of an instability. Observe that there are two stable matches, I and III. This might make you wonder whether among all stable matchings, are some better than others? What would “better” mean? We will not consider this issue here, but it is an interesting one.

**The Gale-Shapley Algorithm:** The algorithm that we will describe is essentially due to Gale and Shapley, who considered this problem back in 1962. My presentation will differ slightly from the one given in Kleinberg and Tardos. The general process involves two steps, a man makes a proposal to a woman, and the woman either accepts or declines. (There is an obvious sexual bias here, but if you prefer you can imagine that the women make the proposals and the men accept or decline. Remarkably these two variants offer significantly different benefits. In particular, it can be shown that whichever group does the proposing tends to get assigned mates of higher preference.) The algorithm presented in the text is actually quite general, since it does not mandate the order in which the men make their proposals. The version that I will present has a somewhat more parallel flavor, and is a bit easier to work out by hand.

Pretend the algorithm is operating in the role of a matchmaker. It operates in the series of *rounds*. In the initial round, the matchmaker tells all the men to go out and propose marriage to their first-choice women. Each man then proposes to the woman at the top of his list.

Each of the women then receives either no proposal (if she was not the first choice of any man) or one or more proposals (if she was the first choice of one or more men). The algorithm instructs the women to respond to the proposals according to the following rules. If no one proposed to you, be patient. (We will prove that eventually someone will.) If one or more men proposed, respond affirmatively to the one you prefer the most, become *engaged* to him, and reject the proposals of the rest. This ends the first round.

After the first round, there are some happy men who are engaged, and some lonely ones who are not. In round two, the matchmaker says to the unengaged men, “Go out and propose again, to your second choice.” While the engaged men do nothing, the unengaged men send out another round of proposals. This time, the matchmaker says to the women: use the same rules as before, with the following change—if you are currently engaged, and receive proposals of marriage from men that you prefer more than your fiancé, you may reject your current fiancé, and reengage yourself to the new man that you prefer the most. Thus a man who is happily engaged at the end of the first round may find himself suddenly unengaged at the end of the second round. (Tough beans, brother.)

As before, some men are engaged and some are not. In the next round, the matchmaker tells each unengaged man to propose to the woman *he prefers most*, among those women to whom he has *not yet proposed*. Again, the matchmaker tells each woman to *accept the best proposition*, either sticking with her current mate or switching to a new suitor according to her preference list. Each time a man proposes, it is with greater desperation, since he is working down his preference list. Each time a woman changes her mate, she becomes happier, because her new mate is someone she prefers more! The rounds continue until there are no more proposals to be made. (We’ll see that everyone is engaged at this point.)

We have presented the code for the algorithm in the following code block. Our presentation is not based on the above rounds-structure, but rather in the form that Kleinberg and Tardos present it, where in each iteration a single proposal is made and decided upon. (The round-based version differs only in that all unengaged men act at once, rather than individually.) An example of this algorithm on the preferences given above is shown in Fig. 1.

```

// Input: 2n preference lists, each consisting of n names.
// Output: A matching that pairs each man with each woman.
Initially all men and all women are unengaged;
while (there is an unengaged man who hasn't yet proposed to every woman) {
  Let m be any such man;
  Let w be the highest woman on his list to whom he has not yet proposed;
  if (w is unengaged) then (m, w) are now engaged;
  else {
    Let m' be the man w is engaged to currently;
    if (w prefers m to m') {
      Break the engagement (m', w);
      Create the new engagement (m, w);
      m' is now unengaged;
    }
  }
}

```

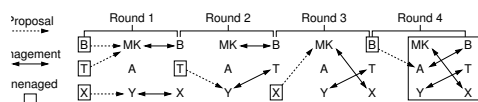


Fig. 1: Example of the round form version of the GS Algorithm on the preference lists given earlier.

**Correctness of the Gale-Shapley Algorithm:** Here are some easy observations regarding the Gale-Shapley (GS) Algorithm. We won't bother to prove them, since they are essentially one-line proofs. But check that you believe them.

**Lemma 1:** Once a woman becomes engaged, she remains engaged for the remainder of the algorithm, and her fiancé can only get better over time in terms of her preference list.

**Lemma 2:** The fiancées assigned to each man decrease over time in terms of his preference list.

Next we show that the algorithm terminates.

**Lemma 3:** The GS Algorithm terminates after at most  $n^2$  iterations of the while loop.

**Proof:** Consider the pairs  $(m, w)$  in which man  $m$  has not yet proposed to woman  $w$ . Initially there are  $n^2$  such pairs, but with each iteration of the while loop, at least one man proposes to one woman. Once a man proposes to a woman, he will never propose to her again. Thus, after  $n^2$  iterations, no one is left to propose.

The above lemma does not imply that the algorithm succeeds in finding a pairing between all the pairs (stable or not), and so we prove this next. By the way, such a 1-to-1 pairing is called a *perfect matching*.

**Lemma 4:** On termination of the GS algorithm, the set of engagements form a perfect matching.

**Proof:** Every time we create a new engagement we break an old one. Thus, each woman is engaged to exactly one man, and vice versa. The only thing that could go wrong is that, at the end of the algorithm, some man, call him Mr. Lonelyheart, is unengaged after exhausting his list. Since there is a 1-to-1 correspondence between engaged men and engaged women, this would imply that some woman, call her Ms. Desperate, is also unengaged. From Lemma 1 we know that once a woman is asked, she will become engaged and will remain engaged henceforth (although possibly to different mates). This implies that Ms. Desperate has never been asked. But she appears on Mr. Lonelyheart's list, and therefore she must have been asked, a contradiction.

Finally, we show that the resulting perfect matching is indeed stable. This establishes the correctness of the GS algorithm formally.

**Lemma 5:** The matching output by the GS algorithm is a stable matching.

**Proof:** Suppose to the contrary that there is some instability in the final output. This means that there are two pairs output  $(m, w)$  and  $(m', w')$  where

- $m$  prefers  $w'$  to his assigned mate  $w$ , and
- $w'$  prefers  $m$  to her assigned mate  $m'$ ,

(and hence  $m$  and  $w'$  would be inclined to elope).

Let's see why this cannot happen. Observe that since  $m$  prefers  $w'$  he proposed to his dreamboat  $w'$  before that plain-jane  $w$ . What went wrong with his plans? Either the lovely  $w'$  was already engaged to some dreamy hunk and rejected the offer outright or she took his offer initially but later opted for someone who she liked better and broke the engagement with  $m$  off. (Recall from Lemma 1 that once engaged, a woman's fiancé only improves over time.) In either case, the lovely  $w'$  winds up with a veritable Greek god of a man—someone she prefers more than  $m$ , and definitely someone she prefers more than the dirty low-life scum  $m'$  who she ranked even lower than  $m$ . Thus, the pair  $(m', w')$  could never have been generated by the algorithm, a contradiction.

**Algorithm Efficiency:** Now that we have established the correctness of the GS Algorithm formally, we turn to the question of its execution time. Let us consider generally how can we evaluate the efficiency of any algorithm. The fundamental measure of efficiency that any user would care about is how fast does it run on my favorite input on my own machine. Unfortunately, this is far too specific to be significant value. As mentioned in the previous class, a natural way to discuss the running time would be to consider the number of steps taken on some ideal computer. This necessitates ignoring constant factors, since presumably different implementations of essentially the same algorithm would involve slightly different (constant factor) differences. Furthermore, faster machines would execute this program at different speeds, but the differences could be related by a constant scaling factor. The running time will depend on some measure of the problem's size and complexity. A most natural way of measuring the complexity of a given problem instance is the size of its input. (Depending on the particular problem, there may be other factors that are involved. For example, some problems, even of the same size, are naturally well-conditioned and easier to solve than ill-conditioned problems. But the notion of what is well-conditioned and ill-conditioned is very much problem specific, so we will not consider this.) For the stable marriage problem, a natural parameter for describing the problem instance size is  $n$ , the number of men and women involved.

There are many different inputs of size  $n$ . On some the algorithm may be much faster than others. As mentioned in the previous lecture, it is common (but not necessarily best) to consider *worst-case running time*, that is, the maximum running time over all inputs of size  $n$ .

Next we could ask, "Efficient relative to what?" We saw already that the GS algorithm terminates after  $n^2$  iterations. Is this a reasonable number? As some measure of how efficient an algorithm is, we might ask about its performance relative to a dumb brute-force search. In the case of the stable marriage problem, we could simply enumerate all of the possible pairings of men to women, and then for each check that it is a valid and stable matching. Since for each of the  $n$  men there are  $n$  choices for a mate, this would suggest that such a naive algorithm would have a running time that grows roughly as fast as  $n^n$ , which would be unimaginably huge for anything other than the smallest values of  $n$ .

A commonly agreed upon (albeit rather weak) notion of efficiency is that the worst-case running time of the algorithm, expressed as a function of  $n$ , should be bounded above by some polynomial function of  $n$ , that is, as a function of the form  $cn^d$ , where  $c$  and  $d$  are positive constants that do not depend on  $n$ . Such an algorithm is called a *polynomial-time algorithm*. This is in contrast to running times like  $2^n$ ,  $n!$ , or  $n^n$ , which arise for many brute-force algorithms, and are certainly not polynomial time. Observe that some common running time functions, such as  $n \log n$  are not polynomials, they are bounded above by polynomial, and so satisfy this definition.

This brings us to the point of asymptotics and asymptotic analysis of algorithms. We will continue with this in the next lecture.

## Algorithm Design Review: Mathematical Background

**Read:** Review Chaps. 1–2 in KT or Chaps. 1-9 in CLRS. Some of the material on recurrences is discussed in Chapt. 5.

**Algorithm Analysis:** In the next couple of lectures we will review some of the basic elements of algorithm analysis, which were covered in previous courses. These include basic algorithm design, proofs of correctness, analysis of running time, and mathematical basics, such as asymptotics, summations, and recurrences.

**Asymptotics:** Asymptotics involves O-notation (“big-Oh”) and its many relatives,  $\Omega$ ,  $\Theta$ ,  $o$  (“little-Oh”),  $\omega$  (“little-omega”). Asymptotic notation provides us with a way to simplify the messy functions that often arise in analyzing algorithm running times by ignoring constant factors and concentrating on the trends for large values of  $n$ . For example, suppose we had three choices of algorithms for solving a given problem, with the following three rather messy looking running times:

$$\begin{aligned}f_1(n) &= 43n^2 \log^4 n + n^3 \log n + 52n \log n \\f_2(n) &= 15n^2 + 7n \log^3 n \\f_3(n) &= 3n + 4 \log_5 n + 19n^2.\end{aligned}$$

By reducing them to their simplest asymptotic forms

$$f_1(n) \in \Theta(n^3 \log n) \quad f_2(n) \in \Theta(n^2) \quad f_3(n) \in \Theta(n^2),$$

we can see that the first function has a significantly larger growth rate than the other two, which have the same growth rates.

Since asymptotics were covered in earlier courses, I will assume that you are familiar with the definitions. For example, perhaps the best known is the definition of big-Oh. Typically we have two functions,  $T(n)$  denotes the running time of some algorithm in question, and  $f(n)$  represents the simpler asymptotic form that we want to use in representing the algorithm’s growth rate.

**Definition:** Given two nonnegative functions  $T(n)$  and  $f(n)$ :

- $T(n) \in O(f(n))$  if there exists constants  $c > 0$  and  $n_0 \geq 0$  such that,  $T(n) \leq c \cdot f(n)$ , for all  $n \geq n_0$ .
- $T(n) \in \Omega(f(n))$  if there exists constants  $c > 0$  and  $n_0 \geq 0$  such that,  $T(n) \geq c \cdot f(n)$ , for all  $n \geq n_0$ .
- $T(n) \in \Theta(f(n))$  if  $T(n) \in (O(f(n)) \cap \Omega(f(n)))$ .

We refer you to a standard reference, such as CLRS, for definitions of the other asymptotic forms. Note that, in contrast to most source, I use the notation  $T(n) \in O(f(n))$ , rather than the more common  $T(n) = O(f(n))$ . The reason is that  $O(f(n))$  is not really a function. Rather, it is more accurate to view it as a class of functions.

If you are more comfortable with manipulating limits (and L’Hôpital’s rule), there is a limit version of the above definitions which may be easier to work with. Here is an example.

**Lemma:** Let  $T(n)$  and  $f(n)$  be two nonnegative functions such that

$$\lim_{n \rightarrow \infty} \frac{T(n)}{f(n)}$$

exists and is equal to some constant  $c > 0$ , then  $T(n) \in \Theta(f(n))$ .

Here are a few facts to remember about asymptotic notation:

**Ignore constant factors:** Multiplicative constant factors are ignored. For example,  $347n$  is  $\Theta(n)$ . Constant factors appearing exponents cannot be ignored. For example,  $2^{3n}$  is *not*  $O(2^n)$ .

**Focus on large  $n$ :** Asymptotic analysis means that we consider trends for large values of  $n$ . Thus, the fastest growing function of  $n$  is the only one that needs to be considered. For example,  $3n^2 \log n + 25n \log n + (\log n)^7$  is  $\Theta(n^2 \log n)$ .

**Polylog, polynomial, and exponential:** These are the most common functions that arise in analyzing algorithms:

**Polylogarithmic:** Powers of  $\log n$ , such as  $(\log n)^7$ . We will usually write this as  $\log^7 n$ .

**Polynomial:** Powers of  $n$ , such as  $n^4$  and  $\sqrt{n} = n^{1/2}$ .

**Exponential:** A constant (not 1) raised to the power  $n$ , such as  $3^n$ .

An important fact is that polylogarithmic functions are strictly asymptotically smaller than polynomial function, which are strictly asymptotically smaller than exponential functions (assuming the base of the exponent is bigger than 1). For example, if we let  $\prec$  mean “asymptotically strictly smaller” then we have

$$\log^a n \prec n^b \prec c^n$$

for any  $a, b$ , and  $c$ , provided that  $b > 0$  and  $c > 1$ .

**Logarithm Simplification:** It is a good idea to first simplify terms involving logarithms. For example, the following formulas are useful. Here  $a, b, c$  are constants:

$$\begin{aligned}\log_b n &= \frac{\log_a n}{\log_a b} = \Theta(\log_a n) \\ \log_a(n^c) &= c \log_a n = \Theta(\log_a n) \\ b^{\log_a n} &= n^{\log_a b}.\end{aligned}$$

Avoid using  $\log n$  in exponents. The last rule above can be used to achieve this. For example, rather than saying  $3^{\log_2 n}$ , express this as  $n^{\log_2 3} \approx n^{1.585}$ .

Following the conventional sloppiness, I will often say  $O(n^2)$ , when in fact the stronger statement  $\Theta(n^2)$  holds. (This is just because it is easier to say “oh” than “theta”.)

**Summations:** Summations naturally arise in the analysis of iterative algorithms. Also, more complex forms of analysis, such as recurrences, are often solved by reducing them to summations. Solving a summation means reducing it to a *closed-form formula*, that is, one having no summations, recurrences, integrals, or other complex operators. In algorithm design it is often not necessary to solve a summation exactly, since an asymptotic approximation or close upper bound is usually good enough. Here are some common summations and some tips to use in solving summations.

**Constant Series:** For integers  $a$  and  $b$ ,

$$\sum_{i=a}^b 1 = \max(b - a + 1, 0).$$

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

**Arithmetic Series:** For  $n \geq 0$ ,

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

This is  $\Theta(n^2)$ . (The starting bound could have just as easily been set to 1 as 0.)

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

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

If  $0 < c < 1$  then this is  $\Theta(1)$ , no matter how large  $n$  is. If  $c > 1$ , then this is  $\Theta(c^n)$ , that is, the entire sum is proportional to the last element of the series.

**Quadratic Series:** For  $n \geq 0$ ,

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

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

$$\sum_{i=0}^{n-1} ic^i = c + 2c^2 + 3c^3 \dots + nc^n = \frac{(n-1)c^{(n+1)} - nc^n + c}{(c-1)^2}.$$

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

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

$$H_n = \sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = (\ln n) + O(1).$$

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

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

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

**Linearity of Summation:** Constant factors and added terms can be split out to make summations simpler.

$$\sum (4 + 3i(i-2)) = \sum 4 + 3i^2 - 6i = \sum 4 + 3 \sum i^2 - 6 \sum i.$$

Now the formulas can be to each summation individually.

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

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

**Example: Previous Larger Element** As an example of the use of summations in algorithm analysis, consider the following simple problem. We are given a sequence of numeric values,  $\langle a_1, a_2, \dots, a_n \rangle$ . For each element  $a_i$ , for  $1 \leq i \leq n$ , we want to know the index of the rightmost element of the sequence  $\langle a_1, a_2, \dots, a_{i-1} \rangle$  whose

value is strictly larger than  $a_i$ . If no element of this subsequence is larger than  $a_i$  then, by convention, the index will be 0. (Or, if you like, you may imagine that there is a fictitious sentinel value  $a_0 = \infty$ .)

More formally, for  $1 \leq i \leq n$ , define  $p_i$  to be

$$p_i = \max\{j \mid 0 \leq j < i \text{ and } a_j > a_i\},$$

where  $a_0 = \infty$ . A more visual way to understand this problem is to imagine that  $a_i$  is the height of the  $i$ th telephone pole in a sequence, and if we shoot a bullet horizontally to the left from the top of the  $i$ th pole, we want to know which pole we will hit first. (See Fig. 2 below.)

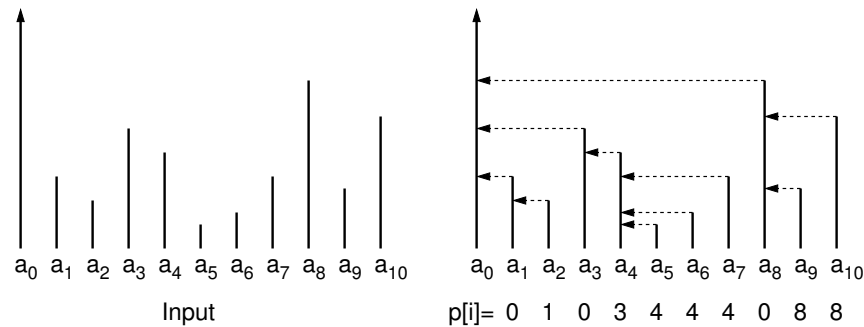


Fig. 2: Example of the previous larger element problem.

There is an  $O(n)$  time solution to this problem. (You should think about it.) However, here, I will describe a much less efficient  $O(n^2)$  time algorithm.

---

Previous Larger Element (Naive Solution)

```
// Input: An array of numeric values a[1..n]
// Returns: An array p[1..n] where p[i] contains the index of the previous
// larger element to a[i], or 0 if no such element exists.
previousLarger(a[1..n]) {
    for (i = 1 to n)
        j = i-1;
        while (j > 0 and a[j] <= a[i]) j--;
        p[i] = j;
    }
    return p
}
```

---

The correctness of this algorithm is almost trivial, but (for the sake of completeness) let us make a couple of observations. The inner while loop has two ways of terminating, one if  $a[j] > a[i]$ , in which case we have found a large element, and the other if  $j = 0$ , implying that no larger element was found.

The time spent in this algorithm is dominated by the time spent in the inner ( $j$ ) loop. On the  $i$ th iteration of the outer loop, the inner loop is executed from  $i - 1$  down to either 0 or the first index whose associated value exceeds  $a[i]$ . In the worst case, this loop will always go all the way to 0. (Can you see what sort of input would give rise to this case?) Thus the total running time (up to constant factors) can be expressed as:

$$\begin{aligned} T(n) &= \sum_{i=1}^n \sum_{j=0}^{i-1} 1 \\ &= 1 + 2 + \dots + (n - 2) + (n - 1) \\ &= \sum_{i=1}^n i. \end{aligned}$$

We can solve this summation directly by applying the above formula for the arithmetic series, which yields

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

An interesting question to consider at this point is, what would the average-case running time be if the elements of the array were random values. Note that if  $i$  is large, it seems that it would be quite unlikely to go through all  $i$  iterations of the inner while loop, because the chances of coming across a larger element would seem pretty high. But how many iterations would you expect on average? A constant number?  $O(\log i)$ ?  $O(\sqrt{i})$ .  $O(i/2)$ ? This is a topic for probabilistic analysis of algorithms, which we may revisit later.

As mentioned above, there is a simple  $O(n)$  time algorithm for this problem. As an exercise, see if you can find it.

**Recurrences:** Another useful mathematical tool in algorithm analysis will be recurrences. They arise naturally in the analysis of divide-and-conquer algorithms. Recall that these algorithms have the following general structure.

**Divide:** Divide the problem into two or more subproblems (ideally of roughly equal sizes),

**Conquer:** Solve each subproblem recursively, and

**Combine:** Combine the solutions to the subproblems into a single global solution.

How do we analyze recursive procedures like this one? If there is a simple pattern to the sizes of the recursive calls, then the best way is usually by setting up a *recurrence*, that is, a function which is defined recursively in terms of itself. Here is a typical example. Suppose that we break the problem into two subproblems, each of size roughly  $n/2$ . (We will assume exactly  $n/2$  for simplicity.). The additional overhead of splitting and merging the solutions is  $O(n)$ . When the subproblems are reduced to size 1, we can solve them in  $O(1)$  time. We will ignore constant factors, writing  $O(n)$  just as  $n$ , yielding the following recurrence:

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

Note that, since we assume that  $n$  is an integer, this recurrence is not well defined unless  $n$  is a power of 2 (since otherwise  $n/2$  will at some point be a fraction). To be formally correct, I should either write  $\lfloor n/2 \rfloor$  or restrict the domain of  $n$ , but I will often be sloppy in this way.

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

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

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

defined for  $n \geq 0$ .

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

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

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

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

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

## Basic Graph Algorithms: BFS and DFS

**Read:** Chapt. 3 in KT. Our notation is based on the presentation given in CLRS.

**Graph Algorithms:** A graph  $G = (V, E)$  is a structure that represents a discrete set  $V$  objects, called *nodes* or *vertices*, and a set of pairwise relations  $E$  between these objects, called *edges*. Edges may be *directed* from one node to another or may be *undirected*. The term *digraph* is sometimes used to indicate the directed edges are present. Fundamental to graphs are notions such as *paths*, which are formed by a sequence of edges joined end to end, *cycles*, which are closed paths, and *connectivity*, which refers to a number of graph properties having to do with the nature reachability through paths. Following CLRS, we will usually abuse notation and use  $V$  and  $E$  to denote both the set of nodes and edges as well as the numbers of nodes and edges, respectively. In KT, the convention is to use  $n$  for the number of nodes and  $m$  for the number of edges.

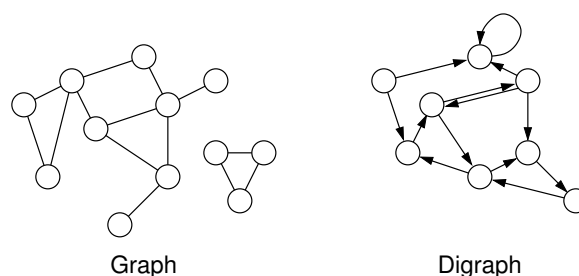


Fig. 3: Example of a graph and digraph.

Graphs are important computational structures because they can be used in such a wide variety of applications, including transportation networks (e.g., road or rail systems), communication networks, information networks (like the Web), and social networks (for modeling family, friend, or organizational structures), and dependency networks (such as precedence relationships in scheduling).

There are a number special classes of graphs that arise in various applications. Examples include *undirected trees*, which are connected, acyclic (undirected) graphs, and *bipartite graphs*, in which the nodes are partitioned into two classes (e.g., men and women) and edges connect nodes of one class to the other. An important special case of digraphs are *DAGs*, which are directed acyclic graphs. They arise often in dependency networks.

We refer the reader to standard algorithm references for further definitions and discussion of graph representations, including *adjacency matrices* and *adjacency lists*. Unless otherwise specified, we will assume that graphs are represented using the adjacency list structure, because it achieves optimal space of  $O(V + E)$ .

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

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

In order to implement BFS we need some way to ascertain which vertices have been visited and which haven't. Initially all vertices (except the source) are colored white, meaning that they are *undiscovered*. When a vertex

has first been *discovered*, it is colored gray (and is part of the frontier). When a gray vertex is *processed*, then it becomes black.

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

---

Breadth-First Search

```

BFS(G, s) {
  for each (u in V) {                                // initialization
    color[u] = white
    d[u]      = infinity
    pred[u]  = null
  }
  color[s] = gray                                  // initialize source s
  d[s] = 0
  Q = {s}                                          // put s in the queue
  while (Q is nonempty) {
    u = dequeue from head of Q                    // u is the next to visit
    for each (v in Adj[u]) {
      if (color[v] == white) {                   // if neighbor v undiscovered
        color[v] = gray                          // ...mark it discovered
        d[v]      = d[u]+1                        // ...set its distance
        pred[v]  = u                             // ...and its predecessor
        insert v at tail of Q                    // ...put it in the queue
      }
    }
    color[u] = black                              // we are done with u
  }
}

```

---

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

It is not hard to prove that if  $G$  is an undirected graph, then cross edges always go between two nodes that are at most one level apart in the BFS tree. (Can you see why this must be true?) Below is a sketch of a proof that on termination,  $d[v]$  is equal to the distance from  $s$  to  $v$ . (See the CLRS for a detailed proof.)

**Theorem:** Let  $\delta(s, v)$  denote the length (number of edges) on the shortest path from  $s$  to  $v$ . Then, on termination of the BFS procedure,  $d[v] = \delta(s, v)$ .

**Proof:** (Sketch) The proof is by induction on the length of the shortest path. Let  $u$  be the predecessor of  $v$  on some shortest path from  $s$  to  $v$ , and among all such vertices the first to be processed by the BFS. Thus,  $\delta(s, v) = \delta(s, u) + 1$ . When  $u$  is processed, we have (by induction)  $d[u] = \delta(s, u)$ . Since  $v$  is a neighbor of  $u$ , we set  $d[v] = d[u] + 1$ . Thus we have

$$d[v] = d[u] + 1 = \delta(s, u) + 1 = \delta(s, v),$$

as desired.

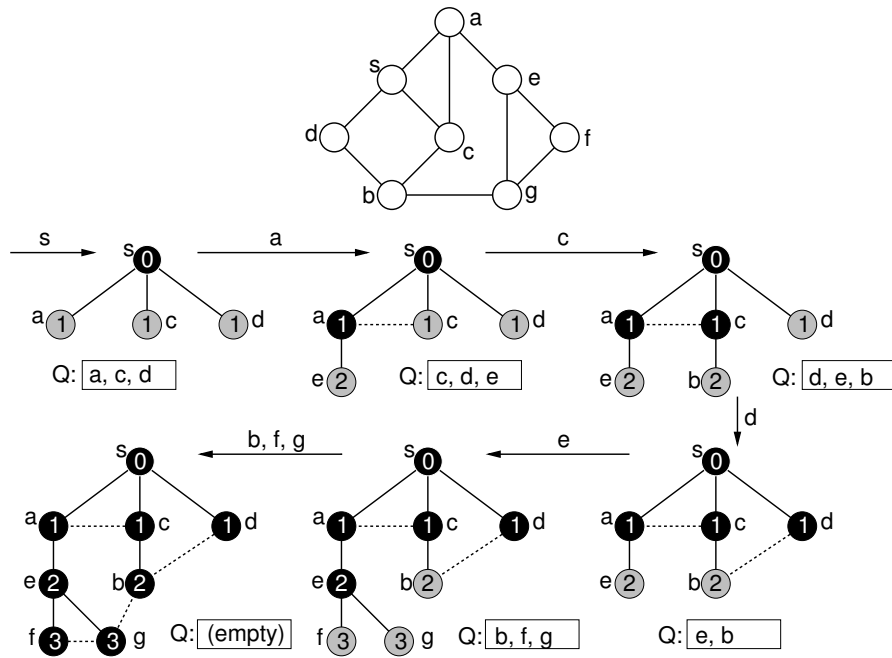


Fig. 4: Breadth-first search example.

**Analysis:** The running time analysis of BFS is similar to the running time analysis of many graph traversal algorithms. Recall that (through some abuse of notation) we let  $V = |V|$  and  $E = |E|$ . Observe that the initialization portion requires  $\Theta(V)$  time. The real meat is in the traversal loop. Since we never visit a vertex twice, the number of times we go through the while loop is at most  $V$  (exactly  $V$  assuming each vertex is reachable from the source). The number of iterations through the inner for loop is proportional to  $\deg(u) + 1$ . (The  $+1$  is because even if  $\deg(u) = 0$ , we need to spend a constant amount of time to set up the loop.) Summing up over all vertices we have the running time

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

The analysis is essentially the same for directed graphs.

**Depth-First Search:** The next traversal algorithm that we will study is called *depth-first search*. As the name suggests, in contrast to BFS where we strive for maximal breadth in our search, here the approach is to plunge as far into the graph as possible and backtracking when there is nothing new to explore.

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

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

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

We use four auxiliary arrays. As before, we maintain a color for each vertex: white means *undiscovered*, gray means *discovered* but not finished processing, and black means *finished*. As before, we also store predecessor

pointers, pointing back to the vertex that discovered a given vertex. We will also associate two numbers with each vertex. These are *time stamps*. When we first discover a vertex  $u$  store a counter in  $d[u]$  and when we are finished processing a vertex we store a counter in  $f[u]$ . The purpose of the time stamps will be explained later. (Note: Do not confuse the discovery time  $d[v]$  with the distance  $d[v]$  from BFS.) The algorithm is shown in code block below, and illustrated in Fig. 5. As with BFS, DFS induces a tree structure. We will discuss this tree structure further below.

---

Depth-First Search

```

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

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

```

---

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

First observe that if we ignore the time spent in the recursive calls, the main DFS procedure runs in  $O(V)$  time. Observe that each vertex is visited exactly once in the search, and hence the call `DFSVisit()` is made exactly once for each vertex. We can just analyze each one individually and add up their running times. Ignoring the time spent in the recursive calls, we can see that each vertex  $u$  can be processed in  $O(1 + \text{outdeg}(u))$  time. Thus the total time used in the procedure is

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

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

**Tree structure:** DFS naturally imposes a tree structure (actually a collection of trees, or a forest) on the structure of the graph. This is just the recursion tree, where the edge  $(u, v)$  arises when processing vertex  $u$  we call `DFSVisit(v)` for some neighbor  $v$ . For directed graphs the other edges of the graph can be classified as follows:

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

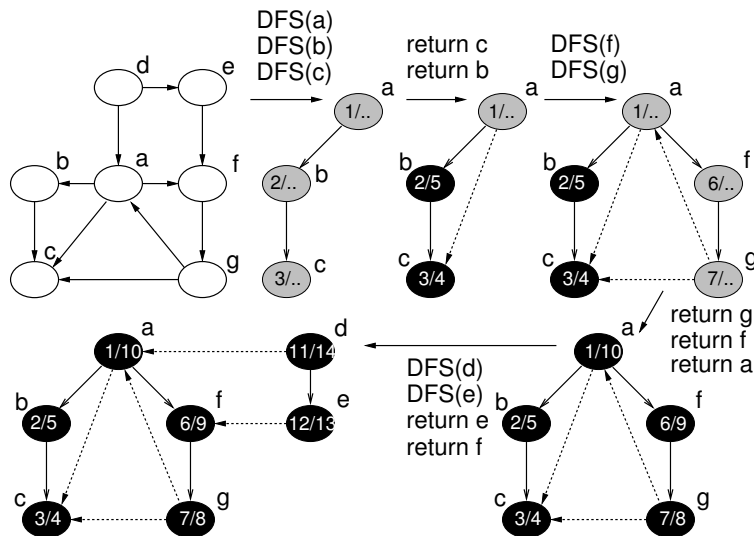


Fig. 5: Depth-First search tree.

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

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

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

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

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

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

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

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

**Lemma:** Given a digraph  $G = (V, E)$ , consider any DFS forest of  $G$ , and consider any edge  $(u, v) \in E$ . If this edge is a tree, forward, or cross edge, then  $f[u] > f[v]$ . If the edge is a back edge then  $f[u] \leq f[v]$ .

**Proof:** For tree, forward, and back edges, the proof follows directly from the parenthesis lemma. (E.g. for a forward edge  $(u, v)$ ,  $v$  is a descendent of  $u$ , and so  $v$ 's start-finish interval is contained within  $u$ 's, implying that  $v$  has an earlier finish time.) For a cross edge  $(u, v)$  we know that the two time intervals are disjoint. When we were processing  $u$ ,  $v$  was not white (otherwise  $(u, v)$  would be a tree edge), implying that  $v$  was started before  $u$ . Because the intervals are disjoint,  $v$  must have also finished before  $u$ .

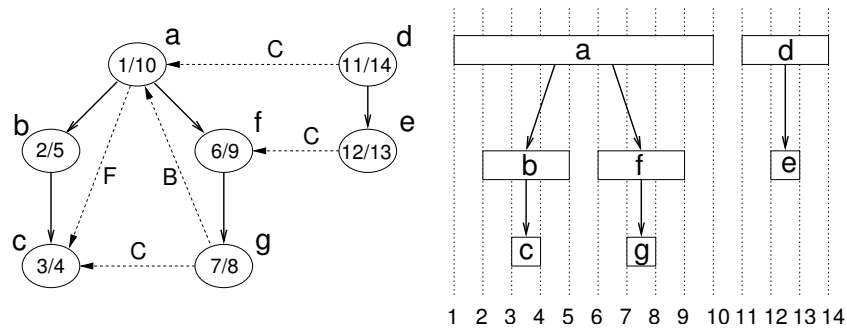


Fig. 6: Parenthesis Lemma.

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

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

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

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

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

## Applications of DFS

**Read:** For topological ordering, see Sect. 3.6 of KT. The algorithm that we will present is based on a presentation in Sects. 22.3–22.5 in CLRS. The material on articulation points is not covered in either book.

**Applications of DFS:** Last time we introduced depth-first search (DFS). Today, we discuss some applications of this powerful and efficient method of graph traversal.

**Directed Acyclic Graphs:** As the name suggests, a *directed acyclic graph*, or *DAG*, is a directed graph that has no cycles. DAGs arise in many applications where there are precedence or ordering constraints. For example, if there are a series of tasks to be performed, and certain tasks must precede other tasks (e.g., in construction you have to build the walls before you install the windows). In general a *precedence constraint graph* is a DAG in which vertices are tasks and the edge  $(u, v)$  means that task  $u$  must be completed before task  $v$  begins.

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

To compute a topological ordering is actually very easy. We will present an algorithm based on DFS. In Kleinberg and Tardos an alternative algorithm is presented. Recall our earlier comments on the nature of finish times. After running any DFS on a graph, if  $(u, v)$  is a tree, forward, or cross edge, then the finish time of  $u$  is greater than the finish time of  $v$ . Since a DAG is acyclic, there can be no back edges. Thus, in order to produce a topological ordering of the vertices it suffices to output the vertices in reverse order of finish times. To do this

---

```

topSort(G) {
  for each (u in V) color[u] = white;    // initialize
  L = an empty linked list;
  for each (u in V)
    if (color[u] == white) topVisit(u);
  return L;                               // the final topol order
}

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

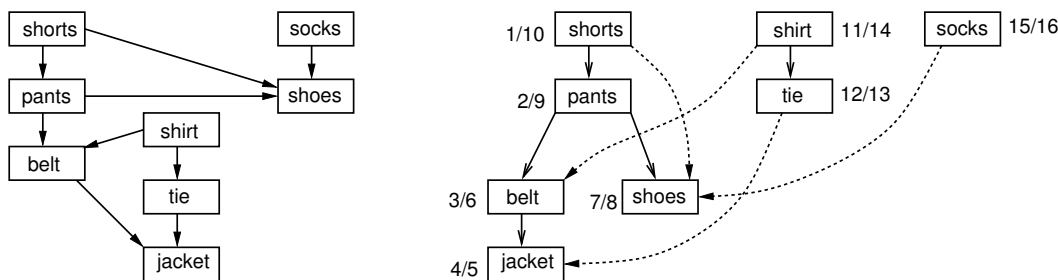
```

---

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

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

As an example we consider the DAG presented in CLRS for Professor Bumstead's order of dressing (see Fig. 7). Bumstead lists the precedences in the order in which he puts on his clothes in the morning. We performed our depth-first search in a different order from the one given in CLRS, and hence we obtained a different final ordering. However both orderings are legitimate, given the precedence constraints. As with depth-first search, the running time of topological sort is  $\Theta(V + E)$ . (Observe that there are many possible topological orderings and many possible DFS searches, depending on the order in which vertices are visited. A question worth pondering is whether every possible topological ordering arises from some DFS search.)



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

Fig. 7: Topological ordering example.

**Articulation Points and Biconnected Graphs:** Next, we consider another application of DFS, this time to a problem on undirected graphs. Let  $G = (V, E)$  be an *connected* undirected graph. We begin with the following definitions (see Fig. 8):

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

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

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

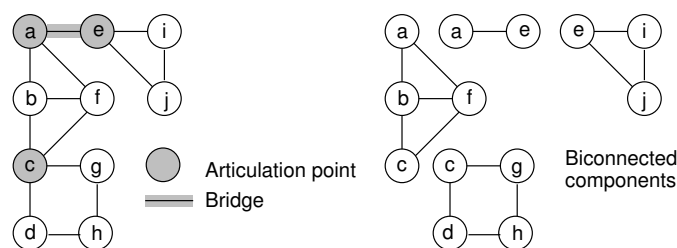


Fig. 8: Articulation Points and Bridges

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

In order to define the notion of biconnected components, we partition the edges of the graph into equivalence classes. We say that two edges  $e_1$  and  $e_2$  are *cocyclic* if either  $e_1 = e_2$  or if there is a simple cycle that contains both edges. It is not hard to verify that this defines an equivalence relation on the edges of a graph. Notice that if two edges are cocyclic, then there are essentially two different ways of getting from one edge to the other (by going around the cycle each way). This suggests the following definition (see the right side of Fig. 8).

**Biconnected components:** of a graph are the equivalence classes of the cocyclicity relation.

Observe that a graph is biconnected if and only if it consists of a single biconnected component, that is, if it contains no articulation points. The articulation points are the places where one biconnected component ends and another one begins. You might think for a while why this is so.

We will give an algorithm for computing articulation points. The algorithms for computing bridges and biconnected components are simple modifications of this procedure.

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

We assume that  $G$  is connected (if not, we can apply this algorithm to each individual connected component). So we may assume that there is only one tree in the DFS forest. Because  $G$  is undirected, the DFS tree has a simpler structure than for digraphs. First off, we cannot distinguish between forward edges and back edges, and we just call them back edges. Also, there are no cross edges. (You should take a moment to convince yourself why this is true.)

For now, let us consider the typical case of a vertex  $u$ , where  $u$  is not a leaf, and  $u$  is not the root. Let’s let  $v_1, v_2, \dots, v_k$  be the children of  $u$ . For each child there is a subtree of the DFS tree rooted at this child. If for some child, there is no back edge going to a proper ancestor of  $u$ , then if we were to remove  $u$ , this subtree would become disconnected from the rest of the graph, and hence  $u$  is an articulation point. On the other hand, if every one of the subtrees rooted at the children of  $u$  have back edges to proper ancestors of  $u$ , then if  $u$  is removed, the graph remains connected (the backedges hold everything together). This leads to the following (see Fig. 9).

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

Please check this condition carefully to see that you understand it. In particular, notice that the condition for whether  $u$  is an articulation point depends on a test applied to its children. This is the most common source of confusion for this algorithm.

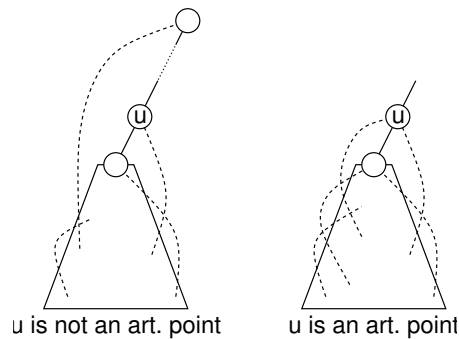


Fig. 9: Observation 1.

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

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

What about the root? Clearly we cannot apply condition 1, since the root has no ancestor. Instead observe that, because there are no cross edges between the subtrees of the root, if the root has two or more children then it is an articulation point (since its removal separates these two subtrees). On the other hand, if the root has only a single child, then (as in the case of leaves) its removal does not disconnect the DFS tree, and hence cannot disconnect the graph in general.

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

These conditions are illustrated in Fig. 10 on the left. The shaded nodes satisfy the conditions for being an articulation points and the others do not. This is an excellent example of the power of DFS. The DFS tree structure has provided us with a way to classify the various conditions.

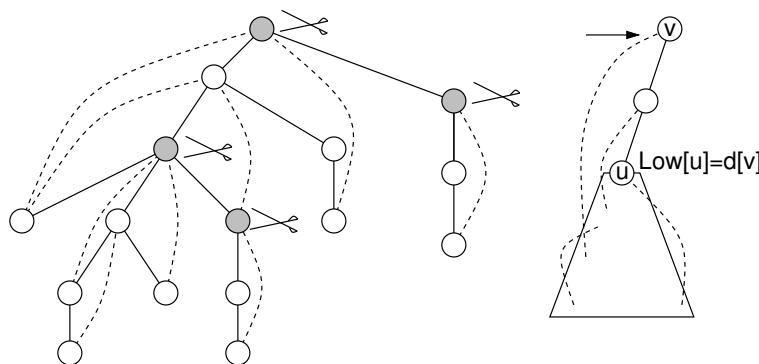


Fig. 10: Articulation Points and the definition of  $Low[u]$ .

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

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

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

**Low:** Define  $\text{Low}[u]$  to be the minimum of  $d[u]$  and

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

The term “descendent” is used in the nonstrict sense, that is,  $v$  may be equal to  $u$  (see Fig. 10 right).

Intuitively,  $\text{Low}[u]$  is the closest to the root that you can get in the tree by taking any one backedge from either  $u$  or any of its descendants. (Beware of this notation: “Low” means low discovery time, not “low” in our drawing of the DFS tree. In fact  $\text{Low}[u]$  tends to be “high” in the tree, in the sense of being close to the root.) Also note that you may consider *any* descendent of  $u$ , but you may only follow *one* back edge.

To compute  $\text{Low}[u]$  we use the following simple rules: Suppose that we are performing DFS on the vertex  $u$ .

**Initialization:**  $\text{Low}[u] = d[u]$ .

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

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

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

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

The complete algorithm for computing articulation points is given below. The main procedure for DFS is the same as before, except that it calls the following routine rather than `DFSvisit()`.

An example is shown in the following figure. As with all DFS-based algorithms, the running time is  $\Theta(V + E)$ . There are some interesting problems that we still have not discussed. We did not discuss how to compute the bridges of a graph. This can be done by a small modification of the algorithm above. We’ll leave it as an exercise. (Notice that if  $\{u, v\}$  is a bridge then it does not follow that  $u$  and  $v$  are both articulation points.) Another question is how to determine which edges are in the biconnected components. A hint here is to store the edges in a stack as you go through the DFS search. When you come to an articulation point, you can show that all the edges in the biconnected component will be consecutive in the stack.

```

findArticPts(u) {
  color[u] = gray;
  Low[u] = d[u] = ++time;           // set discovery time and init Low
  for each (v in Adj(u)) {
    if (color[v] == white) {       // (u,v) is a tree edge
      pred[v] = u;                 // v's parent is u
      findArticPts(v);
      Low[u] = min(Low[u], Low[v]); // update Low[u]
      if (pred[u] == NULL) {       // root: apply Observation 3
        if (this is u's second child)
          Add u to set of articulation points;
      }
      else if (Low[v] >= d[u]) {   // internal: apply Observation 1
        Add u to set of articulation points;
      }
    }
    else if (v != pred[u]) {       // (u,v) is a back edge
      Low[u] = min(Low[u], d[v]);  // update Low[u]
    }
  }
}

```

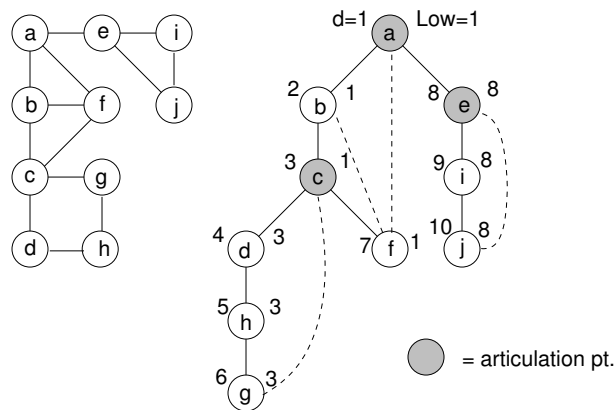


Fig. 11: Articulation Points.

## Greedy Algorithms: Interval Scheduling and Fractional Knapsack

**Read:** Sects. 16.1 and 16.2 in CLRS or Sect. 4.1 of KT.

**Greedy Algorithms:** In many optimization algorithms a series of selections need to be made. Today we will consider a simple design technique for optimization problems, called *greedy algorithms*. Intuitively, a greedy algorithm is one that builds up an optimal solution for some problem by “myopically” selecting the best alternative with each step. When applicable, this method typically leads to very simple and efficient algorithms. However, it is not a very powerful technique, in the sense that the solutions that arise from greedy algorithms are not always optimal. (We will discuss other more powerful techniques, such as dynamic programming, elsewhere.) Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (nonoptimal solution strategies), are often used in finding good approximations. Thus, this is a nice algorithm design technique to be aware of.

Over the next few lectures, will give some examples of problems that can be solved by greedy algorithms. Here we present two examples, interval scheduling and the fractional knapsack problem.

**Interval Scheduling:** Scheduling problems are among the most fundamental optimization problems. Interval scheduling is one of the simplest formulations. We are given a set  $S = \{1, 2, \dots, n\}$  of  $n$  *activity requests* that are to be scheduled to use some resource, where each activity must be started at a given start time  $s_i$  and ends at a given finish time  $f_i$ . For example, these might be lectures that are to be given in a lecture hall, where the lecture times have been set up in advance, or requests for boats to use a repair facility while they are in port.

Because there is only one resource, and some start and finish times may overlap (and two lectures cannot be given in the same room at the same time), not all the requests can be honored. We say that two activities  $i$  and  $j$  are *noninterfering* if their start-finish intervals do not overlap, more formally,  $[s_i, f_i) \cap [s_j, f_j) = \emptyset$ . (Note that by defining the intervals *half open*, two consecutive activities are not considered to interfere. For example, the intervals  $[1, 13)$  and  $[13, 25)$  do not overlap.) *Interval scheduling* is the problem of selecting a maximum-size set of mutually noninterfering activities for use of the resource. Notice that goal here is maximum *number* of activities, not maximum utilization. Thus, we would much rather schedule many short activities rather than one long one. Of course different criteria could be considered, but the greedy approach may not be optimal in general.

How do we schedule the largest number of activities on the resource? There are a number ideas on how to proceed. As we shall see, there are a number of seemingly reasonable approaches that do not work.

**Earliest Activity First:** Let us repeatedly schedule the activity with the earliest start time, provided that it does not overlap any of the previously scheduled activities.

Although this will produce a valid schedule, it is not too hard to see that this will not be optimal in general. A single very long activity with an early start time might consume the entire schedule.

**Shortest Activity First:** The previous counterexample suggests that we should prefer short activities over long ones. This suggests the following greedy strategy. Repeatedly select the activity with the smallest duration ( $f_i - s_i$ ) and schedule it, provided that it does not interfere with any previously scheduled activities. Although this may seem like a reasonable strategy, this also turns out to be nonoptimal. (Think about this and see if you can discover a counterexample.)

**Lowest Conflict Activity First:** Counterexamples to the previous strategy arise because there may be activities of short duration, but that overlap lots of other activities. Intuitively, we to avoid overlaps, because they limit our ability to schedule future tasks. So, let us count for each activity the number of activities it overlaps. Then, we schedule the activity that overlaps the smallest number of other activities. Then eliminate it and all overlapping tasks, and update the overlap counts. Repeat until no more tasks remain.

Although at first glance, this seems to address the shortcomings of the previous methods, it too is not optimal. (See Fig. 4.1(c) in KT for a counterexample.)

If at first you don't succeed, keep trying. Here is a greedy strategy that does work. The intuition is the same. Since we do not like activities that take a long time, let us select the activity that finishes first and schedule it. Then, we skip all activities that interfere with this one, and schedule the next one that has the earliest finish time, and so on. To make the selection process faster, we assume that the activities have been sorted by their finish times, that is,

$$f_1 \leq f_2 \leq \dots \leq f_n,$$

Assuming this sorting, the pseudocode for the rest of the algorithm is presented below. The output is the list  $A$  of scheduled activities. The variable  $prev$  holds the index of the most recently scheduled activity at any time, in order to determine interferences.

---

Greedy Interval Scheduling

```

schedule(R) {
    // R holds the set of all activity requests
    S = empty; // S holds the schedule
    while (R is nonempty) {
        r = the request of R having the smallest finish time;
        Append r to the end of S;
        Delete from R all requests that overlap r;
    }
    return S;
}

```

---

The easiest way to implement the above algorithm is to first sort the activities in increasing order of finishing time. This takes  $O(n \log n)$  time. Selecting the request with the smallest finish time can be done in  $O(1)$  time by taking the next available entry from the sorted list. The process of deleting overlapping requests can be handled by simply running through the subsequent requests until finding the first whose start time exceeds the finish time of the previous request.

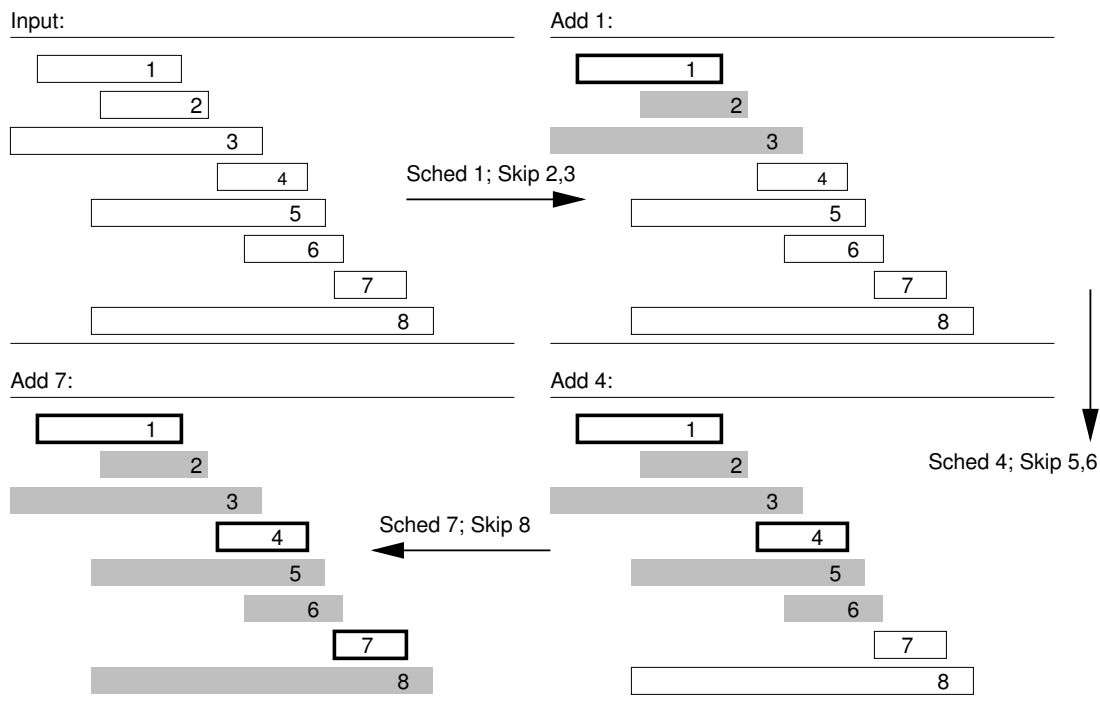


Fig. 12: An example of the greedy algorithm for interval scheduling. The final schedule is  $\{1, 4, 7\}$ .

Fig. 12 shows an example. Each activity is represented by its start-finish time interval. Observe that the intervals are sorted by finish time. Activity 1 is scheduled first. It interferes with Activities 2 and 3. Then Activity 4 is scheduled. It interferes with Activities 5 and 6. Finally, Activity 7 is scheduled, and it interferes with the remaining activity. The final output is  $\{1, 4, 7\}$ . Note that this is not the only optimal schedule.  $\{2, 4, 7\}$  is also optimal.

**Proof of Optimality:** Our proof of optimality is based on showing that the first choice made by the algorithm is the best possible, and then using induction to show that the rest of the choices result in an optimal schedule. Proofs of optimality for greedy algorithms follow a similar structure. Suppose that you have any nongreedy solution. Show that its cost can be reduced by being “greedier” at some point in the solution. This proof is complicated a bit by the fact that there may be multiple solutions. Our approach is to show that any schedule that is not greedy can be made more greedy, without decreasing the number of activities.

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

**Proof:** Consider any optimal schedule  $A$  that is not the greedy schedule. We will construct a new optimal schedule  $A'$  that is in some sense “greedier” than  $A$ . Order the activities in increasing order of finish time. Let  $A = \langle x_1, x_2, \dots, x_k \rangle$  be the activities of  $A$ . Since  $A$  is not the same as the greedy schedule, consider the first activity  $x_j$  where these two schedules differ. That is, we have:

$$\begin{aligned} A &= \langle x_1, x_2, \dots, x_{j-1}, x_j, \dots \rangle \\ G &= \langle x_1, x_2, \dots, x_{j-1}, g_j, \dots \rangle, \end{aligned}$$

where  $g_j \neq x_j$ . (Note that  $k \geq j$ , since otherwise  $G$  would have more activities than the optimal schedule, which would be a contradiction.) The greedy algorithm selects the activity with the earliest finish time that does not conflict with any earlier activity. Thus, we know that  $g_j$  does not conflict with any earlier activity, and it finishes before  $x_j$ .

Consider the modified “greedier” schedule  $A'$  that results by replacing  $x_j$  with  $g_j$  in the schedule  $A$ . (See Fig. 13.) That is,

$$A' = \langle x_1, x_2, \dots, x_{j-1}, g_j, x_{j+1}, \dots, x_k \rangle.$$

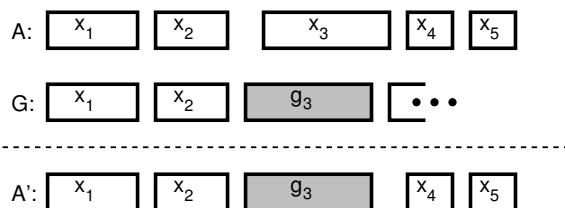


Fig. 13: Proof of optimality for the greedy schedule ( $j = 3$ ).

We assert that this is a feasible schedule. The reason is that  $g_j$  cannot conflict with the earlier activities (since  $G$  is a feasible schedule) and it cannot not conflict with later activities (because, by definition,  $g_j$  finishes before  $x_j$  finishes). Thus, this new schedule “greedier,” and it has the same number of activities as  $A$ . Therefore  $A'$  is also optimal. By repeating this process, we will eventually convert  $A$  into  $G$ , without decreasing the number of activities. Therefore,  $G$  is also optimal.

**Fractional Knapsack Problem:** The classical *0-1 knapsack problem* is a famous optimization problem. A thief is robbing a store, and finds  $n$  items which can be taken. The  $i$ th item is worth  $v_i$  dollars and weighs  $w_i$  pounds, where  $v_i$  and  $w_i$  are integers. He wants to take as valuable a load as possible, but has a knapsack that can only carry  $W$  total pounds. Which items should he take? (The reason that this is called 0-1 knapsack is that each item must be left (0) or taken entirely (1). It is not possible to take a fraction of an item or multiple copies of an

item.) This optimization problem arises in industrial packing applications. For example, you may want to ship some subset of items on a truck of limited capacity.

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

The 0-1 knapsack problem is hard to solve, and in fact it is an NP-complete problem (meaning that there probably doesn't exist an efficient solution). However, there is a very simple and efficient greedy algorithm for the fractional knapsack problem.

As in the case of the other greedy algorithms we have seen, the idea is to find the right order in which to process items. Intuitively, it is good to have high value and bad to have high weight. This suggests that we first sort the items according to some function that is an decreases with value and increases with weight. There are a few choices that you might try here, but only one works. Let  $\rho_i = v_i/w_i$  denote the *value-per-pound ratio*. We sort the items in decreasing order of  $\rho_i$ , and add them in this order. If the item fits, we take it all. At some point there is an item that does not fit in the remaining space. We take as much of this item as possible, thus filling the knapsack entirely. This is illustrated in Fig. 14

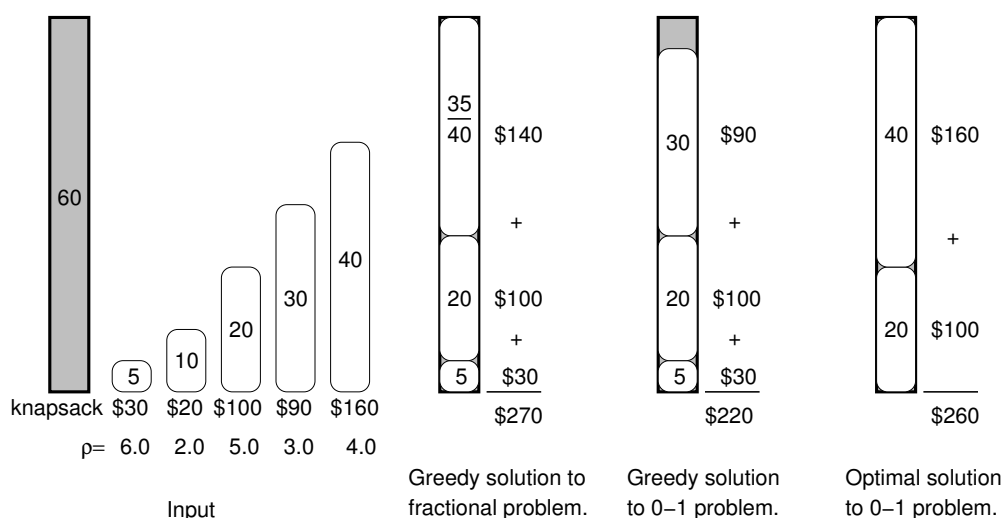


Fig. 14: Example for the fractional knapsack problem.

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

More formally, suppose to the contrary that the greedy algorithm is not optimal. This would mean that there is an alternate selection that is optimal. Sort the items of the alternate selection in decreasing order by  $\rho$  values. Consider the first item  $i$  on which the two selections differ. By definition, greedy takes a greater amount of item  $i$  than the alternate (because the greedy always takes as much as it can). Let us say that greedy takes  $x$  more units of object  $i$  than the alternate does. All the subsequent elements of the alternate selection are of lesser value than  $v_i$ . By replacing  $x$  units of any such items with  $x$  units of item  $i$ , we would increase the overall value of the alternate selection. However, this implies that the alternate selection is not optimal, a contradiction.

**Nonoptimality for the 0-1 Knapsack:** Next we show that the greedy algorithm is not generally optimal in the 0-1 knapsack problem. Consider the example shown in Fig. 14. If you were to sort the items by  $\rho_i$ , then you would first take the items of weight 5, then 20, and then (since the item of weight 40 does not fit) you would settle for

the item of weight 30, for a total value of  $\$30 + \$100 + \$90 = \$220$ . On the other hand, if you had been less greedy, and ignored the item of weight 5, then you could take the items of weights 20 and 40 for a total value of  $\$100 + \$160 = \$260$ . This feature of “delaying gratification” in order to come up with a better overall solution is your indication that the greedy solution is not optimal.

## Greedy Algorithms: Huffman Coding

**Read:** Sect. 4.8 in KT or Sect. 16.3 in CLRS.

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

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

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

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

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

The final 20-character binary string would be “00010010110000100010”.

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

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

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

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

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

probabilities of their occurrences. The expected encoded string length is just  $n$  times the expected encoded character length.

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

Thus, this would represent a 25% savings in expected encoding length. The question that we will consider today is how to form the best code, assuming that the probabilities of character occurrences are known.

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

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

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

Observe that any binary prefix coding can be described by a binary tree in which the codewords are the leaves of the tree, and where a left branch means "0" and a right branch means "1". The length of a codeword is just its depth in the tree. The code given earlier is a prefix code, and its corresponding tree is shown in Fig. 15.

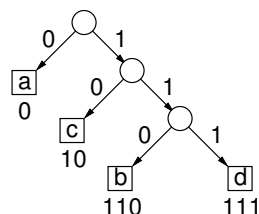


Fig. 15: Prefix codes.

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

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

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

This suggests the following problem:

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

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

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

Another way to think of this, is that we merge  $x$  and  $y$  as the left and right children of a root node called  $z$ . Then the subtree for  $z$  replaces  $x$  and  $y$  in the list of characters. We repeat this process until only one super-character remains. The resulting tree is the final prefix tree. Since  $x$  and  $y$  will appear at the bottom of the tree, it seems most logical to select the two characters with the smallest probabilities to perform the operation on. The result is Huffman's algorithm. It is illustrated in Fig. 16.

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

**Correctness:** The big question that remains is why is this algorithm correct? Recall that the cost of any encoding tree  $T$  is  $B(T) = \sum_x p(x)d_T(x)$ . Our approach will be to show that any tree that differs from the one constructed by Huffman's algorithm can be converted into one that is equal to Huffman's tree without increasing its cost. First, observe that the Huffman tree is a *full binary tree*, meaning that every internal node has exactly two children. It would never pay to have an internal node with only one child (since such a node could be deleted), so we may limit consideration to full binary trees.

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

**Proof:** Let  $T$  be any optimal prefix code tree, and let  $b$  and  $c$  be two siblings at the maximum depth of the tree. Assume without loss of generality that  $p(b) \leq p(c)$  and  $p(x) \leq p(y)$  (if this is not true, then rename these characters). Now, since  $x$  and  $y$  have the two smallest probabilities it follows that  $p(x) \leq p(b)$  and  $p(y) \leq p(c)$ . (In both cases they may be equal.) Because  $b$  and  $c$  are at the deepest level of the tree we know that  $d(b) \geq d(x)$  and  $d(c) \geq d(y)$ . (Again, they may be equal.) Thus, we have  $p(b) - p(x) \geq 0$  and  $d(b) - d(x) \geq 0$ , and hence their product is nonnegative. Now switch the positions of  $x$  and  $b$  in the tree, resulting in a new tree  $T'$ . This is illustrated in Fig. 17.

Next let us see how the cost changes as we go from  $T$  to  $T'$ . Almost all the nodes contribute the same to the expected cost. The only exception are nodes  $x$  and  $b$ . By subtracting the old contributions of these nodes and adding in the new contributions we have

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

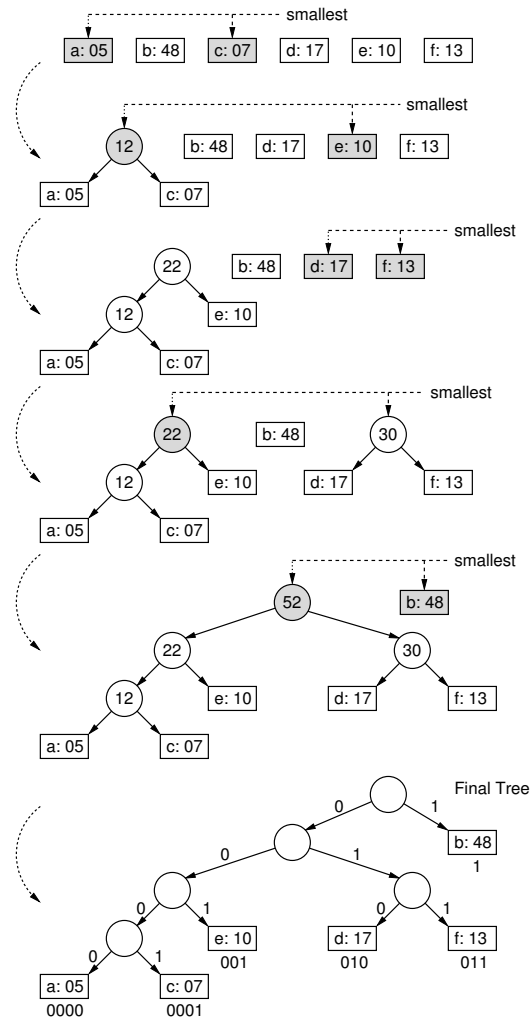


Fig. 16: Huffman's Algorithm.

---

Huffman's Algorithm

```

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

```

---

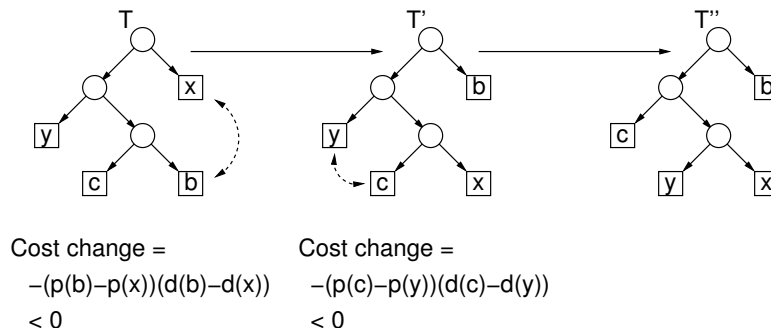


Fig. 17: Correctness of Huffman's Algorithm.

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

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

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

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

Assume inductively that when strictly fewer than  $n$  characters, Huffman's algorithm is guaranteed to produce the optimal tree. We want to show it is true with exactly  $n$  characters. Suppose we have exactly  $n$  characters. The previous claim states that we may assume that in the optimal tree, the two characters of lowest probability  $x$  and  $y$  will be siblings at the lowest level of the tree. Remove  $x$  and  $y$ , replacing them with a new character  $z$  whose probability is  $p(z) = p(x) + p(y)$ . Thus  $n - 1$  characters remain.

Consider any prefix code tree  $T$  made with this new set of  $n - 1$  characters. We can convert it into a prefix code tree  $T'$  for the original set of characters by undoing the previous operation and replacing  $z$  with  $x$  and  $y$  (adding a "0" bit for  $x$  and a "1" bit for  $y$ ). The cost of the new tree is

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

Since the change in cost depends in no way on the structure of the tree  $T$ , to minimize the cost of the final tree  $T'$ , we need to build the tree  $T$  on  $n - 1$  characters optimally. By induction, this exactly what Huffman's algorithm does. Thus the final tree is optimal.

## Minimum Spanning Trees and Kruskal's Algorithm

**Read:** Sect. 4.5 in KT or Sects. 23.1 and 23.2 in CLRS. Information on the Union-Find data structure can be found in Sect. 4.6 of KT and Chapt. 22 in CLRS.

**Minimum Spanning Trees:** A common problem in communications networks and circuit design is that of connecting together a set of nodes (communication sites or circuit components) by a network of minimal total length (where length is the sum of the lengths of connecting wires). We assume that the network is undirected. To

minimize the length of the connecting network, it never pays to have any cycles (since we could break any cycle without destroying connectivity and decrease the total length). Since the resulting connection graph is connected, undirected, and acyclic, it is a *free tree*.

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

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

A *minimum spanning tree* (MST) is a spanning tree of minimum weight. Note that the minimum spanning tree may not be unique, but it is true that if all the edge weights are distinct, then the MST will be distinct (this is a rather subtle fact, which we will not prove). Fig. 18 shows three spanning trees for the same graph, where the shaded rectangles indicate the edges in the spanning tree. The one on the left is not a minimum spanning tree, and the other two are. (An interesting observation is that not only do the edges sum to the same value, but in fact the same set of edge weights appear in the two MST's. Is this a coincidence? We'll see later.)

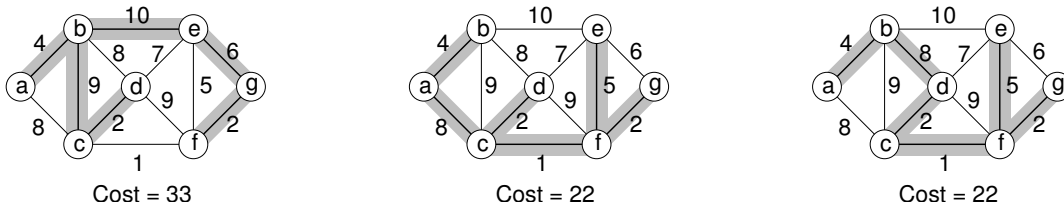


Fig. 18: Spanning trees (the middle and right are minimum spanning trees).

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

Some companies discovered that they could actually reduce their connection costs by opening a new bogus installation. Such an installation served no purpose other than to act as an intermediate point for connections. An example is shown in Fig. 19. On the left, consider four installations that lie at the corners of a  $1 \times 1$  square. Assume that all edge lengths are just Euclidean distances. It is easy to see that the cost of any MST for this configuration is 3 (as shown on the left). However, if you introduce a new installation at the center, whose distance to each of the other four points is  $1/\sqrt{2}$ . It is now possible to connect these five points with a total cost of  $4/\sqrt{2} = 2\sqrt{2} \approx 2.83$ . This is better than the MST.

In general, the problem of determining the lowest cost interconnection tree between a given set of nodes, assuming that you are allowed additional nodes (called *Steiner points*) is called the *Steiner minimum tree* (or SMT for short). An interesting fact is that although there is a simple greedy algorithm for MST's (as we will see below), the SMT problem is much harder, and in fact is NP-hard. (Luckily for AT&T, the US Legal code is rather ambiguous on the point as to whether the phone company was required to use MST's or SMT's in making connections.)

**Generic approach:** We will present two *greedy* algorithms (Kruskal's and Prim's algorithms) for computing a minimum spanning tree. Recall that a *greedy algorithm* is one that builds a solution by repeated selecting the cheapest (or generally locally optimal choice) among all options at each stage. An important characteristic of greedy algorithms is that once they make a choice, they never "unmake" this choice. Before presenting these algorithms, let us review some basic facts about free trees. They are all quite easy to prove.

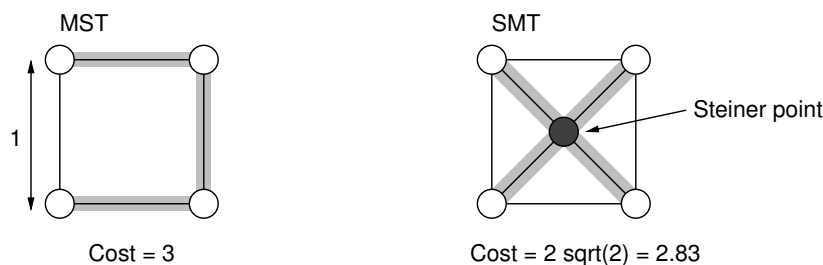


Fig. 19: Steiner Minimum tree.

**Lemma:**

- A free tree with  $n$  vertices has exactly  $n - 1$  edges.
- There exists a unique path between any two vertices of a free tree.
- Adding any edge to a free tree creates a unique cycle. Breaking *any* edge on this cycle restores a free tree.

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

**When is an edge safe?** We consider the theoretical issues behind determining whether an edge is safe or not. Let  $S$  be a subset of the vertices  $S \subseteq V$ . A *cut*  $(S, V - S)$  is just a partition of the vertices into two disjoint subsets. An edge  $(u, v)$  *crosses* the cut if one endpoint is in  $S$  and the other is in  $V - S$ . Given a subset of edges  $A$ , we say that a cut *respects*  $A$  if no edge in  $A$  crosses the cut. It is not hard to see why respecting cuts are important to this problem. If we have computed a partial MST, and we wish to know which edges can be added that do *not* induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate.

An edge of  $E$  is a *light edge* crossing a cut, if among all edges crossing the cut, it has the minimum weight (the light edge may not be unique if there are duplicate edge weights). Intuition says that since all the edges that cross a respecting cut do not induce a cycle, then the lightest edge crossing a cut is a natural choice. The main theorem which drives both algorithms is the following. It essentially says that we can always augment  $A$  by adding the minimum weight edge that crosses a cut which respects  $A$ . (It is stated in complete generality, so that it can be applied to both algorithms.)

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

**Proof:** It will simplify the proof to assume that all the edge weights are distinct. Let  $T$  be any MST for  $G$  (see Fig. 20). If  $T$  contains  $(u, v)$  then we are done. Suppose that no MST contains  $(u, v)$ . We will derive a contradiction.

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

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

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

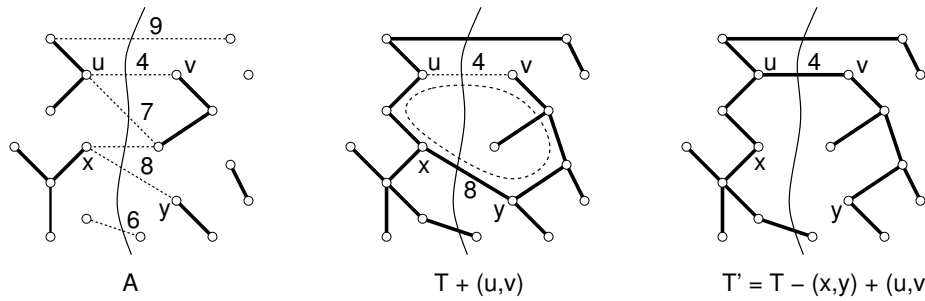


Fig. 20: Proof of the MST Lemma. Edge  $(u, v)$  is the light edge crossing cut  $(S, V - S)$ .

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

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

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

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

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

**Create-Set( $u$ ):** Create a set containing a single item  $v$ .

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

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

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

In Kruskal's algorithm, the vertices of the graph will be the elements to be stored in the sets, and the sets will be vertices in each tree of  $A$ . The set  $A$  can be stored as a simple list of edges. The algorithm is shown below, and an example is shown in Fig. 21.

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

```

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

```

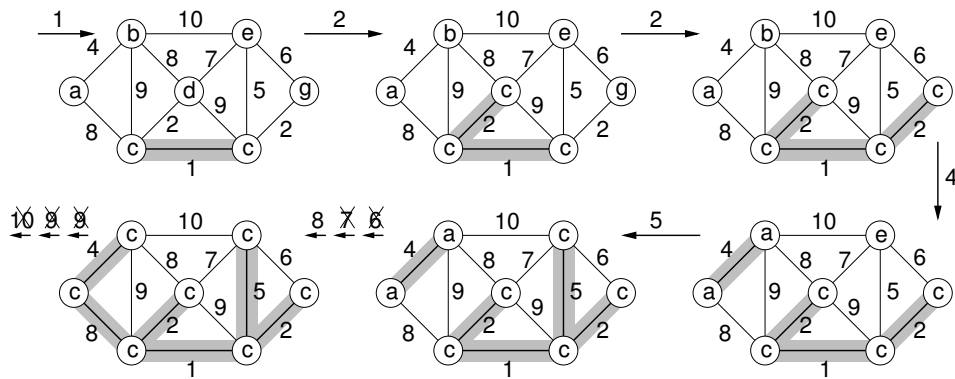


Fig. 21: Kruskal's Algorithm. Each vertex is labeled according to the set that contains it.

## Dijkstra's Algorithm for Shortest Paths

**Read:** Sect. 4.4 in KT and Chapt. 24 in CLRS.

**Shortest Paths:** Consider the problem of computing shortest paths in a directed graph. We have already seen that breadth-first search is an  $O(V + E)$  algorithm for finding shortest paths from a single source vertex to all other vertices, assuming that the graph has no edge weights. Suppose that the graph has edge weights, and we wish to compute the shortest paths from a single source vertex to all other vertices in the graph.

By the way, there are other formulations of the shortest path problem. One may want just the shortest path between a single pair of vertices. Most algorithms for this problem are variants of the single-source algorithm that we will present. There is also a single sink problem, which can be solved in the transpose digraph (that is, by reversing the edges). Computing all-pairs shortest paths can be solved by iterating a single-source algorithm over all vertices, but there are other global methods that are faster.

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

**Single Source Shortest Paths:** The *single source shortest path* problem is as follows. We are given a directed graph with *nonnegative* edge weights  $G = (V, E)$  and a distinguished *source vertex*,  $s \in V$ . The problem is to determine the distance from the source vertex to every vertex in the graph.

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

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

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

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

Consider an edge from a vertex  $u$  to  $v$  whose weight is  $w(u, v)$ . Suppose that we have already computed current estimates on  $d[u]$  and  $d[v]$ . We know that there is a path from  $s$  to  $u$  of weight  $d[u]$ . By taking this path and following it with the edge  $(u, v)$  we get a path to  $v$  of length  $d[u] + w(u, v)$ . If this path is better than the existing path of length  $d[v]$  to  $v$ , we should update  $d[v]$  to the value  $d[u] + w(u, v)$ . This is illustrated in Fig. 22. We should also remember that the shortest path to  $v$  passes through  $u$ , which we do by updating  $v$ 's predecessor pointer.

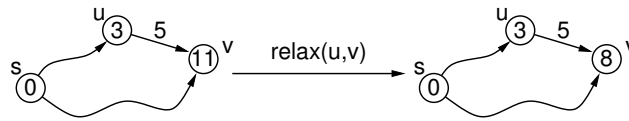


Fig. 22: Relaxation.

Relaxing an edge

---

```

Relax(u, v) {
    if (d[u] + w(u, v) < d[v]) {           // is the path through u shorter?
        d[v] = d[u] + w(u, v)             // yes, then take it
        pred[v] = u                       // record that we go through u
    }
}

```

---

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

It is not hard to see that if we perform  $Relax(u, v)$  repeatedly over all edges of the graph, the  $d[v]$  values will eventually converge to the final true distance value from  $s$ . The cleverness of any shortest path algorithm is to perform the updates in a judicious manner, so the convergence is as fast as possible. In particular, the best possible would be to order relaxation operations in such a way that each edge is relaxed exactly once. Dijkstra's algorithm does exactly this.

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

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

How do we select which vertex among the vertices of  $V - S$  to add next to  $S$ ? Here is where greedy selection comes in. Dijkstra recognized that the best way in which to perform relaxations is by increasing order of distance from the source. This way, whenever a relaxation is being performed, it is possible to infer that result of the relaxation yields the final distance value. To implement this, for each vertex in  $u \in V - S$ , we maintain a distance estimate  $d[u]$ . The greedy thing to do is to take the vertex of  $V - S$  for which  $d[u]$  is minimum, that is, take the unprocessed vertex that is closest (by our estimate) to  $s$ . Later we will justify why this is the proper choice.

In order to perform this selection efficiently, we store the vertices of  $V - S$  in a *priority queue* (e.g. a heap), where the key value of each vertex  $u$  is  $d[u]$ . Also recall that if we implement the priority queue using a heap, we can perform the operations  $insert()$ ,  $extractMin()$ , and  $decreaseKey()$ , on a priority queue of size  $n$  each in  $O(\log n)$  time. Each vertex "knows" its location in the priority queue (e.g. has a cross reference link to the priority queue entry), and each entry in the priority queue "knows" which vertex it represents. It is important when implementing the priority queue that this cross reference information is updated. The algorithm is given in the code block below, and see Fig. 23 for an example.

Notice that the coloring is not really used by the algorithm, but it has been included to make the connection with the correctness proof a little clearer. To analyze Dijkstra's algorithm, we account for the time spent on each vertex as it is extracted from the priority queue. It takes  $O(\log V)$  to extract this vertex from the queue. For each incident edge, we spend potentially  $O(\log V)$  time decreasing the key of the neighboring vertex. Thus the time



the true shortest distance from  $s$  to  $u$ . Since at the end of the algorithm, all vertices are in  $S$ , then all distance estimates are correct.

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

**Proof:** Suppose to the contrary that at some point Dijkstra's algorithm *first* attempts to add a vertex  $u$  to  $S$  for which  $d[u] \neq \delta(s, u)$ . By our observations about relaxation,  $d[u]$  is never less than  $\delta(s, u)$ , thus we have  $d[u] > \delta(s, u)$ . Consider the situation just prior to the insertion of  $u$ , and consider the true shortest path from  $s$  to  $u$ . Because  $s \in S$  and  $u \in V - S$ , at some point this path must first jump out of  $S$ . Let  $(x, y)$  be the first edge taken by the shortest path, where  $x \in S$  and  $y \in V - S$ . (Note that it may be that  $x = s$  and/or  $y = u$ ).

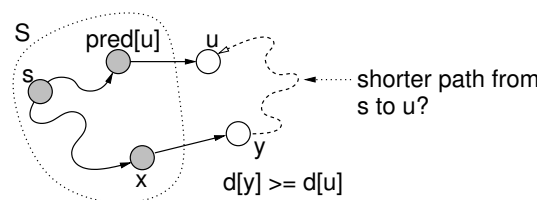


Fig. 24: Correctness of Dijkstra's Algorithm.

Because  $u$  is the first vertex where we made a mistake and since  $x$  was already processed, we have  $d[x] = \delta(s, x)$ . Since we applied relaxation to  $x$  when it was processed, we must have

$$d[y] = d[x] + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y).$$

Since  $y$  appears before  $u$  along the shortest path and edge weights are nonnegative, we have  $\delta(s, y) \leq \delta(s, u)$ . Also, because  $u$  (not  $y$ ) was chosen next for processing, we know that  $d[u] \leq d[y]$ . Putting this together, we have

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

Clearly we cannot have  $\delta(s, u) < \delta(s, u)$ , which establishes the desired contradiction.

## Dynamic Programming: Longest Common Subsequence

**Read:** This algorithm is not covered in KT, but is a simplified version of the Sequence Alignment problem of Section 6.6. This algorithm is presented in Section 15.4 in CLRS.

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

**Substructure:** Decompose your problem into smaller (and hopefully simpler) subproblems. Express the solution of the original problem in terms of solutions for smaller problems.

**Bottom-Up Construction:** Each subproblem has a natural notion of size. Larger subproblems are solved by combining solutions to smaller subproblems.

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

**Optimal substructure:** (Sometimes called the *principle of optimality*.) It states that for the global problem to be solved optimally, each subproblem should be solved optimally. (Not all optimization problems satisfy this. For example, sometimes it is better to lose a little on one subproblem in order to make a big gain on another.)

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

**Strings:** One important area of algorithm design is the study of algorithms for character strings. There are a number of important problems here. Among the most important has to do with efficiently searching for a substring or generally a pattern in large piece of text. (This is what text editors and programs like “grep” do when you perform a search.) In many instances you do not want to find a piece of text exactly, but rather something that is similar. This arises for example in genetics research and in document retrieval on the web. One common method of measuring the degree of similarity between two strings is to compute their longest common subsequence.

**Longest Common Subsequence:** Let us think of character strings as sequences of characters. Given two sequences  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Z = \langle z_1, z_2, \dots, z_k \rangle$ , we say that  $Z$  is a *subsequence* of  $X$  if there is a strictly increasing sequence of  $k$  indices  $\langle i_1, i_2, \dots, i_k \rangle$  ( $1 \leq i_1 < i_2 < \dots < i_k \leq m$ ) such that  $Z = \langle X_{i_1}, X_{i_2}, \dots, X_{i_k} \rangle$ . For example, let  $X = \langle \text{ABRACADABRA} \rangle$  and let  $Z = \langle \text{AADAA} \rangle$ , then  $Z$  is a subsequence of  $X$ .

Given two strings  $X$  and  $Y$ , the *longest common subsequence* of  $X$  and  $Y$  is a longest sequence  $Z$  that is a subsequence of both  $X$  and  $Y$ . For example, let  $X = \langle \text{ABRACADABRA} \rangle$  and let  $Y = \langle \text{YABBADABBADOO} \rangle$ . Then the longest common subsequence is  $Z = \langle \text{ABADABA} \rangle$ . See Fig. 25

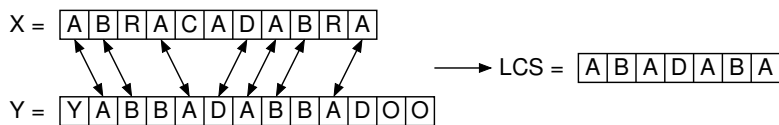


Fig. 25: An example of the LCS of two strings  $X$  and  $Y$ .

The *Longest Common Subsequence Problem* (LCS) is the following. Given two sequences  $X = \langle x_1, \dots, x_m \rangle$  and  $Y = \langle y_1, \dots, y_n \rangle$  determine a longest common subsequence. Note that it is not always unique. For example the LCS of  $\langle \text{ABC} \rangle$  and  $\langle \text{BAC} \rangle$  is either  $\langle \text{AC} \rangle$  or  $\langle \text{BC} \rangle$ .

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

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

The idea will be to compute the longest common subsequence for every possible pair of prefixes. Let  $c[i, j]$  denote the length of the longest common subsequence of  $X_i$  and  $Y_j$ . For example, in the above case we have  $X_5 = \langle \text{ABRAC} \rangle$  and  $Y_6 = \langle \text{YABBAD} \rangle$ . Their longest common subsequence is  $\langle \text{ABA} \rangle$ . Thus,  $c[5, 6] = 3$ .

Which of the  $c[i, j]$  values do we compute? Since we don’t know which will lead to the final optimum, we compute all of them. Eventually we are interested in  $c[m, n]$  since this will be the LCS of the two entire strings. The idea is to compute  $c[i, j]$  assuming that we already know the values of  $c[i', j']$ , for  $i' \leq i$  and  $j' \leq j$  (but not both equal). Here are the possible cases.

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

**Last characters match:** Suppose  $x_i = y_j$ . For example: Let  $X_i = \langle ABCA \rangle$  and let  $Y_j = \langle DACA \rangle$ . Since both end in  $A$ , we claim that the LCS *must* also end in  $A$ . (We will leave the proof as an exercise.) Since the  $A$  is part of the LCS we may find the overall LCS by removing  $A$  from both sequences and taking the LCS of  $X_{i-1} = \langle ABC \rangle$  and  $Y_{j-1} = \langle DAC \rangle$  which is  $\langle AC \rangle$  and then adding  $A$  to the end, giving  $\langle ACA \rangle$  as the answer. (At first you might object: But how did you know that these two  $A$ 's matched with each other. The answer is that we don't, but it will not make the LCS any smaller if we do.) This is illustrated at the top of Fig. 26.

$$\text{if } x_i = y_j \text{ then } c[i, j] = c[i - 1, j - 1] + 1$$

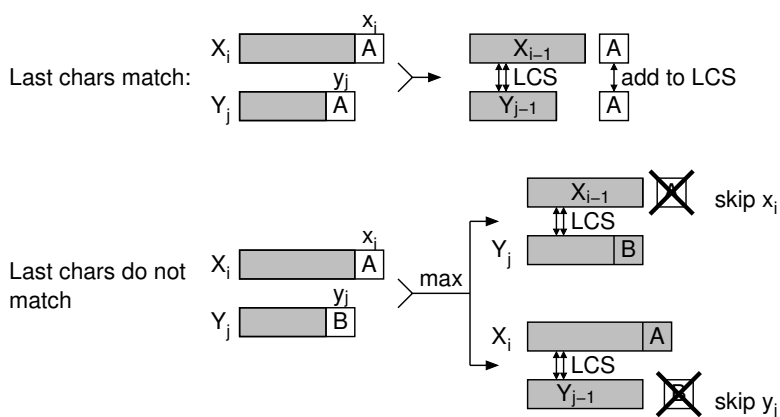


Fig. 26: The possible cases in the DP formulation of LCS.

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

At this point it may be tempting to try to make a “smart” choice. By analyzing the last few characters of  $X_i$  and  $Y_j$ , perhaps we can figure out which character is best to discard. However, this approach is doomed to failure (and you are strongly encouraged to think about this, since it is a common point of confusion.) Instead, our approach is to take advantage of the fact that we have already precomputed smaller subproblems, and use these results to guide us.

In the first case ( $x_i$  is not in the LCS) the LCS of  $X_i$  and  $Y_j$  is the LCS of  $X_{i-1}$  and  $Y_j$ , which is  $c[i - 1, j]$ . In the second case ( $y_j$  is not in the LCS) the LCS is the LCS of  $X_i$  and  $Y_{j-1}$  which is  $c[i, j - 1]$ . We do not know which is the case, so we try both and take the one that gives us the longer LCS. This is illustrated at the bottom half of Fig. 26.

$$\text{if } x_i \neq y_j \text{ then } c[i, j] = \max(c[i - 1, j], c[i, j - 1])$$

We left undone the business of showing that if both strings end in the same character, then the LCS must also end in this same character. To see this, suppose by contradiction that both characters end in  $A$ , and further suppose that the LCS ended in a different character  $B$ . Because  $A$  is the last character of both strings, it follows that this particular instance of the character  $A$  cannot be used anywhere else in the LCS. Thus, we can add it to the end of the LCS, creating a longer common subsequence than the LCS, a contradiction.

Combining these observations we have the following formulation:

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

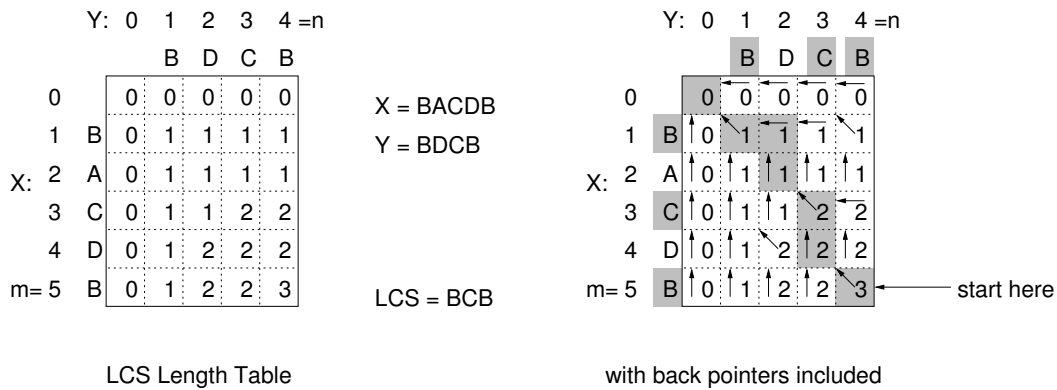


Fig. 27: Longest common subsequence example for the sequences  $X = \langle BACDB \rangle$  and  $Y = \langle BDCB \rangle$ . The numeric table entries are the values of  $c[i, j]$  and the arrow entries are used in the extraction of the sequence.

**Implementing the Formulation:** The task now is to simply implement this formulation. We concentrate only on computing the maximum *length* of the LCS. Later we will see how to extract the actual sequence. We will store some helpful pointers in a parallel array,  $b[0..m, 0..n]$ . The code is shown below, and an example is illustrated in Fig. 27

```

// Build LCS Table
LCS(x[1..m], y[1..n]) {
    // compute LCS table
    int c[0..m, 0..n]
    for i = 0 to m // init column 0
        c[i, 0] = 0; b[i, 0] = skipX
    for j = 0 to n // init row 0
        c[0, j] = 0; b[0, j] = skipY
    for i = 1 to m // fill rest of table
        for j = 1 to n
            if (x[i] == y[j]) // take X[i] (Y[j]) for LCS
                c[i, j] = c[i-1, j-1] + 1; b[i, j] = addXY
            else if (c[i-1, j] >= c[i, j-1]) // X[i] not in LCS
                c[i, j] = c[i-1, j]; b[i, j] = skipX
            else // Y[j] not in LCS
                c[i, j] = c[i, j-1]; b[i, j] = skipY
    return c[m, n] // return length of LCS
}

```

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

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

```

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

```

## Dynamic Programming: Weighted Interval Scheduling

**Read:** Section 6.1 in KT. (Not covered in CLRS.)

**Weighted Interval Scheduling:** Let us consider a variant of a problem that we have seen before, the Interval Scheduling Problem. Recall that in the original (unweighted) version we are given a set  $S = \{1, 2, \dots, n\}$  of  $n$  activity requests, which are to be scheduled to use some resource, where each activity must be started at a given start time  $s_i$  and ends at a given finish time  $f_i$ . We say that two requests are *compatible* if their intervals do not overlap, and otherwise they are said to *interfere*. The objective in the unweighted problem is to select a set of mutually compatible request of maximum size. In *weighted interval scheduling* (WIS), we assume that in addition to the start and finish times, each request is associated with a numeric value, call it  $v_i$ , and the objective is to find a set of compatible requests such that sum of values of the scheduled requests is maximum.

Observe that the original version of the interval scheduling problem can be viewed as a special case of the weighted version, in which all weights are the same (e.g., equal to 1). Although a greedy approach works fine for the unweighted problem, no greedy solution is known for the weighted version. We will consider a method based on dynamic programming.

**Recursive Formulation:** Dynamic programming solutions are based on a decomposition of a problem into smaller subproblems. Let us consider how to do this for the weighted interval scheduling problem. As we did in the greedy algorithm, it will be convenient to sort the requests according to finish time, so that  $f_1 \leq f_2 \leq \dots \leq f_n$ . Given any request  $j$ , define  $p(j)$  to be the largest  $i < j$  such that the  $i$ th and  $j$ th requests are compatible, that is,  $f_i < s_j$ . If no such  $i$  exists, let  $p(j) = 0$ . An example is shown in Fig. 28.

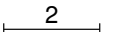
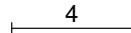

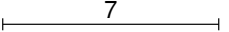
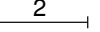

j	Intervals and values	p(j)
1		0
2		0
3		1
4		0
5		3
6		3

Fig. 28: Weighted interval scheduling and p-values.

How shall we define the subproblems? For now, let's just concentrate on computing the optimum total value. Later we will consider how to determine which requests produce that value. A natural idea would be to define

an array  $M[1..n]$ , where  $M[i]$  denotes the maximum possible value (or profit) achievable, if we restrict attention to the first  $i$  requests only. Clearly, the final desired result will be the maximum value using *all* the requests, that is,  $M[n]$ . As a starting point we have  $M[0] = 0$ , which means that we get no value if there are no requests.

In order to compute  $M[j]$  for an arbitrary  $j$ ,  $1 \leq j \leq n$ , we observe that there are two possibilities:

**Request  $j$  is not in the schedule:** If  $j$  is not included in the schedule, then we should do the best we can with the remaining  $j - 1$  requests. Therefore,  $M[j] = M[j - 1]$ .

**Request  $j$  is in the schedule:** If we add request  $j$  to the schedule, then we gain  $v_j$  units of value, but we are now limited as to which other requests we can take. We cannot take any of the requests following  $p(j)$ . Thus we have  $M[j] = v_j + M[p(j)]$ .

Now the question that remains is which of the these two options is best? The simple answer is that we don't know. Thus, let us take the maximum of the two. Thus, we have the recursive scheme given in the following code block. For simplicity, we will assume that values  $v_j$  (given by the array  $v[j]$ ) and  $p(j)$  (given by the array  $p[j]$ ) are global variables (shame on us!) and just pass in the index  $j$ . The function `recursiveWIS(j)` returns the value of  $M[j]$ , as defined above.

---

Recursive WIS

```
recursiveWIS(j) {
    if (j == 0) return 0
    else return max( recursiveWIS(j-1), v[j] + recursiveWIS(p[j]) )
}
```

---

I have left it as self-evident that this simple recursive procedure is correct. Indeed the only subtlety is the inductive observation that in order to compute  $M[j]$  optimally, the two subproblems that are used to make the final result  $M[j - 1]$  and  $M[p(j)]$  should also be computed optimally. This is an example of the principle of optimality, which in this case is clear.<sup>2</sup>

**Memoized Formulation:** The only problem with this elegant and simple recursive procedure is that it has a horrendous running time. To make this concrete, let us suppose that  $p(j) = j - 2$  for all  $j$ . Let  $T(j)$  be the number of recursive function calls that result from `recursiveWIS(j)`. Clearly,  $T(0) = 1$ ,  $T(1) = T(0) + T(0)$ , and for  $j \geq 2$ ,  $T(j) = T(j - 1) + T(j - 2)$ . The resulting series is very similar to the Fibonacci series, which grows exponentially with  $j$ , as seen below.

$j$	0	1	2	3	4	5	6	7	8	...	20	30	50
$T(j)$	1	2	3	5	8	13	21	34	55	...	17,711	2,178,309	32,951,280,099

This may seem ludicrous. (And it is!) Why should it take 32 billion recursive calls to fill in a table with 50 entries? If you unravel a few levels of the recursion, the problem jumps out immediately. The same recursive calls are being made over and over again. This suggests a smarter version of the algorithm. Once a value has been computed for `recursiveWIS(j)` we store it in the array  $M$ , and all future attempts to compute this value will simply access the array, rather than making a recursive call. This technique is called *memoizing*, and is presented in the following code block. You might imagine that we initialize all the  $M[j]$  entries to  $-1$  initially, and use this special value to determine whether an entry has already been computed.

The memoized version runs in  $O(n)$  time. To see this, observe that each invocation of `memoizedWIS` either returns in  $O(1)$  time (with no recursive calls) or it computes one new entry of  $M$ . The number of times the latter can occur is clearly  $n$ .

---

<sup>2</sup>You might think, "This is obvious. Why would it ever be better to solve a subproblem suboptimally?" Suppose, however that you had additional constraints, e.g., you have been told that the final schedule can only have 23 intervals. Now, it might be advantageous to solve a subproblem suboptimally, so that you have a few extra requests to fill at a later time.

---

```

memoizedWIS(j) {
    if (j == 0) return 0
    else if (M[j] has been computed) return M[j]
    else {
        M[j] = max( memoizedWIS(j-1), v[j] + memoizedWIS(p[j]) )
        return M[j]
    }
}

```

---

**Iterative Construction:** Yet another method for computing the values of the array, is to simply fill the table up, one entry at a time. We need to be careful that this is done in such an order that each time we access the array, the entry being accessed is already defined. This is easy here, because we can just compute values in increasing order of  $j$ . We will add one additional piece of information, which will help in reconstructing the final solution. Whenever a choice is made between two options, we'll save a *predecessor pointer*,  $\text{pred}[j]$ , which reminds of which choice we made ( $j - 1$  or  $p(j)$ ). The resulting algorithm is presented in the following code block and it is illustrated in Fig. 29. Clearly the running time is  $O(n)$ .

---

```

iterativeWIS() {
    M[0] = 0
    for (i = 1 to n) {
        if (M[j-1] > v[j] + M[p[j]]) {
            M[j] = M[j-1]; pred[j] = j-1;
        }
        else {
            M[j] = v[j] + M[p[j]]; pred[j] = p[j];
        }
    }
}

```

---

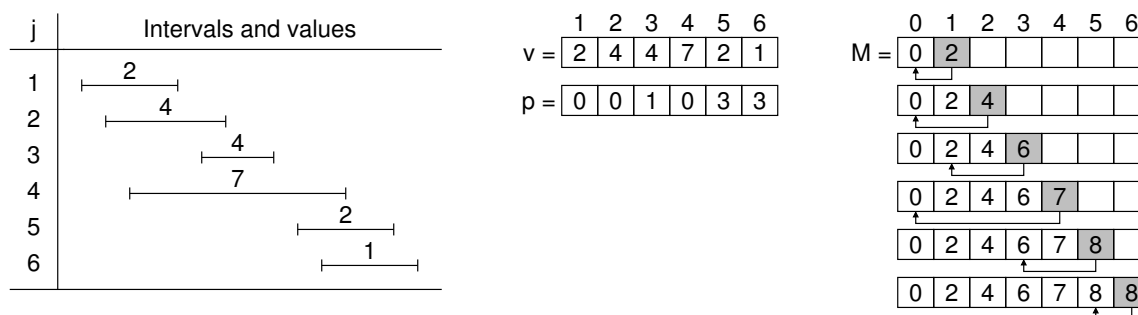


Fig. 29: Example of iterative construction and pred values. The final optimal value is 8. By following the predecessor pointers back from  $M[6]$  we see that the requests that are in the schedule are 5, 3, 1

Do you think that you understand the algorithm now? If so, answer the following question. Would the algorithm be correct if, rather than sorting the requests by finish time, we had instead sorted them by start time? How about if we didn't sort them at all?

**Computing the Final Schedule:** So far we have seen how to compute the value of the optimal schedule, but how do we compute the schedule itself? This is a common problem that arises in many DP problems, since most DP

formulations focus on computing the numeric optimal value, without consideration of the object that gives rise to this value. The solution is the leave ourselves a few hints in order to reconstruct the final result.

In the iterative WIS algorithm we did exactly this. We know that value of  $M[j]$  arose from two distinct possibilities, either (1) we didn't take  $j$  and just used the result of  $M[j - 1]$ , or (2) we did take  $j$ , added its value  $v_j$ , and used  $M[p(j)]$  to complete the result. To remind us of how we obtained the best choice for  $M[j]$  was to store a predecessor pointer  $pred[j]$ .

In order to generate the final schedule, we start with  $M[n]$  and work backwards. In general, if we arrive at  $M[j]$ , we check whether  $pred[j] = p[j]$ . If so, we can surmise that we did use the  $j$ th request, and we continue with  $pred[j] = p[j]$ . If not, then we know that we did not include request  $j$  in the schedule, and we then follow the predecessor link to continue with  $pred[j] = j - 1$ . The algorithm for generating the schedule is given in the code block below.

---

Computing WIS Schedule

```
WIS_schedule() {
    j = n
    sched = (empty list)
    while (j > 0) {
        if (pred[j] == p[j]) {
            prepend j to the front of sched
        }
        j = pred[j]
    }
}
```

---

For example, in the above example, we would start with  $M[6]$ . Its predecessor is  $5 = 6 - 1$ , which means that we did not use request 6 in the schedule. We continued with  $pred[6] = 5$ . We found that  $pred[5] = 3$ , which is not equal to  $5 - 1$ . Therefore, we know that we used request 5 in the final solution, and we continue with 3. Continuing in this manner we obtain the final list  $\langle 5, 3, 1 \rangle$ . Reversing the list gives the final schedule.

## All-Pairs Shortest Paths and the Floyd-Warshall Algorithm

**Read:** Section 25.2 in CLRS. This is not covered in KT. (At least I couldn't find it.)

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

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

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

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

We let  $w_{ij}$  denote the entry in row  $i$  and column  $j$  of  $w$ .

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

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

The output will be an  $n \times n$  distance matrix  $D = d_{ij}$  where  $d_{ij} = \delta(i, j)$ , the shortest path cost from vertex  $i$  to  $j$ . Recovering the shortest paths will also be an issue. To help us do this, we will also compute an auxiliary matrix  $mid[i, j]$ . The value of  $mid[i, j]$  will be a vertex that is somewhere along the shortest path from  $i$  to  $j$ . If the shortest path travels directly from  $i$  to  $j$  without passing through any other vertices, then  $mid[i, j]$  will be set to *null*. These intermediate values behave somewhat like the predecessor pointers in Dijkstra's algorithm, in order to reconstruct the final shortest path in  $\Theta(n)$  time.

**Floyd-Warshall Algorithm:** The Floyd-Warshall algorithm dates back to the early 60's. Warshall was interested in the weaker question of reachability: determine for each pair of vertices  $u$  and  $v$ , whether  $u$  can reach  $v$ . Floyd realized that the same technique could be used to compute shortest paths with only minor variations. The Floyd-Warshall algorithm runs in  $\Theta(n^3)$  time.

As with any DP algorithm, the key is reducing a large problem to smaller problems. A natural way of doing this is by limiting the number of edges of the path, but it turns out that this does not lead to the fastest algorithm (but is an approach worthy of consideration). The main feature of the Floyd-Warshall algorithm is in finding a the best formulation for the shortest path subproblem. Rather than limiting the number of edges on the path, they instead limit the set of vertices through which the path is allowed to pass. In particular, for a path  $p = \langle v_1, v_2, \dots, v_\ell \rangle$  we say that the vertices  $v_2, v_3, \dots, v_{\ell-1}$  are the *intermediate vertices* of this path. Note that a path consisting of a single edge has no intermediate vertices.

**Formulation:** Define  $d_{ij}^{(k)}$  to be the shortest path from  $i$  to  $j$  such that any intermediate vertices on the path are chosen from the set  $\{1, 2, \dots, k\}$ .

In other words, we consider a path from  $i$  to  $j$  which either consists of the single edge  $(i, j)$ , or it visits some intermediate vertices along the way, but these intermediate can only be chosen from among  $\{1, 2, \dots, k\}$ . The path is free to visit any subset of these vertices, and to do so in any order. For example, in the digraph shown in the Fig. 30(a), notice how the value of  $d_{5,6}^{(k)}$  changes as  $k$  varies.

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

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

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

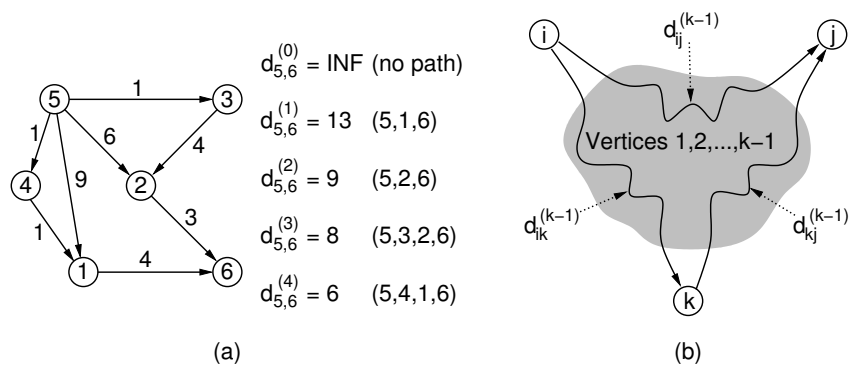


Fig. 30: Limiting intermediate vertices. For example  $d_{5,6}^{(3)}$  can go through any combination of the intermediate vertices  $\{1, 2, 3\}$ , of which  $\langle 5, 3, 2, 6 \rangle$  has the lowest cost of 8.

This suggests the following recursive rule (the DP formulation) for computing  $d^{(k)}$ , which is illustrated in Fig. 30(b).

$$d_{ij}^{(0)} = w_{ij},$$

$$d_{ij}^{(k)} = \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) \quad \text{for } k \geq 1.$$

The final answer is  $d_{ij}^{(n)}$  because this allows all possible vertices as intermediate vertices. We could write a recursive program to compute  $d_{ij}^{(k)}$ , but this will be prohibitively slow because the same value may be reevaluated many times. Instead, we compute it by storing the values in a table, and looking the values up as we need them. Here is the complete algorithm. We have also included mid-vertex pointers,  $mid[i, j]$  for extracting the final shortest paths. We will leave the extraction of the shortest path as an exercise.

---

Floyd-Warshall Algorithm

```

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

```

---

An example of the algorithm's execution is shown in Fig. 31.

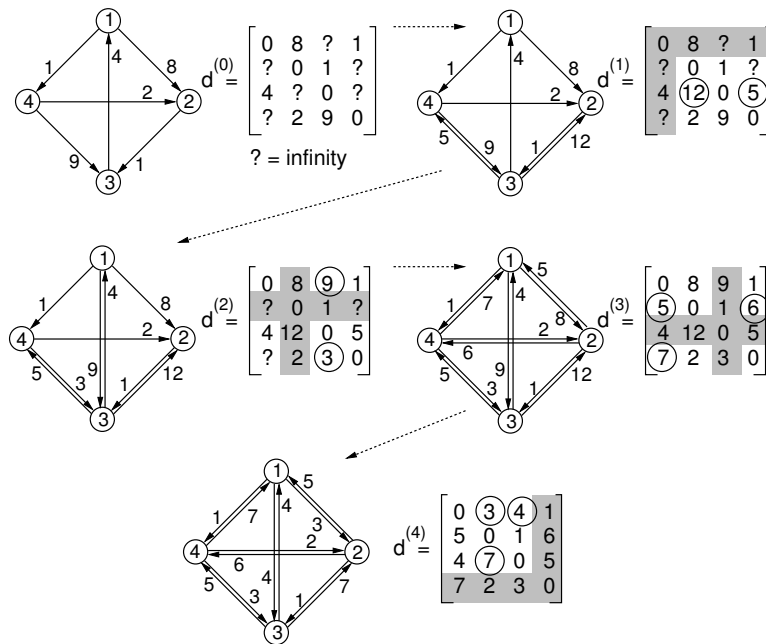


Fig. 31: Floyd-Warshall Example. Newly updates entries are circled.

Clearly the algorithm's running time is  $\Theta(n^3)$ . The space used by the algorithm is  $\Theta(n^2)$ . Observe that we deleted all references to the superscript ( $k$ ) in the code. It is left as an exercise that this does not affect the correctness of the algorithm. (Hint: The danger is that values may be overwritten and then used later in the same phase. Consider which entries might be overwritten and then reused, they occur in row  $k$  and column  $k$ . It can be shown that the overwritten values are equal to their original values.)

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

Printing the Shortest Path

```

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

```

## Dynamic Programming: Chain Matrix Multiplication

**Read:** Chapter 15 of CLRS, and Section 15.2 in particular. This algorithm is not covered in KT.

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

Suppose that we wish to multiply a series of matrices

$$A_1 A_2 \dots A_n$$

Matrix multiplication is an associative but not a commutative operation. This means that we are free to parenthesize the above multiplication however we like, but we are not free to rearrange the order of the matrices. Also recall that when two (nonsquare) matrices are being multiplied, there are restrictions on the dimensions. A  $p \times q$  matrix has  $p$  rows and  $q$  columns. You can multiply a  $p \times q$  matrix  $A$  times a  $q \times r$  matrix  $B$ , and the result will be a  $p \times r$  matrix  $C$ . (See Fig. 32. The number of columns of  $A$  must equal the number of rows of  $B$ .) In particular for  $1 \leq i \leq p$  and  $1 \leq j \leq r$ ,

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

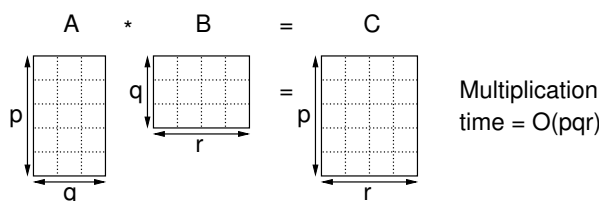


Fig. 32: Matrix Multiplication.

This corresponds to the (hopefully familiar) rule that the  $[i, j]$  entry of  $C$  is the dot product of the  $i$ th (horizontal) row of  $A$  and the  $j$ th (vertical) column of  $B$ . Observe that there are  $pr$  total entries in  $C$  and each takes  $O(q)$  time to compute, thus the total time to multiply these two matrices is proportional to the product of the dimensions,  $pqr$ .

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

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

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

**Chain Matrix Multiplication Problem:** Given a sequence of matrices  $A_1, A_2, \dots, A_n$  and dimensions  $p_0, p_1, \dots, p_n$  where  $A_i$  is of dimension  $p_{i-1} \times p_i$ , determine the order of multiplication (represented, say, as a binary tree) that minimizes the number of operations.

**Important Note:** This algorithm does not perform the multiplications, it just determines the best order in which to perform the multiplications.

**Dynamic Programming Approach:** A naive approach to this problem, namely that of trying all valid ways of parenthesizing the expression, will lead to an exponential running time. We will solve it through dynamic programming.

This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into subproblems, whose solutions can be combined to solve the global problem. As is common to any DP solution, we need to find some way to break the problem into smaller subproblems, and we need to determine a recursive formulation, which represents the optimum solution to each problem in terms of solutions to the subproblems. Let us think of how we can do this.

Since matrices cannot be reordered, it makes sense to think about sequences of matrices. Let  $A_{i..j}$  denote the result of multiplying matrices  $i$  through  $j$ . It is easy to see that  $A_{i..j}$  is a  $p_{i-1} \times p_j$  matrix. (Think about this for a second to be sure you see why.) Now, in order to determine how to perform this multiplication optimally, we need to make many decisions. What we want to do is to break the problem into problems of a similar structure. In parenthesizing the expression, we can consider the highest level of parenthesization. At this level we are simply multiplying two matrices together. That is, for any  $k$ ,  $1 \leq k \leq n - 1$ ,

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

Thus the problem of determining the optimal sequence break down into two questions:

- How do we decide where to split the chain? (what is  $k$ ?)
- How do we parenthesize the subchains  $A_{1..k}$  and  $A_{k+1..n}$ ?

Clearly, the subchain problems can be solved recursively, by applying the same scheme. So, let us think about the problem of determining the best value of  $k$ . At this point, you may be tempted to consider some clever ideas. For example, since we want matrices with small dimensions, pick the value of  $k$  that minimizes  $p_k$ . Although this is not a bad idea, in principle. (After all it might work. It just turns out that it doesn't in this case. This takes a bit of thinking, which you should try.) Instead, as is true in almost all dynamic programming solutions, we will do the dumbest thing of simply considering *all possible* choices of  $k$ , and taking the best of them. Usually trying all possible choices is bad, since it quickly leads to an exponential number of total possibilities. What saves us here is that there are only  $O(n^2)$  different sequences of matrices. (There are  $\binom{n}{2} = n(n-1)/2$  ways of choosing  $i$  and  $j$  to form  $A_{i..j}$  to be precise.) Thus, we do not encounter the exponential growth.

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

**Dynamic Programming Formulation:** We will store the solutions to the subproblems in a table, and build the table in a bottom-up manner. For  $1 \leq i \leq j \leq n$ , let  $m[i, j]$  denote the minimum number of multiplications needed to compute  $A_{i..j}$ . The optimum cost can be described by the following recursive formulation.

**Basis:** Observe that if  $i = j$  then the sequence contains only one matrix, and so the cost is 0. (There is nothing to multiply.) Thus,  $m[i, i] = 0$ .

**Step:** If  $i < j$ , then we are asking about the product  $A_{i..j}$ . This can be split by considering each  $k$ ,  $i \leq k < j$ , as  $A_{i..k}$  times  $A_{k+1..j}$ .

The optimum times to compute  $A_{i..k}$  and  $A_{k+1..j}$  are, by definition,  $m[i, k]$  and  $m[k + 1, j]$ , respectively. We may assume that these values have been computed previously and are already stored in our array. Since  $A_{i..k}$  is a  $p_{i-1} \times p_k$  matrix, and  $A_{k+1..j}$  is a  $p_k \times p_j$  matrix, the time to multiply them is  $p_{i-1}p_kp_j$ . This suggests the following recursive rule for computing  $m[i, j]$ . (See Fig. 33.)

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

It is not hard to convert this rule into a procedure, which is given below. The only tricky part is arranging the order in which to compute the values. In the process of computing  $m[i, j]$  we need to access values  $m[i, k]$  and  $m[k + 1, j]$  for  $k$  lying between  $i$  and  $j$ . This suggests that we should organize our computation according to the

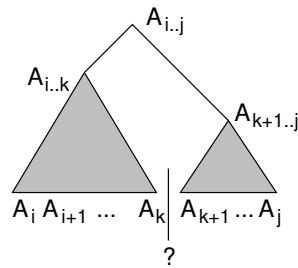


Fig. 33: Dynamic Programming Formulation.

number of matrices in the subsequence. Let  $L = j - i + 1$  denote the length of the subchain being multiplied. The subchains of length 1 ( $m[i, i]$ ) are trivial to compute. Then we build up by computing the subchains of lengths  $2, 3, \dots, n$ . The final answer is  $m[1, n]$ . We need to be a little careful in setting up the loops. If a subchain of length  $L$  starts at position  $i$ , then  $j = i + L - 1$ . Since we want  $j \leq n$ , this means that  $i + L - 1 \leq n$ , or in other words,  $i \leq n - L + 1$ . So our loop for  $i$  runs from 1 to  $n - L + 1$  (in order to keep  $j$  in bounds). The code is presented below. (See Fig. 34 for an example.)

---

Chain Matrix Multiplication

```

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

```

---

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

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

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

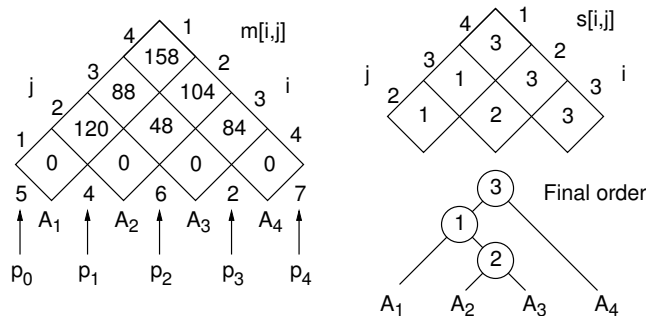


Fig. 34: Chain Matrix Multiplication Example.

Extracting Optimum Sequence

```

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

```

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

## Network Flows

**Read:** Chapt 27 in CLR or Chapt 7 in KT.

**Maximum Flow:** The Max Flow problem is one of the basic problems of algorithm design. Intuitively we can think of a flow network as a directed graph in which fluid is flowing along the edges of the graph. Each edge of the has certain maximum capacity that it can carry. The idea is to find out how much flow we can push from one point to another.

Although we in terms of “fluids” in introducing this problem, the original applications of network flow arose from transportation and communication networks, in an effort to analyze the amount of traffic or communication flowing through network. The method has remarkably diverse applications, however, many of which seem to have nothing to do with traffic, communication, or fluids. It is the simplest problem in a line of many important problems having to do with the movement of commodities through a network. These are often studied in the area of operations research.

**Flow Networks:** A flow network  $G = (V, E)$  is a directed graph in which each edge  $(u, v) \in E$  has a nonnegative capacity  $c(u, v) \geq 0$ . (In KT the capacity of edge  $e$  is denoted by  $c_e$ .) If  $(u, v) \notin E$  we model this by setting  $c(u, v) = 0$ . There are two special vertices: a source  $s$ , and a sink  $t$  (see Fig. 35(a)). We assume that there is no edge entering  $s$  and no edge leaving  $t$ . We also assume that every vertex lies on some path from the source to the sink (for otherwise the vertex is of no use to us). This implies that the digraph is weakly-connected (meaning

that the network is connected if we ignore edge directions), and hence  $e \geq n - 1$ . It will also be convenient to assume that all capacities are integers. (If in some application the capacities are fractional, we can approximate the result by multiplying all the capacities by some very large integer and rounding to the nearest integer.)

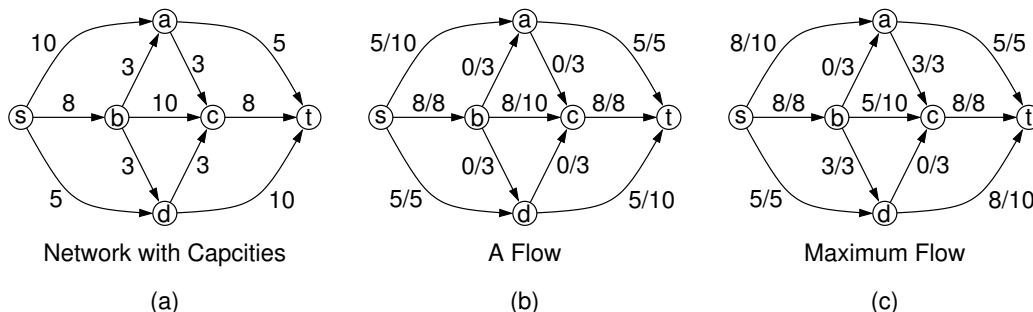


Fig. 35: Networks and Flows.

**Flows and the Maximum Flow Problem:** The next concept to introduce is the notion of the amount of flow in the network. We will adopt the notation used in KT (which differs somewhat from that presented in CLRS). A *flow* (also called an *s-t flow*) is a function  $f$  which maps each edge to a nonnegative real number, that is,  $f : E \rightarrow \mathbb{R}^+$ , which satisfies the following properties:

**Capacity Constraint:** For all  $(u, v) \in E$ ,  $f(u, v) \leq c(u, v)$ .

**Flow conservation:** (also called *flow balance*) For all  $v \in V - \{s, t\}$ , the flow into  $v$  equals the flow out of  $v$ , that is,

$$f^{\text{in}}(v) = f^{\text{out}}(v),$$

where  $f^{\text{in}}(v) = \sum_{(u,v) \in E} f(u, v)$  and  $f^{\text{out}}(v) = \sum_{(v,w) \in E} f(v, w)$ .

An example is shown in Fig. 35(b), where we use the notation  $f/c$  on each edge to denote the flow  $f$  and capacity  $c$  for this edge. Note that flow conservation *does not* apply to the source and sink, since we think of ourselves as pumping flow from  $s$  to  $t$ . The quantity  $f(u, v)$  is called the *flow* from  $u$  to  $v$ . The total *value* of the flow  $f$  is defined as the flow out of  $s$ , that is,

$$|f| = f^{\text{out}}(s) = \sum_{w \in V} f(s, w),$$

(The flow value of the flow shown in Fig. 35(b) is  $5 + 8 + 5 = 18$ .) From flow conservation, it follows easily that this is also equal to the flow into  $t$ , that is,  $f^{\text{in}}(t)$ . We will show this later. Although flow is defined only on edges, it will be convenient to extend the definition to any pair of vertices by defining  $f(u, v) = 0$  if  $(u, v) \notin E$ .

The *maximum-flow problem* is, given a flow network  $G = (V, E)$ , and source and sink vertices  $s$  and  $t$ , find the flow of maximum value from  $s$  to  $t$ . (See Fig. 35(c), which shows a flow of value  $8 + 8 + 5 = 23$ .)

**Ford-Fulkerson Method:** The most basic concept on which all network-flow algorithms work is the notion of *augmenting flows*. The idea is to start with a flow of size zero, and then incrementally make the flow larger and larger by finding a path along which we can push more flow. A path in the network from  $s$  to  $t$  along which more flow can be pushed is called an *augmenting path*. We will define this notion below, but just to see its value we observe that it forms the basis of the simplest and perhaps best known algorithm for computing network flows, called the *Ford-Fulkerson method*. (We do not call it an “algorithm,” since the method of selecting the path is not specified. We will discuss this later.)

Almost all network flow algorithms are based on this simple idea. They only differ in how they decide which path or paths along which to push flow. We will prove that when it is impossible to “push” any more flow

```

FordFulkerson(G, s, t) {
  initialize flow f to 0;
  while (there exists an augmenting path P) {
    augment the flow along P;
  }
  output the final flow f;
}
    
```

through the network, we have reached the maximum possible flow (i.e. a locally maximum flow is globally maximum).

**Residual Network:** To define the notion of an augmenting path, we first define the notion of a residual network. Given a flow network  $G$  and a flow  $f$ , define the *residual network* to be a network having the same vertex set and same source and sink, and whose edges are defined as follows:

**Forward edges:** For each edge  $(u, v)$  for which  $f(u, v) < c(u, v)$ , there are  $c(u, v) - f(u, v)$  additional units of flow that can be pushed along this edge. Create the edge  $(u, v)$  in  $G_f$  and assign it the capacity  $c_f(u, v) = c(u, v) - f(u, v)$ . Note that the flow can never exceed the capacity. The only other possibility is that  $f(u, v) = c(u, v)$ , in which case we say that this edge is *saturated*.

**Backward edges:** For each edge  $(u, v)$  for which  $f(u, v) > 0$ , there are  $f(u, v)$  units of flow that can be “undone” (conceptually cancelling it out by pushing flow in the opposite direction). Create the edge  $(v, u)$  in  $G_f$  and assign it a capacity of  $c_f(v, u) = f(u, v)$ .

An example is shown in Fig. 36(b). The capacity of each edge in the residual network is called the edge’s *residual capacity*. Observe that each edge of  $G$  may give rise to two edges in  $G_f$ , and so the number of edges of  $G_f$  may be up to twice as large as  $G$ . The key observation about the residual network is that if we can push flow through the residual network then we can push this additional amount of flow through the original network.

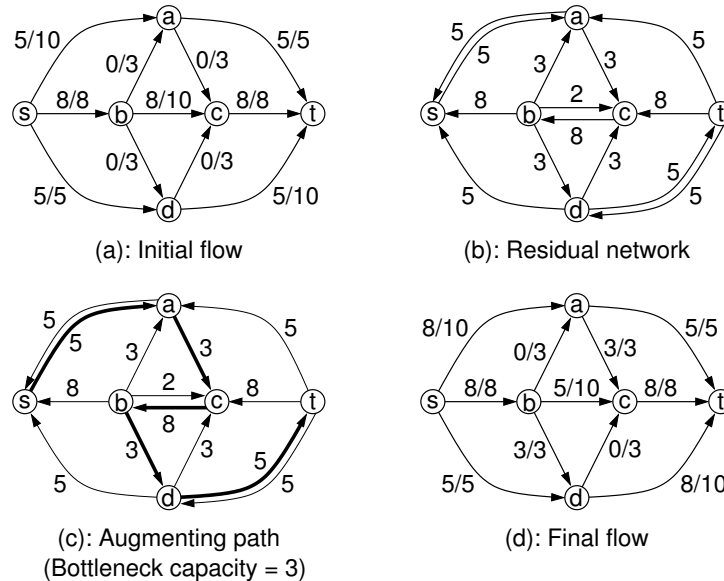


Fig. 36: Residual network and augmenting path.

**Lemma:** Let  $f$  be a flow in  $G$  and let  $f'$  be a flow in  $G_f$ . Then  $(f + f')$  (defined  $(f + f')(u, v) = f(u, v) + f'(u, v)$ ) is a flow in  $G$ . The value of the flow is  $|f| + |f'|$ .

**Proof:** (Sketch) To show that the resulting flow is valid, we need to show that it satisfies flow balance and the capacity constraints. It is easy to see that the capacities of  $G_f$  were exactly designed so that any flow along an edge of  $G_f$  when added to the flow  $f$  of  $G$  will satisfy  $G$ 's capacity constraints. Also, since both flows satisfy flow balance, it is easy to see that their sum will as well (and in fact any linear combination  $\alpha f + \beta f'$  will be balanced as well). Finally, it is easy to see that the flow on each edge of  $G$  is just the sum of the two flows, so the final flow value is their sum.

This lemma suggests that all we need to do to increase the flow is to find any flow in the residual network. This is what we consider next.

**Augmenting Paths:** An *augmenting path* is a simple path  $P$  from  $s$  to  $t$  in  $G_f$ . The *residual capacity* (also called the *bottleneck capacity*) of the path is the minimum capacity of any edge on the path. It is denoted  $c_f(P)$ . Observe that by pushing  $c_f(P)$  units of flow along each edge of the path, we obtain a valid flow in  $G_f$ , and hence by the previous lemma, we can use this to augment the flow in  $G$ .

It is important to keep in mind that while augmentation increases the overall flow, it does *not* necessarily increase the flow on every edge. Given an edge  $(u, v)$  where  $f(u, v) > 0$ , we have the edge  $(v, u)$  in the residual network, and we may undo up to  $f(u, v)$  units flow along this edge.

This establishes that running the Ford-Fulkerson method results in a valid flow. We still need to consider the question of whether it produces the optimal flow. But before doing this, let's consider its implementation.

Determining whether there exists an augmenting path from  $s$  to  $t$  is an easy exercise. First we construct the residual network, and we run either DFS or BFS on the residual network starting at  $s$ . If the search reaches  $t$  then we know that a path exists (and can follow the predecessor pointers backwards to reconstruct it). Since the residual network has  $V$  vertices and at most  $2E = O(E)$  edges, it follows that either DFS or BFS will take  $O(V + E)$  time. Thus, if we need  $M$  augmentations until termination, the overall running time of Ford-Fulkerson is  $O(M(V + E))$ . We will discuss the issue of  $M$  later, but first let us return to the issue of correctness.

**Cuts:** Recall that each augmentation step of Ford-Fulkerson produces a valid flow. Thus, all we need to do to establish its correctness is to prove that when it stops (that is, when no augmenting path exists), we have achieved the maximum flow.

We will begin with some new concepts. Define a *cut* (also called an *s-t cut*) to be a partition of the vertex set into two disjoint subsets  $X \subseteq V$  and  $Y = V - X$ , where  $s \in X$  and  $t \in Y$ . We define the *net flow* from  $X$  to  $Y$  to be the sum of flow from  $X$  to  $Y$  minus the flow from  $Y$  to  $X$ , that is,

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y) - \sum_{y \in Y} \sum_{x \in X} f(y, x).$$

(See Fig. 37.) Observe that  $f(X, Y) = -f(Y, X)$ .

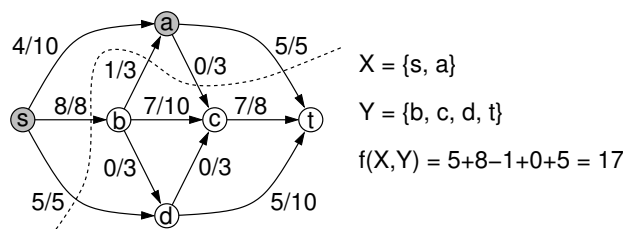


Fig. 37: Flow across the cut  $(\{s, a, b, t\}, \{c, d\})$ .

**Lemma:** Let  $(X, Y)$  be any  $s$ - $t$  cut in a network. Given any flow  $f$ , the value of  $f$  is equal to the net flow across the cut, that is,  $f(X, Y) = |f|$ .

**Proof:** Recall that there are no edges leading into  $s$ , and so we have  $|f| = f^{\text{out}}(s) - f^{\text{in}}(s)$ . Since all the other nodes of  $X$  must satisfy flow conservation it follows that

$$|f| = \sum_{x \in X} (f^{\text{out}}(x) - f^{\text{in}}(x))$$

Now, observe that every edge  $(u, v)$  where both  $u$  and  $v$  are in  $X$  contributes one positive term and one negative term of value  $f(u, v)$  to the above sum, and so all of these cancel out. The only terms that remain are the edges that either go from  $X$  to  $Y$  (which contribute positively) and those from  $Y$  to  $X$  (which contribute negatively). Thus, it follows that the value of the sum is exactly  $f(X, Y)$ , and therefore  $|f| = f(X, Y)$ .

Define the *capacity* of the cut  $(X, Y)$  to be the sum of the capacities of the edges leading from  $X$  to  $Y$ , that is,

$$c(X, Y) = \sum_{x \in X} \sum_{y \in Y} c(x, y).$$

(Note that the capacities of edges from  $Y$  into  $X$  are ignored.) Clearly it is not possible to push more flow through a cut than its capacity. Combining this with the above lemma we have:

**Lemma:** Given any  $s$ - $t$  cut  $(X, Y)$  and any flow  $f$  we have  $|f| \leq c(X, Y)$ .

The optimality of the Ford-Fulkerson method is based on the following theorem, called the Max-Flow, Min-Cut Theorem. Basically, it states that in any flow network the minimum capacity cut acts like a bottleneck to limit the maximum amount of flow. The Ford-Fulkerson method terminates when it finds this bottleneck, and hence it finds the minimum cut and maximum flow.

**Max-Flow Min-Cut Theorem:** The following three conditions are equivalent.

- (i)  $f$  is a maximum flow in  $G$ ,
- (ii) The residual network  $G_f$  contains no augmenting paths,
- (iii)  $|f| = c(X, Y)$  for some cut  $(X, Y)$  of  $G$ .

**Proof:**

- (i)  $\Rightarrow$  (ii): (By contradiction) If  $f$  is a max flow and there were an augmenting path in  $G_f$ , then by pushing flow along this path we would have a larger flow, a contradiction.
- (ii)  $\Rightarrow$  (iii): If there are no augmenting paths then  $s$  and  $t$  are not connected in the residual network. Let  $X$  be those vertices reachable from  $s$  in the residual network, and let  $Y$  be the rest. Clearly,  $(X, Y)$  forms a cut. Because each edge crossing the cut must be saturated with flow, it follows that the flow across the cut equals the capacity of the cut, thus  $|f| = c(X, Y)$ .
- (iii)  $\Rightarrow$  (i): (By contradiction) Suppose that there is a flow  $f'$  whose value exceeds  $|f|$ . Then we would have  $|f'| > c(X, Y)$ , which contradicts the previous lemma.

**Analysis of the Ford-Fulkerson method:** The problem with the Ford-Fulkerson method is that depending on how it picks augmenting paths, it may spend an inordinate amount of time arriving at the final maximum flow. Consider the example shown in Fig. 38. If the algorithm were smart enough to send flow along the top and bottom paths, each of capacity, the algorithm would terminate in two augmenting steps to a total flow of value 200. However, suppose instead that foolishly it attempted to augment first through the path going through the center edge. Then it would be limited to a bottleneck capacity of 1 unit. In the second augmentation, it could now route through the complementary path, this time undoing the flow on the center edge, and again with bottleneck capacity 1. Proceeding in this way, it would take 200 augmentations to get to the final maximum flow. We could, of course,

replace the 100's with as large a number as we like, and thus make the running time arbitrarily bad. We can build the residual network and find an augmenting path in  $O(V + E)$  time. By our assumption that  $V$  is  $O(E)$ , we can express this as  $O(E)$ . The above example shows that Ford-Fulkerson might require as many as  $|f^*|$  augmentations, where  $f^*$  is the maximum flow. Thus, the entire algorithm can take as long as  $O(E|f^*|)$  time in the worst case.

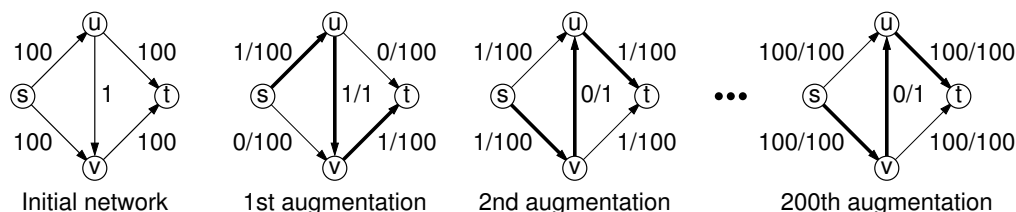


Fig. 38: Bad example for Ford-Fulkerson.

**Improvements to Ford-Fulkerson:** We have shown that if the augmenting path was chosen in a bad way the algorithm could run for a very long time before converging on the final flow. There are a number of alternatives that result in considerably better running times, however. There are two observations to make regarding the above example. First, it chooses a particularly low capacity path along which to augment, and second it chooses a path involving a relatively large number of edges. It turns out that there are two faster algorithms, which result by selecting the augmenting path specifically to avoid these problematic cases.

**Scaling Algorithm:** This is motivated by the intuition that we should augment along a path of maximum bottleneck capacity. Computing such a path is actually not required. We just need a path whose capacity is within a constant factor of the maximum. This will allow us to compute the path more efficiently (using DFS rather than some variant of Dijkstra's algorithm).

The idea is to start with an upper bound on the maximum possible flow. The sum of capacities of the edges leaving  $s$  certainly works:

$$C = \sum_{(s,v) \in E} c(s,v).$$

Next, let  $\Delta$  be the largest power of 2, such that  $\Delta \leq C$ . Given any flow  $f$  (initially the flow of value 0), define  $G_f(\Delta)$  to be the residual network consisting only of edges of residual capacity at least  $\Delta$ . Now, run Ford-Fulkerson on  $G_f(\Delta)$ , until no augmenting paths remain. (Each such augmentation will make big progress, because each augmentation will increase the flow by at least  $\Delta$ .) When no more paths remain, set  $\Delta \leftarrow \Delta/2$ , and repeat. Note that when  $\Delta = 1$ , the algorithm will produce the final maximum flow.

It can be shown (see KL for a proof) that for each value of  $\Delta$ , at most  $O(E)$  augmentations can be performed. Since it takes  $O(E)$  time to find each path (through DFS, say), the time spent for each value of  $\Delta$  is  $O(E^2)$ . Finally, since we half the value of  $\Delta$  with each iteration, it is easy to see that we will consider  $O(\log C)$  values of  $\Delta$ . Therefore, the overall running time is  $O(E^2 \log C)$ .

**Edmonds-Karp Algorithm:** This algorithm is the same as Ford-Fulkerson, but with the change that when finding the augmenting path, we compute the  $s$ - $t$  path in the residual network having the smallest number of edges (e.g., through BFS). It can be shown that the total number of augmenting paths using this method is  $O(V \cdot E)$ . Each augmenting path can be computed in  $O(E)$  time (recall that we assume that  $V$  is  $O(E)$ , and so BFS runs in  $O(V + E) = O(E)$  time). Thus, the overall running time is  $O(V \cdot E^2)$ .

Proving that the number of augmentations is  $O(V \cdot E)$  is not trivial. (See CLRS for a detailed proof.) Here is a sketch. Recall that when an edge becomes saturated (flow equals capacity), we cannot push any more flow down this edge unless we reduce the flow on it first (by pushing flow along the reverse edge in the residual network). It can be shown that each time we do this to some edge  $(u, v)$  (saturate the edge and then unsaturate it), the distance from  $s$  to  $u$  increases by at least 2. It follows that the number of times any

one edge can go through a saturation/unsaturation cycle is at most  $V/2 = O(V)$ . Since each augmenting step of Ford-Fulkerson causes at least one edge to become saturated, the total number of augmentations is  $O(V \cdot E)$ , as desired.

**Preflow-push Algorithm:** This algorithm is among the most efficient known for computing network flows. It is presented in both CLRS and KL, but we will not cover it in detail. Intuitively, it does the Edmonds-Karp algorithm one step better, by generating flows in parallel along many paths. In order to do this, it first generates an invalid flow, called a *preflow*, which satisfies capacity constraints but not flow-balance. It then gradually fixes the preflow to produce a fully valid flow. The fastest implementation of the algorithm runs in  $O(V^3)$  time.

## Applications and Extensions of Network Flow

**Read:** Chapt 27 in CLRS or Sects 7.5 and 7.7 in KT. The material on circulations is not discussed in CLRS.

**Maximum Flow:** We have introduced the concept of network flows in a directed  $s$ - $t$  network, the Min-cut Max-flow Theorem, and we have introduced a number of algorithms including the Ford-Fulkerson method, the scaling algorithm, the Edmonds-Karp algorithm, and the preflow push algorithm. Today we discuss applications and generalizations of this important concept.

**Maximum Matching:** One of the important elements of network flow is that it is a very general algorithm which is capable of solving many problems. Here is an example, which we explain in terms of a dating service, but there are many applications of this in more serious settings.

Consider the following problem, you are running a dating service and there are a set of men  $X$  and a set of women  $Y$ . Using a questionnaire you establish which men are compatible with which women. Your task is to pair up as many compatible pairs of men and women as possible, subject to the constraint that each man is paired with at most one woman, and vice versa. (It may be that some men are not paired with any woman.)

This problem may seem reminiscent of the stable marriage problem that we discussed earlier in the semester, but note that there are no preferences here, only compatibility and incompatibility constraints.

This problem is modelled by giving an undirected graph whose vertex set is  $V = X \cup Y$  and whose edge set consists of pairs  $(u, v)$ ,  $u \in X, v \in Y$  such that  $u$  and  $v$  are compatible (see Fig. 39(a)). The problem is to find a *matching*, that is a subset of edges  $M$  such that for each  $v \in V$ , there is at most one edge of  $M$  incident to  $v$ . The desired matching is the one that has the maximum number of edges, and is called a *maximum matching*.

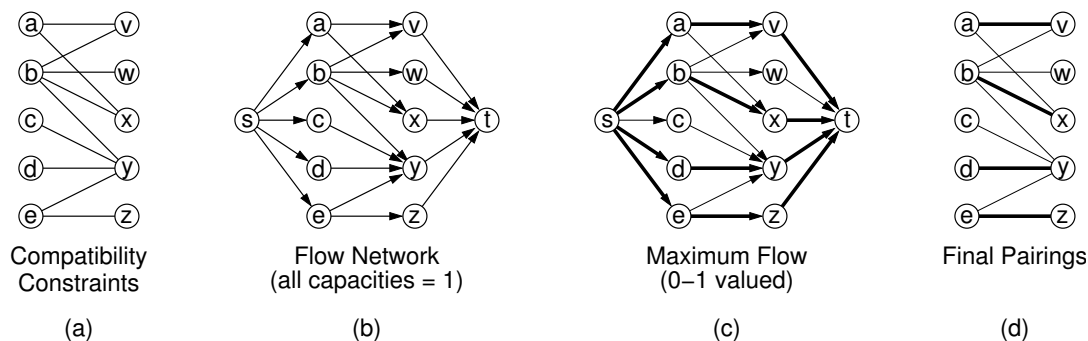


Fig. 39: Reducing bipartite matching to network flow.

The resulting undirected graph has the property that its vertex set can be divided into two groups such that all its edges go from one group to the other. This problem is called the *maximum bipartite matching problem*.

**Reduction to Network Flow:** We claim that if you have an algorithm for solving the network flow problem, then you can use this algorithm to solve the maximum bipartite matching problem.

Construct a flow network  $G' = (V', E')$  as follows. Let  $s$  and  $t$  be two new vertices and let  $V' = V \cup \{s, t\}$ .

$$E' = \{(s, u) \mid u \in X\} \cup \{(v, t) \mid v \in Y\} \cup \{(u, v) \mid (u, v) \in E\}.$$

Set the capacity of all edges in this network to 1 (see Fig. 39(b)).

Now, compute the maximum flow in  $G'$  (see Fig. 39(c)). Although in general it can be that flows are real numbers, observe that the Ford-Fulkerson method will only assign integer value flows to the edges (and this is true of all existing network flow algorithms).

Since each vertex in  $X$  has exactly one incoming edge, it can have flow along at most one outgoing edge, and since each vertex in  $Y$  has exactly one outgoing edge, it can have flow along at most one incoming edge. Thus letting  $f$  denote the maximum flow, we can define a matching

$$M = \{(u, v) \mid u \in X, v \in Y, f(u, v) > 0\}$$

(see Fig. 39(d)).

We claim that this matching is maximum because for every matching there is a corresponding flow of equal value, and for every (integer) flow there is a matching of equal value. Thus by maximizing one we maximize the other.

Because the capacities are so low, we do not need to use a fancy implementation. The total network capacity is at most  $\min(|X|, |Y|) \leq |X| + |Y| = |V|$ . Therefore, even if we run Ford-Fulkerson, the total execution time is  $O(E \cdot C) = O(E \cdot V) = O(V \cdot E)$ .

**Extensions of Network Flow:** Network flow is an important problem because it is useful in a wide variety of applications. We will discuss two useful extensions to the network flow problem. We will show that these problems can be reduced to network flow, and thus a single algorithm can be used to solve both of them. Many computational problems that would seem to have little to do with flow of fluids through networks can be expressed as one of these two extended versions.

**Circulation with Demands:** There are many problems that are similar to network flow in which, rather than transporting flow from a single source to a single sink, we have a collection of *supply nodes* that want to ship flow (or products or goods) and a collection of *demand nodes* that want to receive flow. Each supply node is associated with the amount of product it wishes to ship and each demand node is associated with the amount that it wishes to receive. The question that arises is whether there is some way to get the products from the supply nodes to the demand nodes, subject to the capacity constraints. This is a *decision problem* (or *feasibility problem*), meaning that it has a yes-no answer, as opposed to maximum flow, which is an *optimization problem*.

We can model both supply and demand nodes elegantly by associating a single numeric value with each node, called its *demand*. If  $v \in V$  is a demand node, let  $d_v$  the amount of this demand. If  $v$  is a supply node, we set its demand value to  $-d_v$ . Intuitively, shipping  $x$  units of product is equivalent to demanding receipt of  $-x$  units.<sup>3</sup> If  $v$  is neither a supply or demand node, we let  $d_v = 0$ .

Suppose that we are given a directed graph  $G = (V, E)$  in which each edge  $(u, v)$  is associated with a positive capacity  $c(u, v)$  and each vertex  $v$  is associated with a supply/demand value  $d_v$ . Let  $S$  denote the set of all vertices such that  $d_v < 0$  (supply nodes), and let  $T$  denote the set of vertices such that  $d_v > 0$  (demand nodes). Note that vertices of  $s$  may have incoming edges and vertices of  $t$  may have outgoing edges. (For example, in Fig. 40(a), we show a network in which each node is each labeled with its demand.)

We define a *circulation* to be a function  $f$  that assigns a nonnegative real number to each edge that satisfies the following two conditions. (Recall that  $f^{\text{in}}(v)$  is the sum of flows along incoming edges to  $v$  and  $f^{\text{out}}(v)$  is the sum of flows along outgoing edges from  $v$ .)

---

<sup>3</sup>I would not advise applying this in real life. I doubt that the IRS would appreciate it if your paid your \$100 tax bill by demanding that they send you  $-\$100$  dollars.

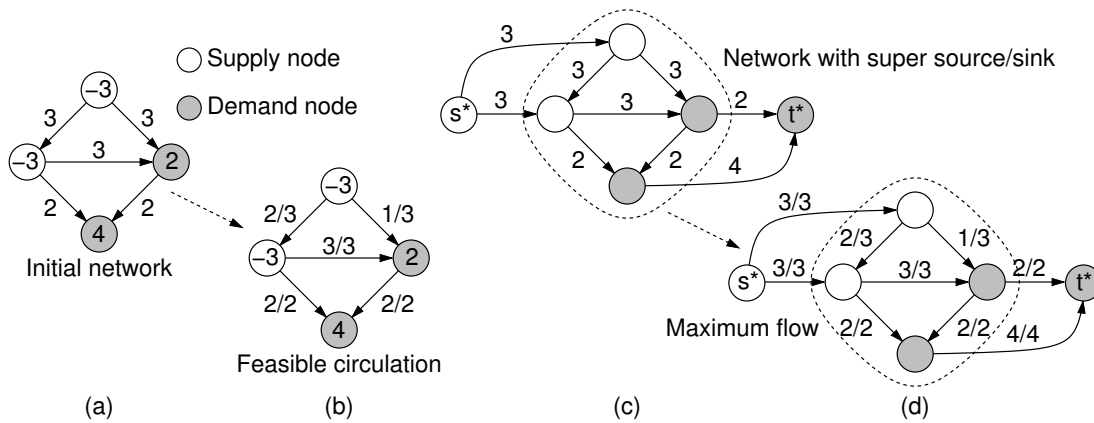


Fig. 40: Reducing the circulation problem to network flow.

**Capacity Constraints:** For each  $(u, v) \in E$ ,  $0 \leq f(u, v) \leq c(u, v)$ .

**Demand Constraints:** For vertex  $v \in V$ ,  $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ .

(In Fig. 40(b), we provide a circulation for the network of part (a).) Observe that demand constraints correspond to the flow-balance in the original max flow problem, since if a vertex is not in  $S$  or  $T$ , then  $d_v = 0$  and we have  $f^{\text{in}}(v) = f^{\text{out}}(v)$ . Also it is easy to see that the total demand must equal the total supply, otherwise we have no chance of finding a feasible circulation. That is, we require that

$$\sum_{v \in V} d_v = 0 \quad \text{or equivalently} \quad -\sum_{v \in S} d_v = \sum_{v \in T} d_v.$$

Let  $D$  denote this common value, called the *total demand*.

We claim that we can convert any instance  $G$  of the circulation problem to an equivalent network flow problem. We assume that total supply equals total demand (since if not we can simply answer “no” immediately.) The reduction is similar in spirit to what we did for bipartite matching. In particular, we create a (standard)  $s$ - $t$  network, called  $G'$ , as follows. First, we create a new *super source* vertex, called  $s^*$ , and a new *super sink* vertex, called  $t^*$ . For each supply node  $v$ , we add an edge  $(s^*, v)$  of capacity  $-d_v$ , and for each demand node  $u$ , we add an edge  $(u, t^*)$  of capacity  $d_u$  (see Fig. 40(c)).

Intuitively, these new edges will be responsible for providing the necessary supply for vertices of  $S$  and draining off the excess demand from the vertices of  $T$ . Suppose that we now compute the maximum flow in  $G'$  (e.g., by the preflow push algorithm). If the flow value is at least  $D$ , then intuitively, we have managed to push enough flow into the network and (by flow balance) enough flow out of the network to satisfy all the demand constraints (see Fig. 40(d)). The following lemma proves formally that this is a necessary and sufficient condition for a circulation to exist.

**Lemma:** There is a feasible circulation in  $G$  if and only if  $G'$  has an  $s^*$ - $t^*$  flow of value  $D$ .

**Proof:** Suppose that there is a feasible circulation  $f$  in  $G$ . The value of this circulation (the net flow coming out of all supply nodes) is clearly  $D$ . We can create a flow  $f'$  of value  $D$  in  $G'$ , by saturating all the edges coming out of  $s^*$  and all the edges coming into  $t^*$ . We claim that this is a valid flow for  $G'$ . Clearly it satisfies all the capacity constraints. To see that it satisfies the flow balance constraints observe that for each vertex  $v \in V$ , we have one of three cases:

- ( $v \in S$ ) The flow into  $v$  from  $s^*$  matches the supply coming out of  $v$  from the circulation.
- ( $v \in T$ ) The flow out of  $v$  to  $t^*$  matches the demand coming into  $v$  from the circulation.
- ( $v \in V - S - T$ ) We have  $d_v = 0$ , which means that it already satisfied flow constraint.

Conversely, suppose that we have a flow  $f'$  of value  $D$  in  $G'$ . It must be that each edge leaving  $s^*$  and each edge entering  $t^*$  is saturated. Therefore, by flow balance of  $f'$ , all the supply nodes and all the demand nodes have achieved their desired supply/demand quotas. Therefore, by ignoring the flows along the edges incident to  $s^*$  and  $t^*$ , we have a feasible circulation  $f$  for  $G$ . This completes the proof.

**Circulations with Upper and Lower Capacity Bounds:** Sometimes, in addition to having a certain maximum flow value, we would also like to impose minimum capacity constraints. That is, given a network  $G = (V, E)$ , for each edge  $(u, v) \in E$  we would like to specify two constraints  $\ell(u, v)$  and  $c(u, v)$ , where  $0 \leq \ell(u, v) \leq c(u, v)$ . A circulation function  $f$  must satisfy the same demand constraints as before, but must also satisfy both the upper and lower flow bounds:

**(New) Capacity Constraints:** For each  $(u, v) \in E$ ,  $\ell(u, v) \leq f(u, v) \leq c(u, v)$ .

**Demand Constraints:** For vertex  $v \in V$ ,  $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ .

Henceforth, we will use the term *upper flow bound* in place of *capacity* (since it doesn't make sense to talk about a lower bound as a capacity constraint). An example is shown in Fig. 41(a).

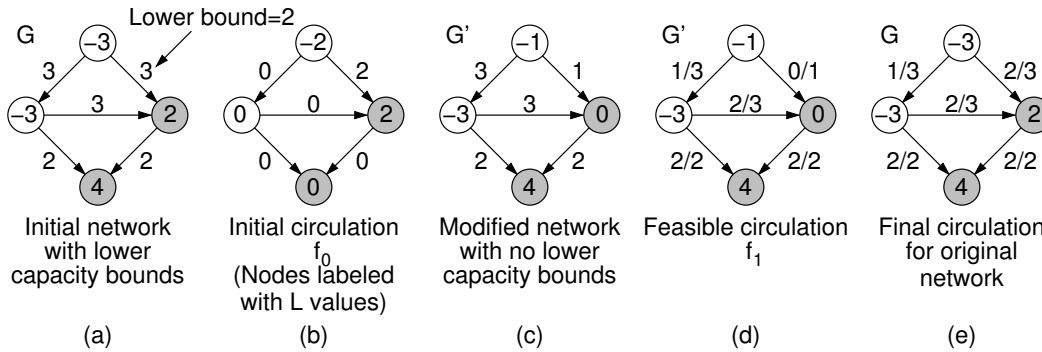


Fig. 41: Reducing the circulation problem with upper and lower flow bounds to a standard circulation problem.

We will reduce this problem to a standard circulation problem (with just the usual upper capacity bounds). To help motivate our reduction, suppose (for conceptual purposes) that we generate an initial circulation  $f_0$ . This circulation will satisfy all the lower flow bounds. In particular, we let  $f_0(u, v) = \ell(u, v)$  (see Fig. 41(b)). Of course, there is no reason to believe that  $f_0$  will satisfy the demand constraints (which, recall, provide for flow balance as well). For each  $v \in V$ , let  $L_v$  denote the excess flow coming into  $v$ , that is

$$L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{(u,v) \in E} \ell(u, v) - \sum_{(v,w) \in E} \ell(v, w).$$

(Note that this may be negative, which means that we have a flow deficit.) If we are lucky and  $L_v = d_v$ , then we satisfy the demand condition for  $v$ , and we are great shape. If not, we will superimpose a circulation  $f_1$  on top of  $f_0$  that will clear out this excess. In particular, we want to generate a net flow of  $d_v$  units coming into  $v$  and cancel out the excess  $L_v$  coming in, which suggests that we want  $f_1$  to satisfy:

$$f_1^{\text{in}}(v) - f_1^{\text{out}}(v) = d_v - L_v.$$

(Observe that if we sum  $f_0$  and  $f_1$ , then the net flow into  $v$  will be  $d_v$ , as desired.) The question is whether we can find such a circulation  $f_1$ . (We will apply the standard circulation algorithm to compute  $f_1$ .) How much capacity do we have with which to generate  $f_1$ ? We have already sent  $\ell(u, v)$  units of flow through the edge  $(u, v)$ , which implies that we have  $c(u, v) - \ell(u, v)$  capacity remaining. (Note that unlike our definition of residual graphs, we do not want to allow for the possibility of “undoing” flow. Can you see why not?)

This motivates the following construction. We create a new network  $G'$  that has all the same vertices and edges of  $G$ . We set the capacity of each edge  $(u, v) \in E$  to  $c(u, v) - \ell(u, v)$  and we set the demand of each node  $v \in V$  to  $d_v - L_v$  (see Fig. 41(c)). This network has now lower flow bounds, and so we may apply invoke the standard network circulation solution to compute a circulation  $f_1$  in  $G'$  (see Fig. 41(d)). The final circulation arises by returning  $f_0 + f_1$ . We claim that this is a valid circulation for  $G$  (with lower flow bounds) if and only if  $f_1$  is a valid circulation for  $G'$ .

**Lemma:** The network  $G$  (with both lower and upper flow bounds) has a feasible circulation if and only if  $G'$  (with only upper capacity bounds) has a feasible circulation.

**Proof:** See KL for a formal proof. Intuitively, if  $G'$  has a feasible circulation  $f'$  then the circulation  $f(u, v) = f'(u, v) + \ell(u, v)$  can be shown to be a valid circulation for  $G$  and it satisfies the lower flow bounds. Conversely, if  $G$  has a feasible circulation (satisfying both the upper and lower flow bounds), then let  $f'(u, v) = f(u, v) - \ell(u, v)$ . As above, it can be shown that  $f'$  is a valid circulation for  $G'$ . (Think of  $f'$  as  $f_1$  and  $f$  as  $f_0 + f_1$ .)

**Application: Survey Design:** Consider the following application problem, which arises in the area of data mining.

A company sells  $k$  different products, and has a database which stores which customers have bought which products recently. We want to send a survey to a subset of  $n$  customers. We will tailor each survey so it is appropriate for the particular customer it is sent to. Here are some guidelines that we want to satisfy:

- The survey sent to a customer will ask questions only about the products this customer has purchased.
- We want to get as much information as possible, but do not want to tire the customer out, so for the  $i$ th customer we will ask questions about at least  $c_i$  products and at most  $c'_i$  products, where  $0 \leq c_i \leq c'_i$ .
- To get a balanced amount of information about each product, for the  $j$ th product we survey at least  $p_j$  customers and at most  $p'_j$  customers about this product.

We can model this as a bipartite graph  $G$ , in which the customers form one of the parts of the network and products form the other part. There is an edge  $(i, j)$  if customer  $i$  has purchased product  $j$ . The flow through each customer node will reflect the number of products this customer is asked about. The flow through each product node will reflect the number of customers that are asked about this product.

This suggests the following network design. Given the bipartite graph  $G$ , we create a directed network as follows (see Fig. 42).

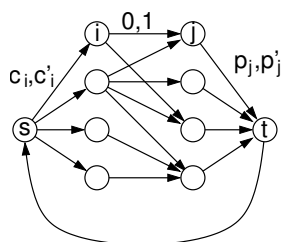


Fig. 42: Reducing the survey design problem to circulation with lower and upper flow bounds.

- For each customer  $i$  who purchased product  $j$  we create a directed edge  $(i, j)$  with lower and upper flow bounds of 0 and 1, respectively. (This reflects the fact that customer  $i$  will be asked at most once about product  $j$ .)
- We create a source vertex  $s$  and connect it to all the customers, where the edge from  $s$  to customer  $i$  has lower and upper flow bounds of  $c_i$  and  $c'_i$ , respectively. (This reflects the minimum and maximum number of products this customer will be asked about.)

- We create a sink vertex  $t$ , and create an edge from product  $j$  to  $t$  with lower and upper flow bounds of  $p_j$  and  $p'_j$ . (This reflects the minimum and maximum number of customers that will be asked about this product.)
- We create an edge  $(s, t)$ . Its lower bound is set to zero and its upper bound can be set to any very large value.
- All node demands are set to 0.

It is easy to see that if  $G$  has a valid (integer valued) circulation then each customer  $i$  receives somewhere between  $c_i$  and  $c'_i$  products to answer questions about, and each product  $j$  is asked about to between  $p_j$  and  $p'_j$  customers. The converse can also be shown. Therefore, there exists a valid circulation in  $G'$  if and only there is a valid survey design.

## NP-Completeness: Languages and the Class NP

**Read:** Chapt 34 in CLRS, up through section 34.2. The background material on NP-completeness that we will discuss today is not really discussed in KL.

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

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

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

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

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

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

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

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

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

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

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

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

**Decision Problems:** Many of the problems that we have discussed involve *optimization* of one form or another: find the shortest path, find the minimum cost spanning tree, find the minimum weight triangulation. For rather technical reasons, most NP-complete problems that we will discuss will be phrased as decision problems. A problem is called a *decision problem* if its output is a simple “yes” or “no” (or you may think of this as True/False, 0/1, accept/reject).

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

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

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

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

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

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

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

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

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

Here is a harder one, though.

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

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

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

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

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

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

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

Fig. 43 illustrates one way that the sets  $P$ , NP, NP-hard, and NP-complete (NPC) *might* look. We say *might* because we do not know whether all of these complexity classes are distinct or whether they are all solvable in polynomial time. There are some problems in the figure that we will not discuss. One is *Graph Isomorphism*, which asks whether two graphs are identical up to a renaming of their vertices. It is known that this problem is in NP, but it is not known to be in  $P$ . The other is QBF, which stands for *Quantified Boolean Formulas*. In this problem you are given a boolean formula with quantifiers ( $\exists$  and  $\forall$ ) and you want to know whether the formula is true or false. This problem is beyond the scope of this course, but may be discussed in an advanced course on complexity theory.

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

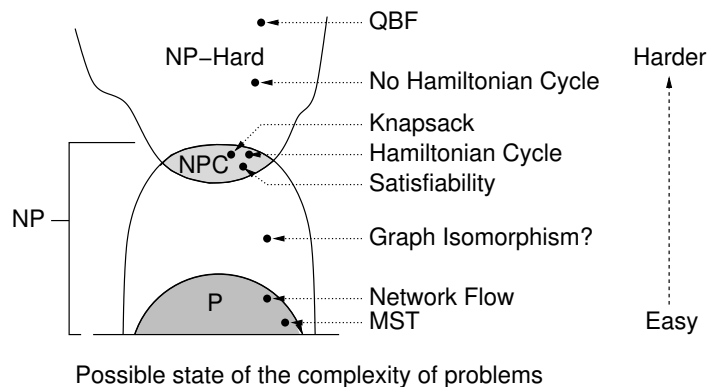


Fig. 43: The (possible) structure of P, NP, and related complexity classes.

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

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

where  $(G)$  denotes an encoding of a graph  $G$  as a string. The Hamiltonian cycle problem seems to be much harder, and there is no known polynomial time algorithm for this problem. For example, Fig. 44 shows two graphs, one which is Hamiltonian and one which is not.

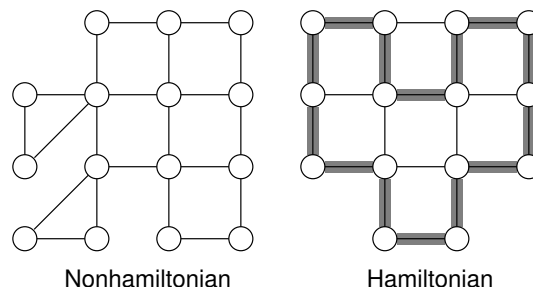


Fig. 44: Hamiltonian cycle.

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

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

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

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

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

### The class NP:

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

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

Like P, NP is a set of languages based on some complexity measure (the complexity of verification). Observe that  $P \subseteq NP$ . In other words, if we can solve a problem in polynomial time, then we can certainly verify membership in polynomial time. (More formally, we do not even need to see a certificate to solve the problem, we can solve it in polynomial time anyway).

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

## NP-Completeness: Reductions

**Read:** CLRS Chapt 34 or KT Chapt 8.

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

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

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

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

**Certificate:** is a piece of evidence that allows us to *verify* in polynomial time that a string is in a given language. For example, suppose that the language is the set of Hamiltonian graphs. To convince someone that a graph is in this language, we could supply the certificate consisting of a sequence of vertices along the cycle. It is easy to access the adjacency matrix to determine that this is a legitimate cycle in  $G$ . Therefore  $\text{HC} \in \text{NP}$ . (A certificate is one way of viewing *nondeterministic computation*, in which a program is allowed to make guesses and verify the correctness of these guesses.)

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

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

Before discussing reductions, let us just consider the following question. Suppose that there are two problems,  $H$  and  $U$ . We know (or you strongly believe at least) that  $H$  is *hard*, that is it cannot be solved in polynomial time. On the other hand, the complexity of  $U$  is *unknown*, but we suspect that it too is hard. We want to prove that  $U$  cannot be solved in polynomial time. How would we do this? We want to show that

$$(H \notin P) \Rightarrow (U \notin P).$$

To do this, we could prove the contrapositive,

$$(U \in P) \Rightarrow (H \in P).$$

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

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

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

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

Coloring arises in various partitioning problems, where there is a constraint that two objects cannot be assigned to the same set of the partition. The term “coloring” comes from the original application which was in map drawing. Two countries that share a common border should be colored with different colors. It is well known that planar graphs can be colored with 4 colors, and there exists a polynomial time algorithm for this. But determining whether 3 colors are possible (even for planar graphs) seems to be hard and there is no known polynomial time algorithm. In Fig. 45, we give two graphs, one is 3-colorable and one is not.

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

**Clique Cover (CCov):** Given a graph  $G = (V, E)$  and an integer  $k$ , can we partition the vertex set into  $k$  subsets of vertices  $V_1, V_2, \dots, V_k$ , such that  $\bigcup_i V_i = V$ , and that each  $V_i$  is a clique of  $G$ .

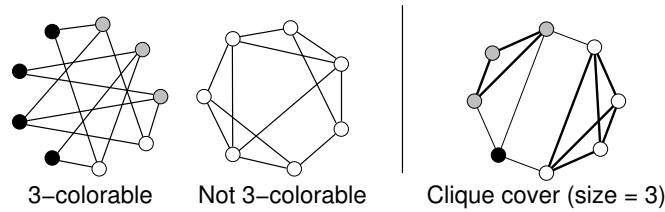


Fig. 45: 3-coloring and Clique Cover.

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

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

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

which you will show by proving the contrapositive

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

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

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

We claim that we can *reduce* the 3-coloring problem to the clique cover problem as follows. Given a graph  $G$  for which we want to determine its 3-colorability, output the pair  $(\overline{G}, 3)$  where  $\overline{G}$  denotes the complement of  $G$ . (That is,  $\overline{G}$  is a graph on the same vertices, but  $(u, v)$  is an edge of  $\overline{G}$  if and only if it is not an edge of  $G$ .) We can then feed the pair  $(\overline{G}, 3)$  into a subroutine for clique cover. This is illustrated in Fig. 46 below.

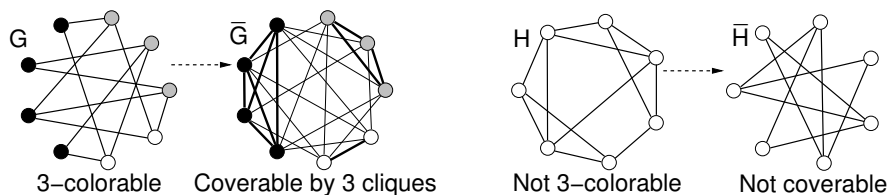


Fig. 46: Clique covers in the complement.

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

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

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

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

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

**Definition:** We say that a language (i.e. decision problem)  $L_1$  is *polynomial-time reducible*<sup>4</sup> to language  $L_2$  (written  $L_1 \leq_P L_2$ ) if there is a polynomial time computable function  $f$ , such that for all  $x$ ,  $x \in L_1$  if and only if  $f(x) \in L_2$ .

In the previous example we showed that

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

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

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

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

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

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

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

**Lemma:** If  $L_1 \leq_P L_2$  and  $L_2 \leq_P L_3$  then  $L_1 \leq_P L_3$ .

The reason is that if two functions  $f(x)$  and  $g(x)$  are computable in polynomial time, then their composition  $f(g(x))$  is computable in polynomial time as well.

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

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

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

(Note that  $L$  does not need to be in NP.)

---

<sup>4</sup>Note that our definition of *reduction* is consistent with that of CLRS, which is sometimes called a *Karp reduction*. In KT, the term *reduction* is used in a more general sense of using the known problem as a black-box, which may be called multiple times. This is sometimes known as a *Cook reduction*. It is not known whether the theory of NP-completeness is essentially different if one definition is used in place of the other.

**Definition:** A language  $L$  is *NP-complete* if:

- (1)  $L \in \text{NP}$  (that is, it can be verified in polynomial time), and
- (2)  $L$  is NP-hard (that is, every problem in NP is polynomially reducible to it).

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

**Lemma:**  $L$  is NP-complete if

- (1)  $L \in \text{NP}$  and
- (2)  $L' \leq_P L$  for some *known* NP-complete language  $L'$ .

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

This gives us a way to prove that problems are NP-complete, once we know that *one* problem is NP-complete. Unfortunately, it appears to be almost impossible to prove that one problem is NP-complete, because the definition says that we have to be able to reduce *every* problem in NP to this problem. There are infinitely many such problems, so how can we ever hope to do this? We will talk about this next time with Cook's theorem. Cook showed that there is one problem called SAT (short for boolean satisfiability) that is NP-complete. To prove a second problem is NP-complete, all we need to do is to show that our problem is in NP (and hence it is reducible to SAT), and then to show that we can reduce SAT (or generally some known NPC problem) to our problem. It follows that our problem is equivalent to SAT (with respect to solvability in polynomial time). This is illustrated in Fig. 47 below.

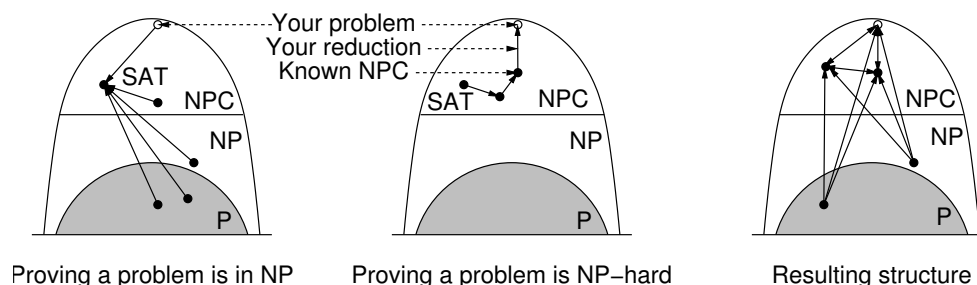


Fig. 47: Structure of NPC and reductions.

## Cook's Theorem, 3SAT, and Independent Set

**Read:** CLRS Chapt 34 or KT Chapt 8. The reduction given here is similar, but not the same as the reduction given in CLRS.

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

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

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

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

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

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

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

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

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

**Lemma:**  $L$  is NP-complete if

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

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

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

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

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

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

A boolean formula is a logical formula which consists of variables  $x_i$ , and the logical operations  $\bar{x}$  meaning the *negation* of  $x$ , *boolean-or* ( $x \vee y$ ) and *boolean-and* ( $x \wedge y$ ). Given a boolean formula, we say that it is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables such that it evaluates to 1. (As opposed to the case where every variable assignment results in 0.)

For example, the following formula

$$F_1 = (x_1 \wedge (x_2 \vee \bar{x}_3)) \wedge ((\bar{x}_2 \wedge \bar{x}_3) \vee \bar{x}_1)$$

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

$$F_2 = (\bar{x}_1 \vee (x_2 \wedge x_3)) \wedge (x_1 \vee (\bar{x}_2 \wedge \bar{x}_3)) \wedge (x_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_3)$$

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

**Cook's Theorem:** SAT is NP complete.

We will not prove this theorem. The proof would take about a full lecture (not counting the week or so of background on Turing machines). In fact, it turns out that a even more restricted version of the satisfiability problem is NP-complete. A *literal* is a variable or its negation  $x$  or  $\bar{x}$ . A formula is in *3-conjunctive normal form* (3-CNF) if it is the boolean-and of clauses where each clause is the boolean-or of exactly 3 literals. For example

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

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

As an aside, note that if we replace the 3 in 3SAT with a 2, then everything changes. If a boolean formula is given in 2-CNF form, then it is possible to determine its satisfiability in polynomial time. (It is a not too difficult exercise to show that this problem can be reduced to computing the strong components of an appropriate a directed graph.) Thus, even a seemingly small change can have a dramatic effect on the complexity of a problem.

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

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

For example, the graph shown in Fig. 48 has an independent set (shown with shaded nodes) of size 4. The independent set problem arises when there is some sort of selection problem, but there are mutual restrictions pairs that cannot both be selected. (For example, you want to invite as many of your friends to your party, but many pairs do not get along, represented by edges between them, and you do not want to invite two enemies.)

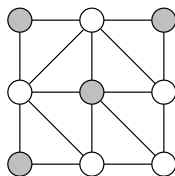


Fig. 48: Independent Set.

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

**Claim:** IS is NP-complete.

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

Secondly, we need to establish that IS is NP-hard, which can be done by showing that some known NP-complete problem (3SAT) is polynomial-time reducible to IS, that is,  $3SAT \leq_P IS$ . (See Fig. 49(a).) Let  $F$  be a boolean formula in 3-CNF form (the boolean-and of clauses, each of which is the boolean-or of 3 literals). We wish to find a polynomial time computable function  $f$  that maps  $F$  into a input for the IS problem, a graph  $G$  and integer  $k$ . That is,  $f(F) = (G, k)$ , such that  $F$  is satisfiable if and only if  $G$  has an independent set of size  $k$ .

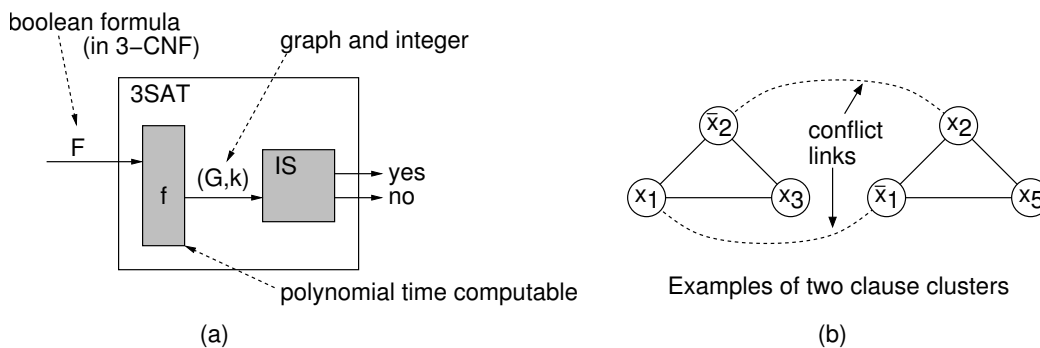


Fig. 49: (a) Reduction of 3-SAT to IS and (b) Clause clusters for the clauses  $(x_1 \vee \bar{x}_2 \vee x_3)$  and  $(\bar{x}_1 \vee x_2 \vee x_5)$ .

This will mean that if we can solve the independent set problem for  $G$  and  $k$  in polynomial time, then we would be able to solve 3SAT in polynomial time.

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

#### What is to be selected?

**3SAT:** Which variables are assigned to be true. Equivalently, which literals are assigned true.

**IS:** Which vertices are to be placed in  $V'$ .

#### Requirements:

**3SAT:** Each clause must contain at least one literal whose value it true.

**IS:**  $V'$  must contain at least  $k$  vertices.

#### Restrictions:

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

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

We want a function  $f$ , which given any 3-CNF boolean formula  $F$ , converts it into a pair  $(G, k)$  such that the above elements are translated properly. Our strategy will be to create one vertex for each literal that appears within each clause. (Thus, if there are  $m$  clauses in  $F$ , there will be  $3m$  vertices in  $G$ .) The vertices are grouped into *clause clusters*, one for each clause.

Selecting a true literal from some clause corresponds to selecting a vertex to add to  $V'$ . We set  $k$  to the number of clauses. This forces the independent set to pick one vertex from each clause, thus, one literal from each clause is true. In order to keep the IS subroutine from selecting two literals from some clause (and hence none from some other), we will connect all the vertices in each clause cluster to each other. To keep the IS subroutine from selecting both a literal and its complement we will put an edge between each literal and its complement, which we call *conflict links*. This enforces the condition that if a literal is put in the IS (set to true) then its complement literal cannot also be true. A formal description of the reduction is given below. The input is a boolean formula  $F$  in 3-CNF, and the output is a graph  $G$  and integer  $k$ .

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

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

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

---

```

 $k \leftarrow$  number of clauses in  $F$ ;
for each clause  $C$  in  $F$ 
    create a clause cluster of 3 vertices from the literals of  $C$ ;
for each clause cluster  $(x_1, x_2, x_3)$ 
    create an edge  $(x_i, x_j)$  between all pairs of vertices in the cluster;
for each vertex  $x_i$ 
    create edges between  $x_i$  and all its complement vertices  $\bar{x}_i$ ; (conflict links)
return  $(G, k)$ ;

```

---

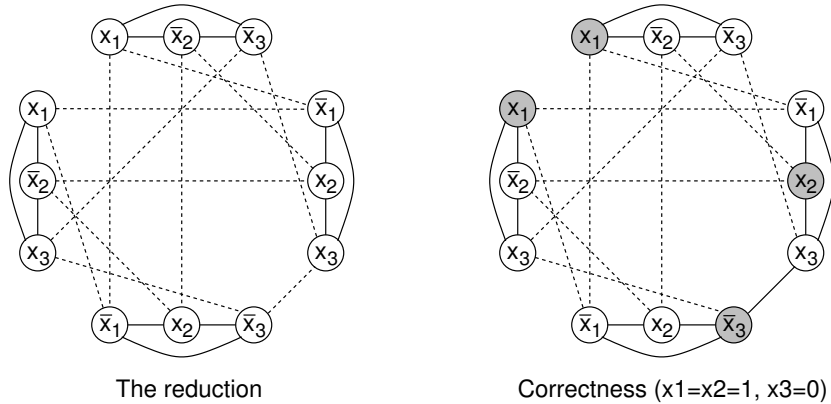


Fig. 50: 3SAT to IS Reduction for  $(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$ .

The reduction produces the graph shown in Fig. 50 and sets  $k = 4$ .

In our example, the formula is satisfied by the assignment  $x_1 = 1$ ,  $x_2 = 1$ , and  $x_3 = 0$ . Note that the literal  $x_1$  satisfies the first and last clauses,  $x_2$  satisfies the second, and  $\bar{x}_3$  satisfies the third. Observe that by selecting the corresponding vertices from the clusters, we get an independent set of size  $k = 4$ .

**Correctness Proof:** We claim that  $F$  is satisfiable if and only if  $G$  has an independent set of size  $k$ . If  $F$  is satisfiable, then each of the  $k$  clauses of  $F$  must have at least one true literal. Let  $V'$  denote the corresponding vertices from each of the clause clusters (one from each cluster). We claim that  $V'$  is an independent set of size  $k$ . Since there are  $k$  clauses, clearly  $|V'| = k$ . We only take one vertex from each clause cluster, and we cannot take two conflicting literals to be in  $V'$ . For each edge of  $G$ , both of its endpoints cannot be in  $V'$ .

Conversely, suppose that  $G$  has an independent set  $V'$  of size  $k$ . We will show how to satisfy  $F$ . First observe that  $V'$  contains exactly one vertex from each clause cluster. Clearly, it cannot contain two or more vertices from any given cluster by independence, and since there are  $k$  clusters, it must contain one from each cluster. Consider the assignment in which we set all of these literals to 1 (true). This assignment satisfies one literal in each clause, as desired. Also, it cannot assign any literal  $x_i$  and its complement  $\bar{x}_i$  to both be true, since by the conflict links they cannot both be in  $V'$ . Clearly, the transformation can be computed in polynomial time. This completes the proof.

We emphasize that our reduction did not attempt to solve the IS problem nor to solve the 3SAT. In particular, the reduction had *no knowledge* of which vertices of  $G$  are in the independent set nor which literals of  $F$  are true. Computing these things would require exponential time (by the best known algorithms). Instead the reduction simply *translated* the input from one problem into an equivalent input to the other problem, while preserving the critical elements to each problem.

## Clique, Vertex Cover, and Dominating Set

**Read:** CLRS Chapt 34 or KT Chapt 8. Dominating set is not given in our texts.

**Recap:** Last time we gave a reduction from 3SAT (satisfiability of boolean formulas in 3-CNF form) to IS (independent set in graphs). Today we give a few more examples of reductions. Recall that to show that a decision problem (language)  $L$  is NP-complete we need to show:

- (i)  $L \in \text{NP}$ . (That is, given an input and an appropriate certificate, we can verify whether the input is in the language), and
- (ii)  $L$  is NP-hard, which we can show by giving a reduction from some known NP-complete problem  $L'$  to  $L$ , that is,  $L' \leq_P L$ . (That is, there is a polynomial time function that transforms an instance  $L'$  into an equivalent instance of  $L$  for the other problem).

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

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

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

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

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

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

The CLIQUE problem is obviously closely related to the independent set problem (IS): Given a graph  $G$  does it have a  $k$  vertex subset that is completely *disconnected*. It is not quite as clear that the vertex cover problem is related. However, the following lemma makes this connection clear as well (see Fig. 51).

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

- (i)  $V'$  is a clique of size  $k$  for the complement,  $\overline{G}$ .
- (ii)  $V'$  is an independent set of size  $k$  for  $G$ .
- (iii)  $V - V'$  is a vertex cover of size  $n - k$  for  $G$ .

**Proof:**

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

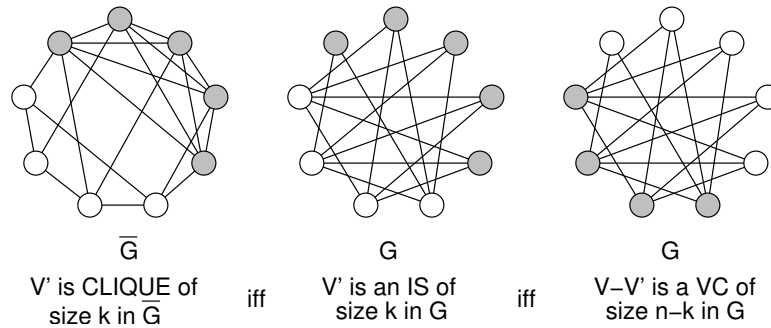


Fig. 51: Clique, Independent set, and Vertex Cover.

- (ii)  $\Rightarrow$  (iii): If  $V'$  is an independent set for  $G$ , then for each  $u, v \in V'$ ,  $\{u, v\}$  is not an edge of  $G$ , implying that every edge in  $G$  is incident to a vertex in  $V - V'$ , implying that  $V - V'$  is a VC for  $G$ .
- (iii)  $\Rightarrow$  (i): If  $V - V'$  is a vertex cover for  $G$ , then for any  $u, v \in V'$  there is no edge  $\{u, v\}$  in  $G$ , implying that there is an edge  $\{u, v\}$  in  $\bar{G}$ , implying that  $V'$  is a clique in  $\bar{G}$ .  $V'$  is an independent set for  $G$ .

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

**Theorem:** CLIQUE is NP-complete.

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

**IS  $\leq_P$  CLIQUE:** We want to show that given an instance of the IS problem  $(G, k)$ , we can produce an equivalent instance of the CLIQUE problem in polynomial time. The reduction function  $f$  inputs  $G$  and  $k$ , and outputs the pair  $(\bar{G}, k)$ . Clearly this can be done in polynomial time. By the above lemma, this instance is equivalent.

**Theorem:** VC is NP-complete.

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

**IS  $\leq_P$  VC:** We want to show that given an instance of the IS problem  $(G, k)$ , we can produce an equivalent instance of the VC problem in polynomial time. The reduction function  $f$  inputs  $G$  and  $k$ , computes the number of vertices,  $n$ , and then outputs  $(G, n - k)$ . Clearly this can be done in polynomial time. By the lemma above, these instances are equivalent.

**Note:** Note that in each of the above reductions, the reduction function did not know whether  $G$  has an independent set or not. It must run in polynomial time, and IS is an NP-complete problem. So it does not have time to determine whether  $G$  has an independent set or which vertices are in the set.

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

**Theorem:** DS is NP-complete.

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

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

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

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

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

Now we can give the complete reduction. Given the pair  $(G, k)$  for the VC problem, we create a graph  $G'$  as follows. Initially  $G' = G$ . For each edge  $\{u, v\}$  in  $G$  we create a new vertex  $w_{uv}$  in  $G'$  and add edges  $\{u, w_{uv}\}$  and  $\{v, w_{uv}\}$  in  $G'$ . Let  $I$  denote the number of isolated vertices and set  $k' = k + I$ . Output  $(G', k')$ . This reduction illustrated in Fig. 52. Note that every step can be performed in polynomial time.

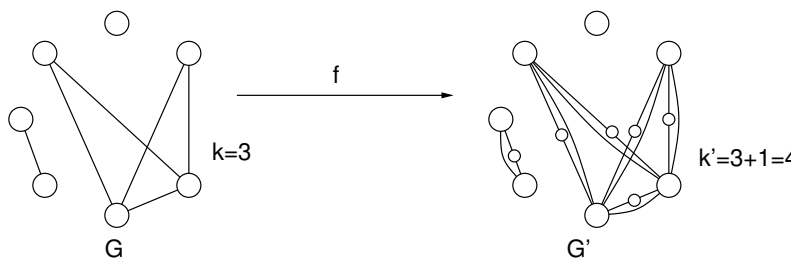


Fig. 52: Dominating set reduction.

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

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

Note that  $|V' \cup V_I|$  might be of size less than  $k + I$ , if there are any isolated vertices in  $V'$ . If so, we can add any vertices we like to make the size equal to  $k'$ .

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

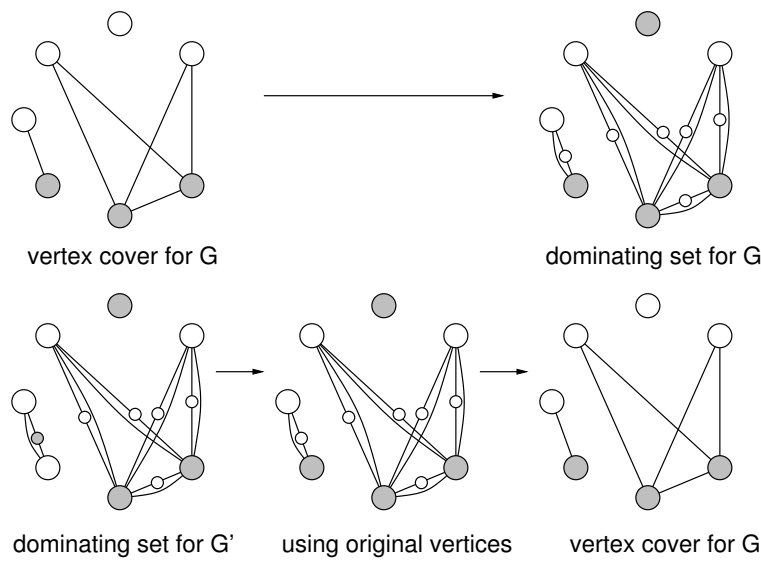


Fig. 53: Correctness of the VC to DS reduction (where  $k = 3$  and  $I = 1$ ).

In either case,  $v$  is either in  $V''$  or adjacent to a vertex in  $V''$ . This is shown in the top part of the following Fig. 53.

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

However, we claim that we never need to use any of the newly created special vertices in  $V'''$ . In particular, if some vertex  $w_{uv} \in V'''$ , then modify  $V'''$  by replacing  $w_{uv}$  with  $u$ . (We could have just as easily replaced it with  $v$ .) Observe that the vertex  $w_{uv}$  is adjacent to only  $u$  and  $v$ , so it dominates itself and these other two vertices. By using  $u$  instead, we still dominate  $u$ ,  $v$ , and  $w_{uv}$  (because  $u$  has edges going to  $v$  and  $w_{uv}$ ). Thus by replacing  $w_{u,v}$  with  $u$  we dominate the same vertices (and potentially more). Let  $V'$  denote the resulting set after this modification. (This is shown in the lower middle part of Fig 53.)

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

## Hamiltonian Path

**Read:** The reduction we present for Hamiltonian Path is completely different from the ones presented CLRS or KT.

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

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

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

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

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

**Claim:** Given the DHP-gadget:

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

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

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

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

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

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

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

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

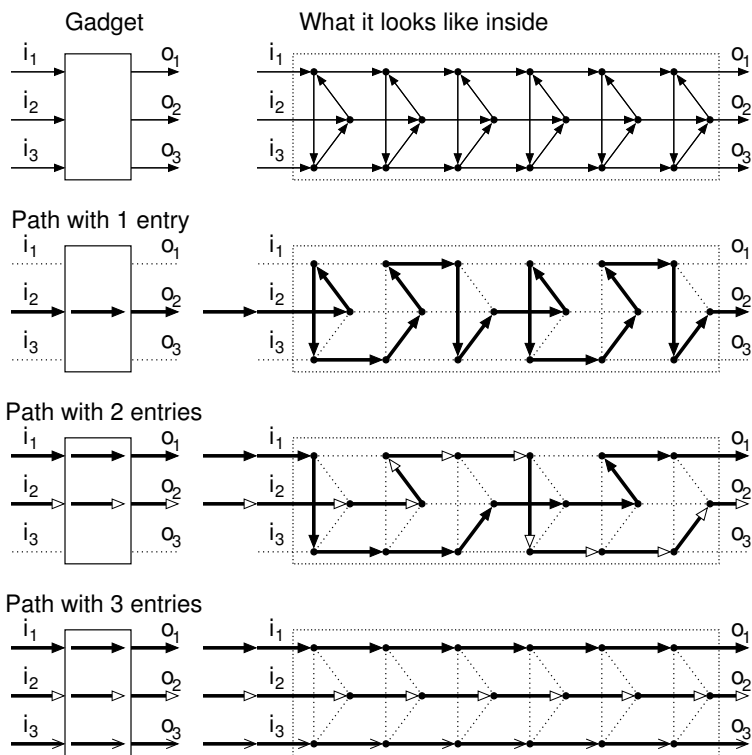


Fig. 54: DHP-Gadget and examples of path traversals.

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

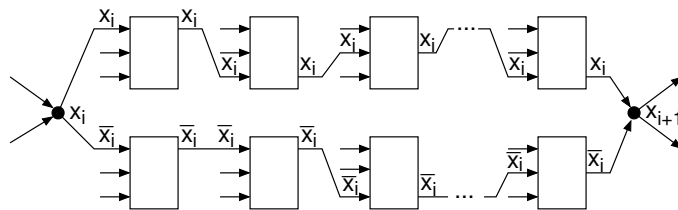


Fig. 55: General structure of reduction from 3SAT to DHP.

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

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

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

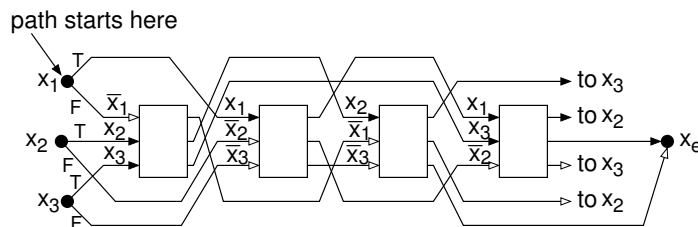


Fig. 56: Example of the 3SAT to DHP reduction.

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

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

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

**Proof:** We need to prove both the “only if” and the “if”.

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

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

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

Also observe that for each variable vertex, the path will proceed along either the true path or the false path. If it proceeds along the true path, set the corresponding variable to 1 and otherwise set it to 0. We will show that the resulting assignment is a satisfying assignment for  $F$ .

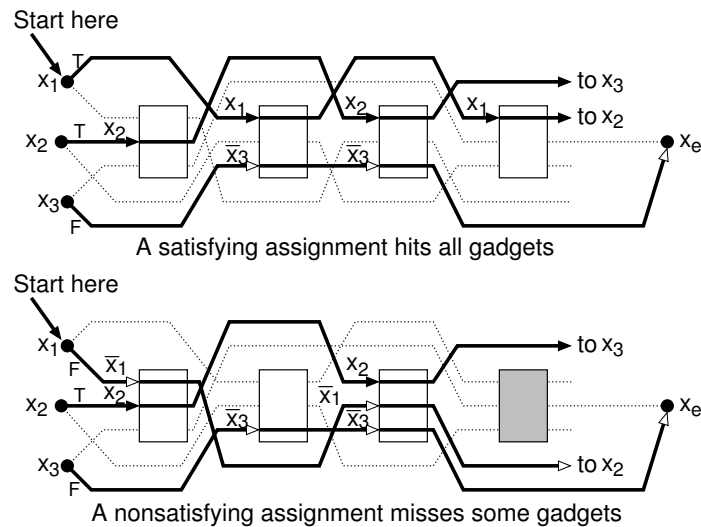


Fig. 57: Correctness of the 3SAT to DHP reduction. The upper figure shows the Hamiltonian path resulting from the satisfying assignment,  $x_1 = 1$ ,  $x_2 = 1$ ,  $x_3 = 0$ , and the lower figure shows the non-Hamiltonian path resulting from the non-satisfying assignment  $x_1 = 0$ ,  $x_2 = 1$ ,  $x_3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

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

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

## Approximation Algorithms: Vertex Cover and TSP

**Read:** Chapt 35 in CLRS. Approximation algorithms are covered in Chapt 11 of KT. The VC approximation is different from the one in KT, and the TSP approximation is not covered there.

**Coping with NP-completeness:** With NP-completeness we have seen that there are many important optimization problems that are likely to be quite hard to solve exactly. Since these are important problems, we cannot simply give up at this point, since people do need solutions to these problems. How do we cope with NP-completeness:

**Use brute-force search:** Even on the fastest parallel computers this approach is viable only for the smallest instances of these problems.

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

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

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

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

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

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

Observe that  $\rho(n)$  is always greater than or equal to 1, and it is equal to 1 if and only if the approximate solution is the true optimum solution.

Some NP-complete problems can be approximated arbitrarily closely. Such an algorithm is given both the input, and a real value  $\epsilon > 0$ , and returns an answer whose ratio bound is at most  $(1 + \epsilon)$ . Such an algorithm is called a *polynomial time approximation scheme* (or *PTAS* for short). The running time is a function of both  $n$  and  $\epsilon$ . As  $\epsilon$  approaches 0, the running time increases beyond polynomial time. For example, the running time might be  $O(n^{\lceil 1/\epsilon \rceil})$ . If the running time depends only on a polynomial function of  $1/\epsilon$  then it is called a *fully polynomial-time approximation scheme*. For example, a running time like  $O((1/\epsilon)^2 n^3)$  would be such an example, whereas  $O(n^{1/\epsilon})$  and  $O(2^{(1/\epsilon)n})$  are not.

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

- For some NP-complete problems, it is very unlikely that any approximation algorithm exists. For example, if the graph TSP problem had an approximation algorithm with a ratio bound of any value less than  $\infty$ , then  $P = NP$ .
- Many NP-complete can be approximated, but the ratio bound is a (slow growing) function of  $n$ . For example, the set cover problem (a generalization of the vertex cover problem), can be approximated to within a factor of  $\ln n$ .
- Some NP-complete problems can be approximated to within a fixed constant factor. We will discuss two examples below.
- Some NP-complete problems have PTAS's. One example is the subset problem and the Euclidean TSP problem.

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

**Vertex Cover:** We begin by showing that there is an approximation algorithm for vertex cover with a ratio bound of 2, that is, this algorithm will be guaranteed to find a vertex cover whose size is at most twice that of the optimum. Recall that a vertex cover is a subset of vertices such that every edge in the graph is incident to at least one of these vertices. The *vertex cover optimization problem* is to find a vertex cover of minimum size.

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

Here is a very simple algorithm, that guarantees an approximation within a factor of 2 for the vertex cover problem. It is based on the following observation. Consider an arbitrary edge  $(u, v)$  in the graph. One of its two vertices *must* be in the cover, but we do not know which one. The idea of this heuristic is to simply put *both* vertices into the vertex cover. (You cannot get much stupider than this!) Then we remove all edges that are incident to  $u$  and  $v$  (since they are now all covered), and recurse on the remaining edges. For every one vertex that must be in the cover, we put two into our cover, so it is easy to see that the cover we generate is at most twice the size of the optimum cover. The approximation is given in Fig. 58. Here is a more formal proof of its approximation bound.

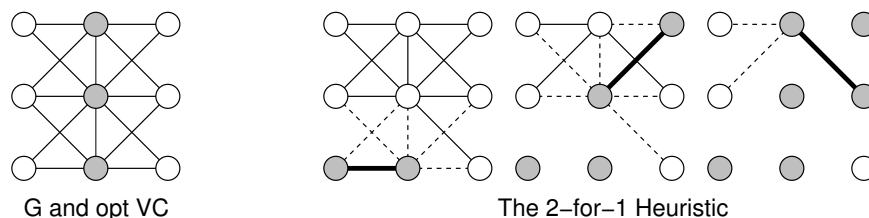


Fig. 58: The 2-for-1 heuristic for vertex cover.

2-for-1 Approximation for VC

```

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

```

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

**Proof:** Consider the set  $C$  output by ApproxVC. Let  $C^*$  be the optimum VC. Let  $A$  be the set of edges selected by the line marked with “(\*)” in Fig. 58. Observe that the size of  $C$  is exactly  $2|A|$  because we add two vertices for each such edge. However note that in the optimum VC one of these two vertices must have been added to the VC, and thus the size of  $C^*$  is at least  $|A|$ . Thus we have:

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

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

**The Greedy Heuristic:** It seems that there is a very simple way to improve the 2-for-1 heuristic. This algorithm simply selects any edge, and adds both vertices to the cover. Instead, why not concentrate instead on vertices of

high degree, since a vertex of high degree covers the maximum number of edges. This is greedy strategy. We saw in the minimum spanning tree and shortest path problems that greedy strategies were optimal.

Here is the greedy heuristic. Select the vertex with the maximum degree. Put this vertex in the cover. Then delete all the edges that are incident to this vertex (since they have been covered). Repeat the algorithm on the remaining graph, until no more edges remain. This algorithm is illustrated in Fig. 59.

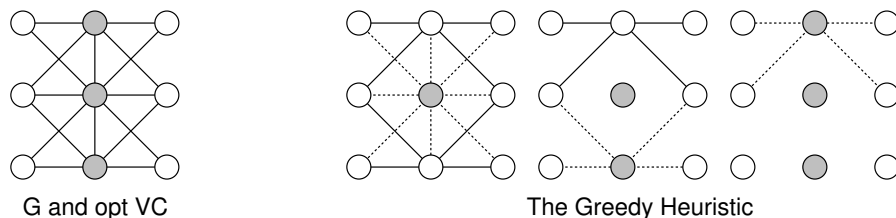


Fig. 59: The greedy heuristic for vertex cover.

Greedy Approximation for VC

```

GreedyVC(G=(V,E)) {
  C = empty-set;
  while (E is nonempty) do {
    let u be the vertex of maximum degree in G;
    add u to C;
    remove from E all edges incident to u;
  }
  return C;
}

```

It is interesting to note that on the example shown in the figure, the greedy heuristic actually succeeds in finding the optimum vertex cover. Can we prove that the greedy heuristic always outperforms the stupid 2-for-1 heuristic? The surprising answer is an emphatic “no”. In fact, it can be shown that the greedy heuristic does not even have a constant performance bound. That is, it can perform arbitrarily poorly compared to the optimal algorithm. It can be shown that the ratio bound grows as  $\Theta(\log \Delta)$ , where  $\Delta$  is the maximum degree of any vertex in the graph. However, it should also be pointed out that the vertex cover constructed by the greedy heuristic is (for typical graphs) smaller than that one computed by the 2-for-1 heuristic, so it would probably be wise to run both algorithms and take the better of the two.

**Reductions and Approximations:** Now that we have a factor-2 approximation for one NP-complete problem (vertex cover), it would seem that through the use of polynomial reductions, we have a factor-2 approximation for all NP-complete problems. Unfortunately, this is not true. The reason is that approximation factors are not generally preserved by transformations.

For example, recall that if  $V'$  is a vertex cover for  $G$ , then  $V - V'$  is an independent set for  $G$ . Suppose that  $G$  has  $n$  vertices, and a minimum vertex cover  $V'$  of size  $k$ . Then our heuristic is guaranteed to produce a vertex cover  $V''$  that is of size at most  $2k$ . If we consider the complement set  $V - V'$ , we know that  $G$  has a maximum independent set of size  $n - k$ . By complementing our approximation  $V - V''$  we have an “approximate” independent set of size  $n - 2k$ . Let us consider the resulting performance ratio

$$\frac{n - k}{n - 2k}.$$

The problem is that this ratio may be arbitrarily large. For example, if  $n = 1001$  and  $k = 500$ , then the ratio is  $501/(1001 - 1000) = 500/1 = 500$ . This is terrible.

The downside of the theory NP-completeness is that it does not apply to approximations. Some NP-complete problems can be approximated very well, and others cannot be approximated well at all (unless  $P = NP$ ).

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

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

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

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

When the underlying cost function satisfies the triangle inequality there is an approximation algorithm for TSP with a ratio-bound of 2. (In fact, there is a slightly more complex version of this algorithm that has a ratio bound of 1.5, but we will not discuss it.) Thus, although this algorithm does not produce an optimal tour, the tour that it produces cannot be worse than twice the cost of the optimal tour.

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

Here is how the algorithm works. Given any free tree there is a tour of the tree called a *twice around tour* that traverses the edges of the tree twice, once in each direction. Fig. 60 shows an example of this.

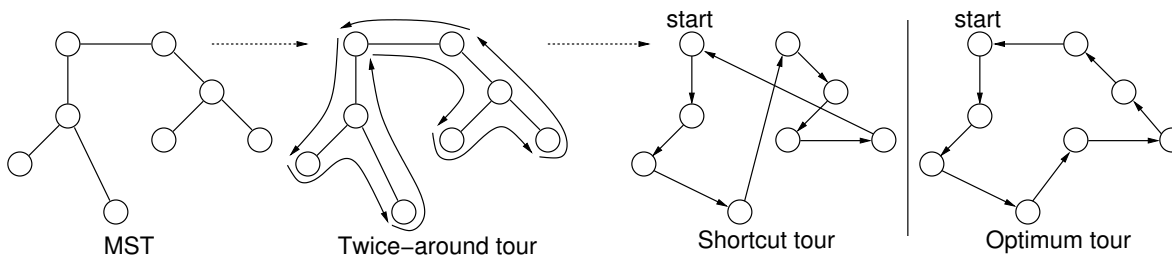


Fig. 60: TSP Approximation.

This path is not simple because it revisits vertices, but we can make it simple by *short-cutting*, that is, we skip over previously visited vertices. Notice that the final order in which vertices are visited using the short-cuts is exactly the same as a preorder traversal of the MST. (In fact, any subsequence of the twice-around tour which visits each vertex exactly once will suffice.) The triangle inequality assures us that the path length will not increase when we take short-cuts.

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

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

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

---

```

ApproxTSP(G=(V,E)) {
  T = minimum spanning tree for G
  r = any vertex
  L = list of vertices visited by a preorder walk of T
      starting with r
  return L
}

```

---

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

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

Combining these we have

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

## Approximations: Set Cover and Bin Packing

**Read:** Chapt 35 of CLRS and Chapt 11 of KT. Bin packing is covered as an exercise in both books. It is closely related to the load balancing problem of Section 11.1 of KT.

**Set Cover:** An important class of optimization problems involves covering a certain domain, with sets of a certain characteristics. Many of these problems can be expressed abstractly as the *Set Cover Problem*. You are given a pair  $(U, F)$  where  $U = \{x_1, x_2, \dots, x_m\}$  is a finite set (a domain of elements), called the *universe*, and  $F = \{S_1, S_2, \dots, S_n\}$  is a family of subsets of  $U$ , such that every element of  $U$  belongs to at least one set of  $F$ . A subset  $C \subseteq F$  is a *cover* if every element of  $U$  belongs to at least one set of  $C$ , that is,

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

In the decision problem formulation we are given an integer  $k$  and want to know whether there exists a set cover of size  $k$ . In the optimization version, we want to compute a cover consisting of the smallest number of subsets. An example is shown in Fig. 61.

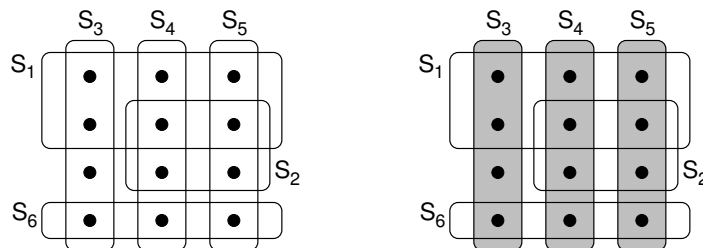


Fig. 61: Set cover. The optimum set cover consists of the three sets  $\{S_3, S_4, S_5\}$ .

Set cover arises in a number of applications. For example, suppose you have a collection of possible location for cell-phone towers. Each tower location provides coverage for some local region. You want to determine the minimum number of towers in which to place your receivers in order to cover the entire city.

A more general formulation (discussed in KT) is a weighted variant, in which each set  $S_i$  is associated with a positive weight  $w_i$ , and the problem is to compute the set cover of minimum weight. (The version described above is equivalent to setting  $w_i = 1$  for all  $i$ .) We will not discuss the weighted version, but the greedy approximation algorithm that we will present and its analysis apply the weighted case. (You might think about how to generalize the algorithm we give and its proof.)

**Complexity of Set Cover:** We have seen special cases of the set cover problems that are NP-complete. For example, Vertex Cover problem is a special case set cover. Given a graph  $G = (V, E)$ , for each vertex  $u \in V$ , let  $E_u$  denote the set of edges incident to  $u$ . Clearly, any  $V' \subseteq V$  is a vertex cover if and only if the corresponding sets covers all the edges, that is,  $\bigcup_{u \in V'} E_u = E$ . More formally, this is an instance of set cover where  $F = \{E_u \mid u \in V\}$  and the universe is  $U = E$ . If we were able to solve Set Cover in polynomial time, we could solve the Vertex Cover problem as well. It follows easily that Set Cover (stated as a decision problem) is NP-complete.

There is a factor-2 approximation for the vertex cover problem, but it cannot be applied to generate a factor-2 approximation for set cover. (Recall that VC is a special case of Set Cover.) In particular, the VC approximation relies on the fact that each element of the domain (an edge) is in exactly 2 sets (one for each of its endpoints). Unfortunately, this is not true for the general set cover problem. In fact, it is known that there is no constant factor approximation to the set cover problem, unless  $P = NP$ . This is unfortunate, because set cover is one of the most pervasive NP-complete problems.

Today we will show that there is a reasonable approximation algorithm, the *greedy heuristic*, which achieves an approximation factor of at most  $\ln m$ , where  $m = |U|$ . (Recall that  $\ln$  denotes the natural logarithm.) KT proves a stronger bound, namely that the approximation factor is at most  $\ln d$ , where  $d = \max_i |S_i|$ . Clearly, this is much better if you have many small sets.)

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

---

Greedy Set Cover

```

Greedy-Set-Cover(U, F) {
    X = U;                               // X stores the uncovered items
    C = empty;                             // C stores the sets of the cover
    while (X is nonempty) {
        select S in F that covers the most elements of X;
        add S to C;
        X = X - S;
    }
    return C
}

```

---

For the example given earlier the greedy-set cover algorithm would select  $S_1$  (since it covers 6 out of 12 elements), then  $S_6$  (since it covers 3 out of the remaining 6), then  $S_2$  (since it covers 2 of the remaining 3) and finally  $S_3$ . Thus, it would return a set cover of size 4, whereas the optimal set cover has size 3. (See Fig. 62.)

**What is the approximation factor?** The problem with the greedy set cover algorithm is that it can be “fooled” into picking the wrong set, over and over again. Consider the example shown in Fig. 63. The optimal set cover consists of sets  $S_5$  and  $S_6$ , each of size 16. Initially all three sets  $S_1$ ,  $S_5$ , and  $S_6$  have 16 elements. If ties are broken in the worst possible way, the greedy algorithm will first select sets  $S_1$ . We remove all the covered elements. Now  $S_2$ ,  $S_5$  and  $S_6$  all cover 8 of the remaining elements. Again, if we choose poorly,  $S_2$  is chosen. The pattern repeats, choosing  $S_3$  (size 4),  $S_4$  (size 2) and finally  $S_5$  and  $S_6$  (each of size 1).

Thus, the optimum cover consisted of two sets, but we picked roughly  $\lg m$ , where  $m = |X|$ , for a ratio bound of  $(\lg m)/2$ . (Recall the  $\lg$  denotes logarithm base 2.) There were many cases where ties were broken badly here, but it is possible to redesign the example such that there are no ties, and yet the algorithm has essentially the same ratio bound.

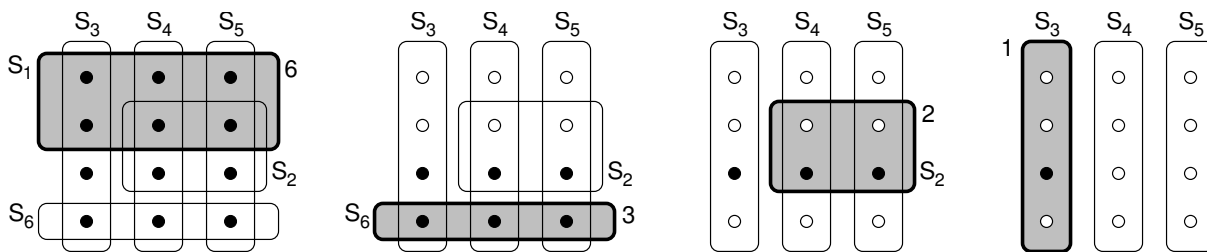


Fig. 62: Example of the greedy algorithm. Final set cover is  $\{S_1, S_6, S_2, S_3\}$ .

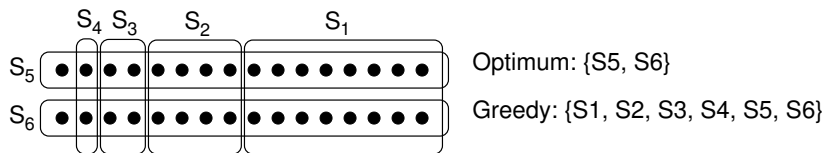


Fig. 63: An example in which the Greedy Set cover performs poorly.

However we will show that the greedy set cover heuristic never performs worse than a factor of  $\ln m$ . (Note that this is natural log, not base 2.)

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

**Lemma:** For all  $c > 0$ ,

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

where  $e$  is the base of the natural logarithm.

**Proof:** We use the fact that for any real  $x$  (positive, zero, or negative),  $1 + x \leq e^x$ . (This follows from the Taylor's expansion  $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \geq 1 + x$ .) Now, if we substitute  $-1/c$  for  $x$  we have  $(1 - 1/c) \leq e^{-1/c}$ . By raising both sides to the  $c$ th power, we have the desired result.

We now prove the approximation bound.

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

**Proof:** We will cheat a bit. Let  $c$  denote the size of the optimum set cover, and let  $g$  denote the size of the greedy set cover minus 1. We will show that  $g/c \leq \ln m$ . (Note that we should really show that  $(g+1)/c \leq \ln m$ . See the book for the correct proof.)

Let's consider how many new elements we cover with each round of the algorithm. Initially, there are  $m_0 = m$  elements to be covered. After the  $i$ th round, let  $m_i$  denote the number of elements remaining to be covered. Since we know that there is a cover of size  $c$  (the optimal cover), by the pigeonhole principal there exists some set that covers at least  $m_0/c$  elements. (If every set covered fewer than  $m_0/c$  elements, then no collection of  $c$  sets could cover all  $m_0$  elements.) Since the greedy algorithm selects the set covering the largest number of remaining elements, it must select a set that covers at least this many elements. The number of elements that remain to be covered is at most

$$m_0 - \frac{m_0}{c} = m_0 \left(1 - \frac{1}{c}\right) = m \left(1 - \frac{1}{c}\right).$$

That is,  $m_1 \leq m(1 - \frac{1}{c})$ .

Let's consider the second round. Again, we know that we can cover the remaining  $m_1$  elements with a cover of size  $c$  (the optimal one), and hence there exists a subset that covers at least  $m_1 \frac{1}{c}$  elements, leaving at most  $m_1(1 - \frac{1}{c}) \leq m(1 - \frac{1}{c})^2$  elements. Thus,  $m_2 \leq m(1 - \frac{1}{c})^2$ .

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

How long can this go on? Since the algorithm ran for  $g + 1$  iterations, we know that just prior to the last iteration we must have had at least one remaining uncovered element, and so we have

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

By the above lemma we have

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

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

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

This completes the proof.

There is anecdotal evidence that, even though the greedy set cover has this relatively bad ratio bound, it tends to perform much better in practice. Thus, the example shown above in which the approximation bound is  $\Omega(\log m)$  is not typical of set cover instances.

**Bin Packing:** Bin packing is another well-known NP-complete problem. This is a partitioning problem where we are given a set of objects that are to be partitioned among a collection of containers, called *bins*. Each bin has the same capacity, and the objective is to use the smallest number of bins to hold all the objects.

More formally, we are given a set of  $n$  objects, where  $s_i$  denotes the *size* of the  $i$ th object. It will simplify the presentation to assume that the sizes have been normalized so that  $0 < s_i < 1$ . We want to put these objects into a set of bins. Each bin can hold a subset of objects whose total size is at most 1. The problem is to partition the objects among the bins so as to use the fewest possible bins. (Note that if your bin size is not 1, then you can reduce the problem into this form by simply dividing all sizes by the size of the bin.)

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

Here is a simple heuristic algorithm for the bin packing problem, called the *first-fit heuristic*. We start with an unlimited number of empty bins. We take each object in turn, and find the first bin that has space to hold this object. We put this object in this bin. The algorithm is illustrated in Fig. 64. We claim that first-fit uses at most twice as many bins as the optimum. That is, if the optimal solution uses  $b_{\text{opt}}$  bins, and first-fit uses  $b_{\text{ff}}$  bins, then we show below that

$$\frac{b_{\text{ff}}}{b_{\text{opt}}} \leq 2.$$

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

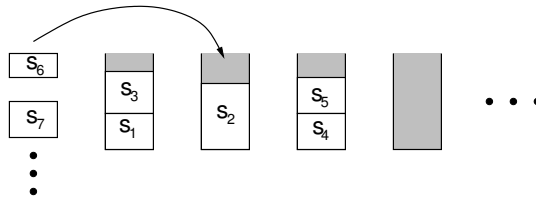


Fig. 64: First-fit Heuristic.

**Proof:** Consider an instance  $\{s_1, \dots, s_n\}$  of the bin packing problem. Let  $S = \sum_i s_i$  denote the sum of all the object sizes. Let  $b_{\text{opt}}$  denote the optimal number of bins, and  $b_{\text{ff}}$  denote the number of bins used by first-fit. First, observe that since no bin can hold more than one unit's worth of items, and we have a total of  $S$  units to be stored, it follows that we need a minimum of  $S$  bins to store everything. (And this would be achieved only if every bin were filled exactly to the top.) Thus,  $b_{\text{opt}} \geq S$ .

Next, we claim that  $b_{\text{ff}} \leq 2S$ . To see this, let  $t_i$  denote the total size of the objects that first-fit puts into bin  $i$ . There cannot be two bins  $i < j$  such that  $t_i + t_j < 1$ . The reason is that any item we decided to put into bin  $j$  must be small enough to fit into bin  $i$ . Thus, the first-fit algorithm would never put such an item into bin  $j$ . In particular, this implies that for all  $i$ ,  $t_i + t_{i+1} \geq 1$  (where indices are taken circularly modulo the number of bins). Thus we have

$$b_{\text{ff}} = \sum_{i=1}^{b_{\text{ff}}} 1 \leq \sum_{i=1}^{b_{\text{ff}}} (t_i + t_{i+1}) = \sum_{i=1}^{b_{\text{ff}}} t_i + \sum_{i=1}^{b_{\text{ff}}} t_{i+1} = S + S = 2S \leq 2b_{\text{opt}},$$

which completes the proof.

There are in fact a number of other heuristics for bin packing. Another example is *best-fit*, which attempts to put the object into the bin in which it fits most closely with the available space (assuming that there is sufficient available space). This is not necessarily a good idea, since it might tend to create very small spaces that will be hard to fill. There is also a variant of first-fit, called *first-fit-decreasing*, in which the objects are first sorted in decreasing order of size. (This makes intuitive sense, because it is best to first load the big items, and then try to squeeze the smaller objects into the remaining space.)

A more careful (an complicated) proof establishes that first-fit has a approximation ratio that is a bit smaller than 2, and in fact  $17/10 = 1.7$  is possible. Best-fit has a very similar bound. It can be shown that first-fit-decreasing has a significantly better bound than either of these. In particular, it achieves a ratio bound of  $11/9 \approx 1.222$ .

## Approximation Algorithms: The $k$ -Center Problem

**Read:** A variant of this problem is discussed in Chapt 11 in KT. The topic is not covered in CLRS.

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

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

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

Consider a subset  $C \subseteq V$  of vertices, the *centers*. For each vertex  $v \in V$  we can associate it with its nearest center in  $C$ . (This is the nearest Blockbuster store to your house). For each center  $c_i \in C$  we define its *neighborhood* to be the subset of vertices for which  $c_i$  is the closest center. (These are the houses that are closest to this center. See Fig. 65.) More formally, define:

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

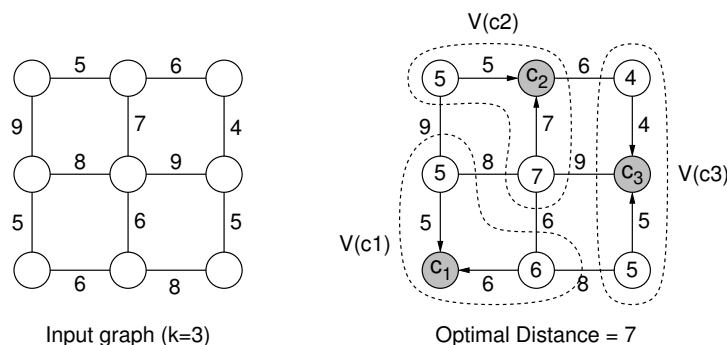


Fig. 65: The  $k$ -center problem with optimum centers  $c_i$  and neighborhood sets  $V(c_i)$ .

Let us assume for simplicity that there are no ties for the distances to the closest center (or that any such ties have been broken arbitrarily). Then  $V(c_1), V(c_2), \dots, V(c_k)$  forms a *partition* of the vertex set of  $G$ . The *bottleneck distance* associated with each center is the distance to its farthest vertex in  $V(c_i)$ , that is,

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

Finally, we define the overall *bottleneck distance* to be

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

This is the maximum distance of any vertex from its nearest center. This distance is critical because it represents the customer that must travel farthest to get to the nearest facility, the *bottleneck vertex*. Given this notation, we can now formally define the problem.

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

The decision-problem formulation of the  $k$ -center problem is NP-complete (reduction from dominating set). A brute force solution to this problem would involve enumerating all  $k$ -element of subsets of  $V$ , and computing  $\Delta(C)$  for each one. However, letting  $n = |V|$  and  $k$ , the number of possible subsets is  $\binom{n}{k} = \Theta(n^k)$ . If  $k$  is a function of  $n$  (which is reasonable), then this an exponential number of subsets. Given that the problem is NP-complete, it is highly unlikely that a significantly more efficient exact algorithm exists in the worst-case. We will show that there does exist an efficient approximation algorithm for the problem.

**Greedy Approximation Algorithm:** Our approximation algorithm is based on a simple greedy algorithm that produces a bottleneck distance  $\Delta(C)$  that is not more than twice the optimum bottleneck distance.

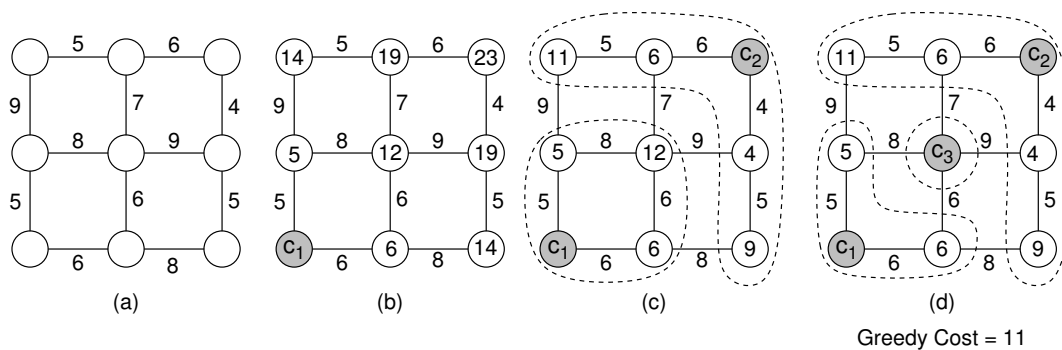


Fig. 66: Greedy approximation to  $k$ -center.

We begin by letting the first center  $c_1$  be *any* vertex in the graph (the lower left vertex, say, in Fig. 66(a)). Compute the distances between this vertex and all the other vertices in the graph (Fig. 66(b)). Consider the vertex that is farthest from this center (the upper right vertex at distance 23 in the figure). This is the bottleneck vertex for  $\{c_1\}$ . We would like to select the next center so as to reduce this distance. So let us just make it the next center, called  $c_2$ . Then again we compute the distances from each vertex in the graph to the *closer* of  $c_1$  and  $c_2$ . (See Fig. 66(c) where dashed lines indicate the neighborhoods of the centers.) Again we consider the bottleneck vertex for the current centers  $\{c_1, c_2\}$ . We place the next center at this vertex (see Fig. 66(d)). Again we compute the distances from each vertex to its nearest center. Repeat this until all  $k$  centers have been selected. In Fig. 66(d), the final three greedy centers are shaded, and the final bottleneck distance is 11.

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

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

Greedy Approximation for  $k$ -center

---

```

KCenterApprox(G, k) {
  C = empty_set
  for each u in V do // initialize distances
    d[u] = INFINITY
  for i = 1 to k do { // main loop
    Find the vertex u such that d[u] is maximum
    Add u to C // u is the current bottleneck vertex
    // update distances
    Compute the distance from each vertex v to its closest
    vertex in C, denoted d[v]
  }
  return C // final centers
}

```

---

We know from Dijkstra's algorithm how to compute the shortest path from a single source to all other vertices in the graph. One way to solve the distance computation step above would be to invoke Dijkstra's algorithm  $i$  times. But there is an easier way. We can modify Dijkstra's algorithm to operate as a *multiple source* algorithm. In particular, in the initialization of Dijkstra's single source algorithm, it sets  $d[s] = 0$  and  $pred[s] = \text{null}$ . In the modified multiple source version, we do this for *all* the vertices of  $C$ . The final greedy algorithm involves running Dijkstra's algorithm  $k$  times (once for each time through the for-loop). Recall that the running time of Dijkstra's algorithm is  $O((V + E) \log V)$ . Under the reasonable assumption that  $E \geq V$ , this is  $O(E \log V)$ .

Thus, the overall running time is  $O(kE \log V)$ .

**Approximation Bound:** How bad could greedy be? We will argue that it has a ratio bound of 2. To see that we can get a factor arbitrarily close to 2, consider a set of  $n$  vertices arranged in a linear graph for some large value of  $n$ , in which all edges are of weight 1, and suppose that  $k = 2$ . The greedy algorithm might pick any initial vertex that it likes. Suppose it picks the leftmost vertex. The next vertex it would pick is the farthest from this, that is the rightmost. (See Fig. 67.) The resulting bottleneck distance is roughly  $n/2$ . On the other hand, had picked two vertices at positions  $n/4$  and  $3n/4$  the bottleneck distance would be nearly  $n/4$ . Thus the ratio is roughly  $(n/2)/(n/4) = 2$ .

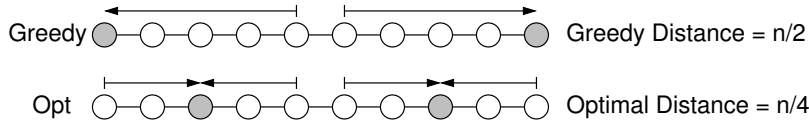


Fig. 67: An example showing that greedy can be a factor 2 from optimal. Here  $k = 2$ .

We want to show that this approximation algorithm always produces a final distance  $\Delta(C)$  that is within a factor of 2 of the distance of the optimal solution.

Let  $O = \{o_1, o_2, \dots, o_k\}$  denote the centers of the optimal solution (shown as black dots in Fig. 68, and the lines show the partition into the neighborhoods for each of these points). Let  $\Delta_* = \Delta(O)$  be the optimal bottleneck distance.

Let  $\mathcal{G} = \{g_1, g_2, \dots, g_k\}$  be the centers found by the greedy approximation (shown as white dots in the figure below). Also, let  $g_{k+1}$  denote the next center that *would have* been added next, that is, the bottleneck vertex for  $\mathcal{G}$ . Let  $\Delta_G$  denote the bottleneck distance for  $\mathcal{G}$ . Notice that the distance from  $g_{k+1}$  to its nearest center is equal  $\Delta_G$ . The proof involves a simple application of the pigeon-hole principal.

**Theorem:** The greedy approximation has a ratio bound of 2, that is  $\Delta_G/\Delta_* \leq 2$ .

**Proof:** Let  $\mathcal{G}' = \{g_1, g_2, \dots, g_k, g_{k+1}\}$  be the  $(k + 1)$ -element set consisting of the greedy centers together with the next greedy center  $g_{k+1}$ . First observe that for  $i \neq j$ ,  $d(g_i, g_j) \geq \Delta_G$ . This follows as a result of our greedy selection strategy. As each center is selected, it is selected to be at the maximum (bottleneck) distance from all the previous centers. As we add more centers, the maximum distance between any pair of centers decreases. Since the final bottleneck distance is  $\Delta_G$ , all the centers are at least this far apart from one another.

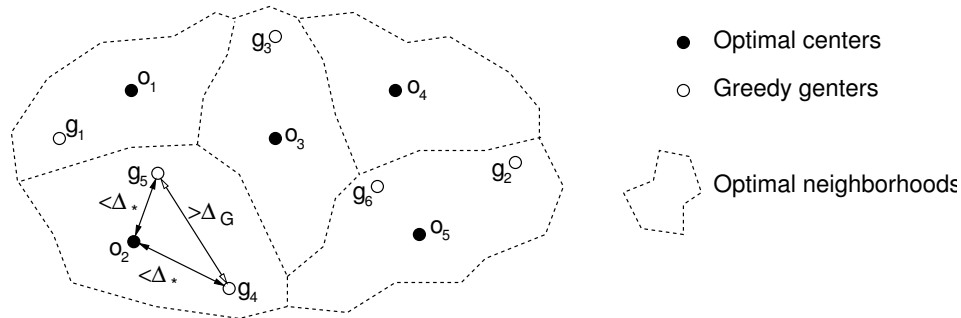


Fig. 68: Analysis of the greedy heuristic for  $k = 5$ .

Each  $g_i \in \mathcal{G}'$  is associated with its closest center in the optimal solution, that is, each belongs to  $V(o_m)$  for some  $m$ . Because there are  $k$  centers in  $O$ , and  $k + 1$  elements in  $\mathcal{G}'$ , it follows from the pigeon-hole

principal, that at least two centers of  $\mathcal{G}'$  are in the same set  $V(o_m)$  for some  $m$ . (In the figure, the greedy centers  $g_4$  and  $g_5$  are both in  $V(o_2)$ ). Let these be denoted  $g_i$  and  $g_j$ .

Since  $\Delta_*$  is the bottleneck distance for  $O$ , we know that the distance from  $g_i$  to  $o_k$  is of length at most  $\Delta_*$  and similarly the distance from  $o_k$  to  $g_j$  is at most  $\Delta_*$ . By concatenating these two paths and the triangle inequality, it follows that there exists a path of length at most  $2\Delta_*$  from  $g_i$  to  $g_j$ , and hence we have  $d(g_i, g_j) \leq 2\Delta_*$ . But from the comments above we have  $d(g_i, g_j) \geq \Delta_G$ . Therefore,

$$\Delta_G \leq d(g_i, g_j) \leq 2\Delta_*.$$

Therefore  $\Delta_G/\Delta_* \leq 2$ , as desired.

## Supplemental Lecture 1: Asymptotics

**Read:** Chapters 2–3 in CLRS.

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

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

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

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

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

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

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

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

**Asymptotic Notation:** To represent the running times of algorithms in a simpler form, we use *asymptotic notation*, which essentially represents a function by its fastest growing term and ignores constant factors. For example, suppose we have an algorithm whose (exact) worst-case running time is given by the following formula:

$$T(n) = 13n^3 + 5n^2 - 17n + 16.$$

As  $n$  becomes large, the  $13n^3$  term dominates the others. By ignoring constant factors, we might say that the running time grows “on the order of”  $n^3$ , which will express mathematically as  $T(n) \in \Theta(n^3)$ . This intuitive definition is fine for informal use. Let us consider how to make this idea mathematically formal.

**Definition:** Given any function  $g(n)$ , we define  $\Theta(g(n))$  to be a set of functions:

$$\Theta(g(n)) = \{f(n) \mid \text{there exist strictly positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}.$$

Let’s dissect this definition. Intuitively, what we want to say with “ $f(n) \in \Theta(g(n))$ ” is that  $f(n)$  and  $g(n)$  are *asymptotically equivalent*. This means that they have essentially the same growth rates for large  $n$ . For example, functions such as

$$4n^2, \quad (8n^2 + 2n - 3), \quad (n^2/5 + \sqrt{n} - 10 \log n), \quad \text{and} \quad n(n - 3)$$

are all intuitively asymptotically equivalent, since as  $n$  becomes large, the dominant (fastest growing) term is some constant times  $n^2$ . In other words, they all grow *quadratically* in  $n$ . The portion of the definition that allows us to select  $c_1$  and  $c_2$  is essentially saying “the constants do not matter because you may pick  $c_1$  and  $c_2$  however you like to satisfy these conditions.” The portion of the definition that allows us to select  $n_0$  is essentially saying “we are only interested in large  $n$ , since you only have to satisfy the condition for all  $n$  bigger than  $n_0$ , and you may make  $n_0$  as big a constant as you like.”

**An example:** Consider the function  $f(n) = 8n^2 + 2n - 3$ . Our informal rule of keeping the largest term and throwing away the constants suggests that  $f(n) \in \Theta(n^2)$  (since  $f$  grows quadratically). Let’s see why the formal definition bears out this informal observation.

We need to show two things: first, that  $f(n)$  does grow asymptotically at least as fast as  $n^2$ , and second, that  $f(n)$  grows no faster asymptotically than  $n^2$ . We’ll do both very carefully.

**Lower bound:**  $f(n)$  grows asymptotically at least as fast as  $n^2$ : This is established by the portion of the definition that reads: (paraphrasing): “there exist positive constants  $c_1$  and  $n_0$ , such that  $f(n) \geq c_1n^2$  for all  $n \geq n_0$ .” Consider the following (almost correct) reasoning:

$$f(n) = 8n^2 + 2n - 3 \geq 8n^2 - 3 = 7n^2 + (n^2 - 3) \geq 7n^2 = 7n^2.$$

Thus, if we set  $c_1 = 7$ , then we are done. But in the above reasoning we have implicitly made the assumptions that  $2n \geq 0$  and  $n^2 - 3 \geq 0$ . These are not true for all  $n$ , but they are true for all sufficiently large  $n$ . In particular, if  $n \geq \sqrt{3}$ , then both are true. So let us select  $n_0 = \sqrt{3}$ , and now we have  $f(n) \geq c_1n^2$ , for all  $n \geq n_0$ , which is what we need.

**Upper bound:**  $f(n)$  grows asymptotically no faster than  $n^2$ : This is established by the portion of the definition that reads “there exist positive constants  $c_2$  and  $n_0$ , such that  $f(n) \leq c_2n^2$  for all  $n \geq n_0$ .” Consider the following reasoning (which is almost correct):

$$f(n) = 8n^2 + 2n - 3 \leq 8n^2 + 2n \leq 8n^2 + 2n^2 = 10n^2.$$

This means that if we let  $c_2 = 10$ , then we are done. We have implicitly made the assumption that  $2n \leq 2n^2$ . This is not true for all  $n$ , but it is true for all  $n \geq 1$ . So, let us select  $n_0 = 1$ , and now we have  $f(n) \leq c_2n^2$  for all  $n \geq n_0$ , which is what we need.

From the lower bound, we have  $n_0 \geq \sqrt{3}$  and from the upper bound we have  $n_0 \geq 1$ , and so combining these we let  $n_0$  be the larger of the two:  $n_0 = \sqrt{3}$ . Thus, in conclusion, if we let  $c_1 = 7$ ,  $c_2 = 10$ , and  $n_0 = \sqrt{3}$ , then we have

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \text{for all } n \geq n_0,$$

and this is exactly what the definition requires. Since we have shown (by construction) the existence of constants  $c_1$ ,  $c_2$ , and  $n_0$ , we have established that  $f(n) \in n^2$ . (Whew! That was a lot more work than just the informal notion of throwing away constants and keeping the largest term, but it shows how this informal notion is implemented formally in the definition.)

Now let's show why  $f(n)$  is not in some other asymptotic class. First, let's show that  $f(n) \notin \Theta(n)$ . If this were true, then we would have to satisfy both the upper and lower bounds. It turns out that the lower bound is satisfied (because  $f(n)$  grows at least as fast asymptotically as  $n$ ). But the upper bound is false. In particular, the upper bound requires us to show "there exist positive constants  $c_2$  and  $n_0$ , such that  $f(n) \leq c_2 n$  for all  $n \geq n_0$ ." Informally, we know that as  $n$  becomes large enough  $f(n) = 8n^2 + 2n - 3$  will eventually exceed  $c_2 n$  no matter how large we make  $c_2$  (since  $f(n)$  is growing quadratically and  $c_2 n$  is only growing linearly). To show this formally, suppose towards a contradiction that constants  $c_2$  and  $n_0$  did exist, such that  $8n^2 + 2n - 3 \leq c_2 n$  for all  $n \geq n_0$ . Since this is true for all sufficiently large  $n$  then it must be true in the limit as  $n$  tends to infinity. If we divide both side by  $n$  we have:

$$\lim_{n \rightarrow \infty} \left( 8n + 2 - \frac{3}{n} \right) \leq c_2.$$

It is easy to see that in the limit the left side tends to  $\infty$ , and so no matter how large  $c_2$  is, this statement is violated. This means that  $f(n) \notin \Theta(n)$ .

Let's show that  $f(n) \notin \Theta(n^3)$ . Here the idea will be to violate the lower bound: "there exist positive constants  $c_1$  and  $n_0$ , such that  $f(n) \geq c_1 n^3$  for all  $n \geq n_0$ ." Informally this is true because  $f(n)$  is growing quadratically, and eventually any cubic function will exceed it. To show this formally, suppose towards a contradiction that constants  $c_1$  and  $n_0$  did exist, such that  $8n^2 + 2n - 3 \geq c_1 n^3$  for all  $n \geq n_0$ . Since this is true for all sufficiently large  $n$  then it must be true in the limit as  $n$  tends to infinity. If we divide both side by  $n^3$  we have:

$$\lim_{n \rightarrow \infty} \left( \frac{8}{n} + \frac{2}{n^2} - \frac{3}{n^3} \right) \geq c_1.$$

It is easy to see that in the limit the left side tends to 0, and so the only way to satisfy this requirement is to set  $c_1 = 0$ , but by hypothesis  $c_1$  is positive. This means that  $f(n) \notin \Theta(n^3)$ .

***O*-notation and  $\Omega$ -notation:** We have seen that the definition of  $\Theta$ -notation relies on proving both a lower and upper asymptotic bound. Sometimes we are only interested in proving one bound or the other. The *O*-notation allows us to state asymptotic upper bounds and the  $\Omega$ -notation allows us to state asymptotic lower bounds.

**Definition:** Given any function  $g(n)$ ,

$$O(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

**Definition:** Given any function  $g(n)$ ,

$$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$$

Compare this with the definition of  $\Theta$ . You will see that *O*-notation only enforces the upper bound of the  $\Theta$  definition, and  $\Omega$ -notation only enforces the lower bound. Also observe that  $f(n) \in \Theta(g(n))$  if and only if  $f(n) \in O(g(n))$  and  $f(n) \in \Omega(g(n))$ . Intuitively,  $f(n) \in O(g(n))$  means that  $f(n)$  grows asymptotically at

the same rate or slower than  $g(n)$ . Whereas,  $f(n) \in O(g(n))$  means that  $f(n)$  grows asymptotically at the same rate or faster than  $g(n)$ .

For example  $f(n) = 3n^2 + 4n \in \Theta(n^2)$  but it is not in  $\Theta(n)$  or  $\Theta(n^3)$ . But  $f(n) \in O(n^2)$  and in  $O(n^3)$  but not in  $O(n)$ . Finally,  $f(n) \in \Omega(n^2)$  and in  $\Omega(n)$  but not in  $\Omega(n^3)$ .

**The Limit Rule for  $\Theta$ :** The previous examples which used limits suggest alternative way of showing that  $f(n) \in \Theta(g(n))$ .

**Limit Rule for  $\Theta$ -notation:** Given positive functions  $f(n)$  and  $g(n)$ , if

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

for some constant  $c > 0$  (strictly positive but not infinity), then  $f(n) \in \Theta(g(n))$ .

**Limit Rule for  $O$ -notation:** Given positive functions  $f(n)$  and  $g(n)$ , if

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

for some constant  $c \geq 0$  (nonnegative but not infinite), then  $f(n) \in O(g(n))$ .

**Limit Rule for  $\Omega$ -notation:** Given positive functions  $f(n)$  and  $g(n)$ , if

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq 0$$

(either a strictly positive constant or infinity) then  $f(n) \in \Omega(g(n))$ .

This limit rule can be applied in almost every instance (that I know of) where the formal definition can be used, and it is almost always easier to apply than the formal definition. The only exceptions that I know of are strange instances where the limit does not exist (e.g.  $f(n) = n^{(1+\sin n)}$ ). But since most running times are fairly well-behaved functions this is rarely a problem.

For example, recall the function  $f(n) = 8n^2 + 2n - 3$ . To show that  $f(n) \in \Theta(n^2)$  we let  $g(n) = n^2$  and compute the limit. We have

$$\lim_{n \rightarrow \infty} \frac{8n^2 + 2n - 3}{n^2} = \lim_{n \rightarrow \infty} 8 + \frac{2}{n} - \frac{3}{n^2} = 8,$$

(since the two fractional terms tend to 0 in the limit). Since 8 is a nonzero constant, it follows that  $f(n) \in \Theta(g(n))$ .

You may recall the important rules from calculus for evaluating limits. (If not, dredge out your calculus book to remember.) Most of the rules are pretty self evident (e.g., the limit of a finite sum is the sum of the individual limits). One important rule to remember is the following:

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

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

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

**Exponentials and Logarithms:** Exponentials and logarithms are very important in analyzing algorithms. The following are nice to keep in mind. The terminology  $\lg^b n$  means  $(\lg n)^b$ .

**Lemma:** Given any positive constants  $a > 1$ ,  $b$ , and  $c$ :

$$\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0 \qquad \lim_{n \rightarrow \infty} \frac{\lg^b n}{n^c} = 0.$$

We won't prove these, but they can be shown by taking appropriate powers, and then applying L'Hôpital's rule. The important bottom line is that polynomials always grow more slowly than exponentials whose base is greater than 1. For example:

$$n^{500} \in O(2^n).$$

For this reason, we will try to avoid exponential running times at all costs. Conversely, logarithmic powers (sometimes called *polylogarithmic functions*) grow more slowly than any polynomial. For example:

$$\lg^{500} n \in O(n).$$

For this reason, we will usually be happy to allow any number of additional logarithmic factors, if it means avoiding any additional powers of  $n$ .

At this point, it should be mentioned that these last observations are really asymptotic results. They are true in the limit for large  $n$ , but you should be careful just how high the crossover point is. For example, by my calculations,  $\lg^{500} n \leq n$  only for  $n > 2^{6000}$  (which is much larger than input size you'll ever see). Thus, you should take this with a grain of salt. But, for small powers of logarithms, this applies to all reasonably large input sizes. For example  $\lg^2 n \leq n$  for all  $n \geq 16$ .

**Asymptotic Intuition:** To get a intuitive feeling for what common asymptotic running times map into in terms of practical usage, here is a little list.

- $\Theta(1)$ : Constant time; you can't beat it!
- $\Theta(\log n)$ : This is typically the speed that most efficient data structures operate in for a single access. (E.g., inserting a key into a balanced binary tree.) Also it is the time to find an object in a sorted list of length  $n$  by binary search.
- $\Theta(n)$ : This is about the fastest that an algorithm can run, given that you need  $\Theta(n)$  time just to read in all the data.
- $\Theta(n \log n)$ : This is the running time of the best sorting algorithms. Since many problems require sorting the inputs, this is still considered quite efficient.
- $\Theta(n^2), \Theta(n^3), \dots$ : Polynomial time. These running times are acceptable either when the exponent is small or when the data size is not too large (e.g.  $n \leq 1,000$ ).
- $\Theta(2^n), \Theta(3^n)$ : Exponential time. This is only acceptable when either (1) your know that you inputs will be of very small size (e.g.  $n \leq 50$ ), or (2) you know that this is a worst-case running time that will rarely occur in practical instances. In case (2), it would be a good idea to try to get a more accurate average case analysis.
- $\Theta(n!), \Theta(n^n)$ : Acceptable only for really small inputs (e.g.  $n \leq 20$ ).

Are there even bigger functions? Definitely! For example, if you want to see a function that grows inconceivably fast, look up the definition of *Ackerman's function* in our text.

**Max Dominance Revisited:** Returning to our Max Dominance algorithms, recall that one had a running time of  $T_1(n) = n^2$  and the other had a running time of  $T_2(n) = n \log n + n(n-1)/2$ . Expanding the latter function and grouping terms in order of their growth rate we have

$$T_2(n) = \frac{n^2}{2} + n \log n - \frac{n}{2}.$$

We will leave it as an easy exercise to show that both  $T_1(n)$  and  $T_2(n)$  are  $\Theta(n^2)$ . Although the second algorithm is twice as fast for large  $n$  (because of the  $1/2$  factor multiplying the  $n^2$  term), this does not represent a significant improvement.

## Supplemental Lecture 2: Max Dominance

**Read:** Review Chapters 1–4 in CLRS.

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

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

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

---

Max Dominance: Sort and Reverse Scan

```
MaxDom3(P, n) {
    Sort P in ascending order by x-coordinate;
    output P[n]; // last point is always maximal
    j = n;
    for i = n-1 downto 1 {
        if (P[i].y >= P[j].y) { // is P[i] maximal?
            output P[i]; // yes..output it
            j = i; // P[i] has the largest y so far
        }
    }
}
```

---

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

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

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

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

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

10. For even larger inputs, say,  $n = 1,000,000$ , we are looking at a ratio of roughly  $1,000,000/20 = 50,000$ . This is quite a significant difference, irrespective of the constant factors.

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

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

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

**Conquer:** Solve each subproblem recursively, and

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

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

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

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

---

Max Dominance: Divide-and-Conquer

```
MaxDom4(P, n) {
    if (n == 1) return {P[1]};           // one point is trivially maximal
    m = n/2;                             // midpoint of list
    M1 = MaxDom4(P[1..m], m);           // solve for first half
    M2 = MaxDom4(P[m+1..n], n-m);      // solve for second half
    return MaxMerge(M1, M2);           // merge the results
}
```

---

The general process is illustrated below.

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

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

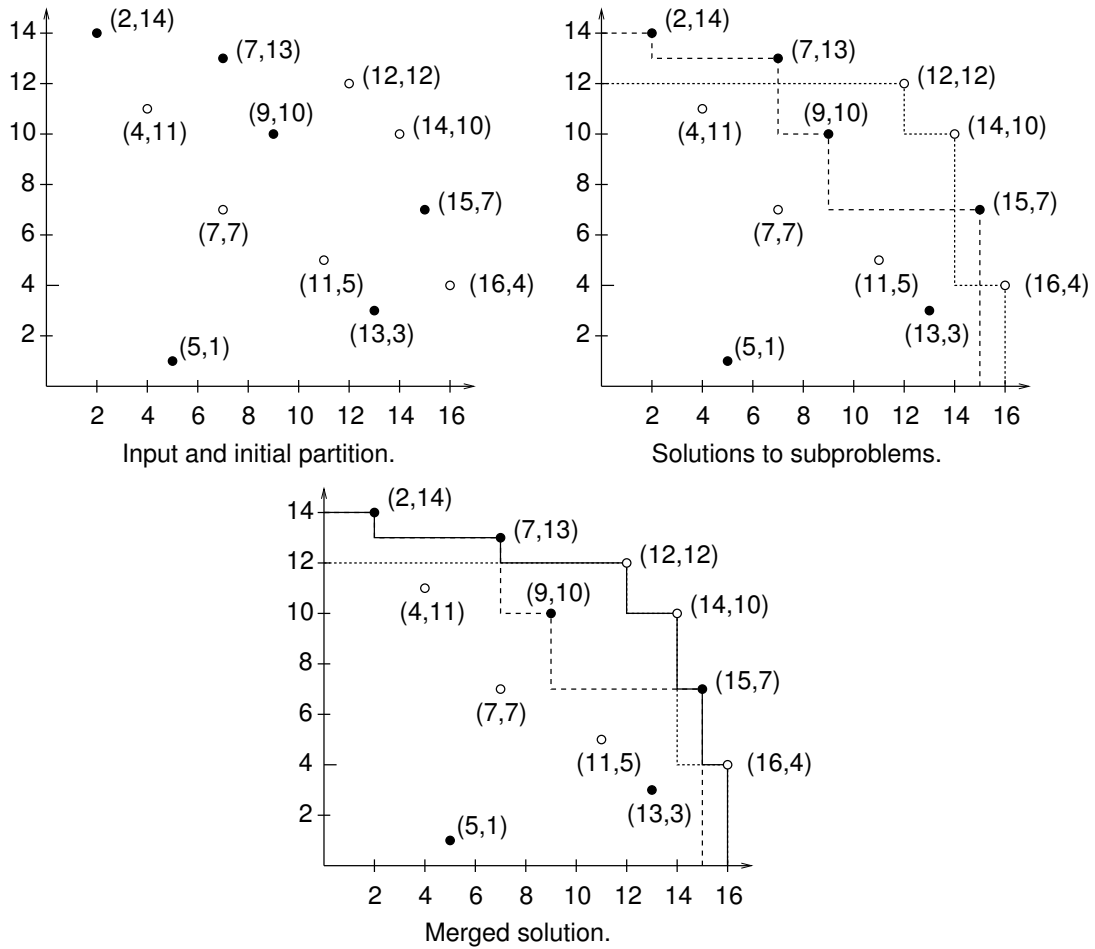


Fig. 69: Divide and conquer approach.

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

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

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

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

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

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

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

defined for  $n \geq 0$ .

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

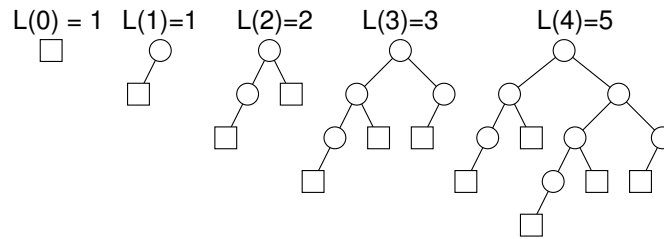


Fig. 70: Minimum-sized AVL trees.

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

**Proof:** We will try to derive the tightest bound we can on the value of  $\phi$ .

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

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

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

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

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

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

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

By the quadratic formula we have

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

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

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

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

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

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

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

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

## Supplemental Lecture 3: Recurrences and Generating Functions

**Read:** This material is not covered in CLR. There a good description of generating functions in D. E. Knuth, *The Art of Computer Programming, Vol 1*.

**Generating Functions:** The method of constructive induction provided a way to get a bound on  $F_n$ , but we did not get an exact answer, and we had to generate a good guess before we were even able to start.

Let us consider an approach to determine an exact representation of  $F_n$ , which requires no guesswork. This method is based on a very elegant concept, called a *generating function*. Consider any infinite sequence:

$$a_0, a_1, a_2, a_3, \dots$$

If we would like to “encode” this sequence succinctly, we could define a polynomial function such that these are the coefficients of the function:

$$G(z) = a_0 + a_1z + a_2z^2 + a_3z^3 + \dots$$

This is called the *generating function* of the sequence. What is  $z$ ? It is just a symbolic variable. We will (almost) never assign it a specific value. Thus, every infinite sequence of numbers has a corresponding generating function, and vice versa. What is the advantage of this representation? It turns out that we can perform arithmetic transformations on these functions (e.g., adding them, multiplying them, differentiating them) and this has a corresponding effect on the underlying transformations. It turns out that some nicely-structured sequences (like the Fibonacci numbers, and many sequences arising from linear recurrences) have generating functions that are easy to write down and manipulate.

Let’s consider the generating function for the Fibonacci numbers:

$$\begin{aligned} G(z) &= F_0 + F_1z + F_2z^2 + F_3z^3 + \dots \\ &= z + z^2 + 2z^3 + 3z^4 + 5z^5 + \dots \end{aligned}$$

The trick in dealing with generating functions is to figure out how various manipulations of the generating function to generate algebraically equivalent forms. For example, notice that if we multiply the generating function by a factor of  $z$ , this has the effect of shifting the sequence to the right:

$$\begin{aligned} G(z) &= F_0 + F_1z + F_2z^2 + F_3z^3 + F_4z^4 + \dots \\ zG(z) &= F_0z + F_1z^2 + F_2z^3 + F_3z^4 + \dots \\ z^2G(z) &= F_0z^2 + F_1z^3 + F_2z^4 + \dots \end{aligned}$$

Now, let’s try the following manipulation. Compute  $G(z) - zG(z) - z^2G(z)$ , and see what we get

$$\begin{aligned} (1 - z - z^2)G(z) &= F_0 + (F_1 - F_0)z + (F_2 - F_1 - F_0)z^2 + (F_3 - F_2 - F_1)z^3 \\ &\quad + \dots + (F_i - F_{i-1} - F_{i-2})z^i + \dots \\ &= z. \end{aligned}$$

Observe that every term except the second is equal to zero by the definition of  $F_i$ . (The particular manipulation we picked was chosen to cause this cancellation to occur.) From this we may conclude that

$$G(z) = \frac{z}{1 - z - z^2}.$$

So, now we have an alternative representation for the Fibonacci numbers, as the coefficients of this function if expanded as a power series. So what good is this? The main goal is to get at the coefficients of its power series expansion. There are certain common tricks that people use to manipulate generating functions.

The first is to observe that there are some functions for which it is very easy to get an power series expansion. For example, the following is a simple consequence of the formula for the geometric series. If  $0 < c < 1$  then

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1-c}.$$

Setting  $z = c$ , we have

$$\frac{1}{1-z} = 1 + z + z^2 + z^3 + \dots$$

(In other words,  $1/(1-z)$  is the generating function for the sequence  $(1, 1, 1, \dots)$ ). In general, given an constant  $a$  we have

$$\frac{1}{1-az} = 1 + az + a^2z^2 + a^3z^3 + \dots$$

is the generating function for  $(1, a, a^2, a^3, \dots)$ . It would be great if we could modify our generating function to be in the form of  $1/(1-az)$  for some constant  $a$ , since then we could then extract the coefficients of the power series easily.

In order to do this, we would like to rewrite the generating function in the following form:

$$G(z) = \frac{z}{1-z-z^2} = \frac{A}{1-az} + \frac{B}{1-bz},$$

for some  $A, B, a, b$ . We will skip the steps in doing this, but it is not hard to verify the roots of  $(1-az)(1-bz)$  (which are  $1/a$  and  $1/b$ ) must be equal to the roots of  $1-z-z^2$ . We can then solve for  $a$  and  $b$  by taking the reciprocals of the roots of this quadratic. Then by some simple algebra we can plug these values in and solve for  $A$  and  $B$  yielding:

$$G(z) = \frac{z}{1-z-z^2} = \left( \frac{1/\sqrt{5}}{1-\phi z} + \frac{-1/\sqrt{5}}{1-\hat{\phi} z} \right) = \frac{1}{\sqrt{5}} \left( \frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z} \right),$$

where  $\phi = (1 + \sqrt{5})/2$  and  $\hat{\phi} = (1 - \sqrt{5})/2$ . (In particular, to determine  $A$ , multiply the equation by  $1 - \phi z$ , and then consider what happens when  $z = 1/\phi$ . A similar trick can be applied to get  $B$ . In general, this is called the method of *partial fractions*.)

Now we are in good shape, because we can extract the coefficients for these two fractions from the above function. From this we have the following:

$$G(z) = \frac{1}{\sqrt{5}} \left( \begin{array}{cccc} 1 & + & \phi z & + & \phi^2 z^2 & + & \dots \\ -1 & + & -\hat{\phi} z & + & -\hat{\phi}^2 z^2 & + & \dots \end{array} \right)$$

Combining terms we have

$$G(z) = \frac{1}{\sqrt{5}} \sum_{i=0}^{\infty} (\phi^i - \hat{\phi}^i) z^i.$$

We can now read off the coefficients easily. In particular it follows that

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n).$$

This is an exact result, and no guesswork was needed. The only parts that involved some cleverness (beyond the invention of generating functions) was (1) coming up with the simple closed form formula for  $G(z)$  by taking appropriate differences and applying the rule for the recurrence, and (2) applying the method of partial fractions to get the generating function into one for which we could easily read off the final coefficients.

This is a rather remarkable, because it says that we can express the integer  $F_n$  as the sum of two powers of to irrational numbers  $\phi$  and  $\hat{\phi}$ . You might try this for a few specific values of  $n$  to see why this is true. By the way, when you observe that  $\hat{\phi} < 1$ , it is clear that the first term is the dominant one. Thus we have, for large enough  $n$ ,  $F_n = \phi^n / \sqrt{5}$ , rounded to the nearest integer.

## Supplemental Lecture 4: Medians and Selection

**Read:** Chapter 9 of CLRS.

**Selection:** We have discussed recurrences and the divide-and-conquer method of solving problems. Today we will give a rather surprising (and very tricky) algorithm which shows the power of these techniques.

The problem that we will consider is very easy to state, but surprisingly difficult to solve optimally. Suppose that you are given a set of  $n$  numbers. Define the *rank* of an element to be one plus the number of elements that are smaller than this element. Since duplicate elements make our life more complex (by creating multiple elements of the same rank), we will make the simplifying assumption that all the elements are distinct for now. It will be easy to get around this assumption later. Thus, the rank of an element is its final position if the set is sorted. The minimum is of rank 1 and the maximum is of rank  $n$ .

Of particular interest in statistics is the *median*. If  $n$  is odd then the median is defined to be the element of rank  $(n + 1)/2$ . When  $n$  is even there are two natural choices, namely the elements of ranks  $n/2$  and  $(n/2) + 1$ . In statistics it is common to return the average of these two elements. We will define the median to be either of these elements.

Medians are useful as measures of the *central tendency* of a set, especially when the distribution of values is highly skewed. For example, the median income in a community is likely to be more meaningful measure of the central tendency than the average is, since if Bill Gates lives in your community then his gigantic income may significantly bias the average, whereas it cannot have a significant influence on the median. They are also useful, since in divide-and-conquer applications, it is often desirable to partition a set about its median value, into two sets of roughly equal size. Today we will focus on the following generalization, called the *selection problem*.

**Selection:** Given a set  $A$  of  $n$  distinct numbers and an integer  $k$ ,  $1 \leq k \leq n$ , output the element of  $A$  of rank  $k$ .

The selection problem can easily be solved in  $\Theta(n \log n)$  time, simply by sorting the numbers of  $A$ , and then returning  $A[k]$ . The question is whether it is possible to do better. In particular, is it possible to solve this problem in  $\Theta(n)$  time? We will see that the answer is yes, and the solution is far from obvious.

**The Sieve Technique:** The reason for introducing this algorithm is that it illustrates a very important special case of divide-and-conquer, which I call the *sieve technique*. We think of divide-and-conquer as breaking the problem into a small number of smaller subproblems, which are then solved recursively. The sieve technique is a special case, where the number of subproblems is just 1.

The sieve technique works in phases as follows. It applies to problems where we are interested in finding a single item from a larger set of  $n$  items. We do not know which item is of interest, however after doing some amount of analysis of the data, taking say  $\Theta(n^k)$  time, for some constant  $k$ , we find that we do not know what the desired item is, but we can identify a large enough number of elements that *cannot* be the desired value, and can be eliminated from further consideration. In particular “large enough” means that the number of items is at least some fixed constant fraction of  $n$  (e.g.  $n/2$ ,  $n/3$ ,  $0.0001n$ ). Then we solve the problem recursively on whatever items remain. Each of the resulting recursive solutions then do the same thing, eliminating a constant fraction of the remaining set.

**Applying the Sieve to Selection:** To see more concretely how the sieve technique works, let us apply it to the selection problem. Recall that we are given an array  $A[1..n]$  and an integer  $k$ , and want to find the  $k$ -th smallest element of  $A$ . Since the algorithm will be applied inductively, we will assume that we are given a subarray  $A[p..r]$  as we did in MergeSort, and we want to find the  $k$ th smallest item (where  $k \leq r - p + 1$ ). The initial call will be to the entire array  $A[1..n]$ .

There are two principal algorithms for solving the selection problem, but they differ only in one step, which involves judiciously choosing an item from the array, called the *pivot element*, which we will denote by  $x$ . Later

we will see how to choose  $x$ , but for now just think of it as a random element of  $A$ . We then partition  $A$  into three parts.  $A[q]$  contains the element  $x$ , subarray  $A[p..q-1]$  will contain all the elements that are less than  $x$ , and  $A[q+1..r]$  will contain all the elements that are greater than  $x$ . (Recall that we assumed that all the elements are distinct.) Within each subarray, the items may appear in any order. This is illustrated below.

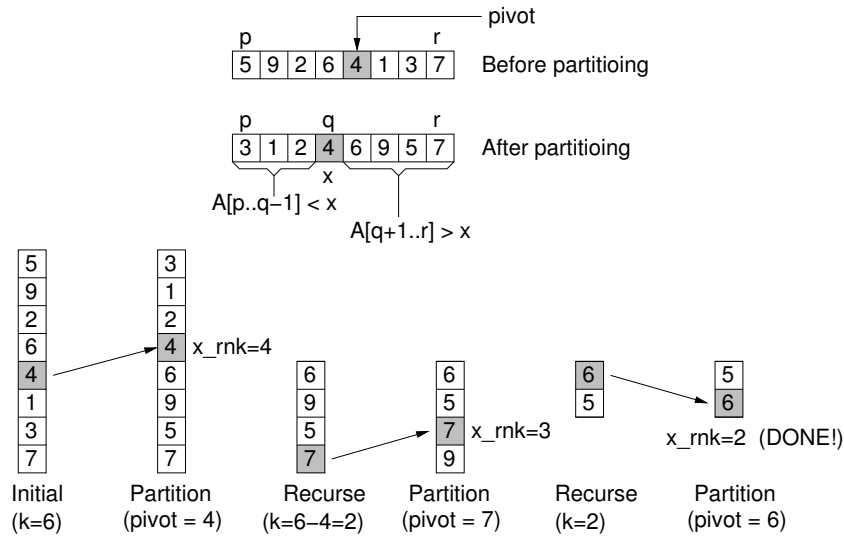


Fig. 71: Selection Algorithm.

It is easy to see that the rank of the pivot  $x$  is  $q - p + 1$  in  $A[p..r]$ . Let  $xRank = q - p + 1$ . If  $k = xRank$ , then the pivot is the  $k$ th smallest, and we may just return it. If  $k < xRank$ , then we know that we need to recursively search in  $A[p..q-1]$  and if  $k > xRank$  then we need to recursively search  $A[q+1..r]$ . In this latter case we have eliminated  $q$  smaller elements, so we want to find the element of rank  $k - q$ . Here is the complete pseudocode.

---

```

                                                                    Selection by the Sieve Technique
Select(array A, int p, int r, int k) {                               // return kth smallest of A[p..r]
    if (p == r) return A[p]                                         // only 1 item left, return it
    else {
        x = ChoosePivot(A, p, r)                                    // choose the pivot element
        q = Partition(A, p, r, x)                                  // partition <A[p..q-1], x, A[q+1..r]>
        xRank = q - p + 1                                          // rank of the pivot
        if (k == xRank) return x                                   // the pivot is the kth smallest
        else if (k < xRank)
            return Select(A, p, q-1, k)                            // select from left subarray
        else
            return Select(A, q+1, r, k-xRank)                      // select from right subarray
    }
}

```

---

Notice that this algorithm satisfies the basic form of a sieve algorithm. It analyzes the data (by choosing the pivot element and partitioning) and it eliminates some part of the data set, and recurses on the rest. When  $k = xRank$  then we get lucky and eliminate everything. Otherwise we either eliminate the pivot and the right subarray or the pivot and the left subarray.

We will discuss the details of choosing the pivot and partitioning later, but assume for now that they both take  $\Theta(n)$  time. The question that remains is how many elements did we succeed in eliminating? If  $x$  is the largest or smallest element in the array, then we may only succeed in eliminating one element with each phase. In fact,

if  $x$  is one of the smallest elements of  $A$  or one of the largest, then we get into trouble, because we may only eliminate it and the few smaller or larger elements of  $A$ . Ideally  $x$  should have a rank that is neither too large nor too small.

Let us suppose for now (optimistically) that we are able to design the procedure `Choose_Pivot` in such a way that it eliminates exactly half the array with each phase, meaning that we recurse on the remaining  $n/2$  elements. This would lead to the following recurrence.

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

We can solve this either by expansion (iteration) or the Master Theorem. If we expand this recurrence level by level we see that we get the summation

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \cdots \leq \sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i}.$$

Recall the formula for the infinite geometric series. For any  $c$  such that  $|c| < 1$ ,  $\sum_{i=0}^{\infty} c^i = 1/(1 - c)$ . Using this we have

$$T(n) \leq 2n \in O(n).$$

(This only proves the upper bound on the running time, but it is easy to see that it takes at least  $\Omega(n)$  time, so the total running time is  $\Theta(n)$ .)

This is a bit counterintuitive. Normally you would think that in order to design a  $\Theta(n)$  time algorithm you could only make a single, or perhaps a constant number of passes over the data set. In this algorithm we make many passes (it could be as many as  $\lg n$ ). However, because we eliminate a constant fraction of elements with each phase, we get this convergent geometric series in the analysis, which shows that the total running time is indeed linear in  $n$ . This lesson is well worth remembering. It is often possible to achieve running times in ways that you would not expect.

Note that the assumption of eliminating half was not critical. If we eliminated even one per cent, then the recurrence would have been  $T(n) = T(99n/100) + n$ , and we would have gotten a geometric series involving  $99/100$ , which is still less than 1, implying a convergent series. Eliminating *any* constant fraction would have been good enough.

**Choosing the Pivot:** There are two issues that we have left unresolved. The first is how to choose the pivot element, and the second is how to partition the array. Both need to be solved in  $\Theta(n)$  time. The second problem is a rather easy programming exercise. Later, when we discuss QuickSort, we will discuss partitioning in detail.

For the rest of the lecture, let's concentrate on how to choose the pivot. Recall that before we said that we might think of the pivot as a random element of  $A$ . Actually this is not such a bad idea. Let's see why.

The key is that we want the procedure to eliminate at least some constant fraction of the array after each partitioning step. Let's consider the top of the recurrence, when we are given  $A[1..n]$ . Suppose that the pivot  $x$  turns out to be of rank  $q$  in the array. The partitioning algorithm will split the array into  $A[1..q-1] < x$ ,  $A[q] = x$  and  $A[q+1..n] > x$ . If  $k = q$ , then we are done. Otherwise, we need to search one of the two subarrays. They are of sizes  $q-1$  and  $n-q$ , respectively. The subarray that contains the  $k$ th smallest element will generally depend on what  $k$  is, so in the worst case,  $k$  will be chosen so that we have to recurse on the larger of the two subarrays. Thus if  $q > n/2$ , then we may have to recurse on the left subarray of size  $q-1$ , and if  $q < n/2$ , then we may have to recurse on the right subarray of size  $n-q$ . In either case, we are in trouble if  $q$  is very small, or if  $q$  is very large.

If we could select  $q$  so that it is roughly of middle rank, then we will be in good shape. For example, if  $n/4 \leq q \leq 3n/4$ , then the larger subarray will never be larger than  $3n/4$ . Earlier we said that we might think of the pivot as a random element of the array  $A$ . Actually this works pretty well in practice. The reason is that roughly half of the elements lie between ranks  $n/4$  and  $3n/4$ , so picking a random element as the pivot will

succeed about half the time to eliminate at least  $n/4$ . Of course, we might be continuously unlucky, but a careful analysis will show that the expected running time is still  $\Theta(n)$ . We will return to this later.

Instead, we will describe a rather complicated method for computing a pivot element that achieves the desired properties. Recall that we are given an array  $A[1..n]$ , and we want to compute an element  $x$  whose rank is (roughly) between  $n/4$  and  $3n/4$ . We will have to describe this algorithm at a very high level, since the details are rather involved. Here is the description for `Select_Pivot`:

**Groups of 5:** Partition  $A$  into groups of 5 elements, e.g.  $A[1..5]$ ,  $A[6..10]$ ,  $A[11..15]$ , etc. There will be exactly  $m = \lceil n/5 \rceil$  such groups (the last one might have fewer than 5 elements). This can easily be done in  $\Theta(n)$  time.

**Group medians:** Compute the median of each group of 5. There will be  $m$  group medians. We do not need an intelligent algorithm to do this, since each group has only a constant number of elements. For example, we could just BubbleSort each group and take the middle element. Each will take  $\Theta(1)$  time, and repeating this  $\lceil n/5 \rceil$  times will give a total running time of  $\Theta(n)$ . Copy the group medians to a new array  $B$ .

**Median of medians:** Compute the median of the group medians. For this, we will have to call the selection algorithm recursively on  $B$ , e.g. `Select(B, 1, m, k)`, where  $m = \lceil n/5 \rceil$ , and  $k = \lfloor (m+1)/2 \rfloor$ . Let  $x$  be this median of medians. Return  $x$  as the desired pivot.

The algorithm is illustrated in the figure below. To establish the correctness of this procedure, we need to argue that  $x$  satisfies the desired rank properties.

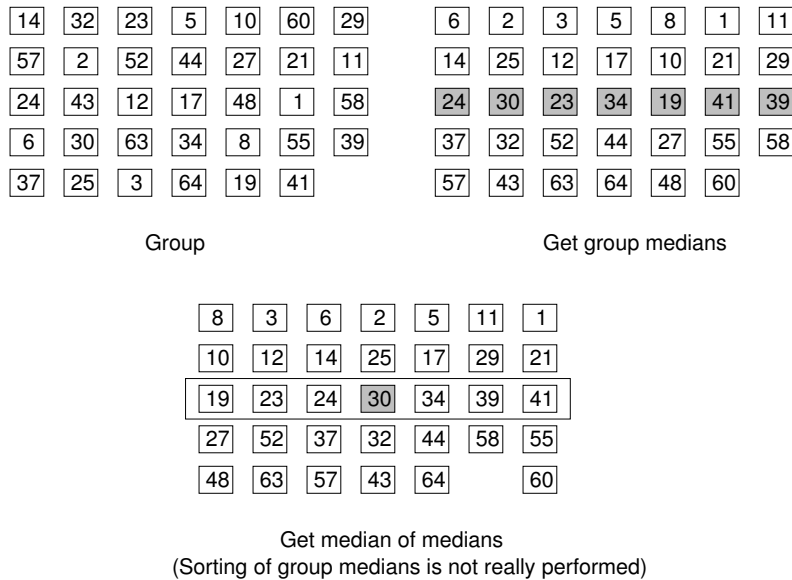


Fig. 72: Choosing the Pivot. 30 is the final pivot.

**Lemma:** The element  $x$  is of rank at least  $n/4$  and at most  $3n/4$  in  $A$ .

**Proof:** We will show that  $x$  is of rank at least  $n/4$ . The other part of the proof is essentially symmetrical. To do this, we need to show that there are at least  $n/4$  elements that are less than or equal to  $x$ . This is a bit complicated, due to the floor and ceiling arithmetic, so to simplify things we will assume that  $n$  is evenly divisible by 5. Consider the groups shown in the tabular form above. Observe that at least half of the group medians are less than or equal to  $x$ . (Because  $x$  is their median.) And for each group median, there are three elements that are less than or equal to this median within its group (because it is the median of its group). Therefore, there are at least  $3((n/5)/2) = 3n/10 \geq n/4$  elements that are less than or equal to  $x$  in the entire array.

**Analysis:** The last order of business is to analyze the running time of the overall algorithm. We achieved the main goal, namely that of eliminating a constant fraction (at least  $1/4$ ) of the remaining list at each stage of the algorithm. The recursive call in `Select()` will be made to list no larger than  $3n/4$ . However, in order to achieve this, within `Select_Pivot()` we needed to make a recursive call to `Select()` on an array  $B$  consisting of  $\lceil n/5 \rceil$  elements. Everything else took only  $\Theta(n)$  time. As usual, we will ignore floors and ceilings, and write the  $\Theta(n)$  as  $n$  for concreteness. The running time is

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ T(n/5) + T(3n/4) + n & \text{otherwise.} \end{cases}$$

This is a very strange recurrence because it involves a mixture of different fractions ( $n/5$  and  $3n/4$ ). This mixture will make it impossible to use the Master Theorem, and difficult to apply iteration. However, this is a good place to apply constructive induction. We know we want an algorithm that runs in  $\Theta(n)$  time.

**Theorem:** There is a constant  $c$ , such that  $T(n) \leq cn$ .

**Proof:** (by strong induction on  $n$ )

**Basis:** ( $n = 1$ ) In this case we have  $T(n) = 1$ , and so  $T(n) \leq cn$  as long as  $c \geq 1$ .

**Step:** We assume that  $T(n') \leq cn'$  for all  $n' < n$ . We will then show that  $T(n) \leq cn$ . By definition we have

$$T(n) = T(n/5) + T(3n/4) + n.$$

Since  $n/5$  and  $3n/4$  are both less than  $n$ , we can apply the induction hypothesis, giving

$$\begin{aligned} T(n) &\leq c\frac{n}{5} + c\frac{3n}{4} + n = cn\left(\frac{1}{5} + \frac{3}{4}\right) + n \\ &= cn\frac{19}{20} + n = n\left(\frac{19c}{20} + 1\right). \end{aligned}$$

This last expression will be  $\leq cn$ , provided that we select  $c$  such that  $c \geq (19c/20) + 1$ . Solving for  $c$  we see that this is true provided that  $c \geq 20$ .

Combining the constraints that  $c \geq 1$ , and  $c \geq 20$ , we see that by letting  $c = 20$ , we are done.

A natural question is why did we pick groups of 5? If you look at the proof above, you will see that it works for any value that is strictly greater than 4. (You might try it replacing the 5 with 3, 4, or 6 and see what happens.)

## Supplemental Lecture 5: Analysis of BucketSort

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

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

Since the distribution is uniform, all of the random variables  $X_i$  have the same probability distribution, so we may as well talk about a single random variable  $X$ , which will work for any bucket. Since we are using a quadratic time algorithm to sort the elements of each bucket, we are interested in the expected sorting time, which is  $\Theta(X^2)$ . So this leads to the key question, what is the expected value of  $X^2$ , denoted  $E[X^2]$ .

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

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

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

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

If you consult a standard textbook on probability and statistics, then you will see the two important facts that we need to know about the binomial distribution. Namely, that its mean value  $E[X]$  and its variance  $\text{Var}[X]$  are

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

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

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

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

## Supplemental Lecture 6: Long Integer Multiplication

**Read:** This material on integer multiplication is not covered in CLRS.

**Long Integer Multiplication:** The following little algorithm shows a bit more about the surprising applications of divide-and-conquer. The problem that we want to consider is how to perform arithmetic on long integers, and multiplication in particular. The reason for doing arithmetic on long numbers stems from cryptography. Most techniques for encryption are based on number-theoretic techniques. For example, the character string to be encrypted is converted into a sequence of numbers, and encryption keys are stored as long integers. Efficient encryption and decryption depends on being able to perform arithmetic on long numbers, typically containing hundreds of digits.

Addition and subtraction on large numbers is relatively easy. If  $n$  is the number of digits, then these algorithms run in  $\Theta(n)$  time. (Go back and analyze your solution to the problem on Homework 1). But the standard algorithm for multiplication runs in  $\Theta(n^2)$  time, which can be quite costly when lots of long multiplications are needed.

This raises the question of whether there is a more efficient way to multiply two very large numbers. It would seem surprising if there were, since for centuries people have used the same algorithm that we all learn in grade school. In fact, we will see that it is possible.

**Divide-and-Conquer Algorithm:** We know the basic grade-school algorithm for multiplication. We normally think of this algorithm as applying on a digit-by-digit basis, but if we partition an  $n$  digit number into two “super digits” with roughly  $n/2$  each into longer sequences, the same multiplication rule still applies.

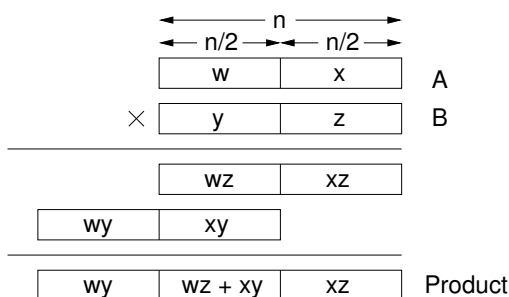


Fig. 73: Long integer multiplication.

To avoid complicating things with floors and ceilings, let's just assume that the number of digits  $n$  is a power of 2. Let  $A$  and  $B$  be the two numbers to multiply. Let  $A[0]$  denote the least significant digit and let  $A[n-1]$  denote the most significant digit of  $A$ . Because of the way we write numbers, it is more natural to think of the elements of  $A$  as being indexed in decreasing order from left to right as  $A[n-1..0]$  rather than the usual  $A[0..n-1]$ .

Let  $m = n/2$ . Let

$$\begin{aligned} w &= A[n-1..m] & x &= A[m-1..0] & \text{and} \\ y &= B[n-1..m] & z &= B[m-1..0]. \end{aligned}$$

If we think of  $w, x, y$  and  $z$  as  $n/2$  digit numbers, we can express  $A$  and  $B$  as

$$\begin{aligned} A &= w \cdot 10^m + x \\ B &= y \cdot 10^m + z, \end{aligned}$$

and their product is

$$\text{mult}(A, B) = \text{mult}(w, y)10^{2m} + (\text{mult}(w, z) + \text{mult}(x, y))10^m + \text{mult}(x, z).$$

The operation of multiplying by  $10^m$  should be thought of as simply shifting the number over by  $m$  positions to the right, and so is not really a multiplication. Observe that all the additions involve numbers involving roughly  $n/2$  digits, and so they take  $\Theta(n)$  time each. Thus, we can express the multiplication of two long integers as the result of four products on integers of roughly half the length of the original, and a constant number of additions and shifts, each taking  $\Theta(n)$  time. This suggests that if we were to implement this algorithm, its running time would be given by the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 4T(n/2) + n & \text{otherwise.} \end{cases}$$

If we apply the Master Theorem, we see that  $a = 4, b = 2, k = 1$ , and  $a > b^k$ , implying that Case 1 holds and the running time is  $\Theta(n^{\lg 4}) = \Theta(n^2)$ . Unfortunately, this is no better than the standard algorithm.

**Faster Divide-and-Conquer Algorithm:** Even though the above exercise appears to have gotten us nowhere, it actually has given us an important insight. It shows that the critical element is the number of multiplications on numbers of size  $n/2$ . The number of additions (as long as it is a constant) does not affect the running time. So, if we could find a way to arrive at the same result algebraically, but by trading off multiplications in favor of additions, then we would have a more efficient algorithm. (Of course, we cannot simulate multiplication through repeated additions, since the number of additions must be a constant, independent of  $n$ .)

The key turns out to be an algebraic “trick”. The quantities that we need to compute are  $C = wy$ ,  $D = xz$ , and  $E = (wz + xy)$ . Above, it took us four multiplications to compute these. However, observe that if instead we compute the following quantities, we can get everything we want, using only three multiplications (but with more additions and subtractions).

$$\begin{aligned} C &= \text{mult}(w, y) \\ D &= \text{mult}(x, z) \\ E &= \text{mult}((w + x), (y + z)) - C - D = (wy + wz + xy + xz) - wy - xz = (wz + xy). \end{aligned}$$

Finally we have

$$\text{mult}(A, B) = C \cdot 10^{2m} + E \cdot 10^m + D.$$

Altogether we perform 3 multiplications, 4 additions, and 2 subtractions all of numbers with  $n/2$  digits. We still need to shift the terms into their proper final positions. The additions, subtractions, and shifts take  $\Theta(n)$  time in total. So the total running time is given by the recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3T(n/2) + n & \text{otherwise.} \end{cases}$$

Now when we apply the Master Theorem, we have  $a = 3$ ,  $b = 2$  and  $k = 1$ , yielding  $T(n) \in \Theta(n^{\lg 3}) \approx \Theta(n^{1.585})$ .

Is this really an improvement? This algorithm carries a larger constant factor because of the overhead of recursion and the additional arithmetic operations. But asymptotics says that if  $n$  is large enough, then this algorithm will be superior. For example, if we assume that the clever algorithm has overheads that are 5 times greater than the simple algorithm (e.g.  $5n^{1.585}$  versus  $n^2$ ) then this algorithm beats the simple algorithm for  $n \geq 50$ . If the overhead was 10 times larger, then the crossover would occur for  $n \geq 260$ . Although this may seem like a very large number, recall that in cryptography applications, encryption keys of this length and longer are quite reasonable.

## Supplemental Lecture 7: Dynamic Programming: 0–1 Knapsack Problem

**Read:** The introduction to Chapter 16 in CLR. The material on the Knapsack Problem is not presented in our text, but is briefly discussed in Section 17.2.

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

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

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

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

subject to

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

Let us assume that the  $v_i$ 's,  $w_i$ 's and  $W$  are all positive integers. It turns out that this problem is NP-complete, and so we cannot really hope to find an efficient solution. However if we make the same sort of assumption that we made in counting sort, we can come up with an efficient solution.

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

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

To compute the entries of the array  $V$  we will imply an inductive approach. As a basis, observe that  $V[0, j] = 0$  for  $0 \leq j \leq W$  since if we have no items then we have no value. We consider two cases:

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

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

Since these are the only two possibilities, we can see that we have the following rule for constructing the array  $V$ . The ranges on  $i$  and  $j$  are  $i \in [0..n]$  and  $j \in [0..W]$ .

$$V[0, j] = 0$$

$$V[i, j] = \begin{cases} V[i - 1, j] & \text{if } w_i > j \\ \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } w_i \leq j \end{cases}$$

The first line states that if there are no objects, then there is no value, irrespective of  $j$ . The second line implements the rule above.

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

An example is shown in the figure below. The final output is  $V[n, W] = V[4, 10] = 90$ . This reflects the selection of items 2 and 4, of values \$40 and \$50, respectively and weights  $4 + 3 \leq 10$ .

The only missing detail is what items should we select to achieve the maximum. We will leave this as an exercise. The key is to record for each entry  $V[i, j]$  in the matrix whether we got this entry by taking the  $i$ th item or leaving it. With this information, it is possible to reconstruct the optimum knapsack contents.

---

```

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

```

---

Values of the objects are  $\langle 10, 40, 30, 50 \rangle$ .  
 Weights of the objects are  $\langle 5, 4, 6, 3 \rangle$ .

		Capacity →										
		$j = 0$	1	2	3	4	5	6	7	8	9	10
Item	Value	Weight	0	0	0	0	0	0	0	0	0	0
1	10	5	0	0	0	0	0	10	10	10	10	10
2	40	4	0	0	0	0	40	40	40	40	40	50
3	30	6	0	0	0	0	40	40	40	40	40	50
4	50	3	0	0	0	50	50	50	50	90	90	90

Final result is  $V[4, 10] = 90$  (for taking items 2 and 4).

Fig. 74: 0-1 Knapsack Example.

## Supplemental Lecture 8: Prim's and Baruvka's Algorithms for MSTs

**Read:** Chapt 23 in CLRS. Baruvka's algorithm is not described in CLRS.

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

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

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

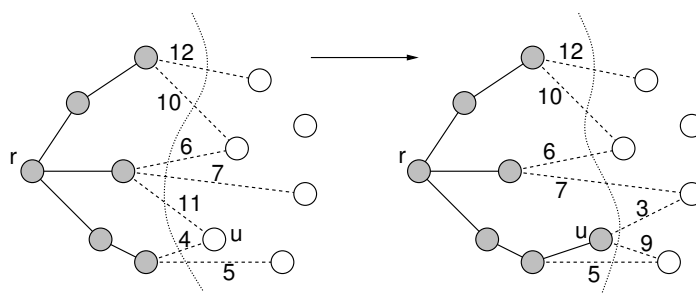


Fig. 75: Prim's Algorithm.

Observe that if we consider the set of vertices  $S$  currently part of the tree, and its complement  $(V - S)$ , we have a cut of the graph and the current set of tree edges  $A$  respects this cut. Which edge should we add next? The MST Lemma from the previous lecture tells us that it is safe to add the *light edge*. In the figure, this is the edge of weight 4 going to vertex  $u$ . Then  $u$  is added to the vertices of  $S$ , and the cut changes. Note that some edges that crossed the cut before are no longer crossing it, and others that were not crossing the cut are.

It is easy to see, that the key questions in the efficient implementation of Prim's algorithm is how to update the cut efficiently, and how to determine the light edge quickly. To do this, we will make use of a *priority queue* data structure. Recall that this is the data structure used in HeapSort. This is a data structure that stores a set of items, where each item is associated with a *key* value. The priority queue supports three operations.

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

**extractMin():** Extract the item with the minimum key value in  $Q$ .

**decreaseKey( $u, new\_key$ ):** Decrease the value of  $u$ 's key value to  $new\_key$ .

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

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

There is a much more elegant solution, and this is what makes Prim's algorithm so nice. For each vertex in  $u \in V - S$  (not part of the current spanning tree) we associate  $u$  with a key value  $key[u]$ , which is the weight of the lightest edge going from  $u$  to any vertex in  $S$ . We also store in  $pred[u]$  the end vertex of this edge in  $S$ . If there is not edge from  $u$  to a vertex in  $V - S$ , then we set its key value to  $+\infty$ . We will also need to know which vertices are in  $S$  and which are not. We do this by coloring the vertices in  $S$  black.

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

Prim's Algorithm

```

Prim(G, w, r) {
  for each (u in V) {                               // initialization
    key[u] = +infinity;
    color[u] = white;
  }
  key[r] = 0;                                       // start at root
  pred[r] = nil;
  Q = new PriorityQueue(V);                         // put vertices in Q
  while (Q.nonEmpty()) {                            // until all vertices in MST
    u = Q.extractMin();                             // vertex with lightest edge
    for each (v in Adj[u]) {
      if ((color[v] == white) && (w(u,v) < key[v])) {
        key[v] = w(u,v);                          // new lighter edge out of v
        Q.decreaseKey(v, key[v]);
        pred[v] = u;
      }
    }
    color[u] = black;
  }
  [The pred pointers define the MST as an inverted tree rooted at r]
}

```

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

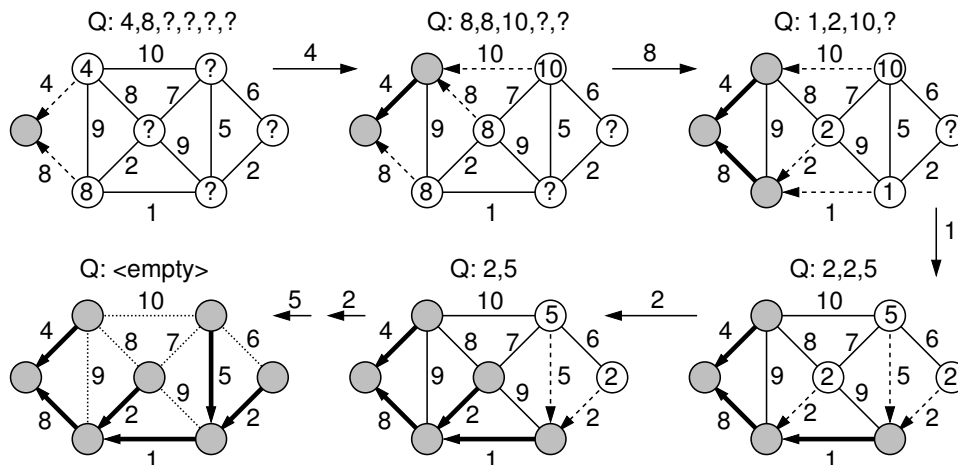


Fig. 76: Prim's Algorithm.

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

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

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

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

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

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

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

---

Baruvka's Algorithm

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

---

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

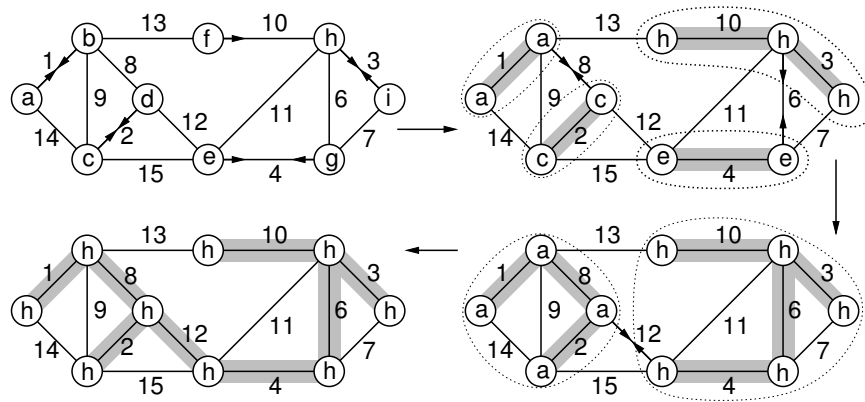


Fig. 77: Baruvka's Algorithm.

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

## Supplemental Lecture 9: Dynamic Programming: Minimum Weight Triangulation

**Read:** This is not covered in CLRS.

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

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

Given two nonadjacent sides  $v_i$  and  $v_j$ , where  $i < j - 1$ , the line segment  $\overline{v_i v_j}$  is a *chord*. (If the polygon is simple but not convex, we include the additional requirement that the interior of the segment must lie entirely in the interior of  $P$ .) Any chord subdivides the polygon into two polygons:  $\langle v_i, v_{i+1}, \dots, v_j \rangle$ , and  $\langle v_j, v_{j+1}, \dots, v_i \rangle$ . A *triangulation* of a convex polygon  $P$  is a subdivision of the interior of  $P$  into a collection of triangles with disjoint interiors, whose vertices are drawn from the vertices of  $P$ . Equivalently, we can define a triangulation as a maximal set  $T$  of nonintersecting chords. (In other words, every chord that is not in  $T$  intersects the interior

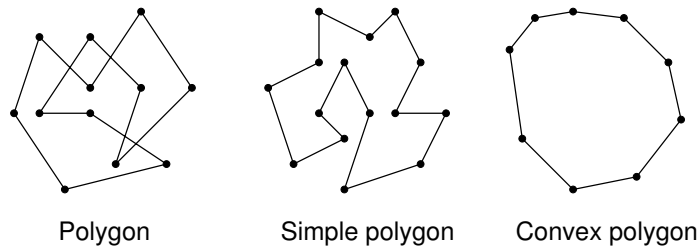


Fig. 78: Polygons.

of some chord in  $T$ .) It is easy to see that such a set of chords subdivides the interior of the polygon into a collection of triangles with pairwise disjoint interiors (and hence the name *triangulation*). It is not hard to prove (by induction) that every triangulation of an  $n$ -sided polygon consists of  $n - 3$  chords and  $n - 2$  triangles. Triangulations are of interest for a number of reasons. Many geometric algorithms operate by first decomposing a complex polygonal shape into triangles.

In general, given a convex polygon, there are many possible triangulations. In fact, the number is exponential in  $n$ , the number of sides. Which triangulation is the “best”? There are many criteria that are used depending on the application. One criterion is to imagine that you must “pay” for the ink you use in drawing the triangulation, and you want to minimize the amount of ink you use. (This may sound fanciful, but minimizing wire length is an important condition in chip design. Further, this is one of many properties which we could choose to optimize.) This suggests the following optimization problem:

**Minimum-weight convex polygon triangulation:** Given a convex polygon determine the triangulation that minimizes the sum of the perimeters of its triangles. (See Fig. 79.)

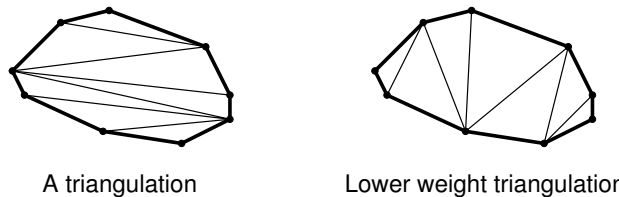


Fig. 79: Triangulations of convex polygons, and the minimum weight triangulation.

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

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

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

**Dynamic Programming Solution:** Let us consider an  $(n + 1)$ -sided polygon  $P = \langle v_0, v_1, \dots, v_n \rangle$ . Let us assume that these vertices have been numbered in counterclockwise order. To derive a DP formulation we need to define a set of subproblems from which we can derive the optimum solution. For  $0 \leq i < j \leq n$ , define  $t[i, j]$  to be the weight of the minimum weight triangulation for the subpolygon that lies to the right of directed chord  $\overrightarrow{v_i v_j}$ , that is, the polygon with the counterclockwise vertex sequence  $\langle v_i, v_{i+1}, \dots, v_j \rangle$ . Observe that if we can compute this quantity for all such  $i$  and  $j$ , then the weight of the minimum weight triangulation of the entire polygon can be extracted as  $t[0, n]$ . (As usual, we only compute the minimum weight. But, it is easy to modify the procedure to extract the actual triangulation.)

As a basis case, we define the weight of the trivial “2-sided polygon” to be zero, implying that  $t[i, i + 1] = 0$ . In general, to compute  $t[i, j]$ , consider the subpolygon  $\langle v_i, v_{i+1}, \dots, v_j \rangle$ , where  $j > i + 1$ . One of the chords of

this polygon is the side  $\overline{v_i v_j}$ . We may split this subpolygon by introducing a triangle whose base is this chord, and whose third vertex is any vertex  $v_k$ , where  $i < k < j$ . This subdivides the polygon into the subpolygons  $\langle v_i, v_{i+1}, \dots, v_k \rangle$  and  $\langle v_k, v_{k+1}, \dots, v_j \rangle$  whose minimum weights are already known to us as  $t[i, k]$  and  $t[k, j]$ . In addition we should consider the weight of the newly added triangle  $\triangle v_i v_k v_j$ . Thus, we have the following recursive rule:

$$t[i, j] = \begin{cases} 0 & \text{if } j = i + 1 \\ \min_{i < k < j} (t[i, k] + t[k, j] + w(v_i v_k v_j)) & \text{if } j > i + 1. \end{cases}$$

The final output is the overall minimum weight, which is,  $t[0, n]$ . This is illustrated in Fig. 80

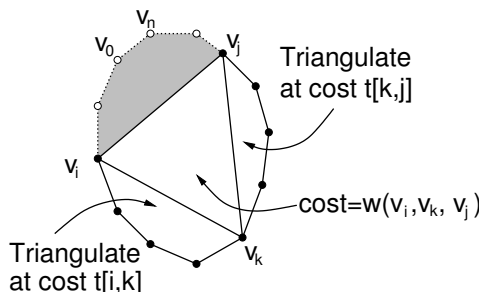


Fig. 80: Triangulations and tree structure.

Note that this has almost exactly the same structure as the recursive definition used in the chain matrix multiplication algorithm (except that some indices are different by 1.) The same  $\Theta(n^3)$  algorithm can be applied with only minor changes.

**Relationship to Binary Trees:** One explanation behind the similarity of triangulations and the chain matrix multiplication algorithm is to observe that both are fundamentally related to binary trees. In the case of the chain matrix multiplication, the associated binary tree is the evaluation tree for the multiplication, where the leaves of the tree correspond to the matrices, and each node of the tree is associated with a product of a sequence of two or more matrices. To see that there is a similar correspondence here, consider an  $(n + 1)$ -sided convex polygon  $P = \langle v_0, v_1, \dots, v_n \rangle$ , and fix one side of the polygon (say  $\overline{v_0 v_n}$ ). Now consider a rooted binary tree whose root node is the triangle containing side  $\overline{v_0 v_n}$ , whose internal nodes are the nodes of the dual tree, and whose leaves correspond to the remaining sides of the tree. Observe that partitioning the polygon into triangles is equivalent to a binary tree with  $n$  leaves, and vice versa. This is illustrated in Fig. 81. Note that every triangle is associated with an internal node of the tree and every edge of the original polygon, except for the distinguished starting side  $\overline{v_0 v_n}$ , is associated with a leaf node of the tree.

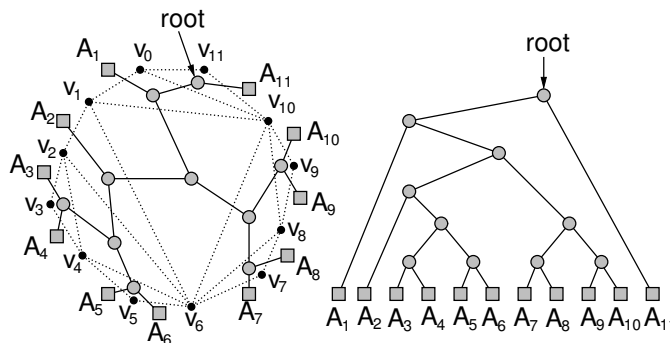


Fig. 81: Triangulations and tree structure.

Once you see this connection. Then the following two observations follow easily. Observe that the associated binary tree has  $n$  leaves, and hence (by standard results on binary trees)  $n - 1$  internal nodes. Since each internal node other than the root has one edge entering it, there are  $n - 2$  edges between the internal nodes. Each internal node corresponds to one triangle, and each edge between internal nodes corresponds to one chord of the triangulation.

## Supplemental Lecture 10: Bellman-Ford Shortest Paths

**Read:** Section 24.1 in CLRS.

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

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

Bellman-Ford Algorithm

```

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

```

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

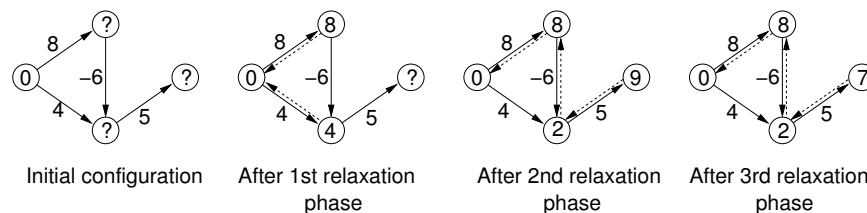


Fig. 82: Bellman-Ford Algorithm.

**Correctness of Bellman-Ford:** I like to think of the Bellman-Ford as a sort of “BubbleSort analogue” for shortest paths, in the sense that shortest path information is propagated sequentially along each shortest path in the graph. Consider any shortest path from  $s$  to some other vertex  $u$ :  $\langle v_0, v_1, \dots, v_k \rangle$  where  $v_0 = s$  and  $v_k = u$ . Since a shortest path will never visit the same vertex twice, we know that  $k \leq V - 1$ , and hence the path consists of at most  $V - 1$  edges. Since this is a shortest path we have  $\delta(s, v_i)$  (the true shortest path cost from  $s$  to  $v_i$ ) satisfies

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

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

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

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

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

## Subset Sum

**Read:** Sections 34.5.5 in CLR.

**Subset Sum:** The Subset Sum problem (SS) is the following. Given a finite set  $S$  of positive integers  $S = \{w_1, w_2, \dots, w_n\}$  and a *target value*,  $t$ , we want to know whether there exists a subset  $S' \subseteq S$  that sums exactly to  $t$ .

This problem is a simplified version of the 0-1 Knapsack problem, presented as a decision problem. Recall that in the 0-1 Knapsack problem, we are given a collection of objects, each with an associated weight  $w_i$  and associated value  $v_i$ . We are given a knapsack of capacity  $W$ . The objective is to take as many objects as can fit in the knapsack’s capacity so as to maximize the value. (In the fractional knapsack we could take a portion of an object. In the 0-1 Knapsack we either take an object entirely or leave it.) In the simplest version, suppose that the value is the same as the weight,  $v_i = w_i$ . (This would occur for example if all the objects were made of the same material, say, gold.) Then, the best we could hope to achieve would be to fill the knapsack entirely. By setting  $t = W$ , we see that the subset sum problem is equivalent to this simplified version of the 0-1 Knapsack problem. It follows that if we can show that this simpler version is NP-complete, then certainly the more general 0-1 Knapsack problem (stated as a decision problem) is also NP-complete.

Consider the following example.

$$S = \{3, 6, 9, 12, 15, 23, 32\} \quad \text{and} \quad t = 33.$$

The subset  $S' = \{6, 12, 15\}$  sums to  $t = 33$ , so the answer in this case is yes. If  $t = 34$  the answer would be no.

**Dynamic Programming Solution:** There is a dynamic programming algorithm which solves the Subset Sum problem in  $O(n \cdot t)$  time.<sup>5</sup>

The quantity  $n \cdot t$  is a polynomial function of  $n$ . This would seem to imply that the Subset Sum problem is in P. But there is a important catch. Recall that in all NP-complete problems we assume (1) running time is measured

<sup>5</sup>We will leave this as an exercise, but the formulation is, for  $0 \leq i \leq n$  and  $0 \leq t' \leq t$ ,  $S[i, t'] = 1$  if there is a subset of  $\{w_1, w_2, \dots, w_i\}$  that sums to  $t'$ , and 0 otherwise. The  $i$ th row of this table can be computed in  $O(t)$  time, given the contents of the  $(i - 1)$ -st row.

as a function of input size (number of bits) and (2) inputs must be encoded in a reasonable succinct manner. Let us assume that the numbers  $w_i$  and  $t$  are all  $b$ -bit numbers represented in base 2, using the fewest number of bits possible. Then the input size is  $O(nb)$ . The value of  $t$  may be as large as  $2^b$ . So the resulting algorithm has a running time of  $O(n2^b)$ . This is polynomial in  $n$ , but exponential in  $b$ . Thus, this running time is not polynomial as a function of the input size.

Note that an important consequence of this observation is that the SS problem is not hard when the numbers involved are small. If the numbers involved are of a fixed number of bits (a constant independent of  $n$ ), then the problem is solvable in polynomial time. However, we will show that in the general case, this problem is NP-complete.

**SS is NP-complete:** The proof that Subset Sum (SS) is NP-complete involves the usual two elements.

- (i)  $SS \in NP$ .
- (ii) Some known NP-complete problem is reducible to SS. In particular, we will show that Vertex Cover (VC) is reducible to SS, that is,  $VC \leq_P SS$ .

To show that SS is in NP, we need to give a verification procedure. Given  $S$  and  $t$ , the certificate is just the indices of the numbers that form the subset  $S'$ . We can add two  $b$ -bit numbers together in  $O(b)$  time. So, in polynomial time we can compute the sum of elements in  $S'$ , and verify that this sum equals  $t$ .

For the remainder of the proof we show how to reduce vertex cover to subset sum. We want a polynomial time computable function  $f$  that maps an instance of the vertex cover (a graph  $G$  and integer  $k$ ) to an instance of the subset sum problem (a set of integers  $S$  and target integer  $t$ ) such that  $G$  has a vertex cover of size  $k$  if and only if  $S$  has a subset summing to  $t$ . Thus, if subset sum were solvable in polynomial time, so would vertex cover.

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

**An Initial Approach:** Here is an idea, which does not work, but gives a sense of how to proceed. Let  $E$  denote the number of edges in the graph. First number the edges of the graph from 1 through  $E$ . Then represent each vertex  $v_i$  as an  $E$ -element bit vector, where the  $j$ -th bit from the left is set to 1 if and only if the edge  $e_j$  is incident to vertex  $v_i$ . (Another way to think of this is that these bit vectors form the rows of an *incidence matrix* for the graph.) An example is shown below, in which  $k = 3$ .

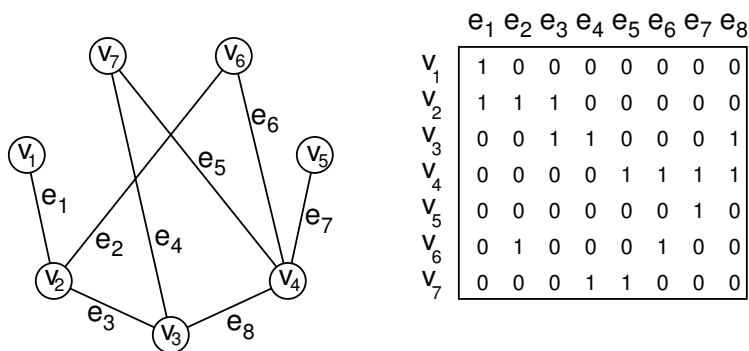


Fig. 83: Encoding a graph as a collection of bit vectors.

Now, suppose we take any subset of vertices and form the logical-or of the corresponding bit vectors. If the subset is a vertex cover, then every edge will be covered by at least one of these vertices, and so the logical-or will be a bit vector of all 1's,  $1111 \dots 1$ . Conversely, if the logical-or is a bit vector of 1's, then each edge has

been covered by some vertex, implying that the vertices form a vertex cover. (Later we will consider how to encode the fact that there only allowed  $k$  vertices in the cover.)

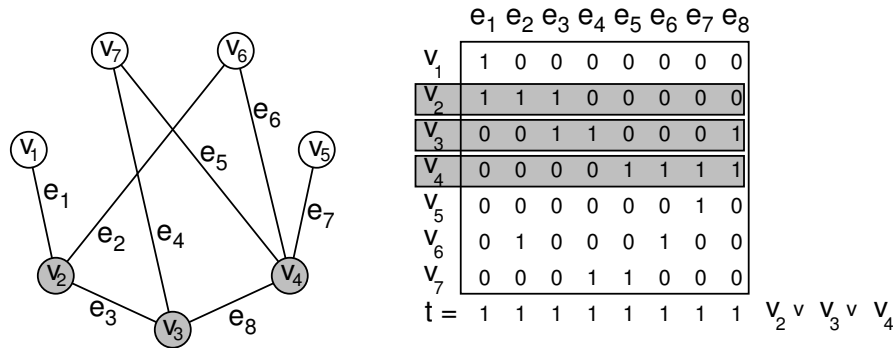


Fig. 84: The logical-or of a vertex cover equals 1111 . . . 1.

Since bit vectors can be thought of as just a way of representing numbers in binary, this is starting to feel more like the subset sum problem. The target would be the number whose bit vector is all 1's. There are a number of problems, however. First, logical-or is not the same as addition. For example, if both of the endpoints of some edge are in the vertex cover, then its value in the corresponding column would be 2, not 1. Second, we have no way of controlling how many vertices go into the vertex cover. (We could just take the logical-or of all the vertices, and then the logical-or would certainly be a bit vectors of 1's.)

There are two ways in which addition differs significantly from logical-or. The first is the issue of carries. For example, the  $1101 \vee 0011 = 1111$ , but in binary  $1101 + 0011 = 1000$ . To fix this, we recognize that we do not have to use a binary (base-2) representation. In fact, we can assume any base system we want. Observe that each column of the incidence matrix has at most two 1's in any column, because each edge is incident to at most two vertices. Thus, if use any base that is at least as large as base 3, we will never generate a carry to the next position. In fact we will use base 4 (for reasons to be seen below). Note that the base of the number system is just for own convenience of notation. Once the numbers have been formed, they will be converted into whatever form our machine assumes for its input representation, e.g. decimal or binary.

The second difference between logical-or and addition is that an edge may generally be covered either once or twice in the vertex cover. So, the final sum of these numbers will be a number consisting of 1 and 2 digits, e.g. 1211 . . . 112. This does not provide us with a unique target value  $t$ . We know that no digit of our sum can be a zero. To fix this problem, we will create a set of  $E$  additional *slack values*. For  $1 \leq i \leq E$ , the  $i$ th slack value will consist of all 0's, except for a single 1-digit in the  $i$ th position, e.g., 00000100000. Our target will be the number 2222 . . . 222 (all 2's). To see why this works, observe that from the numbers of our vertex cover, we will get a sum consisting of 1's and 2's. For each position where there is a 1, we can supplement this value by adding in the corresponding slack value. Thus we can boost any value consisting of 1's and 2's to all 2's. On the other hand, note that if there are any 0 values in the final sum, we will not have enough slack values to convert this into a 2.

There is one last issue. We are only allowed to place only  $k$  vertices in the vertex cover. We will handle this by adding an additional column. For each number arising from a vertex, we will put a 1 in this additional column. For each slack variable we will put a 0. In the target, we will require that this column sum to the value  $k$ , the size of the vertex cover. Thus, to form the desired sum, we must select exactly  $k$  of the vertex values. Note that since we only have a base-4 representation, there might be carries out of this last column (if  $k \geq 4$ ). But since this is the last column, it will not affect any of the other aspects of the construction.

**The Final Reduction:** Here is the final reduction, given the graph  $G = (V, E)$  and integer  $k$  for the vertex cover problem.

- (1) Create a set of  $n$  vertex values,  $x_1, x_2, \dots, x_n$  using base-4 notation. The value  $x_i$  is equal a 1 followed by a sequence of  $E$  base-4 digits. The  $j$ -th digit is a 1 if edge  $e_j$  is incident to vertex  $v_i$  and 0 otherwise.
- (2) Create  $E$  slack values  $y_1, y_2, \dots, y_E$ , where  $y_i$  is a 0 followed by  $E$  base-4 digits. The  $i$ -th digit of  $y_i$  is 1 and all others are 0.
- (3) Let  $t$  be the base-4 number whose first digit is  $k$  (this may actually span multiple base-4 digits), and whose remaining  $E$  digits are all 2.
- (4) Convert the  $x_i$ 's, the  $y_j$ 's, and  $t$  into whatever base notation is used for the subset sum problem (e.g. base 10). Output the set  $S = \{x_1, \dots, x_n, y_1, \dots, y_E\}$  and  $t$ .

Observe that this can be done in polynomial time, in  $O(E^2)$ , in fact. The construction is illustrated in Fig. 85.

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

↑ vertex cover size ( $k=3$ )

Fig. 85: Vertex cover to subset sum reduction.

**Correctness:** We claim that  $G$  has a vertex cover of size  $k$  if and only if  $S$  has a subset that sums to  $t$ . If  $G$  has a vertex cover  $V'$  of size  $k$ , then we take the vertex values  $x_i$  corresponding to the vertices of  $V'$ , and for each edge that is covered only once in  $V'$ , we take the corresponding slack variable. It follows from the comments made earlier that the lower-order  $E$  digits of the resulting sum will be of the form  $222\dots 2$  and because there are  $k$  elements in  $V'$ , the leftmost digit of the sum will be  $k$ . Thus, the resulting subset sums to  $t$ .

Conversely, if  $S$  has a subset  $S'$  that sums to  $t$  then we assert that it must select exactly  $k$  values from among the vertex values, since the first digit must sum to  $k$ . We claim that these vertices  $V'$  form a vertex cover. In particular, no edge can be left uncovered by  $V'$ , since (because there are no carries) the corresponding column would be 0 in the sum of vertex values. Thus, no matter what slack values we add, the resulting digit position could not be equal to 2, and so this cannot be a solution to the subset sum problem.

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

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

vertex cover size

} Vertex values  
(take those in vertex cover)

} Slack values  
(take one for each edge that has only one endpoint in the cover)

Fig. 86: Correctness of the reduction.

## Supplemental Lecture 11: Subset Sum Approximation

**Read:** Section 37.4 in CLR.

**Polynomial Approximation Schemes:** Last time we saw that for some NP-complete problems, it is possible to approximate the problem to within a fixed constant ratio bound. For example, the approximation algorithm produces an answer that is within a factor of 2 of the optimal solution. However, in practice, people would like to control the precision of the approximation. This is done by specifying a parameter  $\epsilon > 0$  as part of the input to the approximation algorithm, and requiring that the algorithm produce an answer that is within a *relative error* of  $\epsilon$  of the optimal solution. It is understood that as  $\epsilon$  tends to 0, the running time of the algorithm will increase. Such an algorithm is called a *polynomial approximation scheme*.

For example, the running time of the algorithm might be  $O(2^{(1/\epsilon)n^2})$ . It is easy to see that in such cases the user pays a big penalty in running time as a function of  $\epsilon$ . (For example, to produce a 1% error, the “constant” factor would be  $2^{100}$  which would be around 4 quadrillion centuries on your 100 Mhz Pentium.) A *fully polynomial approximation scheme* is one in which the running time is polynomial in both  $n$  and  $1/\epsilon$ . For example, a running time of  $O((n/\epsilon)^2)$  would satisfy this condition. In such cases, reasonably accurate approximations are computationally feasible.

Unfortunately, there are very few NP-complete problems with fully polynomial approximation schemes. In fact, recently there has been strong evidence that many NP-complete problems do not have polynomial approximation schemes (fully or otherwise). Today we will study one that does.

**Subset Sum:** Recall that in the subset sum problem we are given a set  $S$  of positive integers  $\{x_1, x_2, \dots, x_n\}$  and a target value  $t$ , and we are asked whether there exists a subset  $S' \subseteq S$  that sums exactly to  $t$ . The optimization problem is to determine the subset whose sum is as large as possible but not larger than  $t$ .

This problem is basic to many packing problems, and is indirectly related to processor scheduling problems that arise in operating systems as well. Suppose we are also given  $0 < \epsilon < 1$ . Let  $z^* \leq t$  denote the optimum sum.

The approximation problem is to return a value  $z \leq t$  such that

$$z \geq z^*(1 - \epsilon).$$

If we think of this as a knapsack problem, we want our knapsack to be within a factor of  $(1 - \epsilon)$  of being as full as possible. So, if  $\epsilon = 0.1$ , then the knapsack should be at least 90% as full as the best possible.

What do we mean by polynomial time here? Recall that the running time should be polynomial in the size of the input length. Obviously  $n$  is part of the input length. But  $t$  and the numbers  $x_i$  could also be huge binary numbers. Normally we just assume that a binary number can fit into a word of our computer, and do not count their length. In this case we will be on the safe side. Clearly  $t$  requires  $O(\log t)$  digits to be store in the input. We will take the input size to be  $n + \log t$ .

Intuitively it is not hard to believe that it should be possible to determine whether we can fill the knapsack to within 90% of optimal. After all, we are used to solving similar sorts of packing problems all the time in real life. But the mental heuristics that we apply to these problems are not necessarily easy to convert into efficient algorithms. Our intuition tells us that we can afford to be a little “sloppy” in keeping track of exactly full the knapsack is at any point. The value of  $\epsilon$  tells us just how sloppy we can be. Our approximation will do something similar. First we consider an exponential time algorithm, and then convert it into an approximation algorithm.

**Exponential Time Algorithm:** This algorithm is a variation of the dynamic programming solution we gave for the knapsack problem. Recall that there we used an 2-dimensional array to keep track of whether we could fill a knapsack of a given capacity with the first  $i$  objects. We will do something similar here. As before, we will concentrate on the question of which sums are possible, but determining the subsets that give these sums will not be hard.

Let  $L_i$  denote a list of integers that contains the sums of all  $2^i$  subsets of  $\{x_1, x_2, \dots, x_i\}$  (including the empty set whose sum is 0). For example, for the set  $\{1, 4, 6\}$  the corresponding list of sums contains  $\langle 0, 1, 4, 5 (= 1 + 4), 6, 7 (= 1 + 6), 10 (= 4 + 6), 11 (= 1 + 4 + 6) \rangle$ . Note that  $L_i$  can have as many as  $2^i$  elements, but may have fewer, since some subsets may have the same sum.

There are two things we will want to do for efficiency. (1) Remove any duplicates from  $L_i$ , and (2) only keep sums that are less than or equal to  $t$ . Let us suppose that we a procedure `MergeLists(L1, L2)` which merges two sorted lists, and returns a sorted lists with all duplicates removed. This is essentially the procedure used in MergeSort but with the added duplicate element test. As a bit of notation, let  $L + x$  denote the list resulting by adding the number  $x$  to every element of list  $L$ . Thus  $\langle 1, 4, 6 \rangle + 3 = \langle 4, 7, 9 \rangle$ . This gives the following procedure for the subset sum problem.

---

Exact Subset Sum

```
Exact_SS(x[1..n], t) {
  L = <0>;
  for i = 1 to n do {
    L = MergeLists(L, L+x[i]);
    remove for L all elements greater than t;
  }
  return largest element in L;
}
```

---

For example, if  $S = \{1, 4, 6\}$  and  $t = 8$  then the successive lists would be

$$\begin{aligned} L_0 &= \langle 0 \rangle \\ L_1 &= \langle 0 \rangle \cup \langle 0 + 1 \rangle = \langle 0, 1 \rangle \\ L_2 &= \langle 0, 1 \rangle \cup \langle 0 + 4, 1 + 4 \rangle = \langle 0, 1, 4, 5 \rangle \\ L_3 &= \langle 0, 1, 4, 5 \rangle \cup \langle 0 + 6, 1 + 6, 4 + 6, 5 + 6 \rangle = \langle 0, 1, 4, 5, 6, 7, 10, 11 \rangle. \end{aligned}$$

The last list would have the elements 10 and 11 removed, and the final answer would be 7. The algorithm runs in  $\Omega(2^n)$  time in the worst case, because this is the number of sums that are generated if there are no duplicates, and no items are removed.

**Approximation Algorithm:** To convert this into an approximation algorithm, we will introduce a “trim” the lists to decrease their sizes. The idea is that if the list  $L$  contains two numbers that are very close to one another, e.g. 91,048 and 91,050, then we should not need to keep both of these numbers in the list. One of them is good enough for future approximations. This will reduce the size of the lists that the algorithm needs to maintain. But, how much trimming can we allow and still keep our approximation bound? Furthermore, will we be able to reduce the list sizes from exponential to polynomial?

The answer to both these questions is yes, provided you apply a proper way of trimming the lists. We will trim elements whose values are sufficiently close to each other. But we should define close in manner that is relative to the sizes of the numbers involved. The trimming must also depend on  $\epsilon$ . We select  $\delta = \epsilon/n$ . (Why? We will see later that this is the value that makes everything work out in the end.) Note that  $0 < \delta < 1$ . Assume that the elements of  $L$  are sorted. We walk through the list. Let  $z$  denote the last untrimmed element in  $L$ , and let  $y \geq z$  be the next element to be considered. If

$$\frac{y - z}{y} \leq \delta$$

then we trim  $y$  from the list. Equivalently, this means that the final trimmed list cannot contain two value  $y$  and  $z$  such that

$$(1 - \delta)y \leq z \leq y.$$

We can think of  $z$  as *representing*  $y$  in the list.

For example, given  $\delta = 0.1$  and given the list

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle,$$

the trimmed list  $L'$  will consist of

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle.$$

Another way to visualize trimming is to break the interval from  $[1, t]$  into a set of *buckets* of exponentially increasing size. Let  $d = 1/(1 - \delta)$ . Note that  $d > 1$ . Consider the intervals  $[1, d], [d, d^2], [d^2, d^3], \dots, [d^{k-1}, d^k]$  where  $d^k \geq t$ . If  $z \leq y$  are in the same interval  $[d^{i-1}, d^i]$  then

$$\frac{y - z}{y} \leq \frac{d^i - d^{i-1}}{d^i} = 1 - \frac{1}{d} = \delta.$$

Thus, we cannot have more than one item within each bucket. We can think of trimming as a way of enforcing the condition that items in our lists are not relatively too close to one another, by enforcing the condition that no bucket has more than one item.

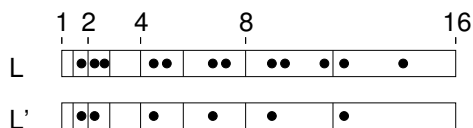


Fig. 87: Trimming Lists for Approximate Subset Sum.

**Claim:** The number of distinct items in a trimmed list is  $O((n \log t)/\epsilon)$ , which is polynomial in input size and  $1/\epsilon$ .

**Proof:** We know that each pair of consecutive elements in a trimmed list differ by a ratio of at least  $d = 1/(1 - \delta) > 1$ . Let  $k$  denote the number of elements in the trimmed list, ignoring the element of value 0. Thus, the smallest nonzero value and maximum value in the trimmed list differ by a ratio of at least  $d^{k-1}$ . Since the smallest (nonzero) element is at least as large as 1, and the largest is no larger than  $t$ , then it follows that  $d^{k-1} \leq t/1 = t$ . Taking the natural log of both sides we have  $(k - 1) \ln d \leq \ln t$ . Using the facts that  $\delta = \epsilon/n$  and the log identity that  $\ln(1 + x) \leq x$ , we have

$$\begin{aligned} k - 1 &\leq \frac{\ln t}{\ln d} = \frac{\ln t}{-\ln(1 - \delta)} \\ &\leq \frac{\ln t}{\delta} = \frac{n \ln t}{\epsilon} \\ k &= O\left(\frac{n \log t}{\epsilon}\right). \end{aligned}$$

Observe that the input size is at least as large as  $n$  (since there are  $n$  numbers) and at least as large as  $\log t$  (since it takes  $\log t$  digits to write down  $t$  on the input). Thus, this function is polynomial in the input size and  $1/\epsilon$ .

The approximation algorithm operates as before, but in addition we call the procedure `Trim` given below.

---

Approximate Subset Sum

```
Trim(L, delta) {
  let the elements of L be denoted y[1..m];
  L' = <y[1]>; // start with first item
  last = y[1]; // last item to be added
  for i = 2 to m do {
    if (last < (1-delta) y[i]) { // different enough?
      append y[i] to end of L';
      last = y[i];
    }
  }
}

Approx_SS(x[1..n], t, eps) {
  delta = eps/n; // approx factor
  L = <0>; // empty sum = 0
  for i = 1 to n do {
    L = MergeLists(L, L+x[i]); // add in next item
    L = Trim(L, delta); // trim away "near" duplicates
    remove for L all elements greater than t;
  }
  return largest element in L;
}
```

---

For example, consider the set  $S = \{104, 102, 201, 101\}$  and  $t = 308$  and  $\epsilon = 0.20$ . We have  $\delta = \epsilon/4 = 0.05$ . Here is a summary of the algorithm's execution.

init:  $L_0 = \langle 0 \rangle$   
 merge:  $L_1 = \langle 0, 104 \rangle$   
 trim:  $L_1 = \langle 0, 104 \rangle$   
 remove:  $L_1 = \langle 0, 104 \rangle$   
  
 merge:  $L_2 = \langle 0, 102, 104, 206 \rangle$   
 trim:  $L_2 = \langle 0, 102, 206 \rangle$   
 remove:  $L_2 = \langle 0, 102, 206 \rangle$   
  
 merge:  $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$   
 trim:  $L_3 = \langle 0, 102, 201, 303, 407 \rangle$   
 remove:  $L_3 = \langle 0, 102, 201, 303 \rangle$   
  
 merge:  $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$   
 trim:  $L_4 = \langle 0, 101, 201, 302, 404 \rangle$   
 remove:  $L_4 = \langle 0, 101, 201, 302 \rangle$

The final output is 302. The optimum is  $307 = 104 + 102 + 101$ . So our actual relative error in this case is within 2%.

The running time of the procedure is  $O(n|L|)$  which is  $O(n^2 \ln t/\epsilon)$  by the earlier claim.

**Approximation Analysis:** The final question is why the algorithm achieves an relative error of at most  $\epsilon$  over the optimum solution. Let  $Y^*$  denote the optimum (largest) subset sum and let  $Y$  denote the value returned by the algorithm. We want to show that  $Y$  is not too much smaller than  $Y^*$ , that is,

$$Y \geq Y^*(1 - \epsilon).$$

Our proof will make use of an important inequality from real analysis.

**Lemma:** For  $n > 0$  and  $a$  real numbers,

$$(1 + a) \leq \left(1 + \frac{a}{n}\right)^n \leq e^a.$$

Recall that our intuition was that we would allow a relative error of  $\epsilon/n$  at each stage of the algorithm. Since the algorithm has  $n$  stages, then the total relative error should be (obviously?)  $n(\epsilon/n) = \epsilon$ . The catch is that these are relative, not absolute errors. These errors do not accumulate additively, but rather by multiplication. So we need to be more careful.

Let  $L_i^*$  denote the  $i$ -th list in the exponential time (optimal) solution and let  $L_i$  denote the  $i$ -th list in the approximate algorithm. We claim that for each  $y \in L_i^*$  there exists a representative item  $z \in L_i$  whose relative error from  $y$  that satisfies

$$(1 - \epsilon/n)^i y \leq z \leq y.$$

The proof of the claim is by induction on  $i$ . Initially  $L_0 = L_0^* = \langle 0 \rangle$ , and so there is no error. Suppose by induction that the above equation holds for each item in  $L_{i-1}^*$ . Consider an element  $y \in L_{i-1}^*$ . We know that

$y$  will generate two elements in  $L_i^*$ :  $y$  and  $y + x_i$ . We want to argue that there will be a representative that is “close” to each of these items.

By our induction hypothesis, there is a representative element  $z$  in  $L_{i-1}$  such that

$$(1 - \epsilon/n)^{i-1}y \leq z \leq y.$$

When we apply our algorithm, we will form two new items to add (initially) to  $L_i$ :  $z$  and  $z + x_i$ . Observe that by adding  $x_i$  to the inequality above and a little simplification we get

$$(1 - \epsilon/n)^{i-1}(y + x_i) \leq z + x_i \leq y + x_i.$$

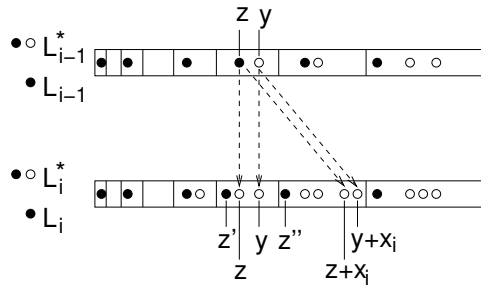


Fig. 88: Subset sum approximation analysis.

The items  $z$  and  $z + x_i$  might not appear in  $L_i$  because they may be trimmed. Let  $z'$  and  $z''$  be their respective representatives. Thus,  $z'$  and  $z''$  are elements of  $L_i$ . We have

$$\begin{aligned} (1 - \epsilon/n)z &\leq z' \leq z \\ (1 - \epsilon/n)(z + x_i) &\leq z'' \leq z + x_i. \end{aligned}$$

Combining these with the inequalities above we have

$$\begin{aligned} (1 - \epsilon/n)^{i-1}(1 - \epsilon/n)y &\leq (1 - \epsilon/n)^i y \leq z' \leq y \\ (1 - \epsilon/n)^{i-1}(1 - \epsilon/n)(y + x_i) &\leq (1 - \epsilon/n)^i (y + x_i) \leq z'' \leq z + x_i. \end{aligned}$$

Since  $z$  and  $z''$  are in  $L_i$  this is the desired result. This ends the proof of the claim.

Using our claim, and the fact that  $Y^*$  (the optimum answer) is the largest element of  $L_n^*$  and  $Y$  (the approximate answer) is the largest element of  $L_n$  we have

$$(1 - \epsilon/n)^n Y^* \leq Y \leq Y^*.$$

This is not quite what we wanted. We wanted to show that  $(1 - \epsilon)Y^* \leq Y$ . To complete the proof, we observe from the lemma above (setting  $a = -\epsilon$ ) that

$$(1 - \epsilon) \leq \left(1 - \frac{\epsilon}{n}\right)^n.$$

This completes the approximate analysis.