CMSC 451
Design and Analysis of Computer Algorithms[1]

David M. Mount
Department of Computer Science
University of Maryland
Fall 2012

Lecture 1: Introduction to Algorithm Design

What is an algorithm? In this course we will study algorithms for interesting computational problems, focusing on principles used to design those algorithms. 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.

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 large programming projects are *structurally complex*, 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 data sets* 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 mathematically. This is among the most basic aspects of good programming. Once you have settled on a good initial design (or perhaps a few good options) you can then prototype the designs and perform experimental studies on their actual efficiency.

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, sorting, and basic graph algorithms. These have been covered in earlier courses, and so we will breeze through them pretty quickly. Next, we will consider a number of common algorithm design techniques, including greedy algorithms, dynamic programming, and augmentation-based methods.

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.

Issues in Algorithm Design: Algorithms are mathematical objects (in contrast to the must 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 may be sufficient. A key concept in establishing an algorithm's correctness is the notion of an *invariant*, that is, a logical assertion about the state of the data at a given point in the algorithm.

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.

Worst-case complexity: Among all inputs of the same size, what is the maximum running time?

Average-case complexity: Among all inputs of the same size, what is the average running time? The average is computed assuming some probability distribution that describes the likelihood that a particular input will arise.

To keep matters simple, we will focus almost exclusively 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 "$theList.\textit{insertAtEnd}(theCurrentElement)$." 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 it 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.

Lecture 2: Algorithm Design: The Stable Marriage Problem

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 residence training, assigning interns to companies, 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 be paired, that is, to *marry*. We will place the algorithm in the role of a metaphorical matchmaker. First, we will use the traditional notion of marriage, the outcome of our process will be a full 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: Brad (B), Tom (T), and Jay-Z (J). There are three women: Angelina (A), Katie (K), and Byounce (Y). Here are their rank orderings (from most to least desired).

	Men			Women	
Brad (B)	Tom (T)	Jay-Z (J)	Katie (K)	Angelina (A)	Byounce (Y)
K	K	Y	J	J	B
A	Y	K	B	T	T
Y	A	A	T	B	J

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 Brad and Tom list Katie 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 can say to another woman, “We each prefer each other to our assigned partners—let’s run away together!” If no such *instability* exists, the pairing is said to be stable.

Definition 1: Given a pair of sets X and Y, a *matching*, is a collection of pairs (x, y), where $x \in X$ and $y \in Y$, and each element of X appears in at most one pair, and each element of Y appears in at most one pair. A matching is *perfect* if every element of X and Y occurs in some pair. (In other words, a perfect matching is a 1-to-1 correspondence between the elements of X and Y.)

Definition 2: Given sets X and Y of equal size and a preference ordering for each element of each set, a perfect matching is *stable* if there is no pair (x, y) that is *not* in the matching and x prefers y to its current match and y prefers x to its current match.

For example, among the following, can you spot which are stable and which are unstable? To make it easier to spot instabilities, after each person I have listed in brackets the people that they would have preferred to their assigned choice.

Assignment I	Assignment II	Assignment III
B [K] ↔ A [J, T]	B [K, A] ↔ Y [ ]	B [K, A] ↔ Y [ ]
T [K] ↔ Y [B]	T [ ] ↔ K [J, B]	T [K, Y] ↔ A [J]
J [Y] ↔ K [ ]	J [Y, K] ↔ A [ ]	J [Y] ↔ K [ ]

The only unstable one is II. Observe that Brad would have preferred Katie over his assigned mate Byounce, and Katie would have preferred Brad to her assigned mate Tom. Thus, the pair of engagements $(B \leftrightarrow Y)$ and $(T \leftrightarrow K)$ 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 algorithm assumes that the process involves two primitives:

Proposal: An unengaged man makes a proposal to a woman

Decision: A woman who receives a proposal can either accept or reject it. If she is already engaged and accepts a proposal, her existing engagement is broken off, and her old mate becomes unengaged.

There is an obvious sexual bias here, since men do the proposing and women do the deciding. It is interesting to consider a more balanced system where either side can offer proposals. (Not surprisingly, it does make a difference whether men or women do the proposing, from the perspective of who tends to get assigned mates of higher preference. We'll leave this question as an exercise.)

It will make the algorithm a bit easier to analyze, if we view it as operating in the series of *rounds*. In each round, each of the men who are currently unengaged offers a proposal to the highest woman on his preference to which hasn't previously proposed. The women receiving the proposal compares the offer to her current mate. If she prefers the new proposal, she accepts it, thus breaking off her old engagement. Otherwise, she rejects it. The rounds continue until no unengaged men remain.

We present the code for the Gale-Shapley 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.

The Gale-Shapley Algorithm

```
// 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
        }
    }
}
```

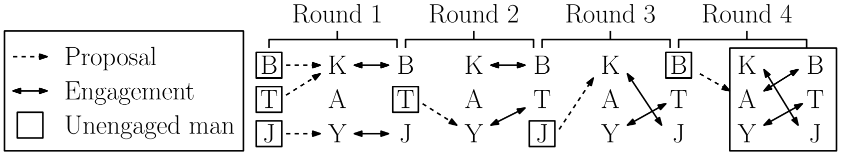


Fig. 1: Example of the round form version of the GS Algorithm on the preference lists given earlier.

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 mate can only get better over time in terms of her preference list.

Lemma 2: The mates 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, at any time, 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 mate 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.

Lecture 3: Algorithm Design Review: Mathematical Background

Algorithm Analysis: In this lecture 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.

Big-O Notation: Asymptotic O-notation ("big-O") provides us with a way to simplify the messy functions that often arise in analyzing the running times of algorithms. The purpose of the notation is to allow us to ignore less important elements, such as constant factors, and focus on important issues, such as the growth rate for large values of n. Here are some typical examples of big-O notation. For clarity, in each case, we have underlined the term that has the fastest growth rate.

$$\begin{array}{rcll} f_1(n) & = & 43n^2\log^4 n + \underline{12n^3\log n} + 52n\log n & \in\ O(n^3\log n) \\ f_2(n) & = & \underline{15n^2} + 7n\log^3 n & \in\ O(n^2) \\ f_3(n) & = & 3n + 4\log_5 n + \underline{91n^2} & \in\ O(n^2). \end{array}$$

Formally, $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that, $f(n) \leq c \cdot g(n)$, for all $n \geq n_0$. Thus, big-O notation can be thought of as a way of expressing a sort of *fuzzy* "$\leq$" relation between functions, where by fuzzy, we mean that constant factors are ignored and we are only interested in what happens as n tends to infinity.

This formal definition is often rather awkward to work with. Perhaps a more intuitive form is based on the notion of limits. An equivalent definition is that $f(n)$ is $O(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) \geq c$, for some constant

$c \geq 0$. For example, if $f(n) = 15n^2 + 7n\log^3 n$ and $g(n) = n^2$, we have $f(n)$ is $O(g(n))$ because

$$\begin{aligned} \lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \left(\frac{15n^2 + 7n\log^3 n}{n^2} \right) = \lim_{n\to\infty} \left(\frac{15n^2}{n^2} + \frac{7n\log^3 n}{n^2} \right) \\ &= \lim_{n\to\infty} \left(15 + \frac{7\log^3 n}{n} \right) = 15. \end{aligned}$$

In the last step of the derivation, we have used the important fact that $\log n$ raised to any positive power grows asymptotically more slowly that n raised to any positive power. The following two facts about limits are useful:

- For $a, b > 0$, $\lim_{n\to\infty} \frac{(\log n)^a}{n^b} = 0$ (polynomials grow faster than polylogs).
- For $a > 0$ and $b > 1$, $\lim_{n\to\infty} \frac{n^a}{b^n} = 0$ (exponentials grow faster than polynomials).
- For $a, b > 1$, $\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = c \neq 0$ (logarithm bases do not matter).
- For $1 < a < b$, $\lim_{n\to\infty} \frac{a^n}{b^n} = 0$ (exponent bases do matter).

Other Asymptotic Forms: Big-O notation has a number of relatives, which are useful for expressing other sorts of relations. These include Ω ("big-omega"), Θ ("theta"), o ("little-oh"), ω ("little-omega"). Let c denote an arbitrary positive constant (not 0 or ∞). These have the following interpretations:

Notation	**Relational Form**	**Limit Definition**
$f(n)$ is $o(g(n))$	$f(n) \prec g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n)$ is $O(g(n))$	$f(n) \preceq g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ or 0
$f(n)$ is $\Theta(g(n))$	$f(n) \approx g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$
$f(n)$ is $\Omega(g(n))$	$f(n) \succeq g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ or ∞
$f(n)$ is $\omega(g(n))$	$f(n) \succ g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Throughout this course, we will not worry about proving these facts, and will instead rely on a fairly intuitive understanding of asymptotic notation.

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 $n \geq 0$,

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

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 $n \geq 0$,

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

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 $n \geq 0$,

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

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

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

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 $n \geq 0$,

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

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, $\langle a_1, a_2, \ldots, a_n \rangle$. For each element a_i, for $1 \leq i \leq n$, we want to know the index of the rightmost element of the sequence $\langle a_1, a_2, \ldots, a_{i-1} \rangle$ 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 ith telephone pole in a sequence, and if we shoot a bullet horizontally to the left from the top of the ith pole, we want to know which pole we will hit first. (See Fig. 2 below.)

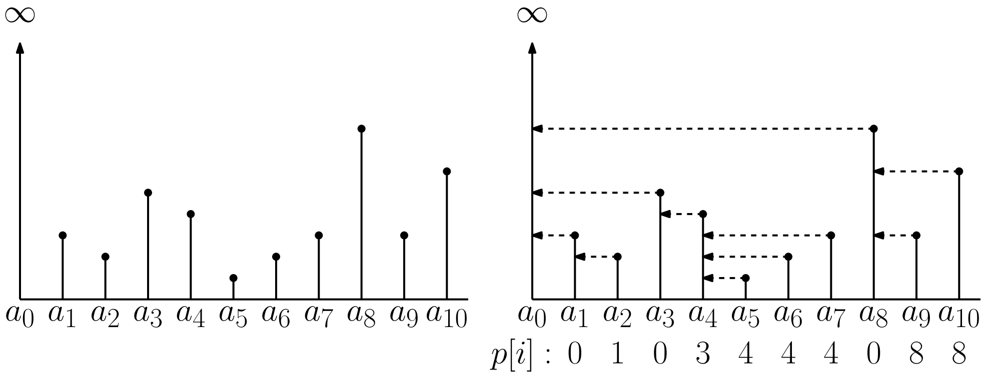

∞
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
∞
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
$p[i]$: 0 1 0 3 4 4 4 0 8 8

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 ith 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:

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

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.

Note that most simple iterative algorithms tend to have polynomial running times where the exponent is an integer, such as $O(n)$, $O(n^2)$, $O(n^3)$, and so on. When you see an algorithm with a noninteger exponent, it is often the result of applying a sophisticated divide-and-conquer algorithm. A famous example of this is Strassen's matrix multiplication algorithm, which has a running time of (roughly) $O(n^{\log_2 7}) = O(n^{2.8074})$. Currently, the best known algorithm for matrix multiplication runs in time $O(n^{2.3727})$.

Lecture 4: Greedy Algorithms for Scheduling

Greedy Algorithms: In an *optimization problem*, we are given an input and asked to compute a structure, subject to various constraints, in a manner that either minimizes cost or maximizes profit. Such problems arise in many applications of science and engineering. Given an optimization problem, we are often faced with the question of whether the problem can be solved efficiently (as opposed to a brute-force enumeration of all possible solutions), and if so, what approach should be used to compute the optimal solution?

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 a 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.

The greedy approach works for a number of optimization problems, including some of the most fundamental optimization problems in computer science (minimum spanning trees, for example). Thus, this is an important algorithm design technique to understand. Even when greedy algorithms do not produce the optimal solution, they often provide fast heuristics (nonoptimal solution strategies) and are often used in finding good approximations.

In this lecture we will discuss some examples of simple scheduling problems that have efficient greedy solutions.

Interval Scheduling: Scheduling problems are among the most fundamental optimization problems. Interval scheduling is one of the simplest formulations. We are given a set $R = \{1, 2, \ldots, n\}$ 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 *conflict* if their start-finish intervals overlap, that is, $[s_i, f_i] \cap [s_j, f_j] \neq \emptyset$. (We do not allow finish time of one request to overlap the start time of another one, but this is easily remedied in practice. For example, a lecture might run from 1:00pm to 1:59pm, and the next runs from 2:00pm to 2:59pm.) Here is a formal problem definition.

Interval scheduling problem: Given a set R of n activities with start-finish times $[s_i, f_i]$ for $1 \leq i \leq n$, determine a subset of R of maximum cardinality consisting of activities that are mutually non-conflicting.

An example of an input and two possible (optimal) solutions is given in Fig. 3. Notice that goal here is maximum *number* of activities. There are many other criteria that could be used in practice. For example, we might want to maximize the amount of time the resource is utilized or we might assign weights to the activities and seek to maximize the weighted sum of scheduled activities.

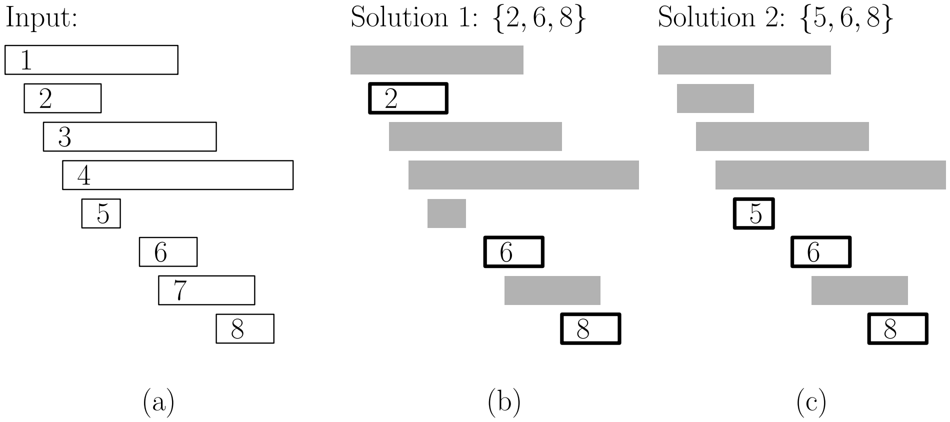


Fig. 3: An input and two possible solutions to the interval scheduling problem.

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 easy to see that this will not be optimal in general. A single very long activity with an early start time would 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 conflict with any previously scheduled activities. Although this may seem like a reasonable strategy, this also turns out to be nonoptimal. (For example, two long nonconflicting activities might have a short activity that overlaps both of them. The algorithm would pick the one short one, thus knocking out both of the long activities.)

Lowest Conflict Activity First: Counterexamples to the previous stratgy 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 other 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, finally, 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 conflict with this one, and schedule the next one that has the earliest finish time, and so on. Call this *Earliest Finish First*. A very rough pseudo-code description for the algorithm is presented below. The output is the list A of scheduled activities.

The above pseudo-code is a bit too sketchy, because it is not quite clear how to implement it. Letting $n = |R|$, a naive implementation would take $O(n^2)$ time. The algorithm can be implemented to run in $O(n \log n)$ time, however. To do this, first sort the activities in increasing order of finishing time. This takes $O(n \log n)$ time. The outer loop considers the task in increasing order of finish times. Each time we schedule a new activity, we maintain the current finish time of this task, call it f. Now, as we consider each successive activity s_i, we perform the following test. If $s_i \leq f$, then the current activity conflicts with the last activity scheduled, and we simply ignore it. On the other hand, if $s_i > f$, then then current activity does not conflict, and we schedule it and set $f = f_i$. Thus, after sorting, we can process each of the remaining activities in $O(1)$ time each, for a total running time of $O((n \log n) + n) = O(n \log n)$.

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

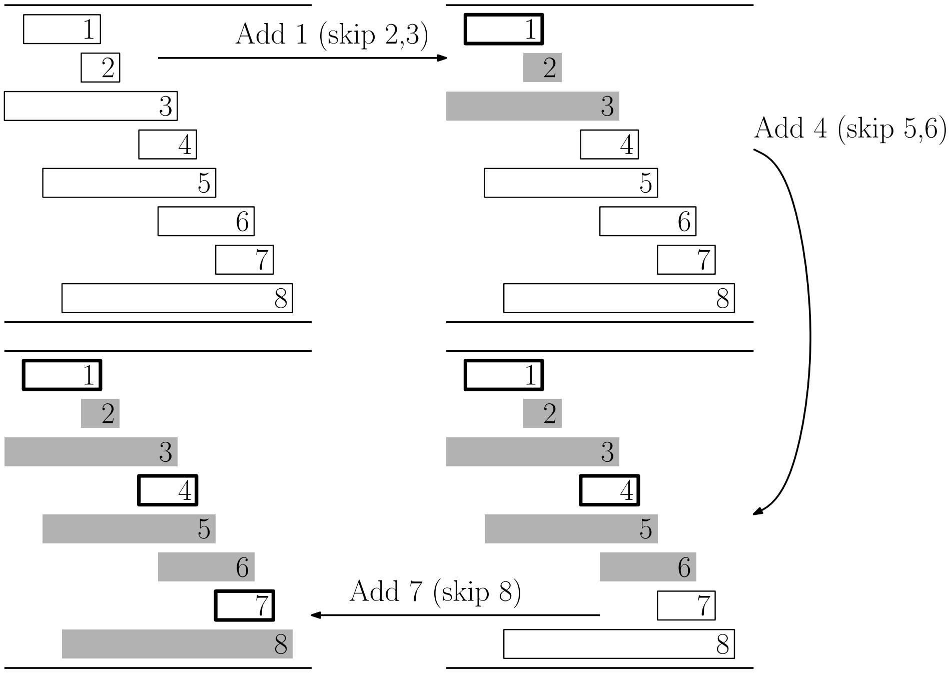


Fig. 4: An example of the greedy algorithm for interval scheduling. The final schedule is $\{1, 4, 7\}$.

Fig. 4 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 conflicts with Activities 2 and 3. Then Activity 4 is scheduled. It conflicts 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.

Correctness: The algorithm's correctness involves two issues. First, is this a valid schedule (in the sense that no scheduled activities conflict) and second, is this schedule optimal (having the maximum number of activities)?

Claim: The greedy algorithm produces a valid schedule.

Proof: Each time we add an activity to our schedule, we remove all conflicting requests from R (basically, we skip over them), therefore, no two conflicting activities can appear in our schedule.

Next, we establish 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 O and let G be the greedy schedule produced by the algorithm. If $O = G$, we are done. Otherwise, we will construct a new "optimal" schedule O' that is more similar to G than O is. By repeating this, eventually we will converge to G.

First, order the activities in increasing order of finish time. Let $O = \langle x_1, x_2, \ldots, x_k\rangle$ be the activities of O. Since O is not the same as the greedy schedule, consider the first activity x_j where these two schedules differ. That is, we have:

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

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.

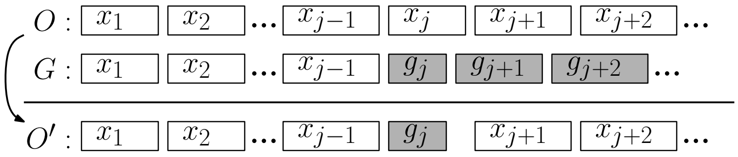


Fig. 5: Proof of optimality for the greedy schedule.

Consider the modified "greedier" schedule O' that results by replacing x_j with g_j in the schedule O (see Fig. 5). That is, $O' = \langle x_1, \ldots, x_{j-1}, g_j, x_{j+1}, \ldots, x_k\rangle$.

We assert that this is also a feasible schedule. The reason is that g_j cannot conflict with the earlier activities (since G is a feasible schedule). Also, it cannot not conflict with later activities (because, by definition, g_j finishes no later than when x_j finishes). Thus, this new "greedier" schedule O' is valid, and, since it has the same number of activities as O, it is also optimal. By repeating this process, we will eventually convert O into G, without decreasing the number of activities. It follows that G is also optimal.

Interval Partitioning: Next, let us consider a variant of the above problem. In interval scheduling, we assumed that there was a single exclusive resource, and our objective was to schedule as many nonconflicting activities as possible on this resource. Let us consider a different formulation, where instead we have an infinite number of possible exclusive resources to use, and we want to schedule *all* the activities using the smallest number resources.

More formally, we are given a collection of activity requests, each with a start and finish time. As before, let $R = \{1, 2, \ldots, n\}$ of n *activity requests*, and let $[s_i, f_i]$ denote the start-finish time of the ith request. Our objective is to find the smallest number d, such that it is possible to partition R into d disjoint subsets $R_1, \ldots, R_d$, such that the events of R_j are nonconflicting, for each j, $1 \leq j \leq d$. For example, we can think of R as representing class-room times, and d represents the number of lecture halls. We want to determine the minimum number of lecture halls, such that we can schedule all the activities in all the lecture halls.

We can view this as a *coloring problem*. In particular, we want to assign colors (positive integers) to the activities such that two conflicting activities must have different colors. (In our example, the colors are rooms, and two lectures at the same time must be assigned to different class rooms.) Our objective is to find the minimum number d, such that it is possible to color each of the activities in this manner.

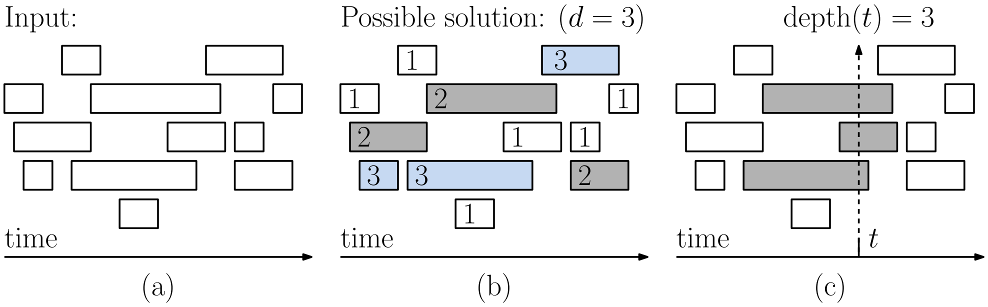


Fig. 6: Interval partitioning: (a) input, (b) possible solution, and (c) depth.

We refer to the subset of activities that share the same color as a *color class*. The activities of each color class are assigned to the same room. (For example, in Fig. 6(a) we give an example with $n = 12$ activities and in (b) show an assignment involving $d = 3$ colors. Thus, the six activities labeled 1 can be scheduled in one room, the three activities labeled 2 can be put in a second room, and the three activities labeled 3 can be put in a third room.)

In general, coloring problems are hard to solve efficiently. However, due to the simple nature of intervals, it is possible to solve the interval partitioning problem quite efficiently by a simple greedy approach. The greedy strategy is to assign each activity the smallest available color that does not conflict with the previously colored activities. The algorithm is presented in the following code block.

(The solution given in Fig. 6(b) comes about by running the above algorithm.) With it's two nested loops, it is easy to see that the algorithm's running time is $O(n^2)$. If we relax the requirement that the color be the smallest available color (instead allowing any available color), it is possible to reduce this to $O(n)$ time with a bit of added cleverness.[2]

To see whether you really understand the algorithm, ask yourself the following question. Why is sorting of the activities essential to the algorithm's correctness? In particular, come up with a set of activities and a method of ordering them so that the above approach will fail to produce the minimum number of colors for your order.

[2] Rather than have the for-loop iterate through just the start times, sort both the start times and the finish times into one large list of size $2n$. Each entry in this sorted lists stores a record consisting of the type of event (start or finish), the index of the activity (a number $1 \leq i \leq n$), and the time of the event (either s_i or f_i). The algorithm visits each time instance from left to right, and while doing this, it maintains a stack containing the collection of *available colors*. It is not hard to show that each of the $2n$ events can be processed in $O(1)$ time. We leave the implementation details as an exercise. The total running time to sort the records is $O((2n)\log(2n)) = O(n \log n)$, and the total processing time is $2n \cdot O(1) = O(n)$. Thus, the overall running time is $O(n \log n)$.

```
greedyPartition(R) {      // R holds the set of all activity requests
    sort activities by increasing start times -- (x1, ..., xn)
    for i = 1 to n do {
        E = emptyset      // E stores set of excluded colors for xi
        for j = 1 to i-1 do {
          if (xj conflicts with xi) add color[i] to E
        }
        Let c be the smallest color not in E
        color[xi] = c
    }
    return color[1..n]
}
```

Correctness: Let us now establish the correctness of the greedy interval partitioning algorithm. We first observe that the algorithm never assigns the same color to two conflicting activities. This is due to the fact that the inner for-loop eliminates the colors of all preceding conflicting tasks from consideration. Thus, the algorithm produces a valid coloring. The question is whether it produces an optimal coloring, that is, one having the minimum number of distinct colors.

To establish this, we will introduce a helpful quantity. Let t be any time instant. Define $\mathrm{depth}(t)$ to be the number of activities whose start-finish interval contains t (see Fig. 6(c)). Given an set $R = \{x_1, \ldots, x_n\}$ of activity requests, define $\mathrm{depth}(R)$ to be the maximum depth over all possible values of t. Since the activities that contribute to $\mathrm{depth}(t)$ conflict with one another, clearly we need at least this many resources to schedule these activities. Therefore, we have the following:

Claim: Given any instance R of the interval partitioning problem, the number of resources needed is at least $\mathrm{depth}(R)$.

This claim states that, if d denotes the minimum number of colors in any schedule, we have $d \geq \mathrm{depth}(R)$. This does not imply, however, that this bound is necessarily achievable. But, in the case of interval partitioning, we can show that the depth bound is achievable, and indeed, the greedy algorithm achieves this bound.

Claim: Given any instance R of the interval partitioning problem, the number of resources produced by the greedy partitioning algorithm is at most $\mathrm{depth}(R)$.

Proof: It will simplify the proof to assume that all start and finish times are distinct. (Let's assume that we have perturbed them infinitesimally to guarantee this.) We will prove a stronger result, namely that at any time t, the number of colors assigned to the activities that overlap time t is at most $\mathrm{depth}(t)$. The result follows by taking the maximum over all times t.

To see why this is true, consider an arbitrary start time s_i during the execution of the algorithm. For an infinitesimally small $\varepsilon > 0$, let $t = s_i - \varepsilon$ denote a time that is just before s_i. (That is, there are no events, start or finish, occurring between t and s_i.) Let d denote the depth at time t. By our hypothesis, just prior to time s_i, the number of colors being used is at most the current depth, which is d. Thus, when time s_i is considered, the depth increases by 1 to $d + 1$. Because at most d colors are in use prior to time s_i, there exists an unused color among the first $d + 1$ colors. Therefore, the total number of colors used at time s_i is $d + 1$, which is not greater than the total depth.

There are many variants of coloring problems, where color assignments are subject to a set of given constraints. We will discuss these later in the semester. Most such formulations are quite hard to solve, and the general problem of computing an optimal coloring of a set of objects subject to an arbitrary set of constraints is NP-hard.

Scheduling to Minimize Lateness: Finally, let us discuss a problem of scheduling a set of tasks where the start and finish times of the tasks are not specified. Let us assume we have a single exclusive resource, and we have n requests for use of the resource. Each task x_i is associated with a *deadline* d_i, which indicates the time by which the task must be completed and a *duration* t_i, which indicates how long the task takes to perform. Thus, the input consists of n pairs (t_i, d_i).

Our goal is to schedule all the tasks so that all the deadlines are satisfied, but of course, two tasks cannot use the resource at the same time. It might be that there are simply too many tasks to satisfy all their deadlines. If so, we define the *lateness* to be the amount by which we exceed the task's deadline. More formally, suppose that we assign task i to start at time $s(i)$. Then this task finishes at time $f(i) = s(i) + t_i$. (For simplicity, we assume that the instant that one task ends, the next one can start. Thus, if task j follows task i, then we allow that $s_j = f_i$. This way, we don't need to insert a tiny time increment to separate the tasks.) We say that task i is *late* if $f(i) > d_i$ and its *lateness* is $\ell_i = \max(0, f(i) - d_i)$.

Our objective is to compute a schedule that minimizes the overall lateness. What do we mean by this? There are a few natural definitions. For example we could chose to minimize:

Maximum lateness: $\max_{1 \le i \le n} \ell_i$

Average lateness: $(1/n) \sum_{i=1}^{n} \ell_i$

Both of these are reasonable objectives. We will focus here on minimizing *maximum lateness*. An example is shown in Fig. 7. The input is given in Fig. 7(a), where the duration is shown by the length of the rectangle and the deadline is indicated by an arrow pointing to a vertical line segment. A possible solution is shown in Fig. 7(b). The width of each red shaded region indicates the amount by which the task exceeds its allowed deadline. The longest such region yields the maximum lateness.

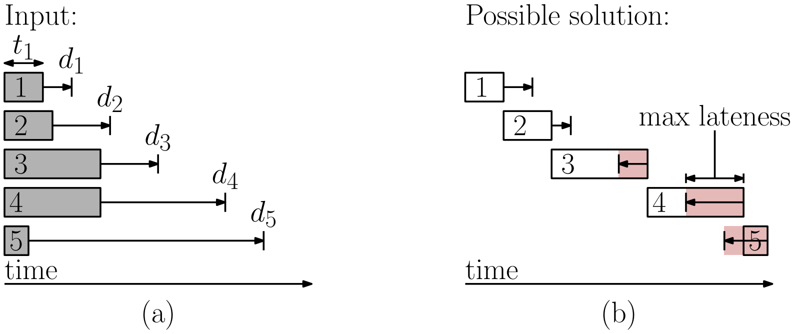


Fig. 7: Scheduling to minimize lateness: (a) input, (b) possible solution.

Greedy Algorithm: Let us present a greedy algorithm for computing a schedule that minimizes maximum lateness. As before, we need to find a quantity upon which to base our greedy choices. Here are some ideas that don't work:

Smallest duration first: Sort tasks by increasing order of duration t_i and schedule them in this order. It is easy to see that this will not give an optimal result, however, because it fails to consider deadlines. A very short job with a deadline way in the future can safely be put off until later in order that more time critical tasks are performed first.

Smallest slack-time first: Define the *slack time* of task x_i as $d_i - t_i$. This statistic indicates how long we can safely wait before starting a task. It would seem intuitively smart to schedule tasks in increasing order of slack-time, but this can also be shown to be suboptimal. Consider, for example a two-task instance where $(t_1, d_1) = (1, 2)$ and $(t_2, d_2) = (10, 10)$. The first task has slackness 1 and the second has slackness 0. But, running the jobs in order of slack time (first task 2 then task 1) would cause task 1 to have a lateness of $11 - 2 = 9$. Running them in the opposite order would result in a maximum lateness of only $11 - 10 = 1$.

So what is the right solution? The best strategy turns out to process the task with the shortest deadline first. That is, sort the tasks in increasing order of d_i, and run them in this order. This strategy is called *shortest deadline first*. It is counterintuitive, because (like smallest duration first) it completely ignores part of the input, namely the running times. Nonetheless, we will show that this is the best possible. The pseudo-code is presented in the following code block.

Greedy Schedule for Minimizing Lateness

```
greedySchedule(T) {     // T holds the set of all tasks
    sort tasks by increasing deadline (d[1] <= ... <= d[n])
    f = 0               // f is the finishing time of last task
    for i = 1 to n do {
        assign task i to start at s[i] = f and finish at f[i] = f + t[i]
        f = f[i]
    }
    return the sequence (s[1],f[1]) ... (s[n],f[n])
}
```

The solution shown in Fig. 7(b) is the result of this algorithm. As before, it is easy to see that this algorithm produces a valid schedule, since we never start a new job until the previous job has been completed. Second, observe that the algorithm's running time is $O(n \log n)$, which is dominated by the time to sort the tasks by their deadline. After this, the algorithm runs in $O(n)$ time.

Correctness: All that remains is to show that the greedy algorithm produces an optimal schedule, that is, one that minimizes the maximum lateness. It would be nice if we could show that every optimal schedule is the same as the greedy schedule, but this is certainly not going to be true. (There may be optimal schedules that are quite different from the greedy schedule, simply because there are tasks whose deadlines are so far in the future that their exact order in the schedule is not critical.) As with the interval scheduling problem, our approach will be to show that is it possible to "morph" any optimal schedule to look like our greedy schedule. In the morphing process, we will show that schedule remains valid, and the maximum lateness can never increase, it can only remain the same or decrease.

To begin, we observe that our algorithm has no *idle time* in the sense that the resource never sits idle during the running of the algorithm. It is easy to see that by moving tasks up to fill in any idle times, we can only reduce lateness. Thus, we have the following.

Claim: There is an optimal schedule with no idle time.

Henceforth, we assume that all schedules are idle free. Let G be the schedule produced by the greedy algorithm, and let O be any optimal schedule. If $G = O$, then greedy is optimal, and we are done. Otherwise, O must contain at least one *inversion*, that is, at least one pair of tasks that have not been scheduled in increasing order of deadline. Let us consider the first instance of such an inversion. That is, let x_i and x_j be the first two consecutive tasks in the schedule O such that $d_j < d_i$. We have:

(a) The schedules O and G are identical up to these two tasks

(b) $d_j < d_i$ (and therefore x_j is scheduled before x_i in schedule G)

(c) x_i is scheduled before x_j in schedule O

This is illustrated in Fig. 8. We will show that by swapping x_i and x_j in O, the maximum lateness cannot increase. Combining this with an inductive argument establishes the optimality of G. In particular, we can start with any optimal schedule, repeatedly search for the first inversion, and then eliminate it by swapping without affecting the schedule's optimality. Eventually the optimal schedule will be morphed into the greedy schedule, implying that greedy is optimal.

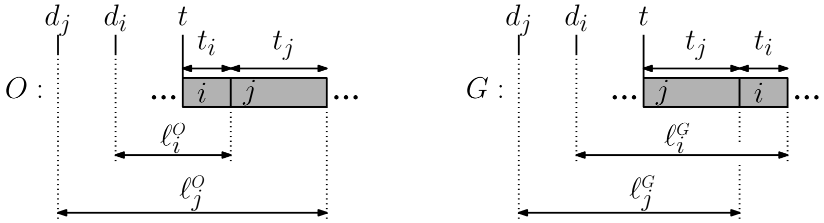


Fig. 8: Optimality of the greedy scheduling algorithm for minimizing lateness.

The reason that swapping x_i and x_j in O does not increase lateness can be seen intuitively from the figure. The lateness is reflected in the length of the horizontal arrowed line segments in Fig. 8. From the figure, it is evident that the worst lateness involves x_j in schedule O (labeled ℓ_j^O). Unfortunately, a picture is not a formal argument. So, let us see if we put this intuition on a solid foundation.

First, let us define some notation. The lateness of task i in schedule O will be denoted by ℓ_i^O and the lateness of task j in O will be denoted by ℓ_j^O. Similarly, let ℓ_i^G and ℓ_j^G denote the respective latenesses of tasks i and j in schedule G. Because the two schedules are identical up to these two tasks, and because there is no slack time in either, the first of the two tasks starts at the same time in both schedules. Let t denote this time (see Fig. 8). In schedule O, task i finishes at time $t + t_i$ and (because it needs to wait for task i to finish) task j finishes as time $t + (t_i + t_j)$. The lateness of each of these tasks is the maximum of 0 and the difference between the finish time and the deadline. Therefore, we have

$$\ell_i^O \;=\; \max(0, t + t_i - d_i) \qquad \text{and} \qquad \ell_j^O \;=\; \max(0, t + (t_i + t_j) - d_j).$$

Applying a similar analysis to G, we can define the latenesses of tasks i and j in G as

$$\ell_i^G \;=\; \max(0, t + (t_i + t_j) - d_i) \qquad \text{and} \qquad \ell_j^G \;=\; \max(0, t + t_j - d_j).$$

The "max" will be a pain to carry around, so to simplify our formulas we will exclude reference to it. (You are encouraged to work through the proof with the full and proper definitions.)

Given the individual latenesses, we can define the maximum lateness contribution from these two tasks for each schedule as

$$L^O \;=\; \max(\ell_i^O, \ell_j^O) \qquad \text{and} \qquad L^G \;=\; \max(\ell_i^G, \ell_j^G).$$

Our objective is to show that by swapping these two tasks, we do not increase the overall lateness. Since this in the only change, it suffices to show that $L^G \le L^O$. To prove this, first observe that, t_i and t_j are nonnegative and $d_j < d_i$ (and therefore $-d_j > -d_i$). Recalling that we are dropping the "max", we have

$$\ell_j^O \;=\; t + (t_i + t_j) - d_j \;>\; t + t_i - d_i \;=\; \ell_i^O.$$

Therefore, $L^O = \max(\ell_i^O, \ell_j^O) = \ell_j^O$. Since $L^G \;=\; \max(\ell_i^G, \ell_j^G)$, in order to show that $L^G \le L^O$, it suffices to show that $\ell_i^G \le L^O$ and $\ell_j^G \le L^O$. By definition we have

$$\ell_i^G \;=\; t + (t_i + t_j) - d_i \;<\; t + (t_i + t_j) - d_j \;=\; \ell_j^O \;=\; L^O,$$

and

$$\ell_j^G \;=\; t + t_j - d_j \;\le\; t + (t_i + t_j) - d_j \;=\; \ell_j^O \;=\; L^O.$$

Therefore, we have $L^G = \max(\ell_i^G, \ell_j^G) \le L^O$, as desired. In conclusion, we have the following.

Claim: The greedy scheduling algorithm minimizes maximum lateness.

Lecture 5: Graphs, Digraphs, and Basic Algorithms

Graphs and Digraphs: Continuing our presentation of greedy algorithms, we will next discuss greedy algorithms for some common problems on graphs. Basic graph concepts have been presented in earlier courses, and so we will present a very quick review of the basic material in today's lecture.

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 "graph" means an undirected graph, and directed graphs are often called *digraphs* (see Fig. 9). Graphs and digraphs provide a flexible mathematical model for numerous application problems involving binary relationships between a discrete collection of object. Examples of graph applications include *communication* and *transportation networks*, *social networks*, *logic circuits*, *surface meshes* used for shape description in computer-aided design and geographic information systems, *precedence constraints* in scheduling systems.

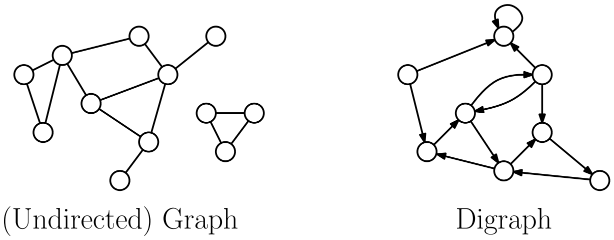


Fig. 9: Graphs and digraphs.

Definition: An *undirected graph* (or simply *graph*) $G = (V, E)$ consists of a finite set V and a set E of *unordered pairs* of distinct vertices.

Definition: A *directed graph* (or *digraph*) $G = (V, E)$ consists of a finite set V and a set E of *ordered pairs* of vertices.

The elements of V are called *vertices* or *nodes* and the elements of E are called *edges* or *arcs*. Observe that multiple edges between the same two vertices are not allowed, but in a directed graph, it is possible to have two oppositely directed edges between the same pair of vertices. For undirected graphs, *self-loop* edges are not allowed, but they are allowed for directed graphs. Directed graphs and undirected graphs are different objects mathematically. Certain notions (such as path) are defined for both, but other notions (such as connectivity and spanning trees) may be defined only for one.

Graph and Digraph Terminology: Given an edge $e = (u, v)$ in a digraph, we say that u is the *origin* of e and v is the *destination* of e. Given an edge $e = \{u, v\}$ in an undirected graph, u and v are called the *endpoints* of e. The edge e is *incident* on (meaning that it touches) both u and v. Given two vertices in a graph or digraph, we say that vertex v is *adjacent* to vertex u if there is an edge $\{u, v\}$ (for graphs) or (u, v) (for digraphs).

In a digraph, the number of edges coming out of v is called its *out-degree*, denoted out-deg(v), and the number of edges coming in is called its *in-degree*, denoted in-deg(v). In an undirected graph we just talk about the *degree* of a vertex as the number of incident edges, denoted $\deg(v)$.

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

In a graph:

Number of edges: $0 \leq m \leq \binom{n}{2} = n(n-1)/2 \in O(n^2)$.

Sum of degrees: $\sum_{v \in V} \deg(v) = 2m$.

In a digraph:

Number of edges: $0 \le m \le n^2$.

Sum of degrees: $\sum_{v \in V}$ in-deg$(v) = \sum_{v \in V}$ out-deg$(v) = m$.

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

Paths and Cycles: A *path* in a graph or digraph is a sequence of vertices $\langle v_0, \ldots, v_k \rangle$ such that (v_{i-1}, v_i) is an edge for $i = 1, \ldots, k$. The *length* of the path is the number of edges, k. A path is *simple* if all vertices and all the edges are distinct. A *cycle* is a path containing at least one edge and for which $v_0 = v_k$. A cycle is *simple* if its vertices (except v_0 and v_k) are distinct, and all its edges are distinct.

A graph or digraph is said to be *acyclic* if it contains no simple cycles. An acyclic connected graph is called a *free tree* or simply *tree* for short (see Fig. 10). (The term "free" is intended to emphasize the fact that the tree has no root, in contrast to a *rooted tree*, as is usually seen in data structures.) An acyclic undirected graph (which need not be connected) is a collection of free trees, and is called a *forest*. An acyclic digraph is called a *directed acyclic graph*, or *DAG* for short (see Fig. 10).

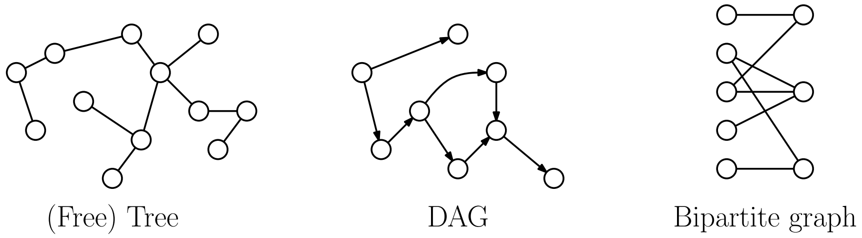


Fig. 10: Illustration of common graph terms.

A *bipartite graph* is one in which the vertices of a graph can be partitioned into two disjoint subsets, denoted V_1 and V_2, such that all the edges have one endpoint in V_1 and one in V_2 (see Fig. 10). Note that every cycle in a bipartite graph contains an even number of edges.

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

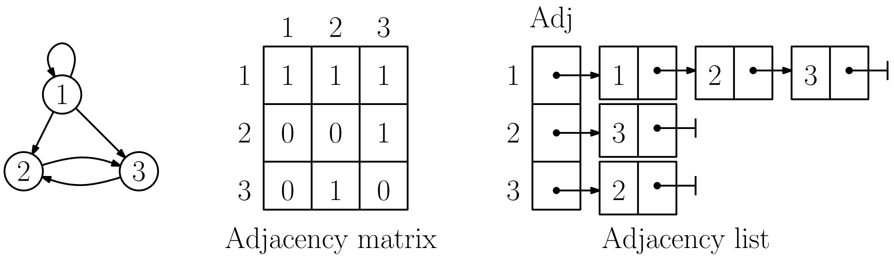
Representations of Graphs and Digraphs: There are two common ways of representing graphs and digraphs. First we show how to represent digraphs. Let $G = (V, E)$ be a digraph with $n = |V|$ and let $m = |E|$. We will assume that the vertices of G are indexed $\{1, 2, \ldots, n\}$.

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

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

(See Fig. 11.) If the digraph has weights we can store the weights in the matrix. For example if $(v, w) \in E$ then $A[v, w] = W(v, w)$ (the weight on edge (v, w)). If $(v, w) \notin E$ then generally $W(v, w)$ need not be defined, but often we set it to some "special" value, e.g. $A(v, w) = -1$, or ∞. (By ∞ we mean some number which is larger than any allowable weight.)

Adjacency List: An array $Adj[1 \ldots n]$ of pointers where for $1 \le v \le n$, $Adj[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements (see Fig. 11).

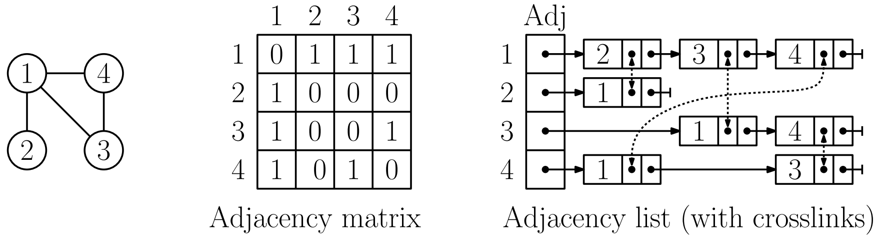


	1	2	3
1	1	1	1
2	0	0	1
3	0	1	0

Fig. 11: Adjacency matrix and adjacency list for digraphs.

We can represent undirected graphs using exactly the same representation, but we will store each edge twice. In particular, we representing the undirected edge $\{v, w\}$ by the two oppositely directed edges (v, w) and (w, v) (see Fig. 12). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

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



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

Fig. 12: Adjacency matrix and adjacency list for graphs.

An adjacency matrix requires $\Theta(n^2)$ storage, and an adjacency list requires $\Theta(n + m)$ storage. The n arises because there is one entry for each vertex in Adj. Since each list has out-deg(v) entries, when this is summed over all vertices, the total number of adjacency list records is $\Theta(m)$. For most applications, the adjacency list representation is standard.

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

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 marked as *undiscovered*. When a vertex is first encountered, it is marked as *discovered* (and is now part of the frontier). When we have finished processing a discovered vertex it becomes *finished*.

The search makes use of a first-in first-out (FIFO) *queue*. (Such a queue 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 *mark*$[u]$ (which stores one of the values "undiscovered," "discovered," or "finished"), *pred*$[u]$ which points to the vertex that discovered u, and $d[u]$, the distance from s to u. For a minimal implementation of BFS, the only quantity really needed is the mark. The other quantities are useful for computing shortest paths. The algorithm is presented in the code block below. A sample trace of the execution is shown in Fig. 13.

Breadth-First Search

```
BFS(G,s) {
    for each (u in V) {                     // initialization
        mark[u] = undiscovered
        d[u]    = infinity
        pred[u] = null
    }
    mark[s] = discovered                    // initialize source s
    d[s] = 0
    Q = {s}                                 // put s in the queue
    while (Q is nonempty) {
        u = dequeue from head of Q          // get next vertex from queue
        for each (v in Adj[u]) {
            if (mark[v] == undiscovered) {  // first time we have seen v?
                mark[v] = discovered        // ...mark v discovered
                d[v]    = d[u]+1            // ...set its distance from s
                pred[v] = u                 // ...and its parent
                append v to the tail of Q   // ...put it in the queue
            }
        }
        mark[u] = finished                  // 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?) The $d[v]$ values store the distances from s, as we prove next.

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

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

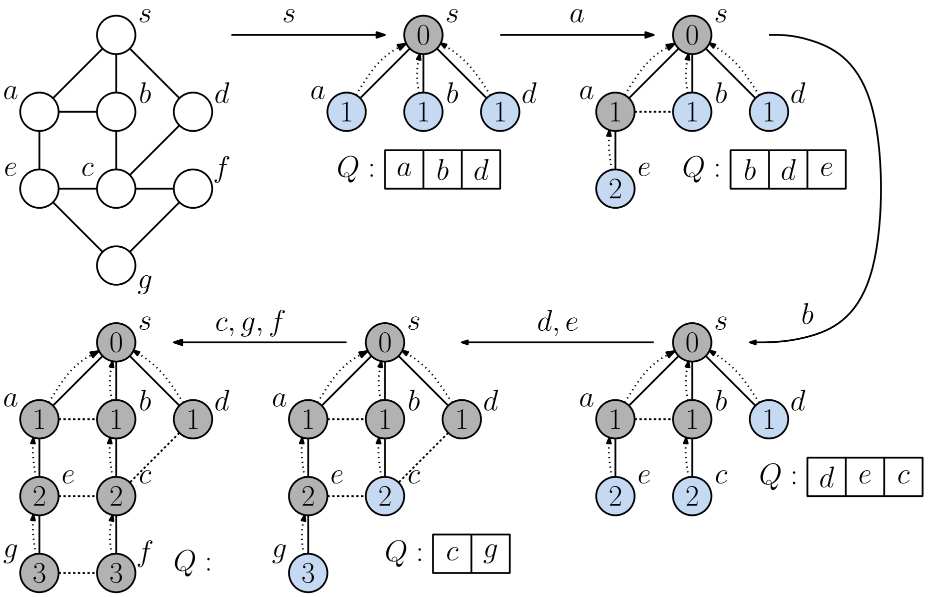


Fig. 13: Breadth-first search. (Tree edges are shown as solid lines, cross edges as dashed lines, and predecessor pointers as arrowed dotted lines.)

as desired. Because the vertices are processed in increasing order of distance from s, v will be discovered by a vertex that is on some shortest path from s to v.

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

$$T(n) \;=\; n + \sum_{u \in V} (\deg(u) + 1) \;=\; n + \left(\sum_{u \in V} \deg(u) \right) + n \;=\; 2n + 2m \;\in\; O(n + m).$$

The analysis is essentially the same for BFS on 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 only 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 mark for each vertex: undiscovered, discovered, finished. Additional information can be stored as part of the traversal process (discovery times, finish times, predecessor pointers), but we will focus on the most basic implementation of the algorithm. As with BFS, DFS induces a tree structure. The algorithm is shown in code block below, and illustrated in Fig. 14.

Depth-First Search

```
DFS(G) {                                      // main program
    for each (u in V) {                       // initialization
        mark[u] = undiscovered
    }
    for each (u in V)
        if (mark[u] == undiscovered)          // found an undiscovered vertex?
            DFSVisit(u)                       // ...start a new search here
}

DFSVisit(u) {                                 // perform a DFS search at u
    mark[u] = discovered                      // u has been discovered
    for each (v in Adj(u)) {
        if (mark[v] == undiscovered) {        // found an undiscovered neighbor
            DFSVisit(v)                       // ...visit it
        }
    }
    mark[u] = finished                        // we're done with u
}
```

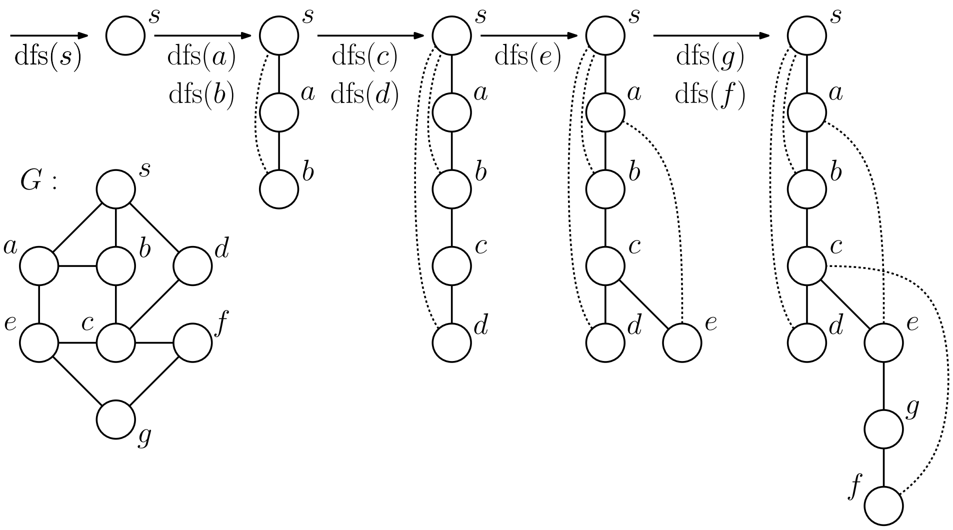


Fig. 14: Depth-First search tree.

Analysis: The running time of DFS is $O(n + m)$. We'll do the analysis for undirected graphs. This is somewhat harder to see than the BFS analysis, because the recursive nature of the algorithm obscures things. First observe that if we ignore the time spent in the recursive calls, the main DFS procedure runs in $O(n)$ time. 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 $O(1 + \deg(u))$ time (the "+1" is needed in case the degree is 0). Thus the total time used in the procedure is

$$T(n) \;=\; n + \sum_{u \in V} (1 + \deg(u)) \;=\; n + \left(\sum_{u \in V} \deg(u) \right) + n \;=\; 2n + m \;\in\; O(n + m).$$

A similar analysis holds if we consider DFS for digraphs.

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 undirected graphs the remaining edges of the graph are called *back edges*. (When performing DFS on directed graphs, there are two other types of edges that arise, forward edges and cross edges.) An important fact about back edges is that they always go between a node and one of its ancestors or one of its descendents. (Can you see why?)

Lecture 6: Dijkstra's Algorithm for Shortest Paths

Shortest Paths: Today we 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. (Thus, distance is the number of edges on a path.) Suppose that each edge $(u, v) \in E$ is associated with an edge weight $w(u, v)$. We define the *length* of a path to be the sum of weights along the edges of the path. We define the *distance* between any two vertices u and v to be the minimum length of any path between the vertices. We will denote this by $\delta(u, v)$. Because a vertex is joined to itself by an empty path, we have $\delta(u, u) = 0$, for all $u \in V$.

There are many ways in which to formulate the shortest path problem. For example, we may want be interested in the shortest path between a single source vertex and a single sink vertex, or we might be given a collection of source-sink pairs. Alternately, in the *single source* shortest-path problem, we are given a source vertex $s \in V$, and we wish to compute shortest paths to all other vertices. (The *single-sink* is a simple variant, which can be obtained by reversing all the edge directions.) Finally, the *all-pairs* shortest path problem involves computing the distances between all pairs of vertices. Of course, in addition to computing the distance between vertices, we will want to provide some intermediate structure that makes it possible to reconstruct the shortest path. Today, we will consider an algorithm for the single-source problem.

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

An important issue in the design of a shortest path algorithm is whether negative-valued edge weights are allowed. (Negative edges weights do not usually arise in transportation networks, but they can arise in financial transaction networks, where a transaction (edge) may result in either a lost or a profit.) In general, the shortest path problem is well defined, even if the graph has negative edge weights, provided that there are no negative cost cycles. (Otherwise you can make the path arbitrarily "short" by iterating forever around such a cycle.) Today, we will present a simple greedy algorithm for the single-source problem, which assumes that the edge weights are nonnegative. The algorithm, called *Dijkstra's algorithm*, was invented by the famous Dutch computer scientist, Edsger Dijkstra[3] in 1959. It is among the most famous algorithms in Computer Science.

In our presentation of the algorithm, we will stress the task of computing just the distance from the source to each vertex (not the path itself). As we did in the breadth-first search algorithm, it will be possible to make a minor modification to compute the paths themselves. As in BFS, we will use *predecessor link*, that point the

[3] Edsger Dijkstra was an important figure in Computer Science, who made a significant impact on approaches to programming, programming languages, and CS education. He was a passionate advocate of elegance in programming and once said, "Elegance is not a dispensable luxury but a quality that decides between success and failure". He also had a reputation for arrogance and not suffering fools lightly. One famous Computer Scientist said of Dijkstra, "You probably know that arrogance, in computer science, is measured in nano-dijkstras."

route back to the source. By reversing the resulting path, we can obtain the shortest path. Since we store one predecessor link per vertex, the total space needed is only $O(n)$.

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 $d[v]$. Intuitively $d[v]$ stores the length of the shortest path from s to v *that the algorithm currently knows of*. Indeed, there will always exist a path of length $d[v]$, but it might not be the ultimate shortest path. Initially, we know of no paths, so $d[v] = \infty$, and $d[s] = 0$. As the algorithm proceeds and sees more and more vertices, it updates $d[v]$ 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 sometimes 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 $d[u] + w(u, v)$ (see Fig. 17.) We should also remember that the shortest path to v passes through u, which we do by setting pred$[v]$ to u.

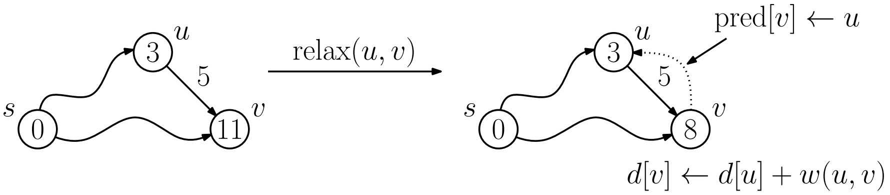


Fig. 15: Relaxation.

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 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 $V \setminus S$ to add to S. (If you haven't seen it before, the notation "$A \setminus B$" means the set A minus the elements of set B. Thus $V \setminus S$ consists of the vertices that are not in S.)

The set S can be implemented using an array of vertex marks. Initially all vertices are marked as "undiscovered," and we set mark$[v]$ = finished to indicate that $v \in S$.

How do we select which vertex among the vertices of $V \setminus S$ 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, we take the vertex of $V \setminus S$ for which $d[u]$ is minimum. That is, we 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 $V \setminus S$ in a *priority queue* (e.g. a heap), where the key value of each vertex u is $d[u]$. We will need to make use of three basic operations that are provided by the priority queue:

Build: Create a priority queue from a list of n elements, each with an associated key value.

Extract min: Remove (and return a reference to) the element with the smallest key value.

Decrease key: Given a reference to an element in the priority queue, decrease its key value to a specified value, and reorganize if needed.

For example, using a standard binary heap (as in heapsort) the first operation can be done in $O(n)$ time, and ther other two can be done in $O(\log n)$ time each. Dijkstra's algorithm is given in the code block below, and see Fig. 15 for an example.

Dijkstra's Algorithm

```
dijkstra(G,w,s) {
    for each (u in V) {                    // initialization
        d[u] = +infinity
        mark[u] = undiscovered
        pred[u] = null
    }
    d[s] = 0                               // distance to source is 0
    Q = a priority queue of all vertices u sorted by d[u]
    while (Q is nonEmpty) {                // until all vertices processed
        u = extract vertex with minimum d[u] from Q
        for each (v in Adj[u]) {
            if (d[u] + w(u,v) < d[v]) {    // relax(u,v)
                d[v] = d[u] + w(u,v)
                decrease v’s key in Q to d[v]
                pred[v] = u
            }
        }
        mark[u] = finished
    }
    [The pred pointers define an ‘‘inverted’’ shortest path tree]
}
```

Notice that the marking 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, recall that $n = |V|$ and $m = |E|$. We account for the time spent on each vertex after it is extracted from the priority queue. It takes $O(\log n)$ to extract this vertex from the queue. For each incident edge, we spend potentially $O(\log n)$ time if we need to decrease the key of the neighboring vertex. Thus the time is $O(\log n + \deg(u) \cdot \log n)$ time. The other steps of the update run in constant time. Recalling that the sum of degrees of the vertices in a graph is $O(m$, the overall running time is given by $T(n, m)$, where

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

Since G is connected, n is asymptotically no greater than m, so this is $O(m \log n)$.

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

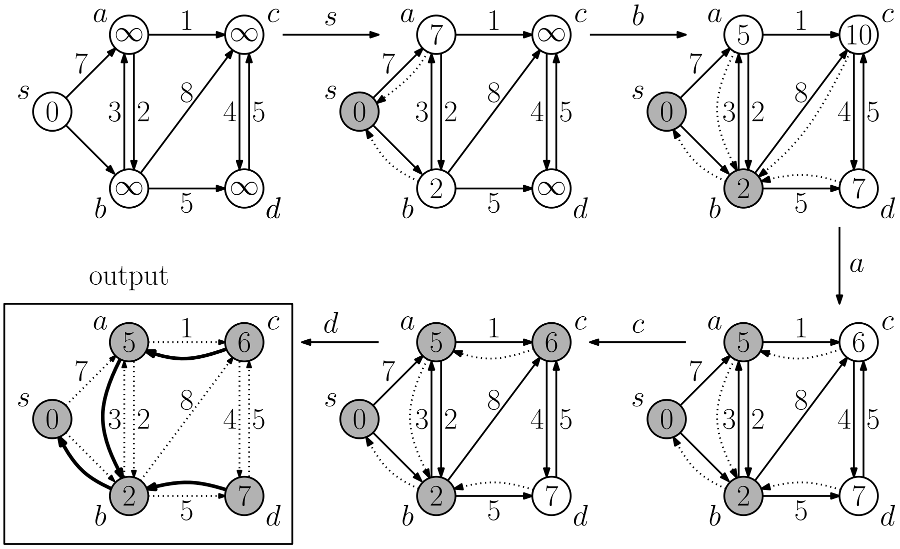


Fig. 16: Dijkstra's Algorithm example.

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 $u \in V \setminus S$, 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 $y \in V \setminus S$ (see Fig. 17). (Note that it may be that $x = s$ and/or $y = u$).

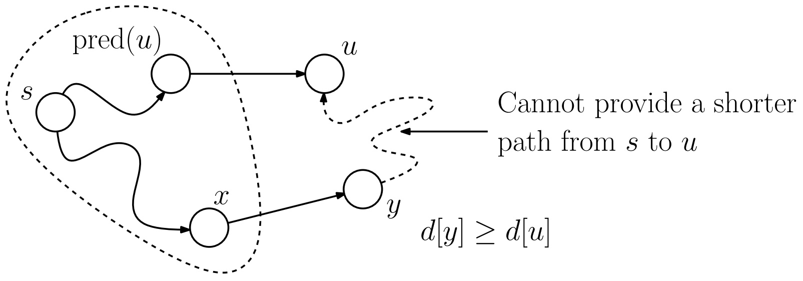


Fig. 17: Correctness of Dijkstra's Algorithm.

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.

Lecture 7: Minimum Spanning Trees and Kruskal's Algorithm

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 spanning tree shown in Fig. 18(a) is not a minimum spanning tree (in fact, it is a maximum weight spanning tree), while the other two are MSTs.

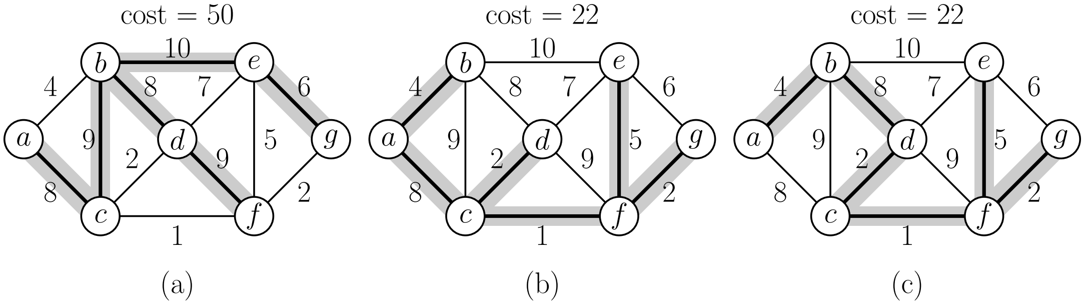


Fig. 18: Spanning trees (the middle and right are minimum spanning trees).

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

Lemma:

(i) A free tree with n vertices has exactly $n - 1$ edges.

(ii) There exists a unique path between any two vertices of a free tree.

(iii) 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 the input graph. 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 is a spanning tree. 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 $(u, v) \in E \setminus A$ 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 $S \subseteq V$. Here are a few useful definitions:

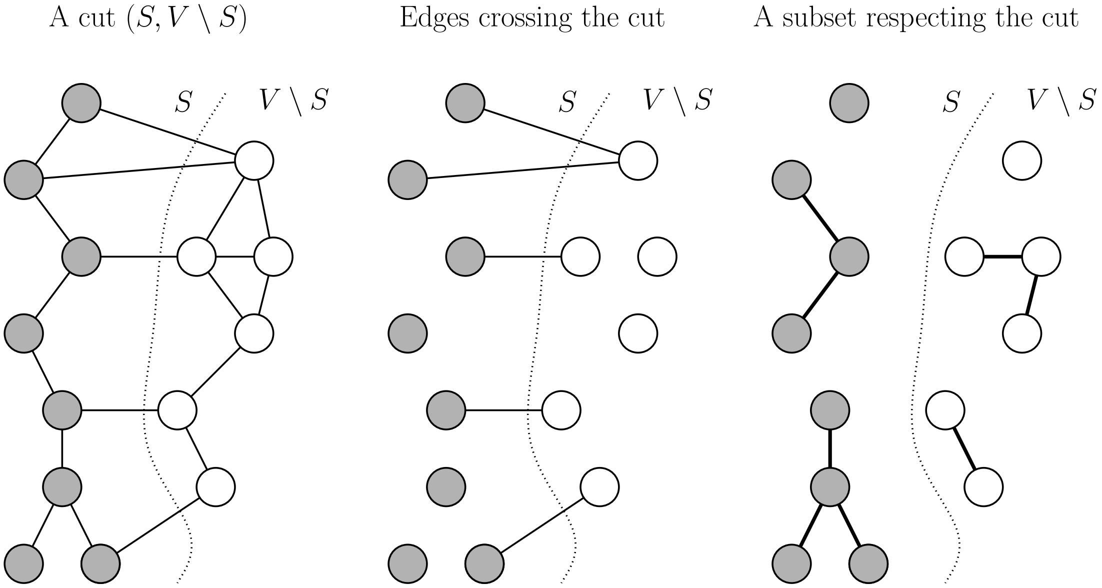


Fig. 19: MST-related terminology.

- A *cut* $(S, V \setminus S)$ 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 $V \setminus S$.
- 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 $(S, V \setminus S)$ 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.

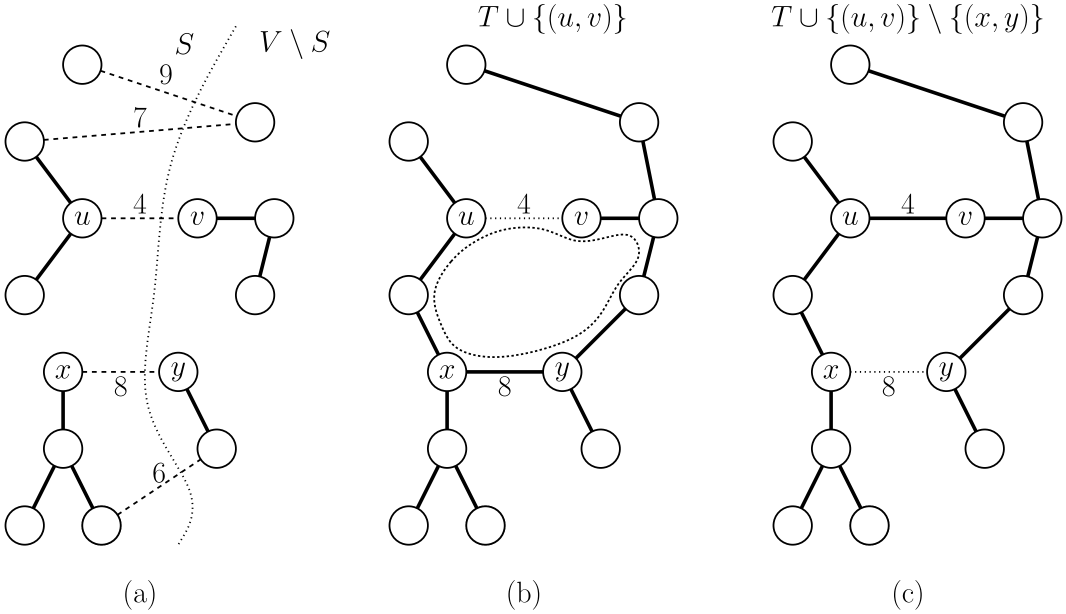


Fig. 20: Proof of the MST Lemma. Edge (u, v) is the light edge crossing cut $(S, V \setminus S)$.

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

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

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

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

Observe that this strategy leads to a correct algorithm. Why? Consider the edge (u, v) that Kruskal's algorithm seeks to add next, and suppose that this edge does not induce a cycle in A. Let A' denote the tree of the forest A that contains vertex u. Consider the cut $(A', V \setminus A')$. 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:

$\text{create}(u)$**:** Create a set containing a single item v.

$\text{find}(u)$**:** Find the set that contains a given item u.

$\text{union}(u, v)$**:** 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 Section 4.6). 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 in the code fragment below, and an example is shown in Fig. 21.

Kruskal's Algorithm

```
kruskalMST(G=(V,E), w) {
    A = {}                                  // initially A is empty
    for (each u in V)
        create-set(u)                       // create a set for each vertex
    sort E in increasing order by weight w
    for each ((u, v) from the sorted list) {
        if (find(u) != find(v)) {   // u and v in different trees
            add (u, v) to A
            union(u, v)
        }
    }
    return A
}
```

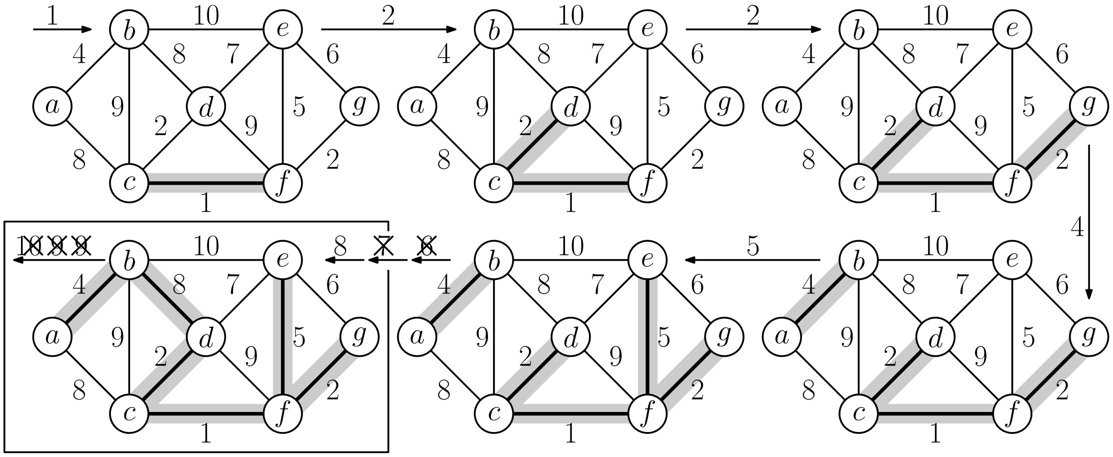


Fig. 21: Kruskal's Algorithm. Each vertex is labeled according to the set that contains it.

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 $\Theta(E \log V)$.

The Union-Find Data Structure: In order to implement the Union-Find data structure we maintain the elements in a forest of *inverted trees*. This means that the pointers in the tree are directed up towards the root. There is no limit on how many children a node can have. The root of a tree has a null parent pointer. Two elements are in the same set if and only if they are in the same tree. For example, consider the domain $\{1, 2, 3, \ldots, 13\}$, and let the current partition be

$$\{1, 6, 7, 8, 11, 12\}, \{2\}, \{3, 4, 5, 13\}, \{9, 10\}.$$

(That is, these sets form the spanning subtrees at some stage of Kruskal's algorithm.) Fig. 22 illustrates one of the many possible ways that these sets might be stored in the data structure. Note that there is no particular order to how the individual trees are structured, as long as they contain the proper elements.

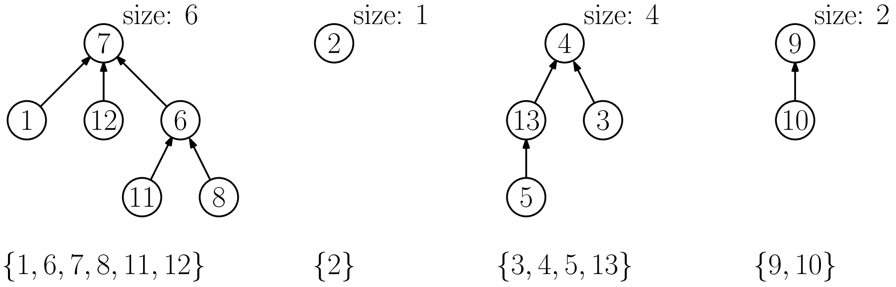


Fig. 22: Example of a Union-Find tree.

Union-Find Operations: In order to implement the operation $\mathrm{create}(u)$ the data structure we create a single node u with a null parent link, and it returns a reference to this node for future reference. To perform the operation $\mathrm{find}(u)$, we find the node containing element u and then just follow parent links up to the root. We return the element in the root node as the "name" of the set. For example, the operation $\mathrm{find}(11)$ would start at the node labeled 11, traverse the two links up to the root node, and return the root node's element 7 as the answer to the query. (Note that the name of the set is not really important. What matters for Kruskal's algorithm is that we can determine whether two nodes are in the same tree of the spanning forest, that is, in the same set. After performing the find operation we check that the two results are equal.)

To perform the operation $\mathrm{union}(u, v)$ we just link the root of one tree into the root of the other tree. For example, to take the union of the set containing $\{3, 4, 5, 13\}$ with the set $\{9, 10\}$ we could make 9 a child of 4 or make 4 a child of 9. Either would be correct, but for the sake of efficiency the order matters. If we link 4 into 9, the height of the resulting tree would be three, whereas if we do it the other way the height of the tree would be only two. (Here the *height* of a tree is the maximum number of edges on any path from a leaf to the root.) It is best to keep the tree's height small, because in doing so we make the find operations go faster.

In order to perform union operations intelligently, we maintain an extra piece of information with the root of each tree, which contains the *size* of the tree, that is, the number of nodes in the tree. When performing a union, we always *merge the smaller tree as a child of the larger one* (with ties broken arbitrarily). This called *balanced merging*. (See Fig. 23.)

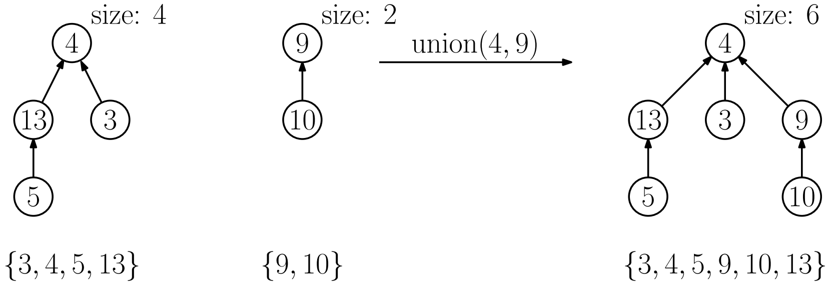


Fig. 23: Union with balanced merging.

Union-Find running time: In the worst case, a find operation takes time proportional to the height of the tree. The key to the efficiency of this procedure is the following observation. (Recall that we use $\lg m$ to denote the logarithm base-2 of m.)

Claim: Assuming balanced merging is used, a Union-Find tree with m elements is of height at most $\lg m$.

Proof: We apply induction on the number of union operations performed to build the tree. For the basis (no unions) we have a tree with one element of height zero. We have $2^0 = 1$, as desired.

For the induction step, suppose that the hypothesis is true for all trees built with strictly fewer than k union operations, and we want to prove the lemma for a Union-Find tree built with exactly k union operations. Let T be the tree and let h denote its height and n denote its number of elements. T was formed by unioning two trees, say T_1 and T_2, of heights h_1 and h_2 and sizes n_1 and n_2, respectively. By the induction hypothesis, we have $h_i \leq \lg n_i$, for $i = 1, 2$. Assume without loss of generality that T_2 was made a child of T_1 (implying that $n_2 \leq n_1$). If $h_2 < h_1$ then the final tree has height

$$h \;=\; \max(h_1, 1 + h_2) \;=\; h_1 \;\leq\; \lg n_1 \;\leq\; \lg n,$$

as desired. On the other hand, if $n_2 = n_1$, then we have $n = n_1 + n_2 = 2n_1$. By symmetry, we may assume that T_2 is made a child of T_1. The height of the final tree is therefore $h = \max(h_1, 1 + h_2) = 1 + h_2$. Thus, we have

$$h \;=\; \max(h_1, 1 + h_2) \;=\; 1 + h_2 \;\leq\; 1 + \lg n_2 \;=\; \lg(2n_2) \;=\; \lg(n_1 + n_2) \;=\; \lg n,$$

which completes the proof.

If there are at most n unions performed, the tree contains at most n elements. Each union or find operation takes time proportional to the height of the tree, which is $O(\log n)$. Therefore, the total time is $O(n \log n)$.

Theorem: After initialization, any sequence of n union and find operations can be performed in time $O(n \log n)$.

Path Compression: Interestingly, it is possible to apply a very simple heuristic improvement to this data structure which provides an impressive improvement in the running time (in particular, it *almost* gets rid of the $\log n$ factor in the running time above).

Here is the intuition. If the user of your data structure repeatedly performs find operations on a leaf at a very low level in the tree then this takes a lot of time. If we were to compress the path on each find operation, then subsequent find's to this element would go much faster. By "compress the path," we mean that when we find the root of the tree, we set all the parent pointers of the nodes on this path to the root directly. For example, in the following figure, when we do a find on 1, we traverse the path through $3, 2, 4$ and then "compress" all the parent pointers of these elements (except 4) to 4. Notice that all other pointers in the tree are unaffected.

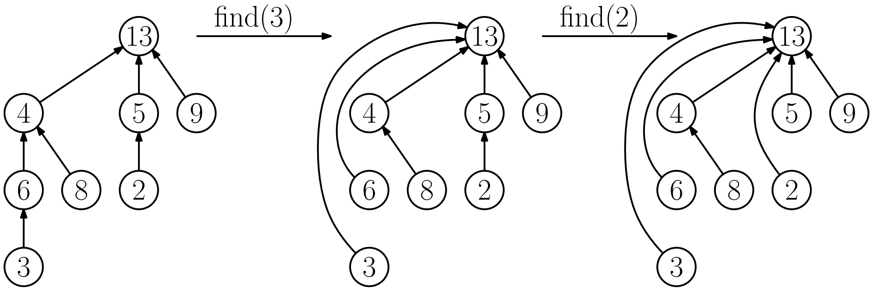


Fig. 24: Path Compression.

While this might just seem like a nice heuristic, it turns out that it makes a significant difference in the running time. However proving just how much better is a very tough exercise. While we will not prove the result, we will simply remark that, but using path compression, the running time of any sequence of n union and find operations (after initialization) runs in $O(n\alpha(n))$, where $\alpha(n)$ is the *absurdly slow growing* function called the *inverse Ackerman function*. For example, $\alpha(n) \leq 5$ under the assumption that n is smaller than the number of particles in the observable universe. (Some would say that this is a pretty reasonable upper bound on the size of any input size your program will see!)

Lecture 8: Greedy Algorithms: Huffman Coding

Huffman Codes: Huffman codes provide a method of encoding data efficiently. Normally when characters are coded using standard codes like ASCII or the Unicode, each character is represented by a fixed-length *codeword* of bits (e.g. 8 or 16 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 $00101\ldots$ how do we know whether the first character of the encoded message is "a" or "b". Conversely, if no codeword is a prefix of any other, then as soon as we see a codeword appearing as a prefix in the encoded text, then we know that we may decode this without fear of it matching some longer codeword. Thus we have the following definition.

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

Observe that any binary prefix coding can be described by a binary tree in which the codewords are the leaves of the tree, and where a left branch means "0" and a right branch means "1". The 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. 25.

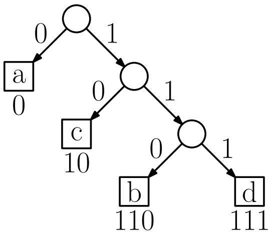


Fig. 25: Prefix codes.

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 an elegant greedy algorithm for finding such a code. It was invented in the 1950's by David Huffman, and is called a *Huffman code*. (While the algorithm is simple, it was not obvious. Huffman was a student at the time, and his professors, Robert Fano and Claude Shannon, two very eminent researchers, had developed their own algorithm, which as suboptimal.) By the way, this code was 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 seem most logical to select the two characters with the smallest probabilities to perform the operation on. The result is Huffman's algorithm. It is illustrated in Fig. 26.

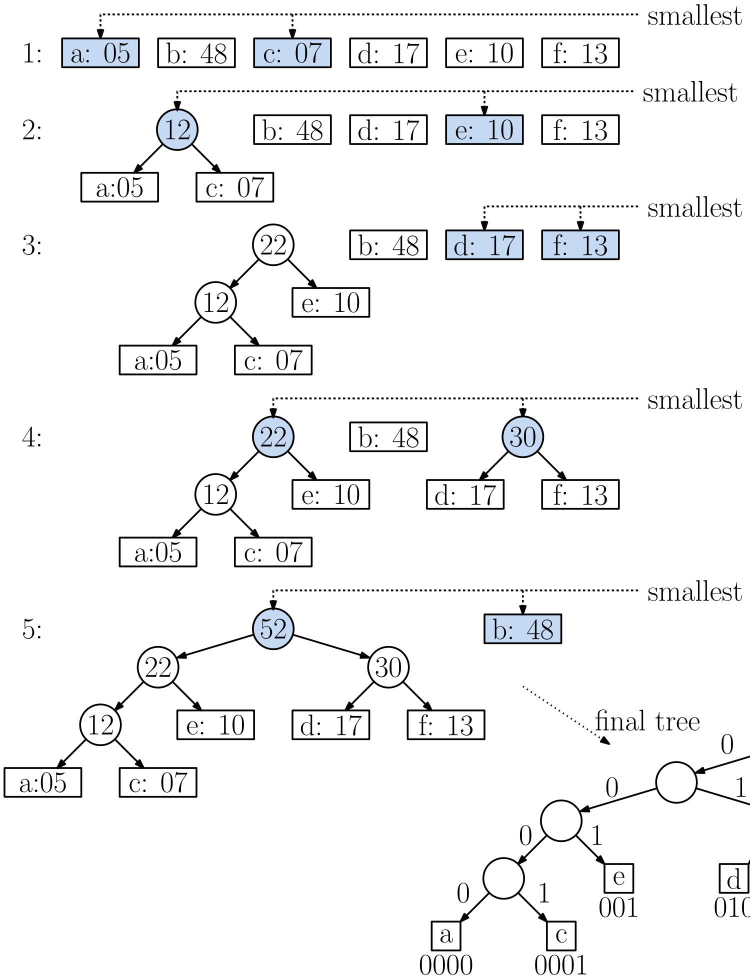


Fig. 26: Huffman's Algorithm.

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

Huffman's Algorithm

```
huffman(int n, char C[1..n]) {
    Q = C                                // insert chars into priority queue
    for (i = 1 to n-1) {                 // repeat until 1 item in queue
        z = new internal tree node
        left[z]  = x = extract-min from Q // extract smallest probabilities
        right[z] = y = extract-min from Q
        prob[z]  = prob[x] + prob[y]      // 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;
}
```

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 $d_T(b) \geq d_T(x)$ and $d_T(c) \geq d_T(y)$. (Again, they may be equal.) Thus, we have $p(b) - p(x) \geq 0$ and $d_T(b) - d_T(x) \geq 0$, 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. 27.

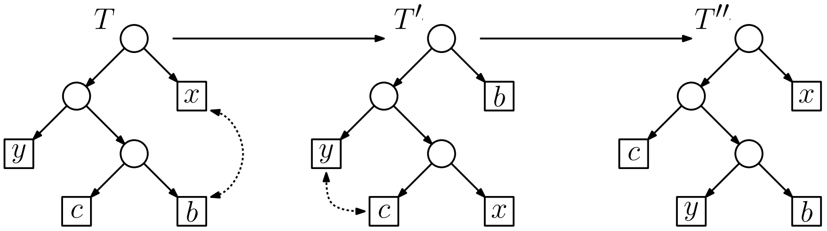


Fig. 27: Correctness of Huffman's Algorithm.

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

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

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

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

Claim: Huffman's algorithm produces 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.

Lecture 9: Divide and Conquer: Mergesort and Inversion Counting

Divide and Conquer: So far, we have been studying a basic algorithm design technique called greedy algorithms. Today, we begin study of a different technique, called *divide and conquer*. The ancient Roman rulers understood this principle well (although they were probably not thinking about algorithms at the time). You divide your enemies (by getting them to distrust each other) and then conquer them one by one. In algorithm design, the idea is to take a problem on a large input, break the input into smaller pieces, solve the pieces individually (usually recursively), and then combine the piecewise solutions into a global solution.

Summarizing, the main elements to a divide-and-conquer solution are

- *Divide* (the problem into a small number of pieces),
- *Conquer* (solve each piece, by applying divide-and-conquer recursively to it), and
- *Combine* (the pieces together into a global solution).

There are a huge number computational problems that can be solved efficiently using divide-and-conquer. Divide-and-conquer algorithms typically involve recursion, since this is usually the most natural way to deal with the "conquest" part of the algorithm. Analyzing the running times of recursive programs is usually done by solving a *recurrence*.

MergeSort: Perhaps the simplest example of a divide-and-conquer algorithm is MergeSort. I am sure you are familiar with this algorithm, but for the sake of completeness, let's recall how it works. We are given an sequence of n numbers, which we denote by A. The objective is to permute the array elements into non-decreasing order. A may be stored as an array or a linked list. Let's not worry about these implementaiton details for now. We will need to assume that, whatever representation we use, we can determine the lists size in constant time, and we can enumerate the elements from left to right.

Here is the basic structure of MergeSort. Let $\mathrm{size}(A)$ denote the number of elements of A.

Basis case: If $\mathrm{size}(A) = 1$, then the array is trivially sorted and we are done.

General case: Otherwise:

Divide: Split A into two subsequences, each of size roughly $n/2$. (More precisely, one will be of size $\lfloor n/2 \rfloor$ and the other of size $\lceil n/2 \rceil$.)

Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Combine: Merge the two sorted subsequences into a single sorted list.

MergeSort: The key to the algorithm is the merging process. Let us assume inductively that the sequence has been split into two, which are presented as two subarrays, $A[p..m]$ and $A[m+1..r]$, each of which has been sorted. The merging process copies the elements of these two subarrays into temporary array B. We maintain two indices i and j, indicating the current elements of the left and right subarrays, respectively. At each step, we copy whichever element is smaller $A[i]$ or $A[j]$ to the next position of B. (Ties are broken in favor of A.) See Fig. 28.)

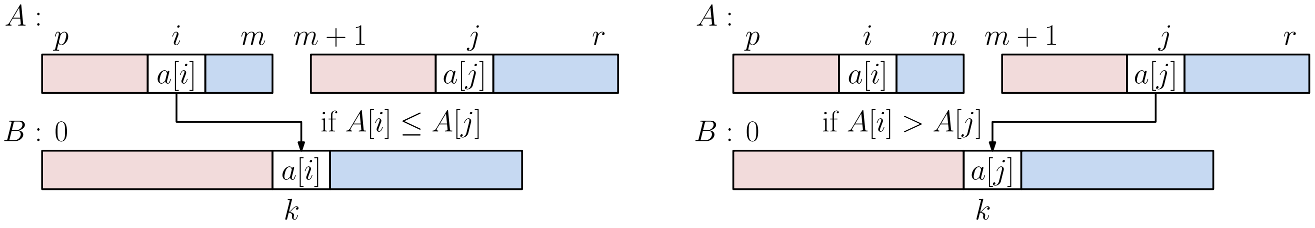


Fig. 28: Merging two sorted lists.

The two code blocks below present the MergeSort algorithm and the merging utility, which merges two sorted lists. Assuming that the input is stored in the array $A[1..n]$, the initial call is $\mathrm{MergeSort}(A, 1, n)$.

MergeSort

```
MergeSort(A, p, r) {                            // sort A[p..r]
    if (p < r) {                                // we have at least 2 items
        m = (p + r)/2                           // midpoint
        MergeSort(A, p, m)                      // sort the left half
        MergeSort(A, m+1, r)                    // sort the right half
        merge(A, p, m, r)                       // merge the two halves
    }
}

merge(A, p, m, r) {                             // merges A[p..m] with A[m+1..r]
    new array B[0..r-p]
    i = p;  j = m+1;  k = 0;                    // initialize indices
    while (i <= m and j <= r) {                 // while both subarrays are nonempty
        if (A[i] <= A[j]) B[k++] = A[i++]       // take next item from left subarray
        else              B[k++] = A[j++]       // take next item from right subarray
    }
    while (i <= m) B[k++] = A[i++]              // copy any extras to B
    while (j <= r) B[k++] = A[j++]
    for (k = 0 to r-p) A[p+k] = B[k]            // copy B back to A
}
```

This completes the description of the algorithm. Observe that of the last two while-loops in the merge procedure, only one will be executed. (Do you see why?) Another question worth considering is the following. Suppose that in the merge function, the statement "`A[i] <= A[j]`" had instead been written "`A[i] < A[j]`"? Would the algorithm still be correct? Can you see any reason for preferring one version over the other? (Hint: Consider what happens when A contains duplicate copies of the same element.)

Fig. 29 shows an example of the execution of MergeSort. The dividing part of the algorithm is shown on the left and the merging part is shown on the right.

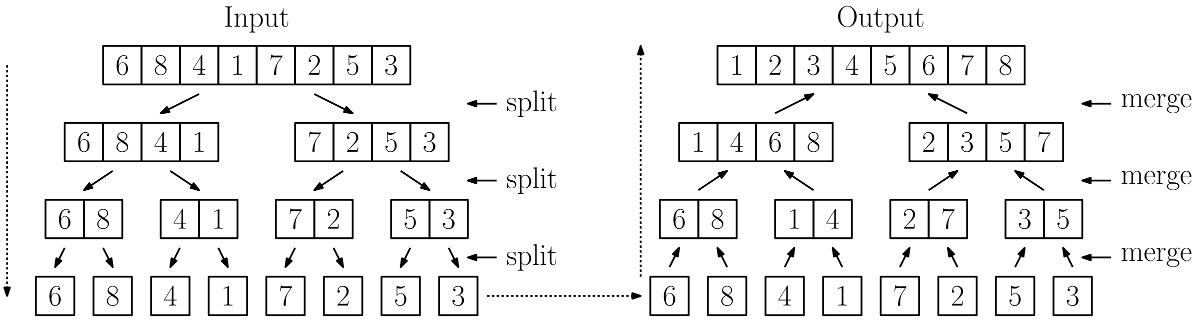


Fig. 29: MergeSort example.

Analysis: Next, let us analyze the running time of MergeSort. First observe that the running time of the procedure $\mathrm{merge}(A, p, m, r)$ is easily seen to be $O(r - p + 1)$, that is, it is proportional to the total size of the two lists being merged. The reason is that, each time through the loop, we succeed in copying one element from $A[p..r]$ to the final output.

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this through the use of a *recurrence*, that is, a function that is defined recursively in terms of itself. Let's see how to apply this to MergeSort. Let $T(n)$ denote the worst case running time of MergeSort on an input of length $n \geq 1$. First observe that if we call MergeSort with a list containing a single element, then the running time is a constant. Since we are ignoring constant factors, we can just write $T(n) = 1$. When we call MergeSort with a list of length $n \geq 2$, e.g. $\mathrm{merge}(A, p, r)$, where $r - p + 1 = n$, the algorithm first computes $m = \lfloor (p + r)/2 \rfloor$. The subarray $A[p..r]$, which contains $r - p + 1$ elements. We'll ignore the floors and ceilings, and simply declare that each subarray is of size $n/2$. Thus, we have

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

Solving the Recurrence: In order to complete the analysis, we need to solve the above recurrence. There are a few ways to solve recurrences. My favorite method is to apply repeated expansion until a pattern emerges. Then, express the result in terms of the number of iterations performed.

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

The above expression as a function of k is messy, but it is useful. We know that $T(1) = 1$. To use that fact, we need to determine what value to set k so that $n/2^k = 1$. Therefore, we have $k = \lg n$.[4] By substituting this value for k, we have $T(n/2^k) = T(1) = 1$ and plugging this into the above formula, we obtain

$$T(n) \;=\; 2^{\lg n} \cdot T(1) + n \lg n \;=\; n \cdot 1 + n \lg n \;=\; O(n \log n),$$

[4] Recall that “lg” means logarithm base 2. This worked because we ignored the floors and ceilings, and hence, treated n as if it were a power of 2. More accurately, we have $k = \lceil \lg n \rceil$.

Therefore, the running time of MergeSort is $O(n \log n)$.

Many of the recurrences that arise in divide-and-conquer algorithms have a similar structure. The following theorem is useful for compute asymptotic bounds for these recurrences.

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

$$T(n) = aT\left(\frac{n}{b}\right) + n^k,$$

defined for $n \geq 0$. (Let us assume that n is a power of b. This doesn't affect the asymptotics. The basis case, $T(1)$ can be any constant value.) Then:

Case 1: if $a > b^k$, then $T(n) \in \Theta(n^{\log_b a})$

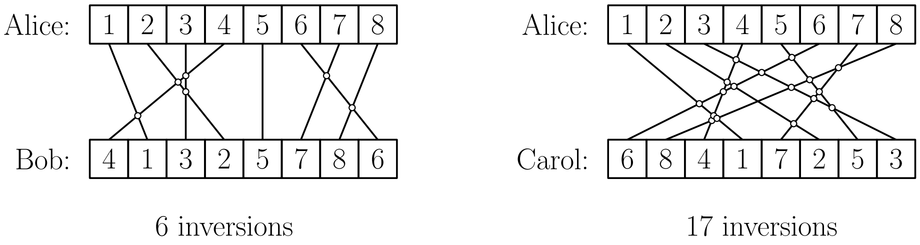
Case 2: if $a = b^k$, then $T(n) \in \Theta(n^k \log n)$

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

Inversion Counting: Let's consider a variant on this. Although the problem description does not appear to have anything to do with sorting or Mergesort, we will see that the solutions to these problems are closely related. Suppose that you are given two rank ordered lists of preferences. For example, suppose that you and bunch of your friends are given a list of 50 popular movies, and you are rank order them from most favorite to least favorite. After this exercise, you want to know which people tended to rank movies in roughly the same way that you did. Here is an example:

Movie Title	Alice	Bob	Carol
Gone with the Wind	1	4	6
Citizen Kane	2	1	8
The Seven Samurai	3	3	4
The Godfather	4	2	1
Titanic	5	5	7
My Cousin Vinny	6	7	2
Star Wars	7	8	5
Plan 9 from Outer Space	8	6	3

Given two such lists, how would you determine their degree of similarity? Here is one possible approach. Given two lists of preferences, L_1 and L_2, define an *inversion* to be a pair of movies x and y, such that L_1 has x before y and L_2 has y before x. Since there are $\binom{n}{2} = n(n-1)/2$ unordered pairs, the maximum number of inversions is $\binom{n}{2}$, which is $O(n^2)$. If the two rankings are the same, then there are no inversions. Thus, the number of inversions can be seen as one possible measure of similarity between two lists of n numbers. (An example is shown in in Fig. 30.)



6 inversions

17 inversions

Fig. 30: Movie preferences and inversions.

We can reduce this problem from one involving two lists to one involving just one. In particular, assume that the first list consists of the sequence $\langle 1, \ldots, n \rangle$. Let the other list be denoted by $\langle a_1, \ldots, a_n \rangle$. (More generally, you can relabel the elements so that the index of the element is its position in the first list.) An *inversion* is a pair of indices (i, j) such that $i < j$, but $a_i > a_j$. Given a list of n (distinct) numbers, our objective is to count the number of inversions.

Naively, we can solve this problem in $O(n^2)$ time. For each a_i, we search all $i + 1 \leq j \leq n$, and increment a counter for every j such that $a_i > a_j$. We will investigate a more efficient method based on divide-and-conquer.

Divide-and-conquer solution: How would we approach solving this problem using divide-and-conquer? Here is one way of doing it:

Basis case: If $\mathrm{size}(A) = 1$, then there are no inversions.

General case: Otherwise:

Divide: Split A into two subsequences, each of size roughly $n/2$.

Conquer: Compute the number of inversions *within* each of the subsequences.

Combine: Count the number of inversions occurring *between* the two sequences.

The computation of the inversions within each subsequence is solved by recursion. The key to an efficient implementation of the algorithm is the step where we count the inversions between the two lists. It will be much easier to count inversions if we first sort the list. In fact, our approach will be to both sort and count inversions at the same time.

Let us assume that the input is given as an array $A[p..r]$. Let us assume inductive that it has been split into two subarrays, $A[p..m]$ and $A[m + 1..r]$, each of which has already been sorted. During the merging process, we maintain two indices i and j, indicating the current elements of the left and right subarrays, respectively (see Fig. 31).

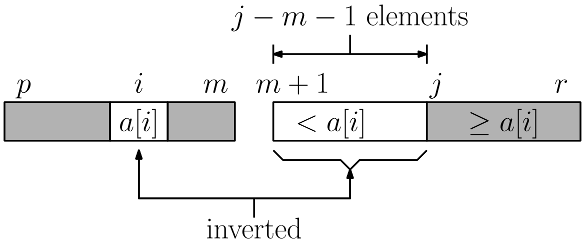


Fig. 31: Counting inversions when $A[i] \leq A[j]$.

Whenever $A[i] > A[j]$ the algorithm advances j. It follows, therefore, that if $A[i] \leq A[j]$, then every element of the subarray $A[m+1..j-1]$ is strictly smaller than $A[i]$. Since the elements of the left subarray appear in the original array before all the elements of the right subarray, it follows that $A[i]$ generates an inversion with *all* the elements of the subarray $A[m + 1..j - 1]$. The number of elements in this subarray is $(j - 1) - (m + 1) + 1 = j - m - 1$. Therefore, before when we process $A[i]$, we increment an inversion counter by $j - m - 1$.

The other part of the code that is affected is when we copy elements from the end of the left subarray to the final array. In this case, each element that is copied generates an inversion with respect to all the elements of the right subarray, that is, $A[m + 1..r]$. There are $r - m$ such elements. We add this value to the inversion counter.

The algorithm is modeled on the same pseudo-code as that used for MergeSort and is presented in the following code block. Assuming that the input is stored in the array $A[1..n]$, the initial call is $\mathrm{InvCount}(A, 1, n)$.

This approach is illustrated in Fig. 32. Observe that inversions are counted in the merging process (shown as small white circles in the figure).

```
    InvCount(A, p, r) {                         // sort A[p..r]
        if (p >= r) return 0                    // 1 element or fewer --> no inversions
        m = (p + r)/2                           // midpoint
        x1 = InvCount(A, p, m)                  // count inversions in the left half
        x2 = InvCount(A, m+1, r)                // sort the right half
        x3 = invMerge(A, p, m, r)               // merge and count inversions
        return x1 + x2 + x3
    }

    invMerge(A, p, m, r) {                      // merges A[p..m] with A[m+1..r]
        new array B[0..r-p]
        i = p;  j = m+1;  k = 0;                // initialize indices
        ct = 0                                  // inversion counter
        while (i <= m and j <= r) {             // while both subarrays are nonempty
            if (A[i] <= A[j]) {
                B[k++] = A[i++]                 // take next item from left subarray
                ct += j - m - 1                 // increment the inversion counter
            }
            else B[k++] = A[j++]                // take next item from right subarray
        }
        while (i <= m) {
            B[k++] = A[i++]                     // copy extras from left subarray to B
            ct += r - m                         // increment inversion counter
        }
        while (j <= r) B[k++] = A[j++]          // copy extras from right subarray to B

        for (k = 0 to r-p) A[p+k] = B[k]        // copy B back to A
    }
```

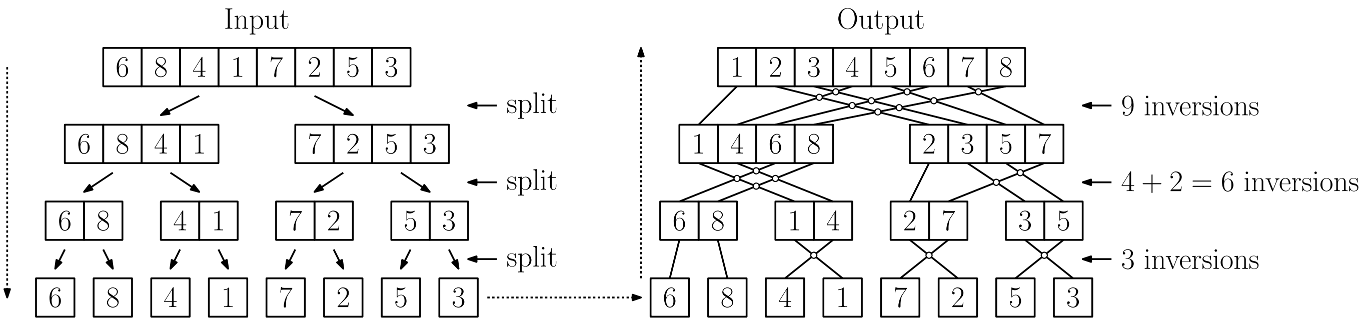


Fig. 32: Inversion counting by divide and conquer.

Lecture 10: Divide-and-Conquer: Closest Pair

Closest Pair: Today, we consider another application of divide-and-conquer, which comes from the field of computational geometry. We are given a set P of n points in the plane, and we wish to find the closest pair of points $p, q \in P$ (see Fig. 33(a)). This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall that, given two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$, their (Euclidean) distance is

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Clearly, this problem can be solved by brute force in $O(n^2)$ time, by computing the distance between each pair, and returning the smallest. Today, we will present an $O(n \log n)$ time algorithm, which is based a clever use of divide-and-conquer.

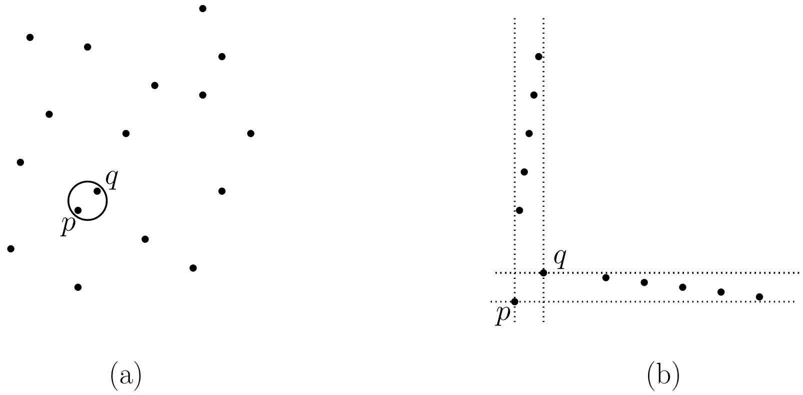


Fig. 33: (a) The closest pair problem and (b) why sorting on x- or y-alone doesn't work.

Before getting into the solution, it is worth pointing out a simple strategy that fails to work. If two points are very close together, then clearly both their x-coordinates and their y-coordinates are close together. So, how about if we *sort* the points based on their x-coordinates and, for each point of the set, we'll consider just *nearby* points in the list. It would seem that (subject to figuring out exactly what "nearby" means) such a strategy might be made to work. The problem is that it could fail miserably. In particular, consider the point set of Fig. 33(b). The points p and q are the closest points, but we can place an arbitrarily large number of points between them in terms of their x-coordinates. We need to separate these points sufficiently far in terms of their y-coordinates that p and q remain the closest pair. As a result, the positions of p and q can be arbitrarily far apart in the sorted order. Of course, we can do the same with respect to the y-coordinate. Clearly, we cannot focus on one coordinate alone.

Divide-and-Conquer Algorithm: Let us investigate how to design an $O(n \log n)$ time divide-and-conquer approach to the problem. The input consists of a set of points P, represented, say, as an array of n elements, where each element stores the (x, y) coordinates of the point. (For simplicity, let's assume there are no duplicate x-coordinates.) The output will consist of a single number, being the closest distance. It is easy to modify the algorithm to also produce the pair of points that achieves this distance.

For reasons that will become clear later, in order to implement the algorithm efficiently, it will be helpful to begin by *presorting* the points, both with respect to their x- and y-coordinates. Let P_x be an array of points sorted by x, and let P_y be an array of points sorted by y. We can compute these sorted arrays in $O(n \log n)$ time. Note that this initial sorting is done only *once*. In particular, the recursive calls do not repeat the sorting process.

Like any divide-and-conquer algorithm, after the initial basis case, our approach involves three basic elements: divide, conquer, and combine.

Basis: If $|P| \leq 3$, then just solve the problem by brute force in $O(1)$ time.

Divide: Otherwise, partition the points into two subarrays P_L and P_R based on their x-coordinates. In particular, imagine a vertical line ℓ that splits the points roughly in half (see Fig. 34). Let P_L be the points to the left of ℓ and P_R be the points to the right of ℓ.

In the same way that we represented P using two sorted arrays, we do the same for P_L and P_R. Since we have presorted P_x by x-coordinates, we can determine the median element for ℓ in constant time. After this, we can partition each of arrays P_x and P_y in $O(n)$ time.

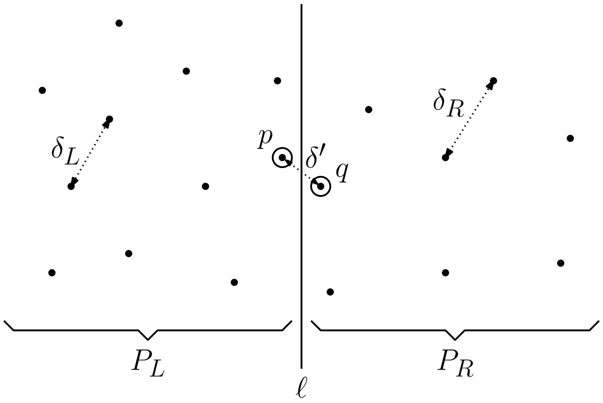


Fig. 34: Divide-and-conquer closest pair algorithm.

Conquer: Compute the closest pair *within* each of the subsets P_L and P_R each, by invoking the algorithm recursively. Let δ_L and δ_R be the closest pair distances in each case (see Fig. 34). Let $\delta = \min(\delta_L, \delta_R)$.

Combine: Note that δ is not necessarily the final answer, because there may be two points that are very close to one another but are on opposite sides of ℓ. To complete the algorithm, we want to determine the closest pair of points *between* the sets, that is, the closest points $p \in P_L$ and $q \in P_R$ (see Fig. 34). Since we already have an upper bound δ on the closest pair, it suffices to solve the following *restricted problem*: if the closest pair (p, q) are within distance δ, then we will return such a pair, otherwise, we may return any pair. (This restriction is very important to the algorithm's efficiency.) In the next section, we'll show how to solve this restricted problem in $O(n)$ time. Given the closest such pair (p, q), let $\delta' = \|pq\|$. We return $\min(\delta, \delta')$ as the final result.

Assuming that we can solve the "Combine" step in $O(n)$ time, it will follow that the algorithm's running time is given by the recurrence $T(n) = 2T(n/2) + n$, and (as in Mergesort) the overall running time is $O(n \log n)$, as desired.

Closest Pair Between the Sets: To finish up the algorithm, we need to compute the closest pair p and q, where $p \in P_L$ and $q \in P_R$. As mentioned above, the algorithm is allowed to make a mistake, but only if there is no such pair that is closer than δ. The input to our algorithm consists of the point set P, the x-coordinate of the vertical splitting line ℓ, and the value of $\delta = \min(\delta_L, \delta_R)$. Recall that our goal is to do this in $O(n)$ time.

This is where the real creativity of the algorithm enters. Observe that if such a pair of points exists, we may assume that both points lie within distance δ of ℓ, for otherwise the resulting distance would exceed δ. Let S denote this subset of P that lies within a vertical strip of width 2δ centered about ℓ (see Fig. 35(a)).[5]

How do we find the closest pair within S? Sorting comes to our rescue. Let $S_y = \langle s_1, \ldots, s_m \rangle$ denote the points of S sorted by their y-coordinates (see Fig. 35(a)). At the start of the lecture, we asserted that considering the

[5] You might be tempted to think that we have pruned away many of the points of P, and this is the source of efficiency, but this is not true. It might very well be that *every* point of P lies within the strip, and so we cannot afford to apply a brute-force solution to our problem.

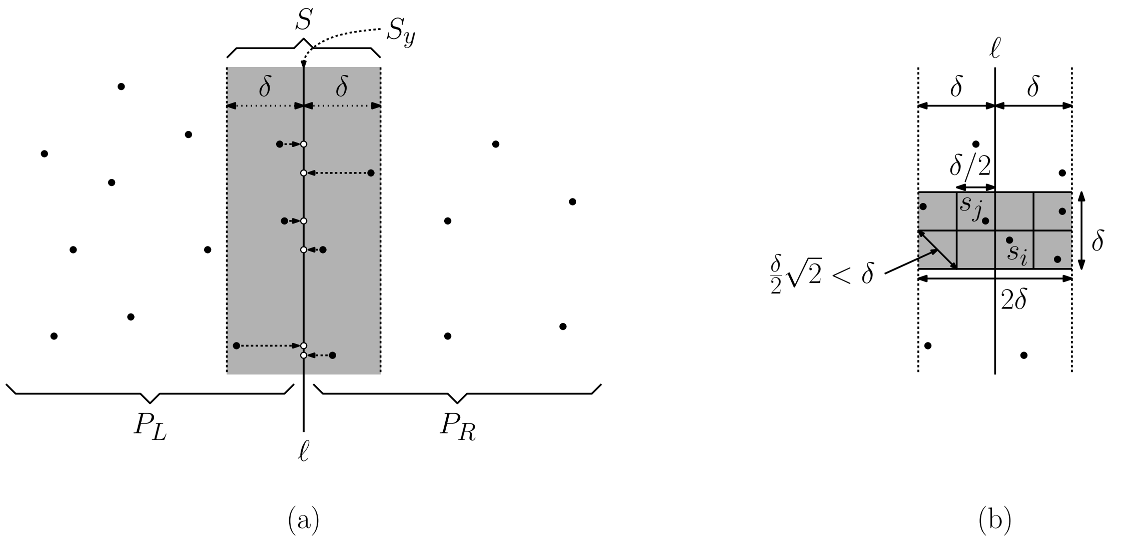


Fig. 35: Closest pair in the strip.

points that are close according to their x- or y-coordinate alone is not sufficient. It is rather surprising, therefore, that this *does* work for the set S_y.

The key observation is that if S_y contains two points that are within distance δ of each other, these two points will be within a constant number of positions of each other in the sorted array S_y. The following lemma formalizes this observation.

Lemma: Given any two points $s_i, s_j \in S_y$, if $\|s_i s_j\| \leq \delta$, then $|j - i| \leq 7$.

Proof: Suppose that $\|s_i s_j\| \leq \delta$. Since they are in S they are within distance δ of ℓ. Clearly, the y-coordinates of these two points can differ by at most δ. So they must both reside in a rectangle of width 2δ and height δ centered about ℓ (see Fig. 35(b)). Split this rectangle into eight identical squares of side length $\delta/2$. A square of side length x has a diagonal of length $x\sqrt{2}$, and no two points within such a square can be farther away than this. Therefore, the distance between any two points lying within one of these eight squares is at most

$$\frac{\delta\sqrt{2}}{2} = \frac{\delta}{\sqrt{2}} < \delta.$$

Since each square lies entirely on one side of ℓ, no square can contain two or more points of P, since otherwise, these two points would contradict the fact that δ is the closest pair seen so far. Thus, there can be at most eight points of S in this rectangle, one for each square. Therefore, $|j - i| \leq 7$.

One issue that we have not yet addressed is how to compute S_y. Recall that we cannot afford to sort these points explicitly, because we may have n points in S, and this part of the algorithm needs to run in $O(n)$ time. This is where presorting comes in. Recall that the points of P_y are already sorted by y-coordinates. To compute S_y, we enumerate the points of P_y, and each time we find a point that lies within the strip, we copy it to the next position of array S_y. This runs in $O(n)$ time, and preserves the y-ordering of the points.

By the way, it is natural to wonder whether 8 in the statement of the lemma is optimal. Getting the best possible value is likely to be a tricky geometric exercise. Our textbook proves a weaker bound of 16. Of course, from the perspective of asymptotic complexity, the exact constant does not matter.

The final algorithm is presented in the code fragment below.

Closest Pair

```
closestPair(P = (Px, Py)) {                             // find closest pair in P
    n = |P|
    if (n <= 3) solve by brute force                    // basis case
    else {
        Find the vertical line L through P’s median     // divide
        Split P into PL and PR (split Px and Py as well)
        dL = closestPair(PL)                            // conquer
        dR = closestPair(PR)
        d = min(dL, dR)
        for (i = 1 to n) {                              // create Sy
            if (Py[i] is within distance d of L) {
                append Py[i] to Sy
            }
        }
        d’ = stripClosest(Sy)                           // closest in strip
        return min(d, d’)                               // overall closest
    }
}

stripClosest(Sy) {                                      // closest in strip
    m = |Sy|
    d’ = infinity
    for (i = 1 to m) {
        for (j = i+1 to min(m, i+7)) {                  // search neighbors
            if (dist(Sy[i], Sy[j]) <= d’) {
                d’ = dist(Sy[i], Sy[j])                 // new closest found
            }
        }
    }
    return d’
}
```

Lecture 11: Dynamic Programming: Weighted Interval Scheduling

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. Dynamic programming is a common technique for solving optimization problems that have clean structural properties. (The meaning of this will become clearer once we have seen a few examples.) There is a superficial resemblance to divide-and-conquer, in the sense that it breaks problems down into smaller subproblems, which can be solved recursively. However, unlike divide-and-conquer problems, in which the subproblems are disjoint, in dynamic programming the subproblems typically overlap each other.

Dynamic programming solutions rely on two important structural qualities, *optimal substructure* and *overlapping subproblems*.

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. While this might seem intuitively obvious, not all optimization problems satisfy this property. For example, it may be advantageous to solve one subproblem suboptimally in order to conserve resources so that another, more critical, subproblem can be solved more optimally.

Overlapping Subproblems: While it may be possible subdivide a problem into subproblems in exponentially many different ways, these subproblems overlap each other in such a way that the number of distinct subproblems is reasonably small, ideally polynomial in the input size. The question is how to generate the solutions to these subproblems. There are two complementary (but essentially equivalent) ways of viewing how a solution is constructed.

Top Down: A solution to a DP problem is expressed recursively. This approach applies recursion directly to solve the problem. However, due to the overlapping nature of the subproblems, the same recursive call is often made many times. An approach, called *memoization*, records the results of recursive calls, so that subsequent calls to a previously solved subproblem are handled by table look-up.

Bottom-up: Although the problem is formulated recursively, the solution is built iteratively by combining the solutions to small subproblems to obtain the solution to larger subproblems. The results are stored in a table.

In the next few lectures, we will consider a number of examples, which will help make these concepts more concrete.

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 $S = \{1, \ldots, n\}$ 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 (see Fig. 36(a)).

In *weighted interval scheduling*, we assume that in addition to the start and finish times, each request is associated with a numeric *weight* or *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 (see Fig. 36(b)).

Observe that the unweighted version of the interval scheduling problem can be viewed as a special case of the weighted version, in which all weights are 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 $f_1 \leq \ldots \leq f_n$.

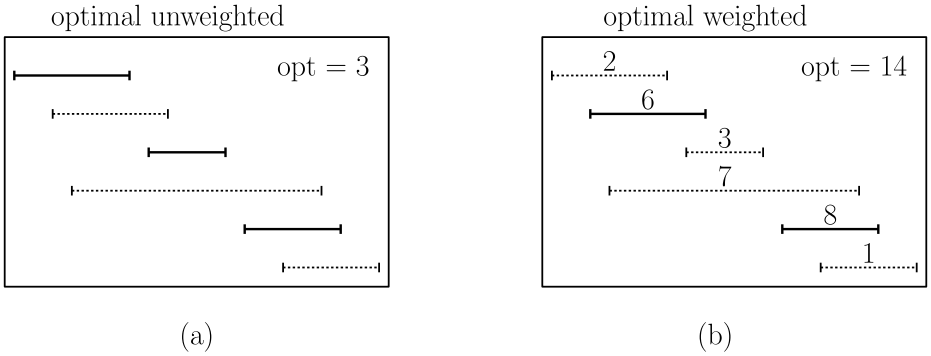


Fig. 36: Weighted and unweighted interval scheduling.

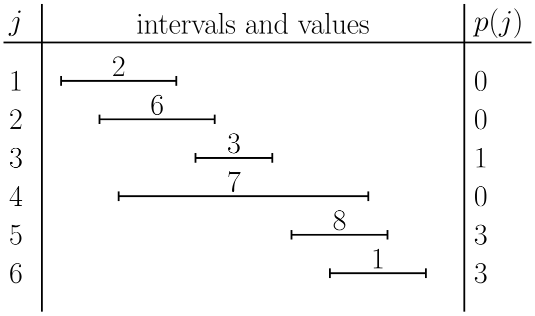


Fig. 37: Weighted interval scheduling input and and p-values.

Given any request j, define $p(j)$ to be the largest $i < j$ such that the ith and jth requests are compatible, that is, $f_i < s_j$. If no such i exists, let $p(j) = 0$ (see Fig. 37).

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 a function opt(i), which denotes the maximum possible value achievable, if we restrict attention to just the first i requests. Clearly, the final desired result will be the maximum value using *all* the requests, that is, opt(n). As a starting point we have opt$(0) = 0$, which means that we get no value if there are no requests.

In order to compute opt(j) for an arbitrary j, $1 \leq j \leq n$, we observe that there are two possibilities:

Request j is not in the optimal schedule: If j is not included in the schedule, then we should do the best we can with the remaining $j - 1$ requests. Therefore, opt$(j) =$ opt$(j - 1)$.

Request j is in the optimal 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 opt$(j) = v_j +$ opt$(p(j))$.

How do we know which of the these two options to select? The answer is fundamental to all DP problems:

DP Selection Principle:
When given a set of feasible options to choose from, try them all and take the best.

This provides us with the following recursive rule:

$$\text{opt}(j) \;=\; \max(\text{opt}(j-1),\; v_j + \text{opt}(p(j))).$$

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

We could express this in pseudocode as follows:

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 opt(j) optimally, the two subproblems that are used to make the final result opt$(j-1)$ and opt$(p(j))$ should also be computed optimally. This is an example of the principle of optimality, which in this case is clear.[6]

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 to opt(0) that result from a single call to opt(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 essentially a 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 just 50 entries? If you look at the recursion tree, the problem jumps out immediately (see Fig 38). The problem is that the same recursive calls are being generated over and over again. But there is no reason to make even a second call this, since they all return exactly the same value.

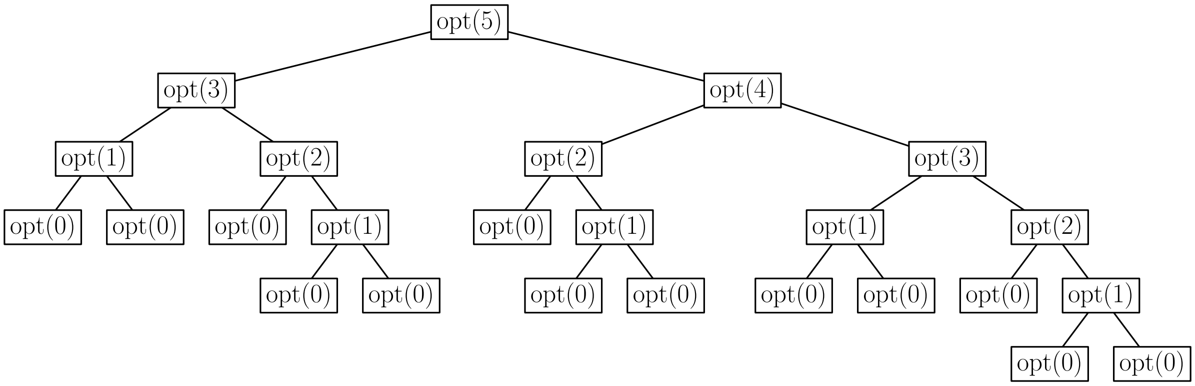


Fig. 38: The exponential nature of recursive-opt.

This suggests a smarter version of the algorithm. Once a value has been computed for $\mathrm{recursive} - \mathrm{opt}(j)$ we store the value in a global array $M[1..n]$, 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 $\mathrm{memoized} - \mathrm{opt}$ 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.

[6] 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.

Memoized Weighted Interval Scheduling

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

Bottom-up Construction: Yet another method for computing the values of the array, is to dispense with the recursion altogether, and 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 schedule. 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. 39. Clearly the running time is $O(n)$.

Iterative Weighted Interval Scheduling

```
iterative-opt() {
    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];
        }
    }
}
```

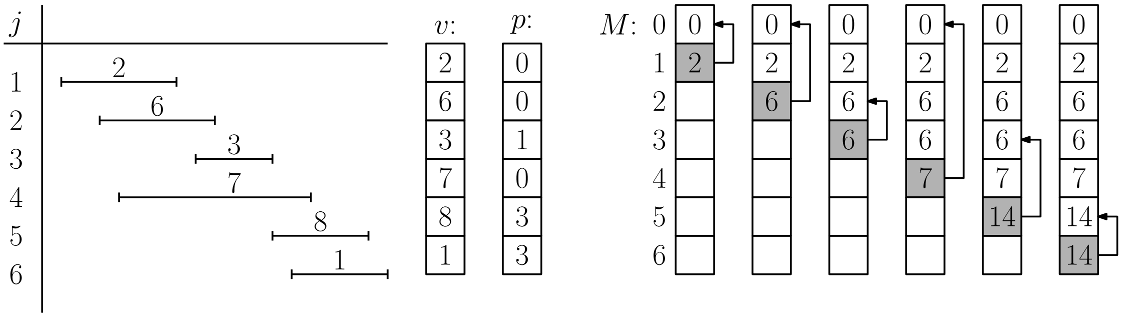


Fig. 39: Example of iterative construction and predecessor values. The final optimal value is 14. By following the predecessor pointers back from $M[6]$ we see that the requests that are in the schedule are 5 and 2.

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 to leave ourselves a few hints in order to reconstruct the final result.

In iterative-opt$()$ 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 used the jth 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 Weighted Interval Scheduling Schedule

```
get-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 Fig. 39 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 $\langle 5, 2\rangle$. Reversing the list gives the final schedule.

Lecture 12: Dynamic Programming: Longest Common Subsequence

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. String searching has many applications in document processing and retrieval and computational biology applied to genomics. An important problem involves determining the degree of similarity between two strings. One common measure of similarity between two strings is the lengths of their longest common subsequence. Today, we will consider an efficient solution to this problem. The same technique can be applied to a variety of string processing problems.

Longest Common Subsequence: Let us think of character strings as sequences of characters. Given two sequences $X = \langle x_1, x_2, \ldots, x_m\rangle$ and $Z = \langle z_1, z_2, \ldots, z_k\rangle$, we say that Z is a *subsequence* of X if there is a strictly increasing sequence of k indices $\langle i_1, i_2, \ldots, i_k\rangle$ ($1 \leq i_1 < i_2 < \ldots < i_k \leq n$) such that $Z = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k}\rangle$. For example, let $X = \langle$ABRACADABRA$\rangle$ and let $Z = \langle$AADAA$\rangle$, then Z is a 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 $X = \langle$ABRACADABRA$\rangle$ and let $Y = \langle$YABBADABBADOO$\rangle$. Then the longest common subsequence is $Z = \langle$ABADABA$\rangle$ (see Fig. 40).

The *Longest Common Subsequence Problem* (LCS) is the following. Given two sequences $X = \langle x_1, \ldots, x_m\rangle$ and $Y = \langle y_1, \ldots, y_n\rangle$ determine the length of their longest common subsequence, and more generally the sequence itself. Note that the subsequence is not necessarily unique. For example the LCS of $\langle$ABC$\rangle$ and $\langle$BAC$\rangle$ is either $\langle$AC$\rangle$ or $\langle$BC$\rangle$.

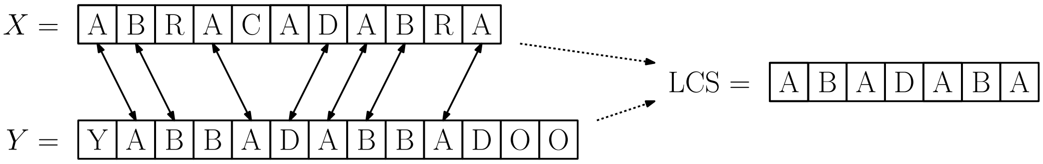


Fig. 40: An example of the LCS of two strings X and Y.

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, $X_i = \langle x_1, \ldots, x_i \rangle$. X_0 is the empty sequence.

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

Let us start by deriving a recursive formulation for computing lcs(i, j). As we have seen with other DP problems, a naive implementation of this recursive rule will lead to a very inefficient algorithm. Rather than implementing it directly, we will use one of the other techniques (memoization or bottom-up computation) to produce a more efficient algorithm.

Basis: If either sequence is empty, then the longest common subsequence is empty. Therefore, lcs$(i, 0) =$ lcs$(j, 0) = 0$.

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’, it is easy to see that the LCS *must* also end in ‘A’. (We will leave the formal proof as an exercise.) There is no harm in assuming that the last two characters of both strings will be matched to each other, since matching the last ‘A’ of one string to an earlier ‘A’ of the other can only limit our options. Since the A is the last character of the LCS, we may find the overall LCS by (1) removing A from both sequences, (2) taking the LCS of $X_{i-1} = \langle ABC \rangle$ and $Y_{j-1} = \langle DAC \rangle$ which is $\langle AC \rangle$, and (3) adding A to the end. This yields $\langle ACA \rangle$ as the LCS. Therefore, the length of the final LCS is the length of LCS$(X_{i-1}, Y_{j-1}) + 1$ (see Fig. 41), which provides us with the following rule:

$$\text{if } x_i = y_j \text{ then lcs}(i, j) = \text{lcs}(i-1, j-1) + 1$$

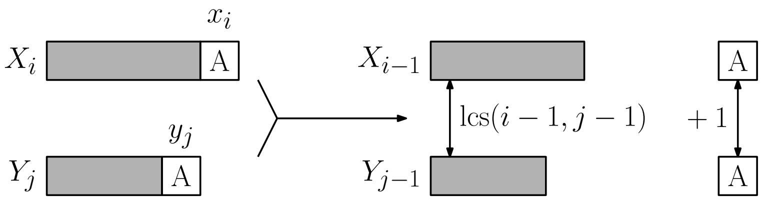


Fig. 41: LCS of two strings whose last characters are equal.

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). Remember the DP selection principle: *When given a set of feasible options to choose from, try them all and take the best.* Let's consider both options, and see which one provides the better result.

Option 1: (x_i is not in the LCS) Since we know that x_i is out, we can infer that the LCS of X_i and Y_j is the LCS of X_{i-1} and Y_j, which is given by lcs$(i-1, j)$.

Option 2: (y_j is not in the LCS) Since y_j is out, we can infer that the LCS of X_i and Y_j is the LCS of X_i and Y_{j-1}, which is given by lcs$(i, j-1)$.

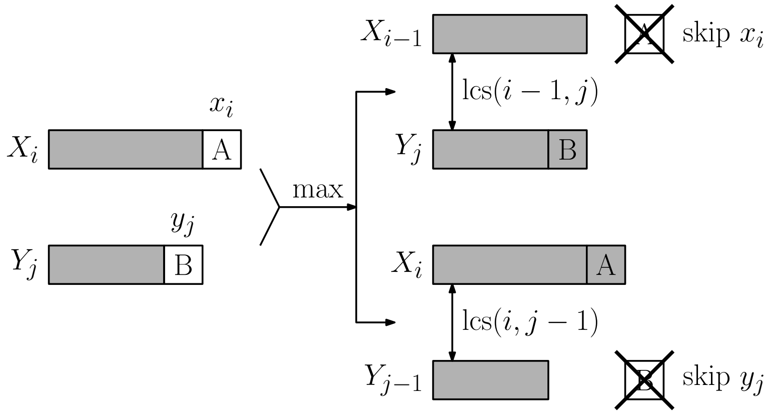


Fig. 42: The possibe cases in the DP formulation of LCS.

We compute both options and take the one that gives us the longer LCS (see Fig. 42).

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

Combining these observations we have the following recursive formulation:

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

As mentioned earlier, a direct recursive implementation of this rule will be very inefficient. Let's consider two alternative approaches to computing it.

Memoized implementation: The principal source of the inefficiency in a naive implementation of the recursive rule is that it makes repeated calls to lcs(i, j) for the same values of i and j. To avoid this, it creates a 2-dimensional array lcs$[0..m, 0..n]$, where $m = |X|$ and $n = |Y|$. The memoized version first checks whether the requested value has already been computed, and if so, it just returns the stored value. Otherwise, it invokes the recursive rule to compute it. See the code block below. The initial call is memoized-lcs(m, n).

The running time of the memoized version is $O(mn)$. To see this, observe that there are $m + 1$ possible values for i, and $n + 1$ possible values for j. Each time we call memoized-lcs(i, j), if it has already been computed then it returns in $O(1)$ time. Each call to memoized-lcs(i, j) generates a constant number of additional calls. Therefore, the time needed to compute the initial value of any entry is $O(1)$, and all subsequent calls with the same arguments is $O(1)$. Thus, the total running time is equal to the number of entries computed, which is $O((m + 1)(n + 1)) = O(mn)$.

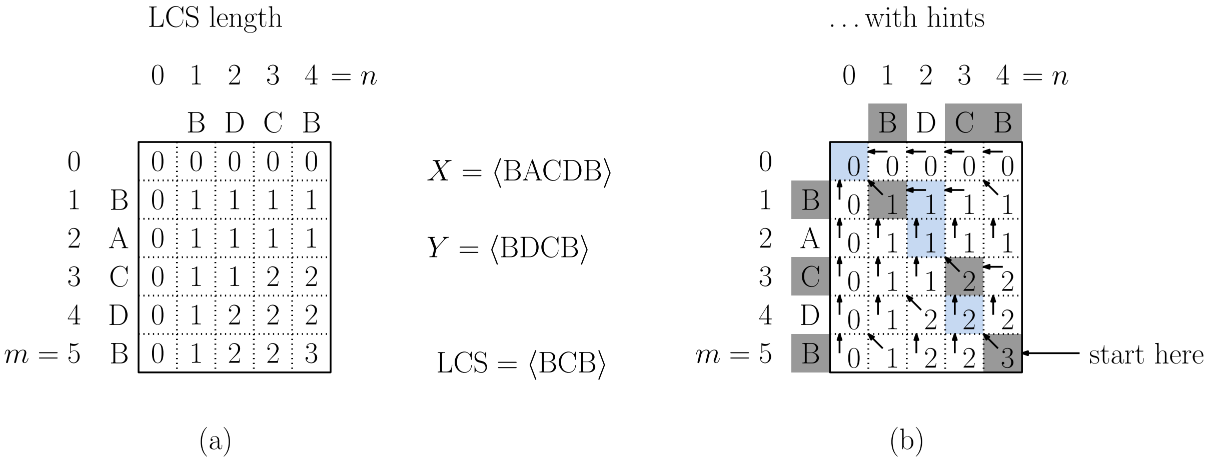
Bottom-up implementation: The alternative to memoization is to just create the lcs table in a bottom-up manner, working from smaller entries to larger entries. By the recursive rules, in order to compute lcs$[i, j]$, we need to have already computed lcs$[i-1, j-1]$, lcs$[i-1, j]$, and lcs$[i, j-1]$. Thus, we can compute the entries row-by-row or column-by-column in increasing order. See the code block below and Fig. 43(a). The running time and space used by the algorithm are both clearly $O(mn)$.

Memoized Longest Common Subsequence

```
memoized-lcs(i,j) {
    if (lcs[i,j] has not yet been computed) {
        if (i == 0 || j == 0)                  // basis case
            lcs[i,j] = 0
        else if (x[i] == y[j])                 // last characters match
            lcs[i,j] = memoized-lcs(i-1, j-1) + 1
        else                                   // last characters don’t match
            lcs[i,j] = max(memoized-lcs(i-1, j), memoized-lcs(i, j-1))
    }
    return lcs[i,j]                            // return stored value
}
```

Bottom-up Longest Common Subsequence

```
bottom-up-lcs() {
    lcs = new array [0..m, 0..n]
    for (i = 0 to m) lcs[i,0] = 0              // basis cases
    for (j = 0 to n) lcs[0,j] = 0
    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
                lcs[i,j] = lcs[i-1, j-1] + 1
            else
                lcs[i,j] = max(lcs[i-1, j], lcs[i, j-1])
        }
    }
    return lcs[m, n]                            // final lcs length
}
```



		0	1	2	3	4 = n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
$m = 5$	B	0	1	2	2	3

Fig. 43: Contents of the lcs array for the input sequences $X = \langle BACDB \rangle$ and $Y = \langle BCDB \rangle$. The numeric table entries are the values of lcs$[i, j]$ and the arrow entries are used in the extraction of the sequence.

Extracting the LCS: The algorithms given so far compute only the length of the LCS, not the actual sequence. The remedy is common to many other DP algorithms. Whenever we make a decision, we save some information to help us recover the decisions that were made. We then work backwards, unraveling these decisions to determine all the decisions that led to the optimal solution. In particular, the algorithm performs three possible actions:

add$_{XY}$: Add $x_i (= y_j)$ to the LCS ($\nwarrow$ in Fig. 43(b)) and continue with lcs$[i-1, j-1]$

skip$_X$: Do not include x_i to the LCS ($\uparrow$ in Fig. 43(b)) and continue with lcs$[i-1, j]$

skip$_Y$: Do not include y_j to the LCS ($\leftarrow$ in Fig. 43(b)) and continue with lcs$[i, j-1]$

An updated version of the bottom-up computation with these added hints is shown in the code block below and Fig. 43(b).

Bottom-up Longest Common Subsequence with Hints

```
bottom-up-lcs-with-hints() {
    lcs = new array [0..m, 0..n]                 // stores lcs lengths
    h = new array [0..m, 0..n]                   // stores hints
    for (i = 0 to m) { lcs[i,0] = 0;  h[i,0] = skipX }
    for (j = 0 to n) { lcs[0,j] = 0;  h[0,j] = skipY }
    for (i = 1 to m) {
        for (j = 1 to n) {
            if (x[i] == y[j])
                { lcs[i,j] = lcs[i-1, j-1] + 1;  h[i,j] = addXY }
            else if (lcs[i-1, j] >= lcs[i, j-1])
                { lcs[i,j] = lcs[i-1, j];  h[i,j] = skipX }
            else
                { lcs[i,j] = lcs[i, j-1];  h[i,j] = skipY }
        }
    }
    return lcs[m, n]                             // final lcs length
}
```

How do we use the hints to reconstruct the answer? We start at the the last entry of the table, which corresponds to lcs(m, n). In general, suppose that we are visiting the entry corresponding to lcs(m, n). If $h[i, j] = \text{add}_{XY}$, we know that $x_i (= y_j)$ is appended to the LCS sequence, and we continue with entry $[i-1, j-1]$. If $h[i, j] =$ skip$_X$ we know that x_i is not in the LCS sequence, and we continue with entry $[i-1, j]$. If $h[i, j] = \text{skip}_Y$ we know that y_j is not in the LCS sequence, and we continue with entry $[i, j-1]$. Because the characters of the LCS are generated in reverse order, we *prepend* each one to a sequence, so that when we are done, the sequence is in proper order.

Extracting the LCS using the Hints

```
get-lcs-sequence() {
    LCS = new empty character sequence
    i = m; j = n                               // start at lower right
    while(i != 0 or j != 0)                    // loop until arriving at upper left
        switch h[i,j]
            case addXY:                        // add x[i] (= y[j])
                prepend x[i] (or equivalently y[j]) to front of LCS
                i--;  j--;    break
            case skipX: i--;  break            // skip x[i]
            case skipY: j--;  break            // skip y[j]
    return LCS
}
```

Lecture 13: Dynamic Programming: Chain Matrix Multiplication

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

$$C \;=\; A_1 \cdot A_2 \cdots A_n$$

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. 44). The number of columns of A must equal the number of rows of B. In particular for $1 \leq i \leq p$ and $1 \leq j \leq r$, we have

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

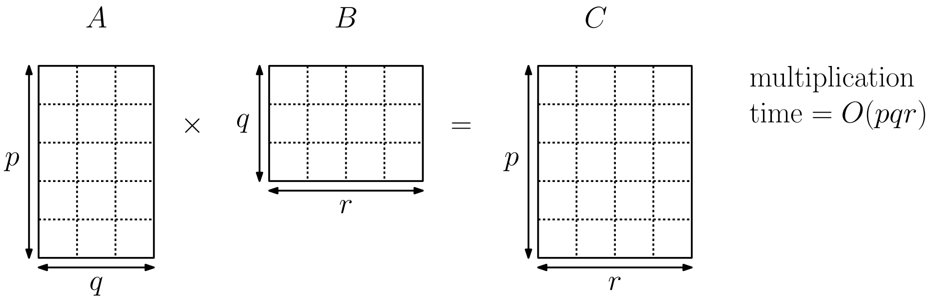


Fig. 44: Matrix Multiplication.

This corresponds to the (hopefully familiar) rule that the $[i,j]$ entry of C is the dot product of the ith (horizontal) row of A and the jth (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 5×4, A_2 be 4×6 and A_3 be 6×2.

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

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

Chain Matrix Multiplication Problem: Given a sequence of matrices $A_1, \ldots, A_n$ and dimensions $p_0, \ldots, p_n$ 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 and the total number of operations.

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, $1 \le k \le n-1$,

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

Thus the problem of determining the optimal sequence reduces to two decisions:

- How do we decide where to split the chain? (what is k?)
- How do we parenthesize the subsequences $A_{1..k}$ 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 the case in the other dynamic programming solutions we have seen, we will try *all possible* choices of k and take the best of them. This is not as inefficient as it might sound, since 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, only polynomial growth.

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

Recursive formulation: Let's explore how to express the optimum cost of multiplication in a recursive form. Later we will consider how to efficiently implement this recursive rule. We will subdivide the problem into subproblems by considering subsequences of matrices. In particular, for $1 \le i \le j \le n$, let $m(i,j)$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The desired total cost of multiplying all the matrices is that of computing the entire chain $A_{1..n}$, which is given by $m(1,n)$. 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 into two groups $A_{i..k}$ times $A_{k+1..j}$, by considering each k, $i \le k < j$ (see Fig. 45).

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 $m(i, j)$.

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

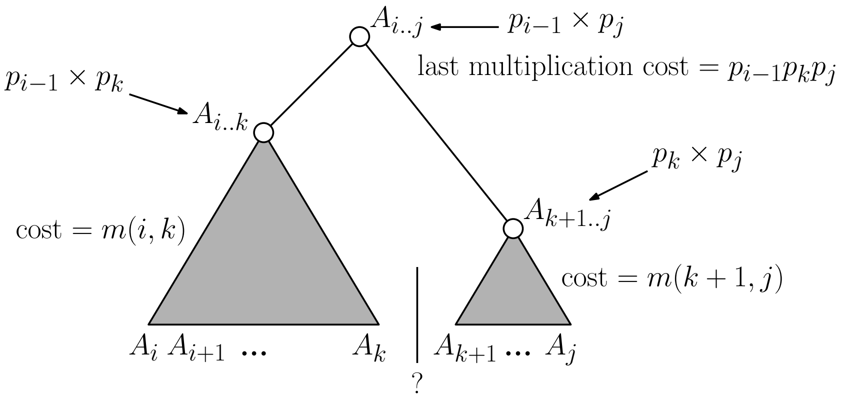


Fig. 45: Dynamic programming decision.

Bottom-up implementation: As with other DP problems, there are two natural implementations of the recursive rule that will lead to an efficient algorithm. One is memoization (which we will leave as an exercise), and the other is bottom-up calculation. We will consider just the latter.

To do this, we will store the values of $m(i, j)$ in a 2-dimensional array $m[1..n, 1..n]$. The trickiest part of the process 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. Note that we cannot just compute the matrix in the simple row-by-row order that we used for the longest common subsequence problem. To see why, suppose that we are computing the values in row 3. When computing $m[3, 5]$, we would need to access both $m[3, 4]$ and $m[4, 5]$, but $m[4, 5]$ is in row 4, which has not yet been computed.

Instead, the trick is to compute *diagonal-by-diagonal* working out from the middle of the array. In particular, we organize our computation according to the number of matrices in the subsequence. For example, $m[3, 5]$ represents a chain of $5 - 3 + 1 = 3$ matrices, whereas $m[3, 4]$ and $m[4, 5]$ each represent chains of only two matrices. We first solve the problem for chains of length 1 (which is trivial), then chains of length 2, and so on, until we come to $m[1, n]$, which is the total chain of length n.

To do this, for $1 \le i \le j \le n$, let $L = j - i + 1$ denote the length of the subchain being multiplied. How shall we set up the loops to do this? The case $L = 1$ is trivial, since there is only one matrix, and nothing needs to be multiplied, so we have $m[i, i] = 0$. Otherwise, our outer loop runs from $L = 2, \ldots, n$. If a subchain of length L starts at position i, then $j = i + L - 1$. Since $j \le n$, we have $i + L - 1 \le n$, or in other words, $i \le n - L + 1$. So our inner loop will be based on i running from 1 up to $n - L + 1$. The code is presented in the code block below. (Also, see Fig. 46 for an example.) We will explain below the purpose of the s array.

The array $s[i, j]$ will be explained below. It will be used to extract the actual multiplication sequence. The running time of the procedure is $O(n^3)$. This is because we have three nested loops, and each can iterate at most n times. (A more careful analysis would show that the total number of iterations grows roughly as $n^3/6$.)

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

```
Matrix-Chain(p[0..n]) {
    s = array[1..n-1, 2..n]
    for (i = 1 to n) m[i, i] = 0                // initialize
    for (L = 2 to n) {                          // L = length of subchain
        for (i = 1 to n - L + 1) {
            j = i + L - 1
            m[i,j] = INFINITY
            for (k = i to j - 1) {              // check all splits
                cost = m[i, k] + m[k+1, j] + p[i-1]*p[k]*p[j]
                if (cost < m[i, j]) {           // found a new optimum?
                    m[i, j] = cost              // ...save its cost
                    s[i, j] = k                 // ...and the split marker
                }
            }
        }
    }
    return m[1, n] (final cost) and s (splitting markers)
}
```

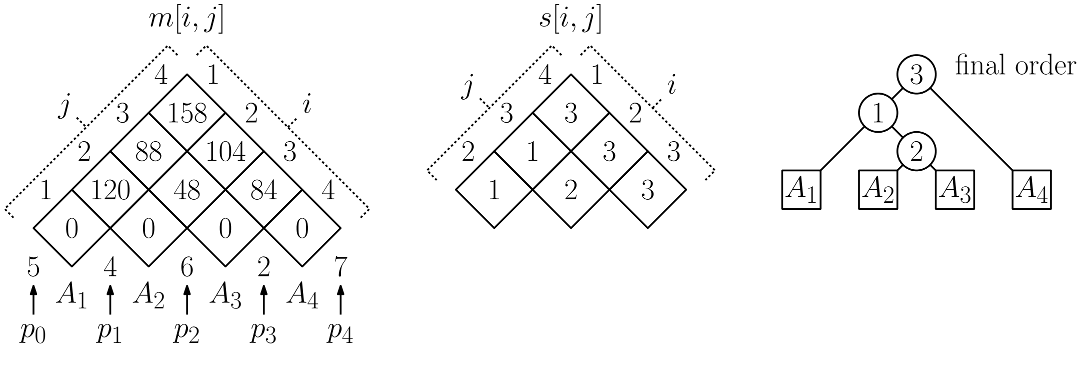


Fig. 46: Chain matrix multiplication for the product $A_1 \cdots A_4$, where A_i is of dimension $p_{i-1} \times p_i$.

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 do-mult does this computation and below returns a matrix (see Fig. 46).

Extracting Optimum Sequence

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

It's a good idea to trace through this example to be sure you understand it.

Lecture 14: Bellman-Ford Shortest Paths

Shortest Paths with Negative Edge Weights: Suppose that we are given a digraph $G = (V, E)$ with numeric edge weights, $w(u, v)$ for each $(u, v) \in E$. We consider the possibility that the edge weights might be negative. (For example, an edge weight might denote the cost of a transaction. Some transactions may have a net benefit, which can be modeled as a negative weight edge.)

Recall that the *length* (or *cost*) of a path is the sum of edge weights along the path. Let s be a designated source vertex. Also recall that the *single-source shortest path problem* is that of computing the length of the shortest path from s to every vertex of the digraph. As before, let $\delta(s, u)$ denote the length of this shortest path from s to u. (By definition $\delta(s, s) = 0$.)

We have seen in earlier lectures that Dijkstra's algorithm solves the single-source shortest path problem, under the assumption that the edge weights are nonnegative.[7] We also saw that shortest paths are undefined if you have cycles of total negative cost. (Because you can make the length shorter and shorter by looping through the cycle as often as you like.)

What if you have negative edge weights, but no negative cost cycles? Dijkstra's algorithm may fail to give a correct answer, but the length of a path is still well defined. Today, we shall present the Bellman-Ford algorithm, which solves this problem. This algorithm is slower that Dijkstra's algorithm in the worst case, running in $\Theta(nm)$ time. (Recall that Dijkstra runs in $O(m \log n)$ time.)

We will simply "assume" that the input digraph has no negative cost cycles. As an exercise, you are encouraged to think about how to modify this algorithm so that, in addition to computing the shortest paths, it can also determine whether any negative cost cycles exist.

Dynamic Programming Approach: As with all DP solutions, our algorithm will be based on solving a series of subproblems. The question is how to define a reasonable subproblem? To help with this, we will first start with an important observation about shortest paths in the absence of negative cost cycles.

Claim: If G has no negative cost cycles, then there is a shortest path from s to any vertex t that does not repeat any node (and in particular, it consists of at most $n - 1$ edges).

[7] You might be tempted to try the following "fix." Let's add a huge constant to every edge weight, so that they are now all nonnegative, and then let's run Dijkstra's algorithm on the resulting graph. As an exercise, show that this method will generally fail. Hint: Consider paths of equal total length but that use a different number of edges.

Proof: Suppose to the contrary that the shortest path from s to t did repeat a node. Then there would be a cycle in the path. Since G has no negative cost cycles, we can remove this cycle from the path without increasing the path's total cost. If we repeat this for every repeated vertex, we will eventually have a path that contains no repetitions.

Since we know that the shortest path contains at most $n-1$ edges, this suggests that we approach the subproblems by limiting the number of edges along the path. Towards this end, for $0 \leq i \leq n-1$ and $v \in V$, define $d(i,v)$ to be equal to the length of the shortest path from s to v that uses at most i edges. By the above claim, if we succeed in computing $d(n-1,v)$ for all $v \in V$, we have solved the problem. (At least we have computed the distance to each node. We should still consider how to recover the shortest paths.)

Let's see if we can derive a recursive rule for computing $d(i,v)$. As always, we need a basis case. Clearly, if $i = 0$, then we have $d(0,s) = 0$ and for all other vertices $d(0,v) = \infty$. Next, let's consider the induction step. We want to compute the shortest path from s to v that uses i edges, under the assumption that we know how to get to every vertex in the graph using $i-1$ edges. Any shortest path from s to v of length i will consist of an initial segment from s to some vertex u of length $i-1$, followed by a single edge from u to v. Since we don't know which vertex u provides the shortest path, we will just consider all possibilities. The basic process is the same *relaxation* step we discussed with Dijkstra's algorithm. In particular, if $d(i-1,u)+w(u,v) < d(i,v)$ then set $d(i,v) = d(i-1,u) + w(u,v)$. Also, to remember the way from v back to the source, we set $\mathrm{pred}[u] = v$.

This suggests the algorithm which is presented in the following code block. A sample execution is illustrated in Fig. 47.

Bellman-Ford Algorithm

```
bellman-ford(G = (V,E)) {
    d = new array[0..n-1, V]
    pred = new array[V]
    for each (u in V) {                // initialization
        d[0, u] = +infinity
        pred[u] = null
    }
    d[0, s] = 0
    for (i = 1 to n-1) {               // repeat n-1 times
        for each (v in V)
         d[i, v] = +infinity
        for each ((u, v) in E) {       // relax along each edge
            if (d[i-1, u] + w(u, v) < d[i, v]) {
                d[i, v] = d[i-1, u] + w(u,v)
                pred[v] = u
            }
        }
    }
}
```

The $\Theta(nm)$ running time is pretty obvious, since there are two main nested loops, one iterated $n-1$ times and the other iterated m times. Note that the space is $O(n^2)$, since i takes on n possible values and v takes on n possible values.

Reducing Space Requirements: It is annoying that the algorithm uses $O(n^2)$ total space. Can we do better? First, off, we may an easy improvement by noting that $d(i,v)$ depends only on the array elements $d(i-1,u)$, for all $u \in V$. This implies that we need only maintain two rows of the d array at any time. This allows us to reduce the space to $O(n)$.

What is rather remarkable however, is that we do not even need the index i. In particular, suppose that we jut eliminate the index i from our array references altogether. For example, the main for-loop would look as follows:

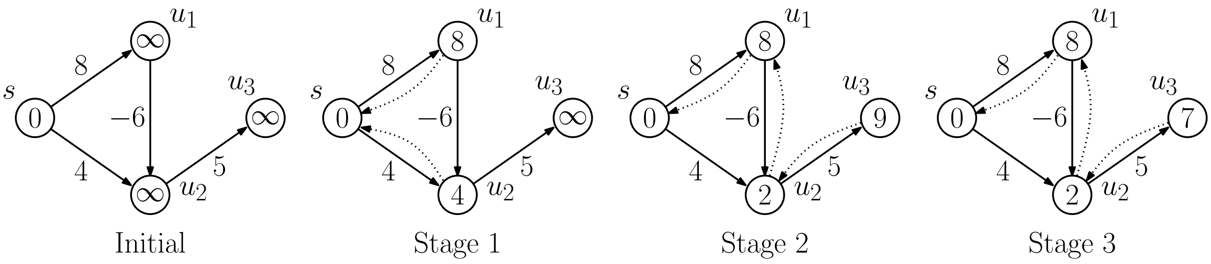


Fig. 47: Bellman-Ford Algorithm.

```
for (i = 1 to n-1) {              // repeat n-1 times
    for each ((u, v) in E) {      // relax along each edge
        if (d[u] + w(u, v) < d[v]) {
            d[v] = d[u] + w(u,v)
            pred[v] = u
        }
    }
}
```

Claim: Consider the modified algorithm, where d is a 1-dimensional array. Then after iteration i of the for-loop:

(i) there exists a path of length $d[v]$ from s to v, and

(2) $d[v]$ is less than or equal to the length of any path from s to v that uses i edges or fewer, that is, $d[v] \leq d[i, v]$.

Proof: To prove (i), observe that the d values arise from relaxations. Each relaxation propagates a path from one node to another along edges of the graph, so there always exists a path of this length. (It need not be a shortest path.)

To prove (ii), observe that whatever relaxation resulted in $d[i, v]$'s value was also considered when updating $d[v]$, and so $d[v]$ can be no larger than $d[i, v]$.

It is interesting to note that after stage i, we may have $d[v] < d[i, v]$. This depends on being very lucky in the ordering of edges in E. For example, consider the graph of Fig. 47. Consider the order in which relaxations are performed in Stage 2. If the relaxation was performed along edge (u_1, u_2) first, we would update $d[u_2] = 8 - 6 = 2$. Then, if after this (still in Stage 2), we performed a relaxation on edge (u_2, u_3), we would now set $d[u_3] = 2 + 5 = 7$. If, on the other hand, we had ordered the edges in the opposite order, however, the (u_2, u_3) relaxation would have been performed *before* we updated $d[u_2]$, and so the result would be exactly as shown in the figure. We would have to wait until Stage 3 for $d[u_3]$ to obtain its final value.

Since we generally cannot predict how the edges are being ordered, we have to run the algorithm all the way to Stage $n - 1$ to be sure we have a correct answer.

Lecture 15: Network Flows: Basic Definitions

Network Flow: "Network flow" is the name of a variety of related graph optimization problems, which are of fundamental value. We are given a *flow network*, which is essentially a directed graph with nonnegative edge weights. We think of the edges as "pipes" that are capable of carrying some sort of "stuff." In applications, this stuff can be any measurable quantity, such as fluid, megabytes of network traffic, commodities, currency, and so on. Each edge of the network has a given *capacity*, that limits the amount of stuff it is able to carry. The idea is to find out how much flow we can push from a designated source node to a designated sink node.

Although the network flow problem is defined in terms of the metaphor of pushing fluids, this problem and its many variations find remarkably diverse applications. These are often studied in the area of operations research.

The network flow problem is also of interest because it is a restricted version of a more general optimization problem, called *linear programming*. A good understanding of network flows is helpful in obtaining a deeper understanding of linear programming.

Flow Networks: A *flow network* is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonegative *capacity* $c(u, v) \geq 0$. (In our book, 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. 48).

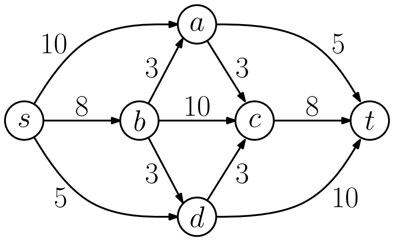


Fig. 48: A flow network.

We assume that there is no edge entering s and no edge leaving t. Such a network is sometimes called an s-t *network*. We also assume that every vertex lies on some path from the source to the sink.[8] This implies that $m \geq n - 1$, where $n = |V|$ and $m = |E|$. It will also be convenient to assume that all capacities are integers. (We can assume more generally that the capacities are rational numbers, since we can convert them to integers by multiplying them by the least common multiple of the denominators.)

Flows, Capacities, and Conservation: Given an s-t network, a *flow* (also called an s-t *flow*) is a function f that maps each edge to a nonnegative real number and satisfies the following properties:

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

Flow conservation (or flow balance): For all $v \in V \setminus \{s, t\}$, the sum of flow along edges into v equals the sum of flows along edges out of v.

We can state flow conservation more formally as follows. First off, let us make the assumption that if (u, v) is *not* an edge of E, then $f(u, v) = 0$. We then define the total flow into v and total flow out of v as:

$$f^{\text{in}}(v) \;=\; \sum_{u \in V} f(u, v) \qquad \text{and} \qquad f^{\text{out}}(v) \;=\; \sum_{w \in V} f(v, w).$$

Then flow conservation states that $f^{\text{in}}(v) = f^{\text{out}}(v)$, for all $v \in V \setminus \{s, t\}$. Note that flow conservation *does not* apply to the source and sink, since we think of ourselves as pumping flow from s to t.

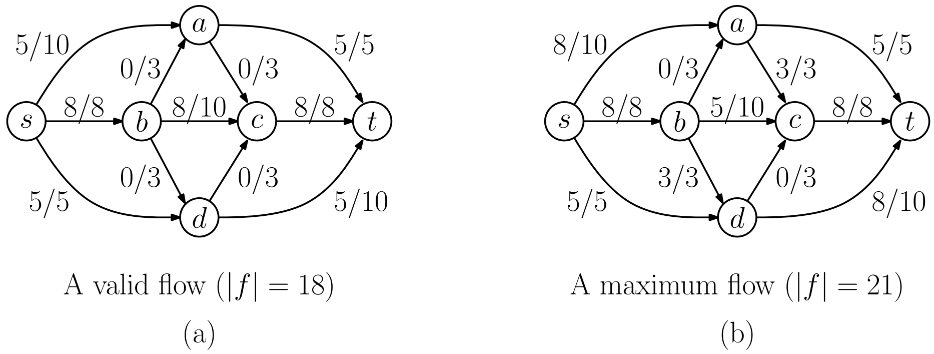
Two examples are shown in Fig. 49, where we use the notation f/c on each edge to denote the flow f and capacity c for this edge.

The quantity $f(u, v)$ is called the *flow* along edge (u, v). We are interested in defining the total flow, that is, the total amount of fluid flowing from s to t. The *value* of a flow f, denoted $|f|$, is defined as the sum of flows out of s, that is,

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

(For example, the value of the flow shown in Fig. 49(a) 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 prove this later.

[8] Neither of these is an essential requirement. Given a network that fails to satisfy these assumptions, we can easily generate an equivalent one that satisfies both.



A valid flow ($|f| = 18$)

(a)

A maximum flow ($|f| = 21$)

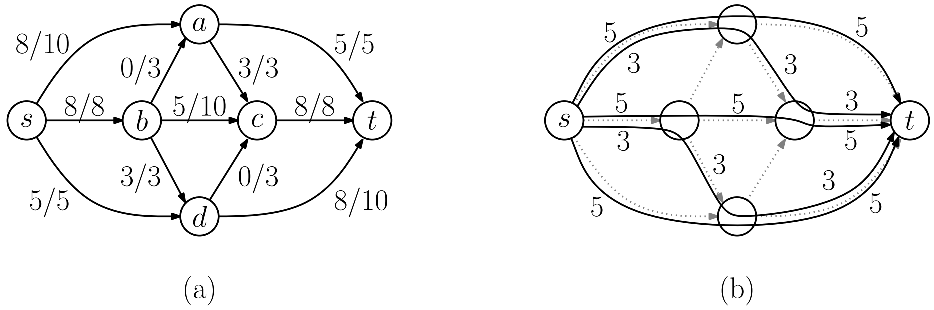
(b)

Fig. 49: A valid flow and a maximum flow.

Maximum Flow: Given an s-t network, an obvious optimization problem is to determine a flow of maximum value. More formally, 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. (For example, in Fig. 49(b) we show flow of value $8 + 8 + 5 = 21$, which can be shown to be the maximum flow for this network.) Note that, although the value of the maximum flow is unique, there may generally be many different flow functions that achieve this value.

Path-Based Flows: The definition of flow we gave above is sometimes call the *edge-based* definition of flows. An alternative, but mathematically equivalent, definition is called the *path-based* definition of flows. Define an s-t *path* to be any simple path from s to t. For example, in Fig. 48, $\langle s, a, t\rangle$, $\langle s, b, a, c, t\rangle$ and $\langle s, d, c, t\rangle$ are all examples of s-t paths. There may generally be an exponential number of such paths (but that is alright, since this just a mathematical definition).

A *path-based flow* is a function that assigns each s-t path a nonnegative real number such that, for every edge $(u, v) \in E$, the sum of the flows on all the paths containing this edge is at most $c(u, v)$. Note that there is no need to provide a flow conservation constraint, because each path that carries a flow into a vertex (excluding s and t), carries an equivalent amount of flow out of that vertex. For example, in Fig. 50(b) we show a path-based flow that is equivalent to the edge-based flow of Fig. 50(a). The paths carrying zero flow are not shown.



(a)

(b)

Fig. 50: (a) An edge-based flow and (b) its path-based equivalent.

The *value* of a path-based flow is defined to be the total sum of all the flows on all the s-t paths of the network. Although we will not prove it, the following claim is an easy consequence of the above definitions.

Claim: Given an s-t network G, under the assumption that there are no edges entering s or leaving t, G has an edge-based flow of value x if and only if G has a path-based flow of value x.

Multi-source, multi-sink networks: It may seem overly restrictive to require that there is only a single source and a single sink vertex. Many flow problems have situations in which many source vertices $s_1, \ldots, s_k$ and many sink vertices $t_1, \ldots, t_l$. This can easily be modeled by just adding a special *super-source* s' and a *super-sink* t', and attaching s' to all the s_i and attach all the t_j to t'. We let these edges have infinite capacity (see Fig. 51). Now by pushing the maximum flow from s' to t' we are effectively producing the maximum flow from all the s_i's to all the t_j's.

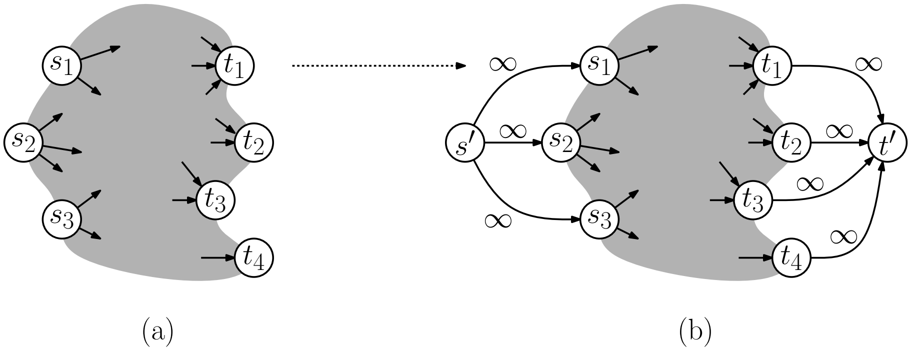


Fig. 51: Reduction from (a) multi-source/multi-sink to (b) single-source/single-sink.

Note that we don't assume any correspondence between flows leaving source s_i and entering t_j. Flows from one source may flow into *any* sink vertex. In some cases, you would like to specify the flow from a certain source must arrive at a designated sink vertex. For example, imagine that the sources are manufacturing production centers and sinks are retail outlets, and you are told the amount of commodity from s_i to arrive at t_j. This variant of the flow problem, called the *multi-commodity flow problem,* is a much harder problem to solve (in fact, some formulations are NP-hard).

Lecture 16: Network Flows: The Ford-Fulkerson Algorithm

Network Flow: We continue discussion of the network flow problem. Last time, we introduced basic concepts, such the concepts s-t networks and flows. Today, we discuss the Ford-Fulkerson Max Flow algorithm, cuts, and the relationship between flows and cuts.

Recall that a *flow network* is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative *capacity* $c(u, v) \geq 0$, with a designated source vertex s and sink vertex t. We assume that there are no edges entering s or exiting t. A *flow* is a function f that maps each edge to a nonnegative real number that does not exceed the edge's capacity, and such that the total flow into any vertex other than s and t equals the total flow out of this vertex. The total *value* of a flow is equal to the sum of flows coming out of s (which, by flow conservation, is equal to the total flow entering t). The objective of the *max flow problem* is to compute a flow of maximum value. Today we present an algorithm for this problem.

Why Greedy Fails: Before considering our algorithm, we start by considering why a simple greedy scheme for computing the maximum flow does not work. The idea behind the greedy algorithm is motivated by the path-based notion of flow. (Recall this from the previous lecture.) Initially the flow on each edge is set to zero. Next, find any path P from s to t, such that the edge capacities on this path are all strictly positive. Let $c_{\min}$ be the minimum capacity of any edge on this path. This quantity is called the *bottleneck capacity* of the path. Push $c_{\min}$ units through this path. For each edge $(u, v) \in P$, set $f(u, v) \leftarrow c_{\min} + f(u, v)$, and decrease the capacity of (u, v) by $c_{\min}$. Repeat this until no s-t path (of positive capacity edges) remains in the network.

While this may seem to be a very reasonable algorithm, and will generally produce a valid flow, it may fail to compute the maximum flow. To see why, consider the network shown in Fig. 52(a). Suppose we push 5 units along the topmost path, 8 units along the middle path, and 5 units along the bottommost path. We have a flow of value 18. After adjusting the capacities (see Fig. 52(b)) we see that there is no path of positive capacity from s to t. Thus, greedy gets stuck.

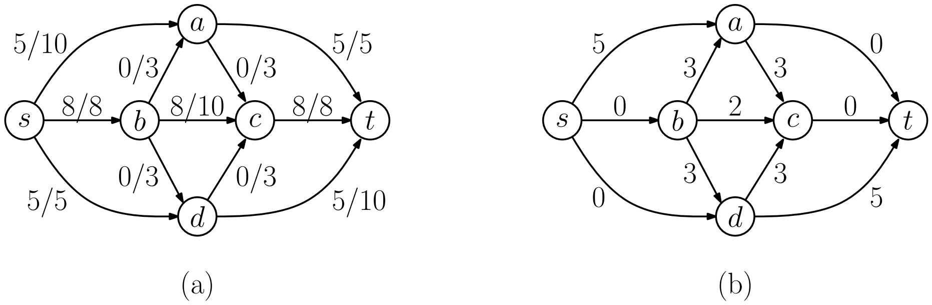


Fig. 52: The greedy flow algorithm can get stuck before finding the maximum flow.

Residual Network: The key insight to overcoming the problem with the greedy algorithm is to observe that, in addition to increasing flows on edges, it is possible to *decrease* flows on edges that already carry flow (as long as the flow never becomes negative). It may seem counterintuitive that this would help, but we shall see that it is exactly what is needed to obtain an optimal solution.

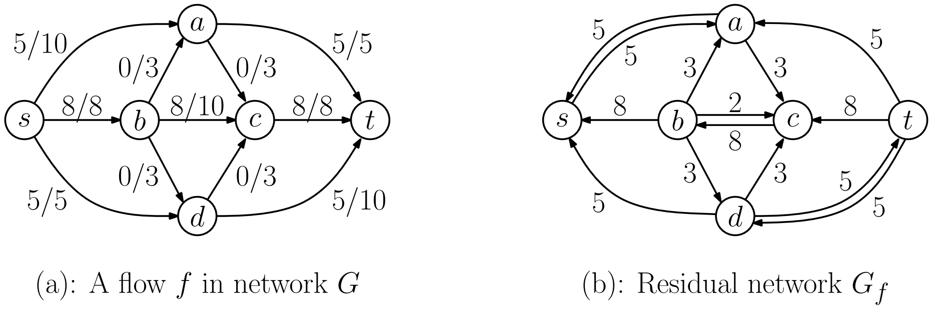
To make this idea clearer, we first need to define the notion of the residual network and augmenting paths. Given a flow network G and a flow f, define the *residual network*, denoted G_f, to be a network having the same vertex set and same source and sink, and whose edges are defined as follows:

Forward edges: For each edge (u, v) for which $f(u, v) < c(u, v)$, create an edge (u, v) in G_f and assign it the capacity $c_f(u, v) = c(u, v) - f(u, v)$. Intuitively, this edge signifies that we can add up to $c_f(u, v)$ additional units of flow to this edge without violating the original capacity constraint.

Backward edges: For each edge (u, v) for which $f(u, v) > 0$, create an edge (v, u) in G_f and assign it a capacity of $c_f(v, u) = f(u, v)$. Intuitively, this edge signifies that we can cancel up to $f(u, v)$ units of flow along (u, v). Conceptually, by pushing positive flow along the reverse edge (v, u) we are decreasing the flow along the original edge (u, v).

Observe that every edge of the residual network has *strictly positive* capacity. (This will be important later on.) Note that each edge in the original network may result in the generation of up to two new edges in the residual network. Thus, the residual network is of the same asymptotic size as the original network.

An example of a flow and the associated residual network are shown in Fig. 53(a) and (b), respectively. For example, the edge (b, c) of capacity 2 signifies that we can add up to 2 more units of flow to edge (b, c) and the edge (c, b) of capacity 8 signifies that we can cancel up to 8 units of flow from the edge (b, c).



(a): A flow f in network G

(b): Residual network G_f

Fig. 53: A flow f and the residual network G_f.

The capacity of each edge in the residual network is called its *residual capacity*. 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. This is formalized in the following lemma. Given two flows f and

f', we define their *sum*, $f + f'$, in the natural way, by summing the flows along each edge. If $f'' = f + f'$, then $f''(u, v) = f(u, v) + f'(u, v)$. Clearly, the value of $f + f'$ is equal to $|f| + |f'|$.

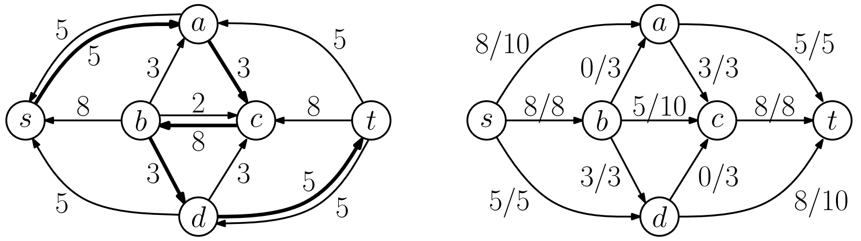
Lemma: Let f be a flow in G and let f' be a flow in G_f. Then $(f + f')$ is a flow in G.

Proof: (Sketch) To show that the resulting flow is valid, we need to show that it satisfies both the capacity constraints and flow conservation. 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 conservation, it is easy to see that their sum will as well. (More generally, any linear combination $\alpha f + \beta f'$ will satisfy flow conservation.)

This lemma suggests that all we need to do to increase the flow is to find any flow in the residual network. This leads to the notion of an augmenting path.

Augmenting Paths and Ford-Fulkerson: Consider a network G, let f be a flow in G, and let G_f be the associated residual network. 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 $c_f(P)$. (Recall that all the edges of G_f are of strictly positive capacity, so $c_f(P) > 0$.) By pushing $c_f(P)$ units of flow along each edge of the path, we obtain a valid flow in G_f, and by the previous lemma, adding this to f results in a valid flow in G of strictly higher value.

For example, in Fig. 54(a) we show an augmenting path of capacity 3 in the residual network for the flow given earlier in Fig. 53. In (b), we show the result of adding this flow to every edge of the augmenting path. Observe that because of the backwards edge (c, b), we have decreased the flow along edge (b, c) by 3, from 8 to 5.



(a): Augmenting path of capacity 3 (b): The flow after augmentation

Fig. 54: Augmenting path and augmentation.

How is this different from the greedy algorithm? The greedy algorithm only increases flow on edges. Since an augmenting path may increase flow on a backwards edge, it may actually *decrease* the flow on some edge of the original network.

This observation naturally suggests an algorithm for computing flows of ever larger value. Start with a flow of weight 0, and then repeatedly find an augmenting path. Repeat this until no such path exists. This, in a nutshell, is the simplest and best known algorithm for computing flows, called the *Ford-Fulkerson method*. (We do not call it an "algorithm," since the method of selecting the augmenting path is not specified. We will discuss this later.) It is summarized in the code fragment below.

There are three issues to consider before declaring this a reasonable algorithm.

- How efficiently can we perform augmentation?
- How many augmentations might be required until converging?
- If no more augmentations can be performed, have we found the max-flow?

Let us consider first the question of how to perform augmentation. First, given G and f, we need to compute the residual network, G_f. This is easy to do in $O(n + m)$ time, where $n = |V|$ and $m = |E|$. We assume that G_f

```
ford-fulkerson-flow(G = (V, E, s, t)) {
    f = 0 (all edges carry zero flow)
    while (true) {
        G’ = the residual-network of G for f
        if (G’ has no s-t augmenting path)
            break                             // no augmentation possible - quit
        P = any-augmenting-path of G’         // augmenting path
        c = minimum capacity edge of P        // amount of augmentation
        augment f by adding c to the flow on every edge of P
    }
    return f
}
```

contains only edges of strictly positive capacity. Next, we need to determine whether there exists an augmenting path from s to t G_f. We can do this by performing either a DFS or BFS in the residual network starting at s and terminating as soon (if ever) t is reached. Let P be the resulting path. Clearly, this can be done in $O(n+m)$ time as well. Finally, we compute the minimum cost edge along P, and increase the flow f by this amount for every edge of P.

Two questions remain: What is the best way to select the augmenting path, and is this correct in the sense of converging to the maximum flow? Next, we consider the issue of correctness. Before doing this, we will need to introduce the concept of a cut.

Cuts: In order to show that Ford-Fulkerson leads to the maximum flow, we need to formalize the notion of a “bottleneck” in the network. Intuitively, the flow cannot be increased forever, because there is some subset of edges, whose capacities eventually become saturated with flow. Every path from s to t must cross one of these saturated edges, and so the sum of capacities of these edges imposes an upper bound on size of the maximum flow. Thus, these edges form a bottleneck.

We want to make this concept mathematically formal. Since such a set of edges lie on every path from s from t, their removal defines a partition separating the vertices that s can reach from the vertices that s cannot reach. This suggests the following concept.

Given a network G, 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 $Y = V \setminus X$, where $s \in X$ and $t \in Y$. We define the *net flow* from X to Y to be the sum of flows from X to Y minus the sum of flows from Y to X, that is,

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

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

For example, Fig. 55 shows a flow of value 17. It also shows a cut $(X,Y) = (\{s,a\},\{b,c,d,t\})$, where $f(X,Y) = 17$.

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 $|f| = f^{\text{out}}(s) = f^{\text{out}}(s) - f^{\text{in}}(s)$. 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

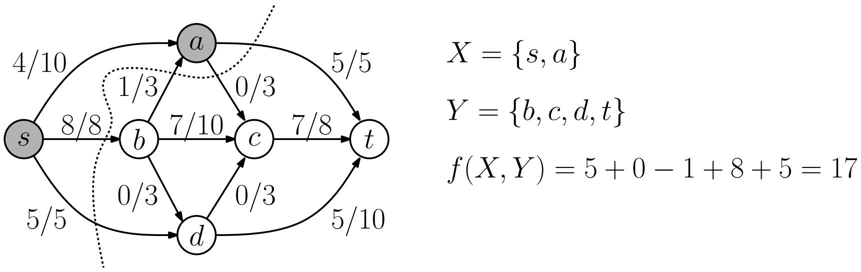


Fig. 55: Flow of value 17 across the cut $(\{s, a\}, \{b, c, d, t\})$.

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 famous 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 on termination, it finds both the minimum cut and the 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) ⇒ (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) ⇒ (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) ⇒ (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.

We have established that, on termination, Ford-Fulkerson generates the maximum flow. But, is it guaranteed to terminate and, if so, how long does it take? We will consider this question next time.

Lecture 17: More on Network Flow

Analysis of Ford-Fulkerson: We have established that, on termination, Ford-Fulkerson generates the maximum flow. But, is it guaranteed to terminate and, if so, how long does it take? First, it is easy to see that it will terminate. Recall that we assumed that all edge capacities are integers. Every augmentation increases the flow by an integer amount, and therefore, after a finite number of augmentations (at most the sum of all the capacities of edges incident to s) the algorithm must terminate.

Recall our convention that $n = |V|$ and $m = |E|$. Since we assume that every vertex is reachable from s, it follows that $m \geq n - 1$. Therefore, $n = O(m)$. Running times of the form $O(n + m)$ can be expressed more simply as $O(m)$.

As we saw last time, the residual network can be computed in $O(n+m) = O(m)$ time and an augmenting path can also be found in $O(m)$ time. Therefore, the running time of each augmentation step is $O(m)$. How many augmentations are needed? Unfortunately, the number could be very large. To see this, consider the example shown in Fig. 56. If the algorithm were smart enough to send flow along the topmost and bottommost paths, each of capacity 100, the algorithm would terminate in just two augmenting steps to a total flow of value 200. However, suppose instead that it foolishly augments 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 will take 200 augmentations until we terminate with the final maximum flow. Without increasing the network's size, we could replace the 100's with as large a number as we like and thus make the running time arbitrarily high.

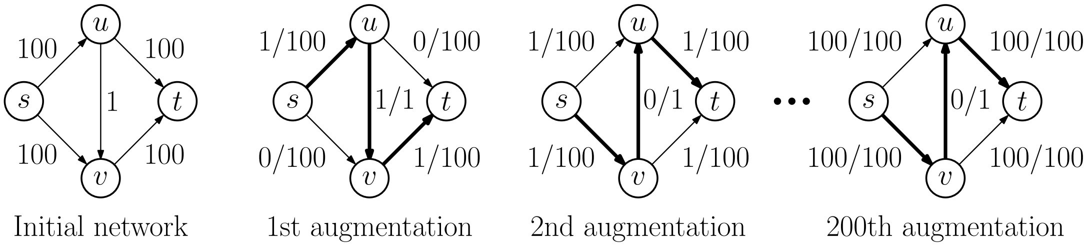


Fig. 56: Bad example for Ford-Fulkerson.

If we let F^* denote the final maximum flow value, the number of augmentation steps can be as high as F^*. If we make the reasonable assumption that each augmentation step takes at least $\Omega(m)$ time, the total running time can be as high as $\Omega(F^* \cdot m)$. Since F^* may be arbitrarily high (it depends neither on n or m), this running time could be arbitrarily high, as a function of n and m.

Faster Max-Flow Algorithms: 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. Below we sketch a few algorithms are more complex than Ford-Fulkerson, but may be superior with respect to asymptotic running times.

Scaling Algorithm: As we saw above, Ford-Fulkerson can perform very badly when the optimum flow is very high. But the above example indicates that we do badly when we augment along paths of very low capacity. What if we were to select paths of high capacity. We could attempt to find the path of maximum capacity, but it turns out that it not necessary to be quite so greedy. Selecting any augmenting path whose residual capacity is within a constant of the maximum is good enough. This gives rise to something called the *scaling algorithm* for max flows.

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

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

Clearly, the maximum flow value cannot exceed C. Next, define Δ to 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* Δ. (That is, we ignore all edges of small capacity.) Repeatedly find an augmenting path in $G_f(\Delta)$, augment the flow along this path, and then compute the residual network $G_{f'}(\Delta)$ for the augmented flow f'. Repeat this until no augmenting paths remain.

Intuitively, each such augmentation has the advantage that it will make big progress, because each augmentation will increase the flow by at least Δ units. When no more augmenting paths remain, set $\Delta \leftarrow \Delta/2$, compute $G_f(\Delta)$ for the new value of Δ, and repeat the process. Eventually, we will have $\Delta = 1$. When the algorithm terminates for $\Delta = 1$, we have the final maximum flow.

It can be shown that for each choice of Δ, the algorithm terminates after $O(m)$ augmentation steps. (This is not trivial. See our text for a proof.) Since each augmentation takes $O(m)$ time, the time spent for each value of Δ is $O(m^2)$. Finally, since we cut the value of Δ in half with each iteration, it is easy to see that we will consider $O(\log C)$ different values of Δ. Whenever C is sufficiently large (that is, when $C/\log C$ is asymptotically larger than m) the scaling algorithm will outperform the Ford-Fulkerson algorithm.

Perhaps more importantly, observe that the total number of bits needed to encode the weights of number of magnitude C is $O(\log C)$. Therefore, the total space needed to encode the input network is $O(m \log C)$. Although the running time of the scaling algorithm is not polynomial in n and m (which would be the ideal), it is polynomial in the *number of bits* needed to encode the input. Thus, it is in some sense a polynomial time algorithm. Algorithms that run in time that is polynomial in the number of bits of input are said to run in *weak polynomial time*. The scaling algorithm is an example of such an algorithm.

Edmonds-Karp Algorithm: Neither of the algorithms we have seen so far runs in "truly" polynomial time (that is, polynomial in n and m, irrespective of the magnitudes of the capacity.) Edmonds and Karp developed the first such algorithm. This algorithm uses Ford-Fulkerson as its basis, 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*. Note that this can be accomplished by using BFS to compute the augmenting path, since BFS effectively finds shortest path based on the number of edges. It can be shown that the total number of augmenting steps using this method is $O(nm)$. (Again, this is not trivial. Our book does not give a proof, but one can be found in the algorithms book by Cormen, Leiserson, Rivest, and Stein.) Recall that each augmenting path can be computed in $O(m)$ time. Thus, the overall running time is $O(nm^2)$.

Other Algorithms: The max-flow problem is widely studied, and there are many different algorithms. Our book discusses one algorithm, called the *pre-flow push algorithm*. There are a number of variants of this algorithm, but the simplest one runs in $O(n^3)$ time. Another quite sophisticated algorithm runs in time $O(\min(n^{2/3}, m^{1/2})m \log n \log U)$, where U is an upper bound on the largest capacity.

Applications of Max-Flow: The network flow problem has a huge number of applications. Many of these applications do not appear at first to have anything to do with networks or flows. This is a testament to the power of this problem. In this lecture and the next, we will present a few applications from our book. (If you need more convincing of this, however, see the exercises in Chapter 7 of KL. There are over 40 problems, most of which involve reductions to network flow.)

Maximum Matching: Earlier in the semester we talked about stable marriage. There are many applications where pairings are to be sought, and there are many criteria for what constitutes a good pairing. We will consider another one here. As in the stable marriage problem, we will present it in the form of a "dating game," but there are many serious applications of this general problem.

Suppose 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 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.) Note that, unlike the stable marriage problem, there are no preferences here, only compatibility and incompatibility constraints.

Recall that an undirected graph $G = (V, E)$ is said to be *bipartite* if V can be partitioned into two sets X and Y, such that every edge has one endpoint in X and the other in Y. This problem can be modeled as an undirected, bipartite 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. 57(a)). Given a graph, a *matching* is defined to be a subset of edges $M \subseteq E$ such that for each $v \in V$, there is at most one edge of M incident to v. Clearly, the objective to the dating problem is to find a maximum matching in G that has the highest cardinality. Such a matching is called a *maximum matching* (see Fig. 57(b)).

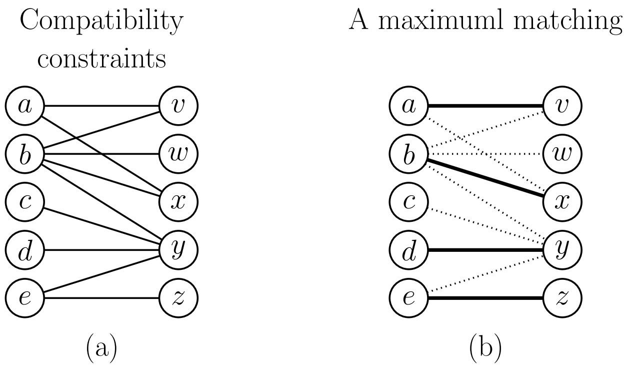


Fig. 57: A bipartite graph G and a maximum matching in G.

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*.

We will now show a reduction from maximum bipartite matching to network flow. In particular, we will show that, given any bipartite graph G on which we want to solve the maximum matching problem, we can convert it into an instance of network flow G', such that the maximum matching on G can be extracted from the maximum flow on G'.

To do this, we 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. 58(a)).

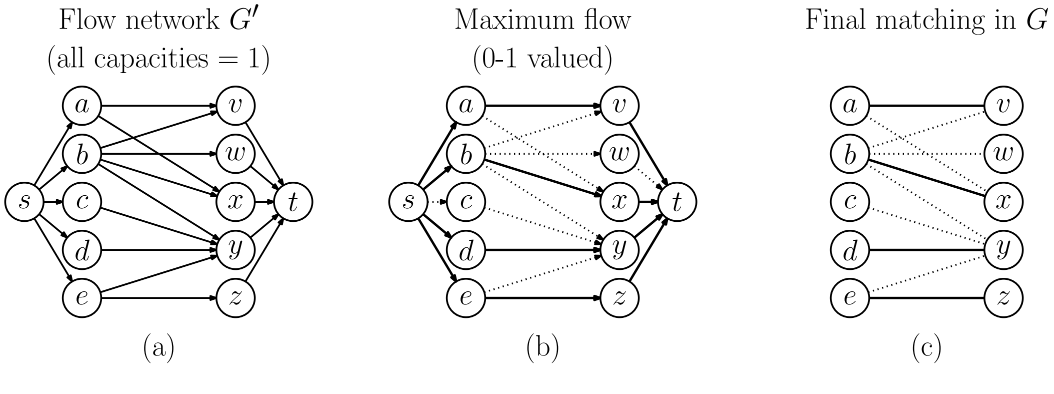

Flow network G' (all capacities = 1)
Maximum flow (0-1 valued)
Final matching in G
s
t
a
b
c
d
e
v
w
x
y
z
(a)
(b)
(c)

Fig. 58: Reducing bipartite matching to network flow.

Now, compute the maximum flow in G' (see Fig. 58(b)). 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. 58(c)).

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. Recall that Ford-Fulkerson runs in time $O(m \cdot F^*)$, where F^* is the final maximum flow. In our case F^* is not very large. In particular, the total capacity of the edges coming out of S is at most $|X| \leq |V| = n$. Therefore, the running time of Ford-Fulkerson on this instance is $O(m \cdot F^*) = O(nm)$.

There are other algorithms for maximum bipartite matching. The best is due to Hopcroft and Karp, and runs in $O(\sqrt{n} \cdot m)$ time.

Lecture 18: Extensions of Network Flow

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 model this by assigning it a negative demand, so that $-d_v$ is its available supply. Intuitively, supplying x units of product is equivalent to demanding receipt of $-x$ units.[9] 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 *supply nodes* ($d_v < 0$), and let T denote the set of *demand nodes* ($d_v > 0$). Note that vertices of S may have incoming edges and vertices of T may have outgoing edges. (For example, in Fig. 59(a), we show a network in which each node is each labeled with its demand.)

Recall that, given a flow f and a node v, $f^{\text{in}}(v)$ 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. We define a *circulation* in G to be a function f that assigns a nonnegative real number to each edge that satisfies the following two conditions.

[9] 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.

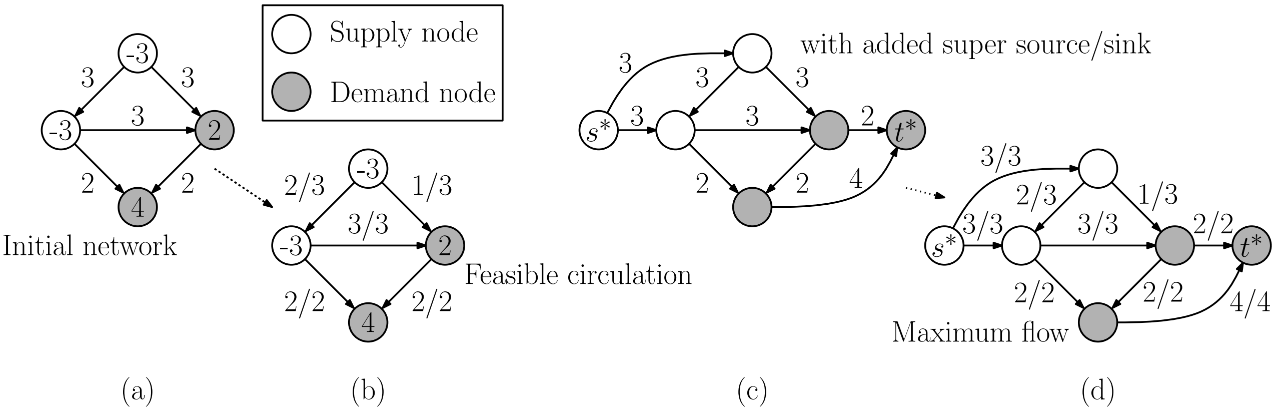


Fig. 59: Reducing the circulation problem to network flow.

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

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

(In Fig. 59(b), we show a valid 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

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

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 $v \in S$, we add an edge (s, v) of capacity $-d_v$, and for each demand node $u \in T$, we add an adge (u, t) of capacity d_v (see Fig. 59(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 the all the demand constraints (see Fig. 59(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.
- ($v \in V \setminus (S \cup T)$) 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 achieve 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 networ $G = (V, E)$, 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 of such a network is shown in Fig. 60(a), and a valid circulation is shown in Fig. 60(b).

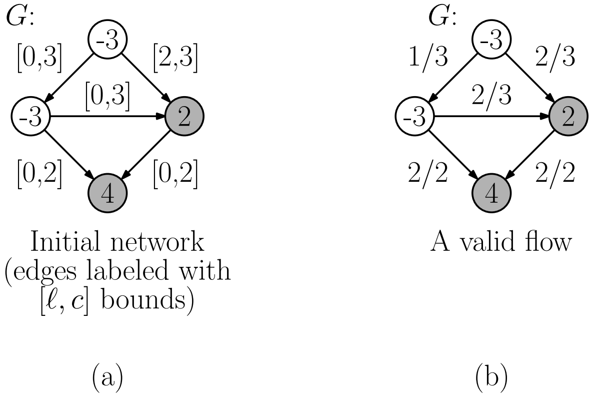


Fig. 60: (a) A network with both upper and lower flow bounds and (b) a valid circulation.

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 (invalid) circulation f_0. This circulation will be defined so it satisfies all the lower flow bounds. In particular, we let $f_0(u, v) = \ell(u, v)$ (see Fig. 61(a)). This circulation may be invalid because f_0 need not 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, then $L_v = d_v$ and we are done. Otherwise, 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 how do we determine whether there exists such a circulation 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 new capacity $c'(u,v)$ of each edge $(u,v) \in E$ to amount of remaining capacity we have, namely $c(u,v) - \ell(u,v)$, and we set the demand d'_v for each node $v \in V$ to the remainder demand $d_v - L_v$ which we had after considering f_0.

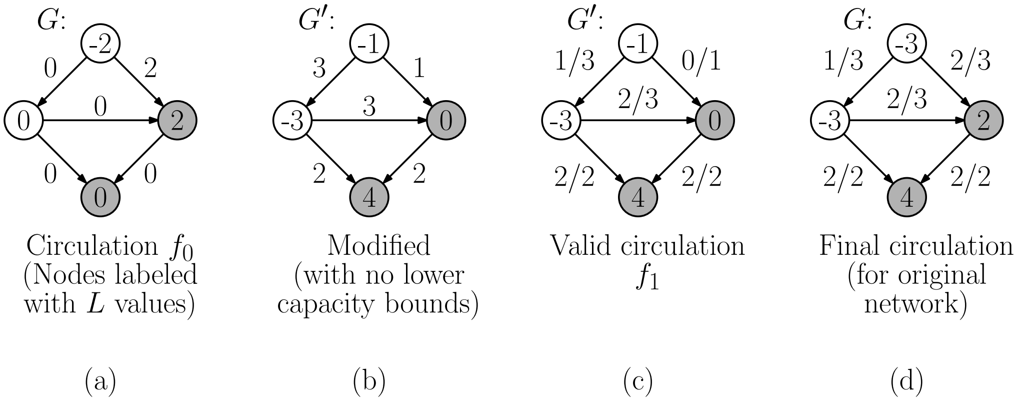

G:
-2
0
2
0
0
2
0
0
0
Circulation f_0 (Nodes labeled with L values)
(a)
G′:
-1
3
1
-3
3
0
2
2
4
Modified (with no lower capacity bounds)
(b)
G′:
-1
1/3
0/1
2/3
-3
0
2/2
2/2
4
Valid circulation f_1
(c)
G:
-3
1/3
2/3
2/3
-3
2
2/2
2/2
4
Final circulation (for original network)
(d)

Fig. 61: Reducing the circulation problem with upper and lower flow bounds to a standard circulation problem.

An example of the resulting circulation network is shown in Fig. 61(b). Note that, unlike G, this network has no lower flow bounds, and so we may apply invoke the *standard* network circulation algorithm to compute a circulation f_1 in G'. The resulting circulation is shown in Fig. 61(c). As mentioned earlier, the final circulation arises by returning $f_0 + f_1$, and is shown in Fig. 61(d).

We prove below that this is a valid circulation for G (with lower flow bounds) if and only if f_1 is a valid circulation 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: (Sketch. 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: To demonstrate the usefulness of circulations with lower flow bounds, let us consider an application problem that arises in the area of data mining. A company sells k different products, and it maintains 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 annoy the customer by asking too many questions. (Otherwise, they will simply not respond.) Based on our knowledge of how many products customer i has purchased, and easily they are annoyed, our marketing people have come up with two bounds $0 \leq c_i \leq c'_i$. We will ask the ith customer about at least c_i products they bought, but (to avoid annoying them) at most c'_i products.

- Again, our marketing people know that we want more information about some products (e.g., new releases) and less about others. To get a balanced amount of information about each product, for the jth product we have two bounds $0 \leq p_j \leq p'_j$, and we will ask at least p_j customers about this product and at most p'_j customers.

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. 62).

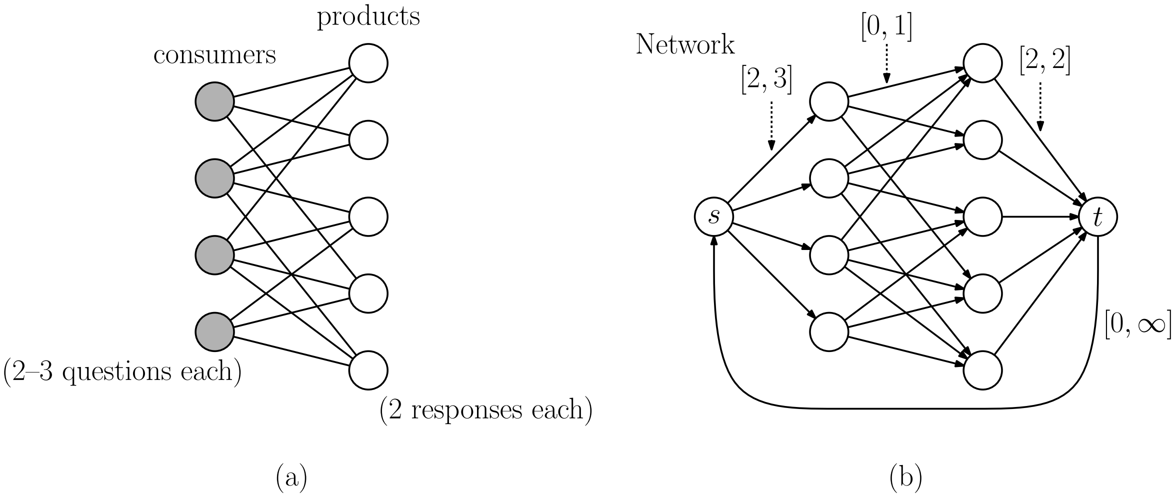


Fig. 62: Reducing the survey design problem to circulation with lower and upper flow bounds.

- For each customer i who purchased product j we create a directed edge (i, j) with an upper flow bounds of 1, respectively. This models the requirement that customer i will be surveyed at most once about product j, and customers will be asked only about products they purchased.
- 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 models the requirement that customer i will be asked about at least c_i products and at most c'_i.
- 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 models the requirement that there are at least p_j and at most p'_j customers will be asked about product j.
- 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. This is needed for technical reasons, since we want a circulation.
- All node demands are set to 0.

It is easy to see that if G has a valid (integer valued) circulation. There is a flow of one unit along edge (i, j) if customer i is surveyed about product j. From our capacity constraints, it follows that 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. Since the node demands are all 0, it follows that the flows through every vertex (including s and t) satisfy flow conservation. This implies that the total number of surveys sent to all the customers (the flow out of s) equals the total number of surveys received on all the products (the flow into t). The converse is also easy to show, namely that a valid survey design implies the existence of a circulation in G. Therefore, there exists a valid circulation in G' if and only there is a valid survey design (see Fig. 63).

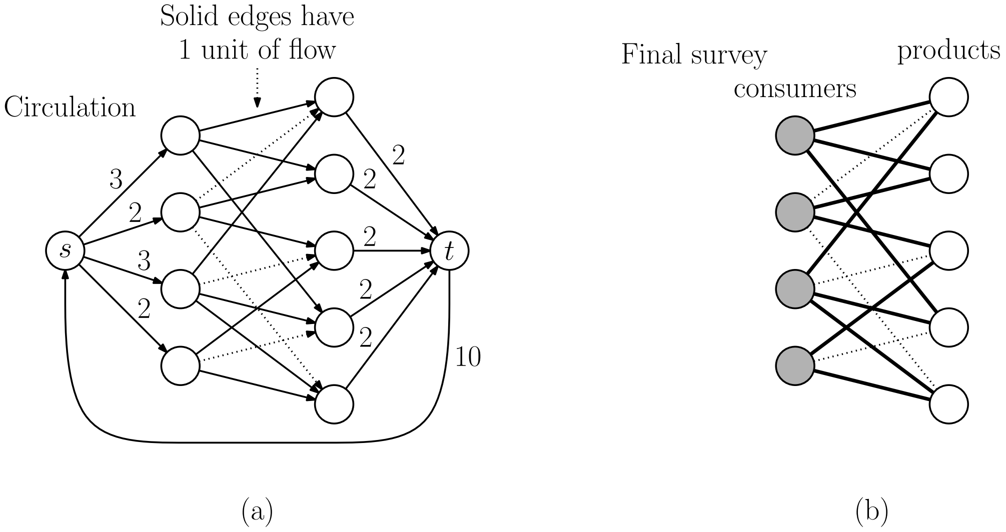


Fig. 63: Reducing the survey design problem to circulation with lower and upper flow bounds.

Lecture 19: NP-Completeness: General Definitions

Efficiency and Polynomial Time: Up to this point of the semester we have been building up your "bag of tricks" for solving algorithmic problems efficiently. 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, etc.), what sort of data structures might be relevant (trees, priority queues, graphs) and what representations would be best (adjacency list, adjacency matrices), what is the running time of your algorithm.

The notion of what we mean by efficient is quite vague. If n is small, a running time of 2^n may be just fine, but when n is huge, even n^2 may be unacceptably slow. In an effort put matters on a clear mathematical basis, algorithm designers observed that there are two very general classes of combinatorial problems: those that can be solved by an intelligent search process and those that involve simple brute-force search. Since most combinatorial problems involve choosing from an exponential set of possibilities, the key distinguishing feature in most cases was whether there existed a *polynomial time algorithm* for solving the problem.

Recall that an algorithm is said to run in *polynomial time* if its worst-case running time is $O(n^c)$, where c is a nonnegative constant. (Note that running times like $O(n \log n)$ are also polynomial time, since $n \log n = O(n^2)$.) A computational problem is said to be solved *efficiently* if it is solvable in polynomial time. Higher worst-case running times, such as 2^n, $n!$, and n^n are not polynomial time.

You can't be serious! You would be quite right to object to this "definition" of efficiently solvable for a number of reasons. First off, if you are interested only in small values of n, a running time of 2^n with a small constant factor may be vastly superior to an algorithm that runs in $O(n^{20})$ and/or where the asymptotic notation hides huge constant factors. There are many problems for which good *average case* solutions exist, but the worst case complexity, which may only arise in very rare instances) may be very bad. On modern architectures, practical efficiency is a function of many issues that have to do with the machine's internal architecture, such as whether the algorithm can be compiled so it makes good use of the machines many processing cores or whether it has good performance with respect to the machine's cache and memory structure.

In spite of its many drawbacks, defining "efficiently solvable" to be "worst-case polynomial time solvable" has a number of mathematical advantages. For example, since the composition of two polynomials is a polynomial, a polynomial time algorithm that makes a polynomial number of calls to a polynomial time function, runs in polynomial time. (For example, an algorithm that makes $O(n^2)$ calls to a function that takes $O(n^3)$ time runs in $O(n^5)$ time, which is still polynomial. This would not be have been true had we defined "efficient" to mean solvable in, say, $O(n^2)$ time.) Also, because we focus on worst-case complexity, we do not need to worry about the distribution of inputs.

Even though you might not agree that all polynomial time algorithms are "efficient," (ignoring the issue of average-case performance) we can hopefully agree that exponential time algorithms, such as those running in 2^n time, are certainly *not* efficient, assuming that n is sufficiently large.

The Emergence of Hard Problems: Near the end of the 60's, although there was great success in finding efficient solutions to many combinatorial problems, there was also a growing list of problems which were "hard" in the sense that no known efficient algorithmic solutions existed for these problems.

A remarkable discovery was made about this time. Many of these believed hard problems turned out to be equivalent, 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. An example of some of these problems is shown in Table 1.

Table 1: Some polynomial-time solvable problems and equivalent (and believed hard) problems.

Complexity Class	**Examples**
Polynomial Time	Minimum Spanning Trees, Shortest Paths, Chain Matrix Multiplication, LCS, Stable Marriage, Maximum Matching Network Flows, Minimum Cut
Equivalent (Believed Hard)	Vertex Cover, Hamiltonian Cycle Boolean Satisfiability, Set Cover, Clique Cover Clique, Independent Set, Graph Coloring Hitting Set, Feedback Vertex Set

The mathematical theory, which was developed by Richard Karp and Stephen Cook, gave rise to the notions of P, NP, and NP-completeness. Since then, thousands of problems were identified as being in this equivalence class. It is widely believed that none of them can be solved in polynomial time, but there is no proof of this fact. This has given rise to one of the biggest open problems in computer science: Is P $=$ NP?

We will investigate this class in the next few lectures. Note that represents a radical departure from what we have been doing so far this semester. 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?

Reasonable Input Encodings: When trying to show the impossibility of achieving a task efficiently, it is important to define terms precisely. Otherwise, we might be beaten by clever cheats. We will treat the input to our problems as a string over some alphabet that has a constant number, but at least two, characters (e.g., a binary bit string or a Unicode encoding). If you think about it for just a moment, every data structure that we have seen this semester can be *serialized* into such a string, without increasing its size significantly.

How are inputs to be encoded? Observe that if you encode an integer in a very inefficient manner, for example, using *unary notation* (so that 8 is represented as 11111111), rather than an efficient encoding (say in binary or decimal[10]), the length of the string increases by exponentially. Why should we care? Observe that if the input size grows exponentially, then an algorithm that ran in exponential time for the short input size may now run in linear time for the long input size. We consider this a cheat because we haven't devised a faster algorithm, we have just made our measuring yardstick much much longer.

[10] The exact choice of the numeric base is not important so long as it is as least 2, since all base representations can be converted to each other with only a constant factor change in the length.

All the representations we have seen this semester (e.g., sets as lists, graphs as adjacency lists or adjacency matrices, etc.) are considered to be reasonable. To determine whether some new representation is reasonable, it should be as concise as possible (in the worst case) and/or it should be possible to convert from an existing reasonable representation to this new form in polynomial time.

Decision Problems and Languages: 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 maximum flow. 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).

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 certainly the more general optimization problem certainly cannot be solved efficiently either. (In fact, if you can solve a decision problem efficiently, it is almost always possible to construct an efficient solution to the optimization problem, but this is a technicality that we won't worry about now.)

Observe that a decision problem can also be thought of as a language recognition problem. For example, we could define a language MST encoding the minimum spanning tree problem as:

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

(Again, when we say (G, k), we mean a reasonable encoding of the pair G and k as a string.) What does it mean to solve the decision problem? When presented with a specific input string $x = \text{serialize}(G, k)$, the algorithm would answer "yes" if $x \in \text{MST}$, that is, if 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. Thus, decision problems are equivalent to language membership problems.

Given an input x, how would we determine whether $x \in \text{MST}$? First, we would decode x as G and k. We would then feed these into any efficient minimum spanning tree algorithm (Kruskal's, say). If the final cost of the spanning tree is at most k, we accept x and otherwise we reject it.

The Class P: We now present an important definition:

Definition: P is the set of all languages (i.e., decision problems) for which membership can be determined in (worst case) polynomial time.

Intuitively, P corresponds to the set of all decisions problems that can be solved efficiently, that is, in polynomial time. Note P is not a language, rather, it is a set of languages. A set of languages that is defined in terms of how hard it is to determine membership is called a *complexity class*. (Therefore, P is a complexity class.)

Since Kruskal's algorithm runs in polynomial time, we have MST $\in$ P. We could define equivalent languages for all of the other optimization problems we have seen this year (e.g., shortest paths, max flow, min cut).

To show that not all languages are (obviously) in P, consider the following:

$$\text{HC} = \{G \mid G \text{ has a simple cycle that visits every vertex of } G\}.$$

Such a cycle is called a *Hamiltonian cycle* and the decision problem is the *Hamiltonian Cycle Problem*. (In Fig. 64(a) we show an example of a Hamiltonian cycle in a graph. If you think that the problem is easy to solve, try to solve the problem on the graph shown in Fig. 64(b), which has one less vertex. Either find a Hamiltonian cycle in this graph or show than none exists. If you thought that was easy, imagine a tessellation of the plane with a million of these triangular configurations, each slightly different than the next.)

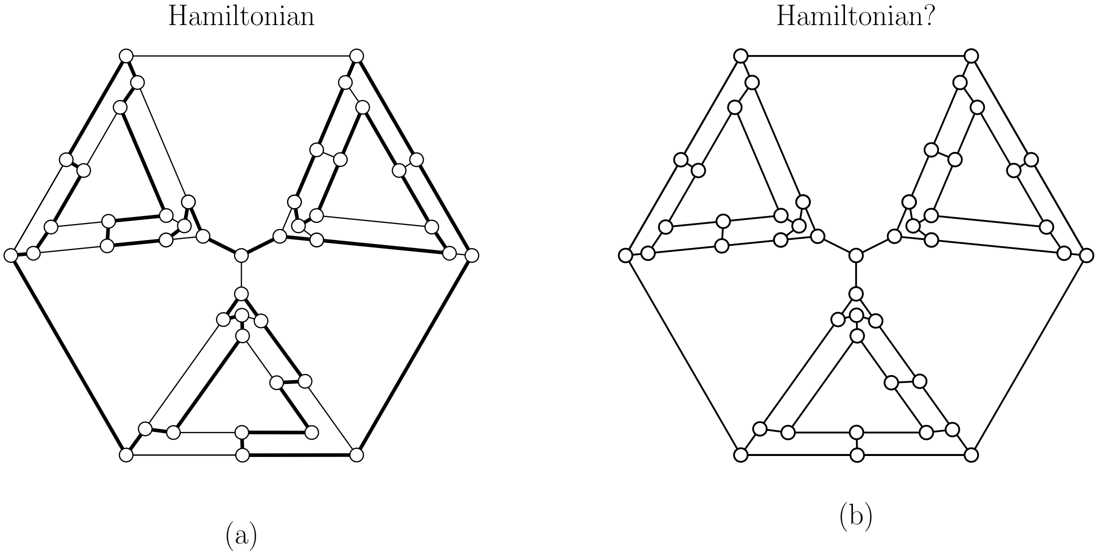


Fig. 64: The Hamiltonian cycle (HC) problem.

Is HC $\in$ P? No one knows the answer for sure, but it is conjectured that it is not. (In fact, we will show that later that HC 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.

Polynomial Time Verification and Certificates: In order to define NP-completeness, we need to first define NP. Unfortunately, providing a rigorous definition of NP will involve a presentation of the notion of *nondeterministic* models of computation, and will take us away from our main focus. (Formally, NP stands for *nondeterministic polynomial time*.) Instead, we will present a very simple, "hand-wavy" definition, which will suffice for our purposes.

To do so, it is important to first introduce the notion of a verification algorithm. Many language recognition problems that may be *hard to solve*, but they have the property that they are *easy to verify* that a string is in the language. Recall the Hamiltonian cycle problem defined above. As we saw, there is no obviously efficient way to find a Hamiltonian cycle in a graph. However, suppose that a graph did have a Hamiltonian cycle and someone wanted to convince us of its existence. This person would simply tell us the vertices in the order that they appear along the cycle. It would be a very easy matter for us to inspect the graph and check that this is indeed a legal cycle that it visits all the vertices 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 has one. (You might ask, but what if the graph did not have one? Don't worry. A verification process is not required to do anything if the input is not in the language.)

The given cycle in the above example is called a *certificate*. A certificate is a piece of information which allows us to verify that a given string is in a language in polynomial time.

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. If there exists a verification algorithm that runs in polynomial time, we say that L can be *verified in polynomial time*.

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

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

There is no known polynomial time verification algorithm for either of these. For example, 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 certainly show us one Hamiltonian cycle, but it is unclear that they could provide us with any easily verifiable piece of information that would demonstrate that this is the only one.

The class NP: We can now define the complexity class NP.

Definition: NP is the set of all languages that can be verified in polynomial time.

Observe that if we can solve a problem efficiently without a certificate, we can certainly solve given the additional help of a certificate. Therefore, P $\subseteq$ NP. 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.

There is one last ingredient that will be needed before defining NP-completeness, namely the notion of a *polynomial time reduction*. We will discuss that next time.

Lecture 20: NP-Completeness: Reductions

Recap: Recall that we have introduced a number of concepts on the way to defining NP-completeness.

Decision Problems/Language recognition: are problems for which the answer is either yes or no. These can also be thought of as language recognition problems, assuming that the input has been encoded as a string. For example:

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

P: is the class of all decision problems which can be solved in polynomial time. While MST $\in$ P, we do not know whether HC $\in$ P (but we suspect not).

Certificate: is a piece of evidence that allows us to *verify* in polynomial time that a string is in a given language. For example, the language HC above, a certificate could be a sequence of vertices along the cycle. (If the string is not in the language, the certificate can be anything.)

NP: is defined to be the class of all languages that can be *verified* in polynomial time. (Formally, it stands for *Nondeterministic Polynomial time*.) Clearly, P $\subseteq$ NP. It is widely believed that P $\neq$ NP.

To define NP-completeness, we need to introduce the concept of a reduction.

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.

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*. We want to prove that U is also hard. How would we do this? We effective want to show that

$$(H \notin \text{P}) \;\Rightarrow\; (U \notin \text{P}).$$

To do this, we could prove the contrapositive,

$$(U \in \mathrm{P}) \;\Rightarrow\; (H \in \mathrm{P}).$$

To show that U is not solvable in polynomial time, we will suppose (towards a contradiction) that a polynomial time algorithm for U did existed, and then we will use this algorithm to solve H in polynomial time, thus yielding a contradiction.

To make this more concrete, suppose that we have a subroutine[11] that can solve any instance of problem U in polynomial time. Given an input x for the problem H, we could translate it into an *equivalent* input x' for U. By "equivalent" we mean that $x \in H$ if and only if $x' \in U$ (see Fig. 65). Then we run our subroutine on x' and output whatever it outputs.

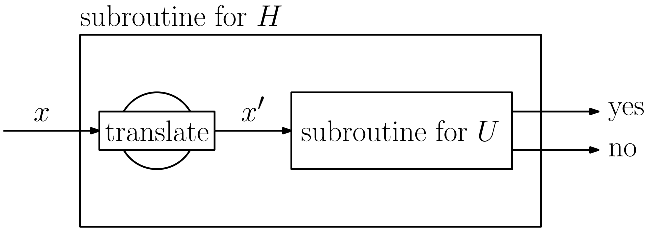


Fig. 65: Reducing H to U.

It is easy to see that if U is solvable in polynomial time, then so is H. We assume that the translation module runs in polynomial time. If so, we say we have a *polynomial reduction* of problem H to problem U, which is denoted $H \leq_P U$. More specifically, this is called a *Karp reduction*.

More generally, we might consider calling the subroutine multiple times. How many times can we call it? Since the composition of two polynomials is a polynomial, we may call it any polynomial number of times. A reduction based on making multiple calls to such a subroutine is called a *Cook reduction*. Although Cook reductions are theoretically more powerful than Karp reductions, every NP-completeness proof that I know of is based on the simpler Karp reductions.

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 three different "colors", such that no two adjacent vertices have the same label (see Fig. 66(a) and (b)).

Coloring arises in various partitioning problems, where there is a constraint that two objects cannot be assigned to the same set of the partition. It is well known that planar graphs can be colored with four colors, and there exists a polynomial time algorithm for doing this. But determining whether three colors are possible (even for planar graphs) seems to be hard, and there is no known polynomial time algorithm.

The 3Col problem will play the role of the known hard problem H. To play the role of 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 distinct 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 $V_1, \ldots, V_k$ such that each V_i is a clique of G (see Fig. 66(c)).

[11] It is important to note here that this supposed subroutine for U is a *fantasy*. We know (or strongly believe) that H cannot be solved in polynomial time, thus we are essentially proving that such a subroutine cannot exist, implying that U cannot be solved in polynomial time.

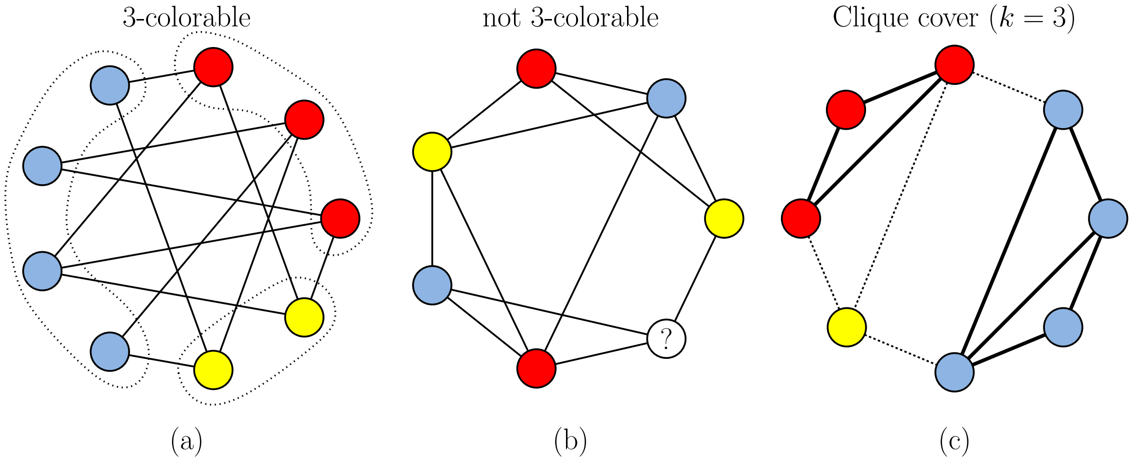


Fig. 66: 3-coloring and Clique Cover.

The clique cover problem arises in 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 at most k groups.

We want to prove that CCov is hard, under the assumption that 3Col is hard, that is,

$$(\text{3Col} \notin \text{P}) \implies (\text{CCov} \notin \text{P}).$$

Again, we'll prove the contrapositive:

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

Let us assume that we have access to a polynomial time subroutine CCov(G, k). Given a graph G and an integer k, this subroutine returns true (or "yes") if G has a clique cover of size k and false otherwise. How can we use this *alleged* subroutine to solve the well-known hard 3Col problem? We need to find a translation, that maps an instance G for 3-coloring into an instance (G', k) for clique cover (see Fig. 67).

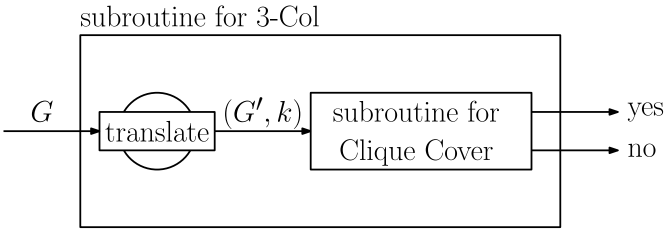
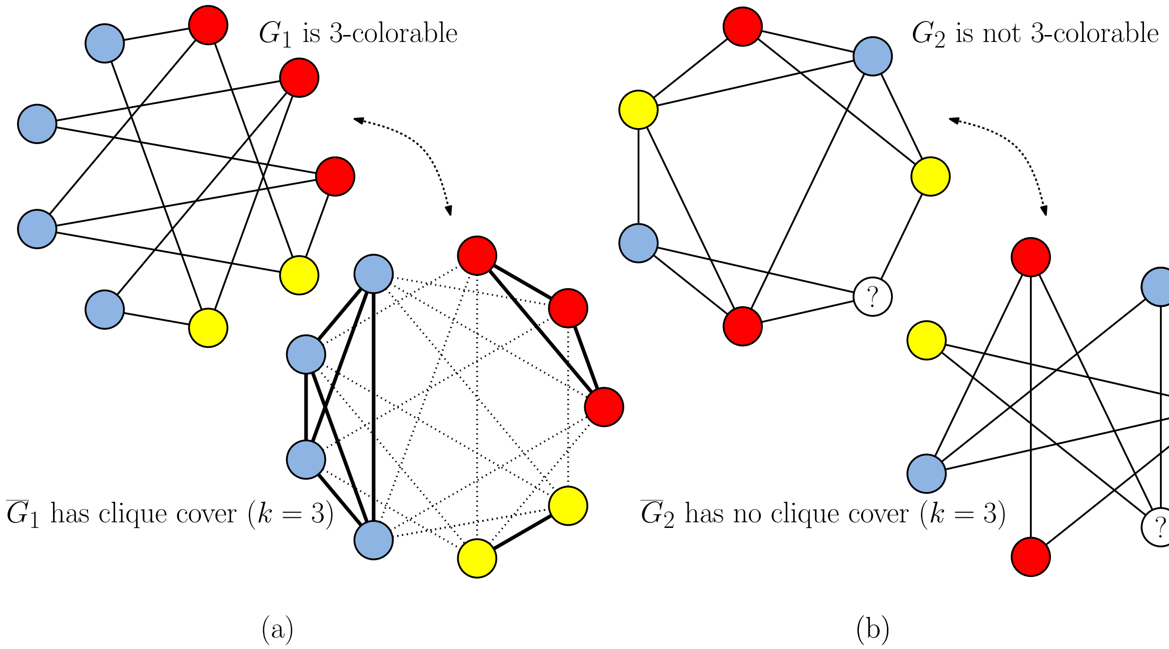


Fig. 67: Reducing 3Col to CCov.

Observe that both problems involve partitioning the vertices up into groups. There are two differences. First, in the 3-coloring problem, the number of groups is fixed at three. In the Clique Cover problem, the number is given as an input. Second, in the 3-coloring problem, in order for two vertices to be in the same group they should *not* have an edge between them. In the Clique Cover problem, for two vertices to be in the same group, they *must* have an edge between them. Our translation therefore, should convert edges into non-edges and vice versa.

This suggests the following idea for reducing the 3-coloring problem to the Clique Cover problem. Given a graph G, let $\overline{G}$ denote the *complement graph*, where two distinct nodes are connected by an edge if and only if



G_1 is 3-colorable
$\overline{G}_1$ has clique cover $(k = 3)$
(a)
G_2 is not 3-colorable
?
?
$\overline{G}_2$ has no clique cover $(k = 3)$
(b)

Fig. 68: Clique covers in the complement.

they are not adjacent in G. Let G be the graph for which we want to determine its 3-colorability. The translator outputs the pair $(\overline{G}, 3)$. We then feed the pair $(G', k) = (\overline{G}, 3)$ into a subroutine for clique cover (see Fig. 68).

The following formally establishes the correctness of this reduction.

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

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

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

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

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

In the previous example we showed that 3Col $\leq_P$ CCov, and in particular, $f(G) = (\overline{G}, 3)$. 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 $L_2 \notin P$.

Because the composition of two polynomials is a polynomial, we can chain reductions together.

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

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 $L' \leq_P L$, for all $L' \in$ NP. (Note that L does not need to be in NP.)

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

(1) $L \in$ NP (that is, it can be verified in polymomial 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) $L \in$ NP and

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

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

This gives us a way to prove that problems are NP-complete, once we know that *one* problem is NP-complete. Unfortunately, it appears to be almost impossible to prove that one problem is NP-complete, because the definition says that we have to be able to reduce *every* problem in NP to this problem. There are infinitely many such problems, so how can we ever hope to do this?

We will talk about this next time with Cook's theorem. Cook showed that there is one problem called SAT (short for *boolean satisfiability*) that is NP-complete. To prove a second problem is NP-complete, all we need to do is to show that our problem is in NP (and hence it is reducible to SAT), and then to show that we can reduce SAT (or generally some known NPC problem) to our problem. It follows that our problem is equivalent to SAT (with respect to solvability in polynomial time). This is illustrated in Fig. 69 below.

Lecture 21: Cook's Theorem, 3SAT, and Independent Set

Recap: Recall the following definitions, which were given in earlier lectures.

P: is the set of decisions problems solvable in polynomial time, or equivalently, the set of languages for which membership can be determined in polynomial time.

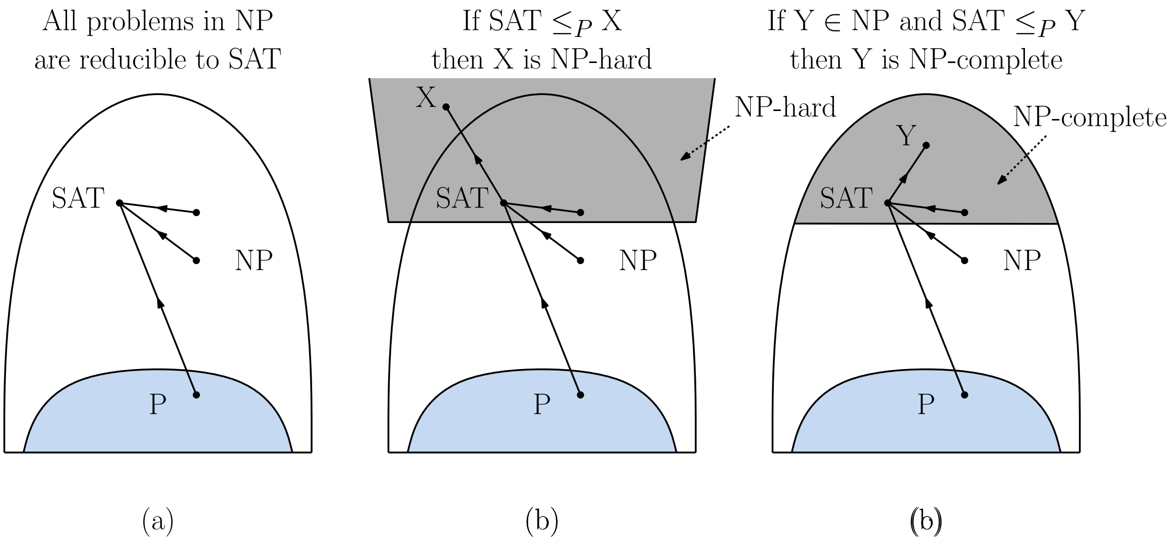


Fig. 69: Structure of NPC and reductions.

NP: is the set of languages that can be *verified* in polynomial time, or equivalently, that can be solved in polynomial time by a "guessing computer", whose guesses are guaranteed to produce an output of "yes" if at all possible.

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 way 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, $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ implies $L_1 \leq_P L_3$.

NP-Hard: L is NP-hard if for all $L' \in$ NP, $L' \leq_P L$. By transitivity of $\leq_P$, we can say that L is NP-hard if $L' \leq_P L$ for some known NP-hard problem L'.

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

It follows from these definitions that:

- If *any* NP-hard problems is solvable in polynomial time, then *every* NP-complete problem (in fact, every problem in NP) is also solvable in polynomial time.
- If *any* NP-complete problem cannot be solved in polynomial time, then *every* NP-complete problem (in fact, 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).

Cook's Theorem: To get the ball rolling, we need to prove that there is *at least one* NP-complete problem. Stephen Cook achieved this task. This first NP-complete problem involves boolean formulas. A boolean formula consists of variables (say x, y, and z) and the logical operations *not* (denoted $\overline{x}$), *and* (denoted $x \wedge y$), and *or* (denoted $x \vee y$).

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, consider the following formula:

$$F_1(x, y, z) \;=\; (x \wedge (y \vee \overline{z})) \wedge ((\overline{y} \wedge \overline{z}) \vee \overline{x}).$$

F_1 is satisfiable, by the assignment $x = 1$ and $y = z = 0$. On the other hand, the formula

$$F_2(x, y) \;=\; (\overline{z} \vee x) \wedge (z \vee y) \wedge (\overline{x} \wedge (\overline{y})$$

is not satisfiable since every possible assignment of 0-1 values to x, y, and z evaluates to 0.

The *boolean satisfiability problem* (SAT) is as follows: given a boolean formula F, is it possible to assign truth values (0/1, true/false) to F's variables, so that it evaluates to true?

Cook's Theorem: SAT is NP-complete.

A complete proof would take about a full lecture (not counting the week or so of background on nondeterminism and Turing machines). Here is an intuitive justification.

SAT is in NP: We nondeterministically guess truth values to the variables. (In the context of verification, the certificate consists of the assignment of values to the variables.) We then plug the values into the formula and evaluate it. Clearly, this can be done in polynomial time.

SAT is NP-Hard: To show that the 3SAT is NP-hard, Cook reasoned as follows. First, every NP-problem can be encoded as a program that runs in polynomial time on a given input, subject to a number of nondeterministic guesses. Since the program runs in polynomial time, we can express its execution on a specific input as straight-line program (that is, one containing no loops or function calls) that contains a polynomial number of lines of code in your favorite programming language. We then compile each line of code into machine code, and convert each machine code instruction into an equivalent boolean circuit. Finally, we can express each of these circuits equivalently as a boolean formula.

The nondeterministic choices can be implemented as boolean variables in this formula, whose values take on the possible values of 0 and 1. By definition of nondeterminism, the program answers "yes" if there is some choice of decisions that leads to an output of "yes". In our context, this means that there is some way of assigning 0-1 values to the variables so that our circuit produces an output of 1, that is, if the associated boolean formula is satisfied.

Therefore, if you *could* determine the satisfiability of this formula in polynomial time, you could determine whether the original nondeterministic program output "yes" in polynomial time.

Cook proved that satisfiability in NP-hard even for boolean formulas of a special form. To define this form, we start by defining a *literal* to be either a variable or its negation, that is, x or $\overline{x}$. A formula is said to be in *3-conjunctive normal form* (3-CNF) if it is the boolean-and of clauses where each clause is the boolean-or of exactly three literals. For example

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

is in 3-CNF form. The *3-CNF satisfiability problem* (3SAT) is the problem of determining whether a 3-CNF[12] boolean formula is satisfiable.

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 G shown in Fig. 70 has an independent set (shown with shaded nodes) of size 4, but there is no independent set of size 5. Therefore $(G, 4) \in$ IS but $(G, 5) \notin$ IS. 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.)

[12] Is there something special about the number 3? 1SAT is trivial to solve. 2SAT is trickier, but it can be solved in polynomial time (by reduction to DFS on an appropriate directed graph). kSAT is NP-complete for any $k \geq 3$.

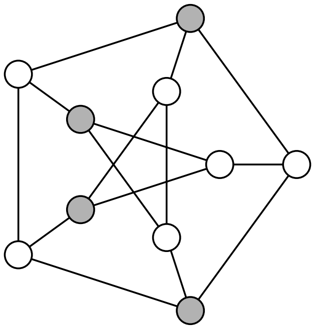


Fig. 70: A graph with an independent set of size $k = 4$.

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 polynomially reducible to IS, that is, 3SAT $\leq_P$ IS (see Fig. 71(a)). Let F be a boolean formula in 3-CNF form. We wish to find a polynomial time computable function f that maps F into a input for the IS problem, a graph G and integer k. That is, $f(F) = (G, k)$, such that F is satisfiable if and only if G has an independent set of size k. This will imply that if we could solve the independent set problem for G and k in polynomial time, then we would be able to solve 3SAT in polynomial time.

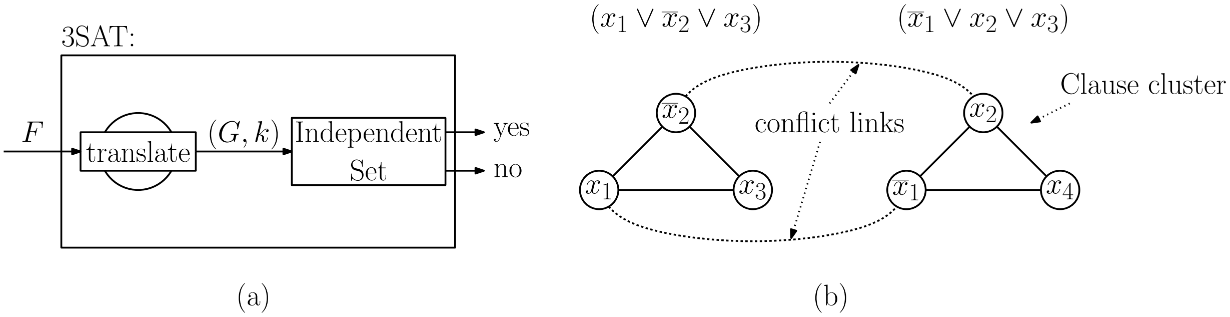


Fig. 71: (a) Reduction of 3-SAT to IS and (b) Clause clusters for the clauses $(x_1 \vee \overline{x}_2 \vee x_3)$ and $(\overline{x}_1 \vee x_2 \vee x_5)$.

Since this is the first nontrivial reduction we will do, let's take a moment to think about the process by which we develop a reduction. An important aspect to reductions is that we *do not* know whether the formula is satisfiable, we *don't know* which variables should be true or false, and we *don't have time* to determine this. (Remember: It is NP-complete!) The translation function f must operate without knowledge of the answer.

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'.

Idea: Let's create a vertex in G for each literal in each clause. A natural approach would be that if a literal is true, then it will correspond to putting the vertex in the independent set. Unfortunately, this will not quite work. Instead, we observe that *at least one* vertex of each clause must be true. We will take *exactly one* such literal from each clause to put into our independent set.

Requirements:

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

IS: V' must contain at least k vertices.

Idea: Let's group vertices into groups of three, one group per clause. As mentioned above, exactly one vertex of each group must be in any independent set. We'll set k equal to the number of clauses to enforce this condition.

Restrictions:

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

IS: If u and v are adjacent, then both u and v cannot be in the independent set.

Conclusion: We'll put an edge between two vertices if they correspond to complimentary literals.

In summary, our strategy will be to create clusters of three vertices, one for each literal in each clause. We call these *clause clusters*(see Fig. 71(b)). Since each clause must have at least one true literal, we will model this by forcing the IS algorithm to select one (and only one) vertex per clause cluster. Let's set k to the number of clauses. But, this does not force us to select one true literal from each clause, since we might take two from some clause cluster and zero from another. To prevent this, we will connect all the vertices within each clause cluster to each other. At most one can be taken to be in any independent set. Since we need to select k vertices, this will force us to pick exactly one from each cluster.

To enforce the restriction that only one of x and $\overline{x}$ can be set to 1, we create edges between all vertices associated with x to all vertices associated with $\overline{x}$. We call these *conflict links*. A formal description of the reduction is given below. The input is a boolean formula F in 3-CNF, and the output is a graph G and integer k.

3SAT to IS reduction

```
$k \leftarrow$ number of clauses in $F$
**for each** (clause $(x_1 \vee x_2 \vee x_3)$ in $F$)
    create a clause cluster consisting of three vertices labeled $x_1$, $x_2$, and $x_3$
    create edges $(x_1, x_2)$, $(x_2, x_3)$, $(x_3, x_1)$ between all pairs of vertices in the cluster
**for each** (vertex $x_i$)
    create edges between $x_i$ and all its complement vertices $\overline{x}_i$ (conflict links)
**return** $(G, k)$
```

Given any reasonable encoding of F, it is an easy programming exercise to create G in polynomial time. As an example, suppose that we are given the 3-CNF formula:

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

The reduction produces the graph shown in Fig. 72. The clauses clusters appear in clockwise order starting from the top.

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 $\overline{x}_3$ satifies the third. Observe that by selecting the corresponding vertices from the clusters, we obtain an independent set of size $k = 4$.

Correctness: We'll show that F is satisfiable if and only if G has an independent set of size k.

$(\Rightarrow)$: If F is satisfiable, then each of the k clauses of F must have at least one true literal. Select such a literal from each clause. 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'. Therefore V' is an independent set of size k.

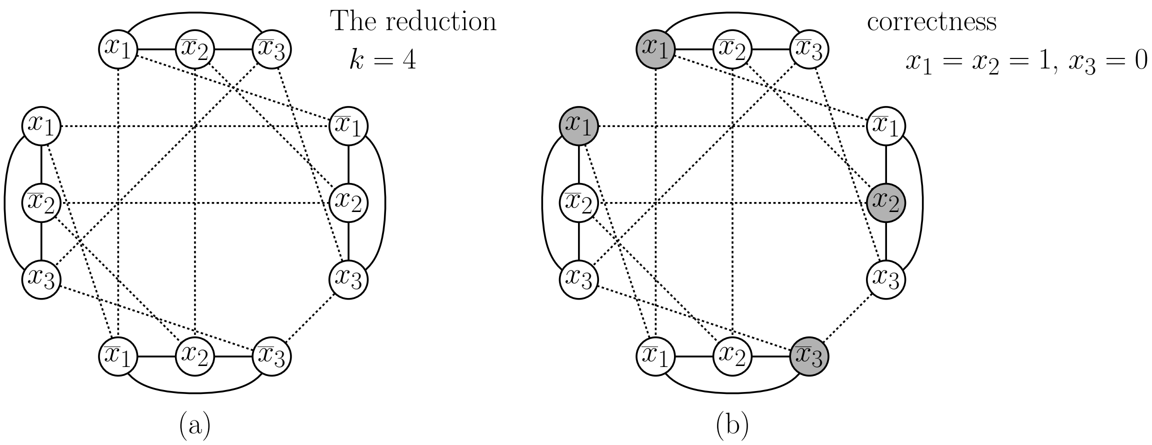


Fig. 72: 3SAT to IS reduction.

($\Leftarrow$) : Suppose that G has an independent set V' of size k. We cannot select two vertices from a clause cluster, and since there are k clusters, V' has exactly one vertex from each clause cluster. Note that if a vertex labeled x is in V' then the adjacent vertex $\overline{x}$ cannot also be in V'. Therefore, there exists an assignment in which every literal corresponding to a vertex appearing in V' is set to 1. Such an assignment satisfies one literal in each clause, and therefore the entire formula is satisfied.

Let us emphasize a few things about this reduction:

- Every NP-complete problem has three similar elements: (a) something is being selected, (b) something is forcing us to select a sufficient number of such things (requirements), and (c) something is limiting our ability to select these things (restrictions). A reduction's job is to determine how to map these similar elements to each other.
- Our reduction did not attempt to solve the 3SAT problem. (As a sign of this, observe that whatever we did for one literal, we did for all.) Remember this rule! If your reduction treats some entities different other, based on what you think the final answer may be, you are very likely making a mistake. Remember, these problems are NP-complete!

We now have the following picture of the world of NP-completeness. By Cook's Theorem, we know that every problem in NP is reducible to 3SAT. When we showed that IS $\in$ NP, it followed immediately that IS $\leq_P$ 3SAT. When we showed that 3SAT $\leq_P$ IS, we established their equivalence (up to polynomial time). By transitivity, it follows that all problems in NP are now reducible to IS (see Fig. 73).

Lecture 22: Clique, Vertex Cover, and Dominating Set

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) $L \in$ NP. (That is, given an input and an appropriate certificate, we can guess the solution and 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).

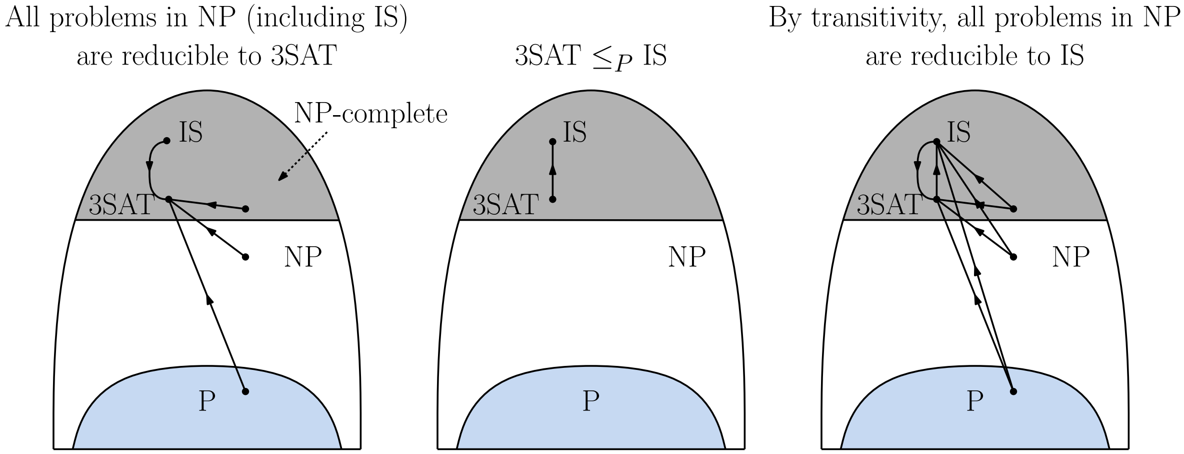

All problems in NP (including IS) are reducible to 3SAT
IS
NP-complete
3SAT
NP
P
3SAT $\leq_P$ IS
IS
3SAT
NP
P
By transitivity, all problems in NP are reducible to IS
IS
3SAT
NP
P

Fig. 73: Our updated picture of NP-completeness.

Some Easy Reductions: Next, let us consider some closely related NP-complete problems:

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?

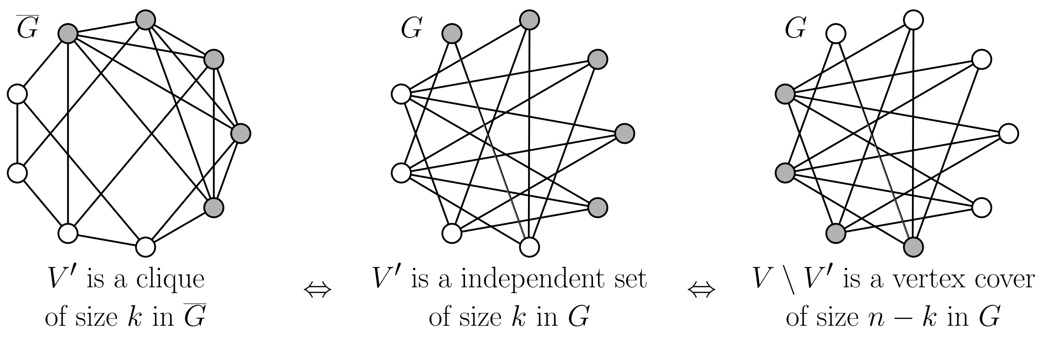
Don't confuse the clique (CLIQUE) problem with the clique-cover (CC) problem that we discussed in an earlier lecture. The clique problem seeks to find a single clique of size k, and the clique-cover problem seeks to partition the vertices into k groups, each of which is a clique.

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 two minutes of the nearest fire station. We create a graph in which two locations are adjacent if they are within two minutes of each other. A minimum sized dominating set will be a minimum set of locations such that every other location is reachable within two 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. 74). Given a graph G, recall that $\overline{G}$ is the *complement graph* where edges and non-edges are reverse. Also, recall that $A \setminus B$ denotes set resulting by removing the elements of B from A.

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, $\overline{G}$



V' is a clique of size k in $\overline{G}$ $\Leftrightarrow$ V' is a independent set of size k in G $\Leftrightarrow$ $V \setminus V'$ is a vertex cover of size $n-k$ in G

Fig. 74: Clique, Independent set, and Vertex Cover.

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

(iii) $V \setminus V'$ is a vertex cover of size $n-k$ for G, (where $n = |V|$)

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.

(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 $V \setminus V'$, implying that $V \setminus V'$ is a vertex cover for G.

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

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: We guess the k vertices that will form the clique. We can easily verify in polynomial time that all pairs of vertices in the set are adjacent (e.g., by inspection of $O(k^2)$ entries of the adjacency matrix).

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 $(\overline{G}, k)$. 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 at least one member of the dominating set, as opposed to each *edge* being *incident* to at least one member of the vertex cover. Obviously, if G is connected and has a vertex cover of size k, then it has a dominating set of size k (the same set of vertices), but the converse is not necessarily true.

However, the similarity suggests that if VC in NP-complete, then DS is likely to be NP-complete as well. The main result of this section is just this.

Theorem: DS is NP-complete.

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

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 n_I denote the number of isolated vertices. The number of vertices to request for the dominating set will be $k' = k + n_I$.

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 $k' = k + n_I$. Output (G', k'). This reduction illustrated in Fig. 75. Note that every step can be performed in polynomial time.

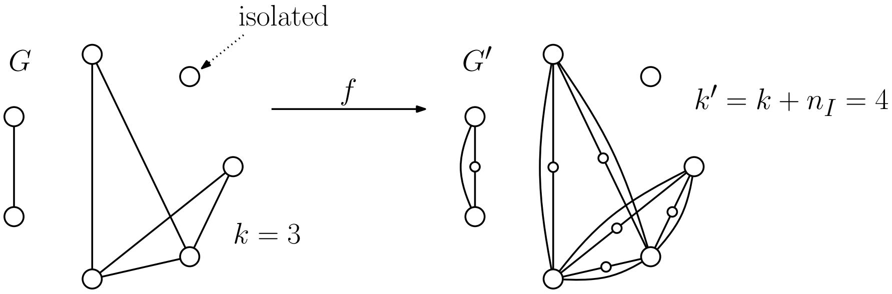


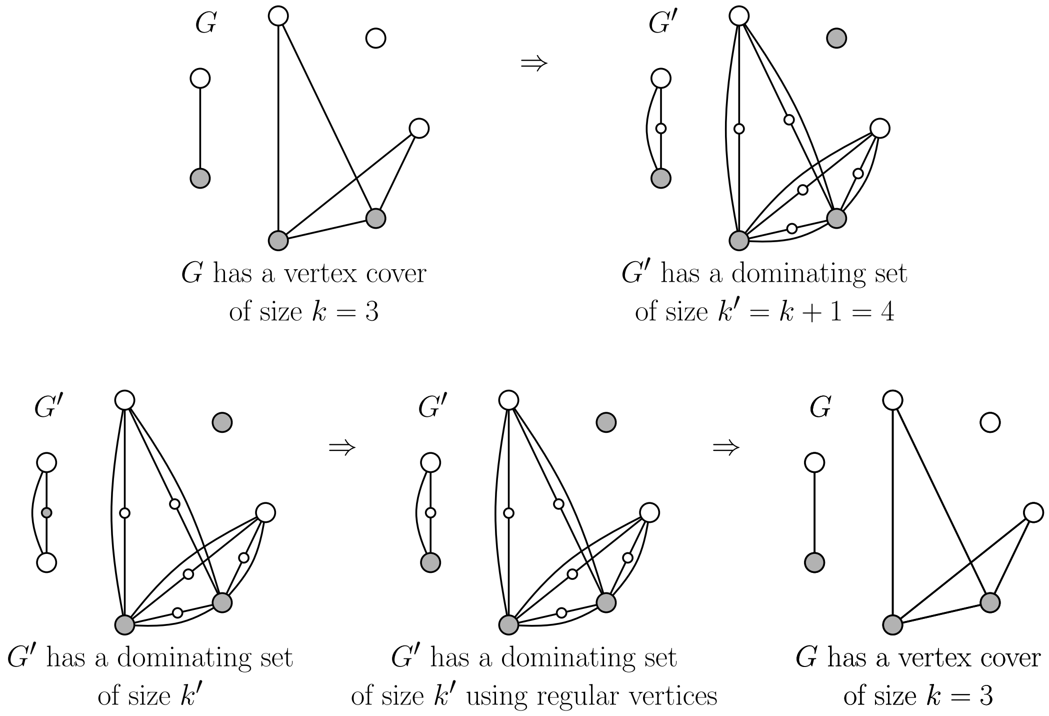
Fig. 75: Dominating set reduction with $k = 3$ and one isolated vertex.

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

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

Note that $|V' \cup V_I|$ might be of size less than $k + n_I$, 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'. 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. 76.



G has a vertex cover of size $k = 3$

G' has a dominating set of size $k' = k + 1 = 4$

G' has a dominating set of size k'

G' has a dominating set of size k' using regular vertices

G has a vertex cover of size $k = 3$

Fig. 76: Correctness of the VC to DS reduction (where $k = 3$ and $I = 1$).

Conversely, we claim that if G' has a dominating set V'' of size $k' = k + n_I$ then G has a vertex cover V' of size k. Note that all n_I isolated vertices of G' must be in the dominating set. First, let $V''' = V'' \setminus V_I$ 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 76.)

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

Lecture 23: Approximation Algorithms: Vertex Cover and TSP

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:

Brute-force search: This is usually only a viable option for small input sizes (e.g., $n \leq 20$).

Heuristics: This is a strategy for producing a valid solution, but may be there no guarantee on how close it is to optimal.

General Search Algorithms: There are a number of very powerful techniques for solving general combinatorial optimization problems. These go under various names such as *branch-and-bound*, *Metropolis-Hastings*, *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 guaranteed to be within some 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, vertex cover optimization problem is to find the vertex cover of minimum size, the clique optimization problem is to find the clique of maximum size. An approximation algorithm is one that returns a legitimate answer, but not necessarily one of the optimal size.

How do we measure how good an approximation algorithm is? We define the *performance ratio* 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 have $C(I)/C^*(I) \geq 1$. For a maximization problem we have $C^*(I)/C(I) \geq 1$. In either case, we want the ratio to be as small as possible. For any input size n, we say that the approximation algorithm achieves *performance 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 equal to 1 if and only if the approximate solution is the true optimum solution.

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.

- Some NP-complete are *inapproximable* in the sense no polynomial time algorithm achieves a ratio bound smaller than ∞ unless P $=$ NP.
- Some NP-complete can be approximated, but the ratio bound is a *function of* n. For example, the set cover problem (a generalization of the vertex cover problem), can be approximated to within a factor of $O(\log n)$.
- Some NP-complete problems can be approximated and the ratio bound is a *constant*.
- Some NP-complete problems can be approximated *arbitrarily well*. In particular, the user provides a parameter $\varepsilon > 0$ and the algorithm achieves a ratio bound of $(1 + \varepsilon)$. Of course, as ε approaches 0 the algorithm's running time gets worse. If such an algorithm runs in polynomial time for any fixed ε, it is called a *polynomial time approximation scheme* (PTAS).

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 (See Fig. 77).

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.

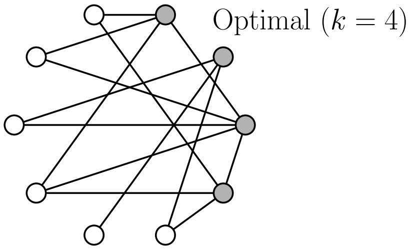


Fig. 77: Vertex cover (optimal solution).

Here is an very simple algorithm, that guarantees an approximation within a factor of 2 for the vertex cover problem. It is based on the following observation. Consider an arbitrary edge (u, v) in the graph. One of its two vertices *must* be in the cover, but we do not know which one. The idea of this heuristic is to simply put *both* vertices into the vertex cover. (You cannot get much stupider than this!)

We call this the *2-for-1 heuristic*. More formally, the algorithm runs in a series of stages. Initially the cover is empty. During each stage we select an arbitrary edge (u, v) from the graph and add both u and v to the current cover. We then remove *all* the edges that are incident to either u or v (since these edges are now all covered). We repeat until G has no more edges. (The algorithm shown in the following code fragment, and it is illustrated in Fig. 78.)

2-for-1 Approximation for VC

```
    two-for-one-VC(G=(V,E)) {
        C = empty
        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: two-for-one-VC(G) achieves a performance ratio of 2.

Proof: returns a vertex cover for G that is at most twice the size of the optimal vertex cover. Consider the set C output by two-for-one-VC(G). Let C^* be the optimum vertex cover. Let A be the set of edges selected by the line marked with “$(*)$” in the code fragment. Because we add both endpoints of each edge of A to C, we have $|C| = 2|A|$. However, the optimum vertex cover C^* must contain at least one of these two vertices. Therefore, we have $|C^*| \geq |A|$. Therefore

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

as desired.

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

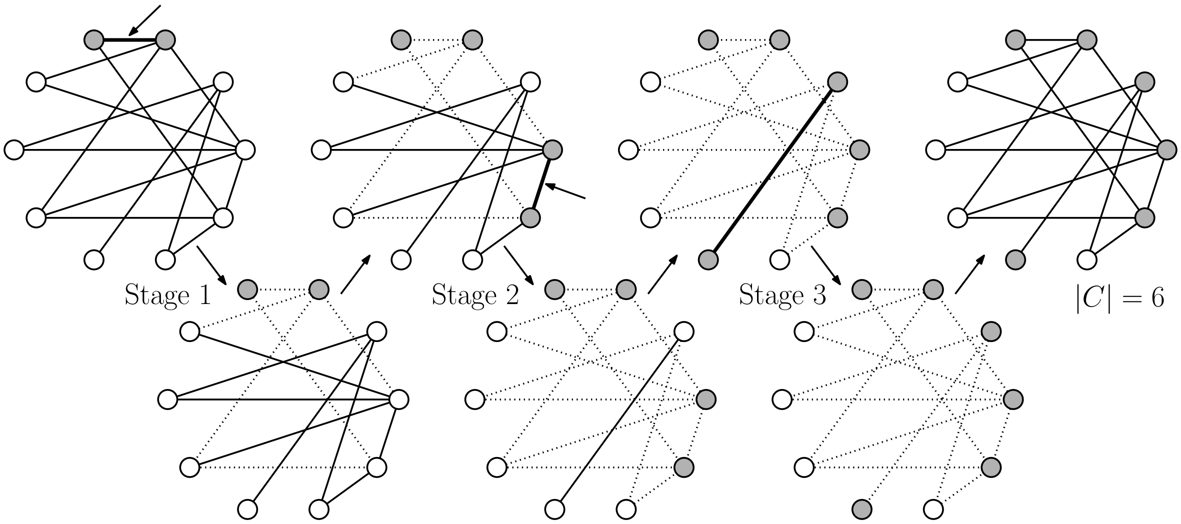


Fig. 78: The 2-for-1 heuristic for vertex cover.

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 given in the code fragment below and is illustrated in Fig. 79.)

Greedy Approximation for VC

```
greedy-VC(G=(V,E)) {
    C = empty
    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 Fig. 79, the greedy heuristic actually succeeds in finding the optimum vertex cover. Given that it is more clever than the 2-for-1 heuristic, we might be inclined to conjecture that the greedy heuristic always does at least as well as the 2-for-1 heuristic. It is surprising, however, that answer is “no”. Moreover, it can be shown that the greedy heuristic does *not* even achieve a constant performance bound. Indeed there exist graphs having n vertices such that the greedy heuristic achieves a performance ration of $\Theta(\log n)$. It should be mentioned, however, that experimental studies show that greedy actually works quite well in practice, and for “typical” graphs, it will perform better than the 2-for-1 heuristic.

Reductions and Approximations: Now that we have a factor-2 approximation for one NP-complete problem (vertex cover), you might be tempted to believe that we now 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

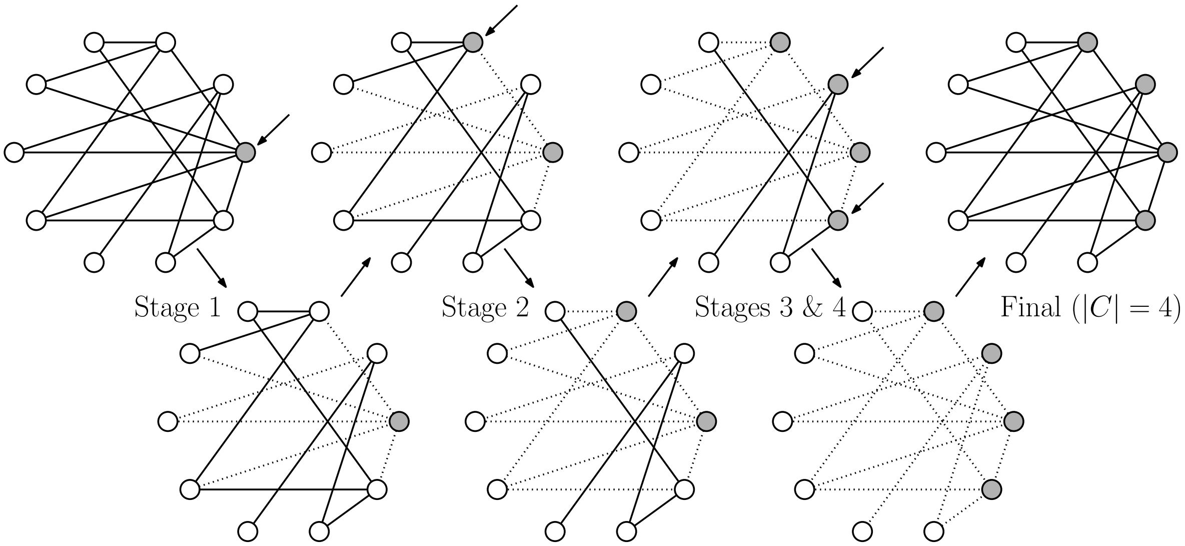


Fig. 79: The greedy heuristic for vertex cover.

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 $n - 2k$. Therefore, through the use of the reduction we achieve a performance ration of

$$\rho(n, k) \;=\; \frac{n - k}{n - 2k}.$$

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.

Traveling Salesman with Triangle Inequality: 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 $w(u, v)$ denote the weight on edge (u, v). Given a set of edges A forming a tour we define $W(A)$ to be the sum of edge weights in A.

For many of the applications of TSP, the edge weights satisfy a property called the *triangle inequality*. Intuitively, this says that the direct path from u to x, is never longer than an indirect path. More formally, for all $u, v, x \in V$

$$w(u, v) \;\leq\; w(u, x) + w(x, v).$$

There are many examples of graphs that satisfy the triangle inequality. For example, given any weighted graph, if we define $w(u, v)$ to be the shortest path length between u and v, 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 $w(u, v)$ 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 just 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, Kruskal'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 (see Fig. 80).

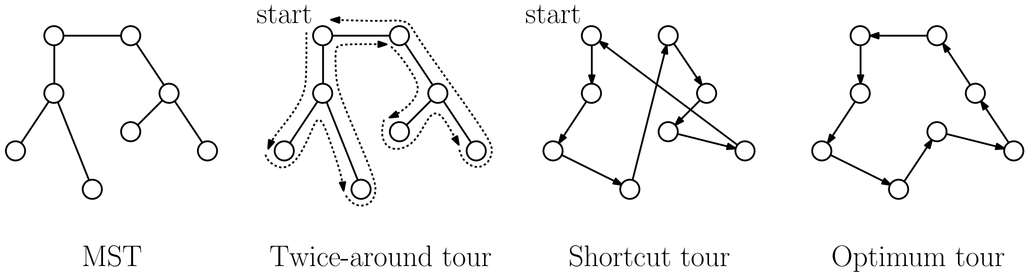


Fig. 80: TSP Approximation.

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.

TSP Approximation

```
approx-TSP(G=(V,E)) {
    T = minimum spanning tree for G
    r = any vertex
    H = list of vertices visited by a preorder walk of T starting at r
    return L
}
```

Claim: Approx-TSP achieves a performance ratio 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

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

Now observe that the twice around tour of T has cost $2 \cdot W(T)$, 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

$$W(H) \leq 2 \cdot W(T).$$

Combining these we have

$$W(H) \leq 2 \cdot W(T) \leq 2 \cdot W(H^*) \qquad \Rightarrow \qquad \frac{W(H)}{W(H^*)} \leq 2,$$

as desired.

Supplemental Lecture 1: Max Dominance

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

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

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

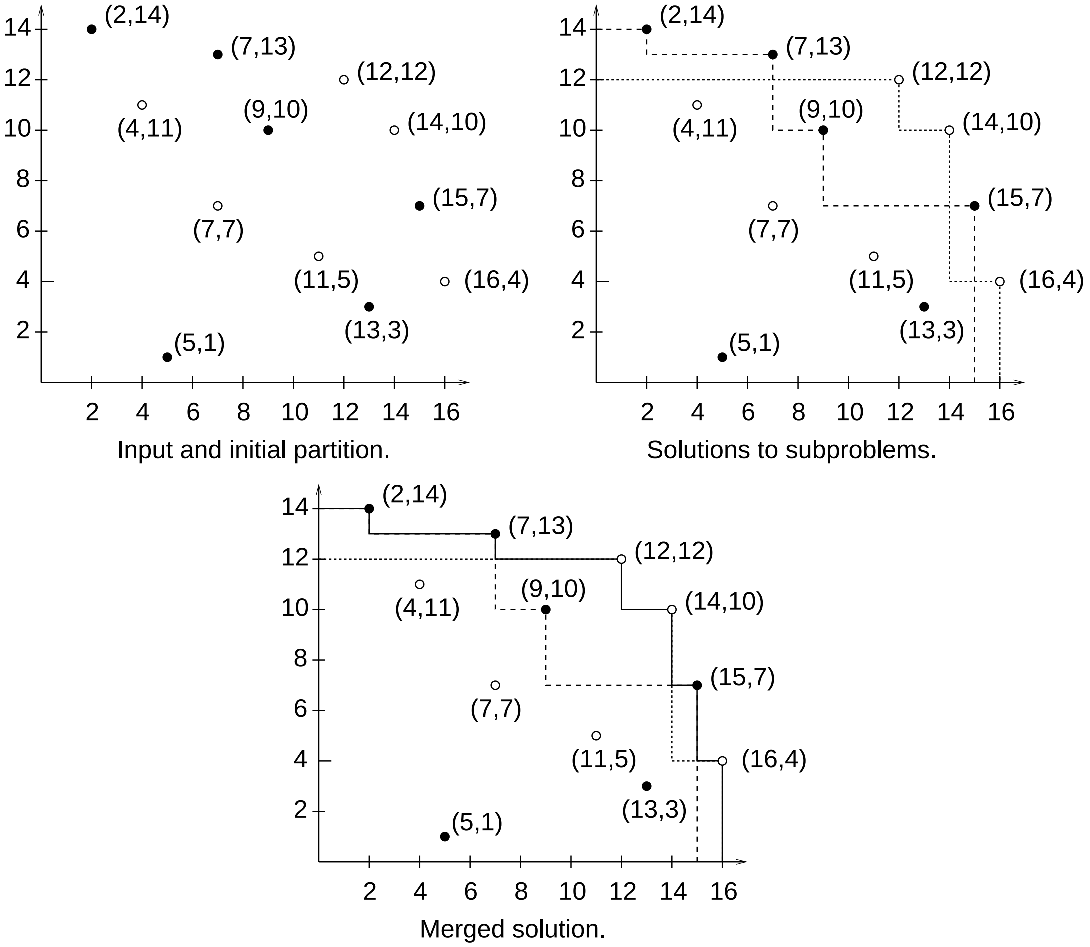
```
MaxDom4(P, n) {
    if (n == 1) return {P[1]};          // one point is trivially maximal
    m = n/2;                            // midpoint of list
    M1 = MaxDom4(P[1..m], m);           // solve for first half
    M2 = MaxDom4(P[m+1..n], n-m);       // solve for second half
    return MaxMerge(M1, M2);            // merge the results
}
```

The general process is illustrated below.

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

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

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



Input and initial partition.

Solutions to subproblems.

Merged solution.

Fig. 81: Divide and conquer approach.

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.

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

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

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

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

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

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

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

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

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

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

The Fibonacci numbers arise in data structure design. If you study AVL (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 $L(i) = F_{i+1}$.

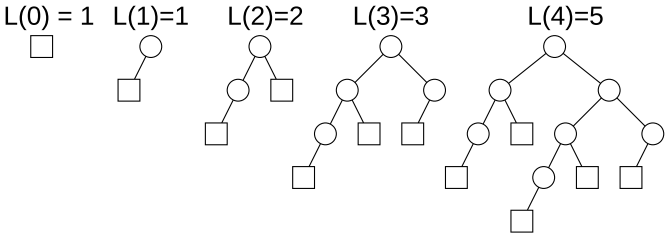


Fig. 82: Minimum-sized AVL trees.

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

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

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

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

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

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

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

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

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

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

By the quadratic formula we have

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

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

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

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

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

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

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

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

Supplemental Lecture 2: Recurrences and Generating Functions

Generating Functions: The method of constructive induction provided a way to get a bound on F_n, but we did not get an exact answer, and we had to generate a good guess before we were even able to start.

Let us consider an approach to determine an exact representation of F_n, which requires no guesswork. This method is based on a very elegant concept, called a *generating function*. Consider any infinite sequence:

$$a_0, a_1, a_2, a_3, \ldots$$

If we would like to "encode" this sequence succinctly, we could define a polynomial function such that these are the coefficients of the function:

$$G(z) = a_0 + a_1 z + a_2 z^2 + a_3 z^3 + \ldots$$

This is called the *generating function* of the sequence. What is z? It is just a symbolic variable. We will (almost) never assign it a specific value. Thus, every infinite sequence of numbers has a corresponding generating function, and vice versa. What is the advantage of this representation? It turns out that we can perform arithmetic transformations on these functions (e.g., adding them, multiplying them, differentiating them) and this has a corresponding effect on the underlying transformations. It turns out that some nicely-structured sequences (like the Fibonacci numbers, and many sequences arising from linear recurrences) have generating functions that are easy to write down and manipulate.

Let's consider the generating function for the Fibonacci numbers:

$$\begin{aligned} G(z) &= F_0 + F_1 z + F_2 z^2 + F_3 z^3 + \ldots \\ &= z + z^2 + 2z^3 + 3z^4 + 5z^5 + \ldots \end{aligned}$$

The trick in dealing with generating functions is to figure out how various manipulations of the generating function to generate algebraically equivalent forms. For example, notice that if we multiply the generating function by a factor of z, this has the effect of shifting the sequence to the right:

$$\begin{array}{rcccccccccccc} G(z) & = & F_0 & + & F_1 z & + & F_2 z^2 & + & F_3 z^3 & + & F_4 z^4 & + & \ldots \\ zG(z) & = & & & F_0 z & + & F_1 z^2 & + & F_2 z^3 & + & F_3 z^4 & + & \ldots \\ z^2 G(z) & = & & & & & F_0 z^2 & + & F_1 z^3 & + & F_2 z^4 & + & \ldots \end{array}$$

Now, let's try the following manipulation. Compute $G(z) - zG(z) - z^2G(z)$, and see what we get

$$\begin{aligned} (1 - z - z^2)G(z) &= F_0 + (F_1 - F_0)z + (F_2 - F_1 - F_0)z^2 + (F_3 - F_2 - F_1)z^3 \\ &\qquad + \ldots + (F_i - F_{i-1} - F_{i-2})z^i + \ldots \\ &= z. \end{aligned}$$

Observe that every term except the second is equal to zero by the definition of F_i. (The particular manipulation we picked was chosen to cause this cancellation to occur.) From this we may conclude that

$$G(z) = \frac{z}{1 - z - z^2}.$$

So, now we have an alternative representation for the Fibonacci numbers, as the coefficients of this function if expanded as a power series. So what good is this? The main goal is to get at the coefficients of its power series expansion. There are certain common tricks that people use to manipulate generating functions.

The first is to observe that there are some functions for which it is very easy to get an power series expansion. For example, the following is a simple consequence of the formula for the geometric series. If $0 < c < 1$ then

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1 - c}.$$

Setting $z = c$, we have

$$\frac{1}{1 - z} = 1 + z + z^2 + z^3 + \ldots$$

(In other words, $1/(1 - z)$ is the generating function for the sequence $(1, 1, 1, \ldots)$. In general, given an constant a we have

$$\frac{1}{1 - az} = 1 + az + a^2z^2 + a^3z^3 + \ldots$$

is the generating function for $(1, a, a^2, a^3, \ldots)$. It would be great if we could modify our generating function to be in the form of $1/(1 - az)$ for some constant a, since then we could then extract the coefficients of the power series easily.

In order to do this, we would like to rewrite the generating function in the following form:

$$G(z) \;=\; \frac{z}{1-z-z^2} \;=\; \frac{A}{1-az} + \frac{B}{1-bz},$$

for some A, B, a, b. We will skip the steps in doing this, but it is not hard to verify the roots of $(1-az)(1-bz)$ (which are $1/a$ and $1/b$) must be equal to the roots of $1-z-z^2$. We can then solve for a and b by taking the reciprocals of the roots of this quadratic. Then by some simple algebra we can plug these values in and solve for A and B yielding:

$$G(z) \;=\; \frac{z}{1-z-z^2} \;=\; \left(\frac{1/\sqrt{5}}{1-\phi z} + \frac{-1/\sqrt{5}}{1-\hat{\phi}}\right) \;=\; \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}}\right),$$

where $\phi = (1+\sqrt{5})/2$ and $\hat{\phi} = (1-\sqrt{5})/2$. (In particular, to determine A, multiply the equation by $1-\phi z$, and then consider what happens when $z = 1/\phi$. A similar trick can be applied to get B. In general, this is called the method of *partial fractions*.)

Now we are in good shape, because we can extract the coefficients for these two fractions from the above function. From this we have the following:

$$\begin{array}{rclrcrcrcl} G(z) & = & \frac{1}{\sqrt{5}}\,(& 1 & + & \phi z & + & \phi^2 z^2 & + & \ldots \\ & & & -1 & + & -\hat{\phi} z & + & -\hat{\phi}^2 z^2 & + & \ldots\,) \end{array}$$

Combining terms we have

$$G(z) = \frac{1}{\sqrt{5}} \sum_{i=0}^{\infty} (\phi^i - \hat{\phi}^i) z^i.$$

We can now read off the coefficients easily. In particular it follows that

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \widehat{\phi^n}).$$

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 ϕ 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 3: Medians and Selection

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 kth 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 element 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.

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 kth 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.

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.

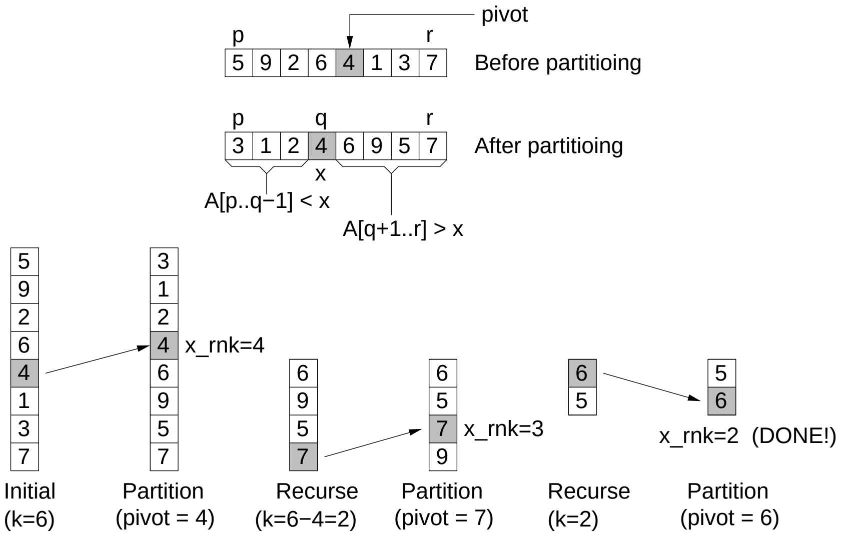


Fig. 83: Selection Algorithm.

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
    }
}
```

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 is 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 kth 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.

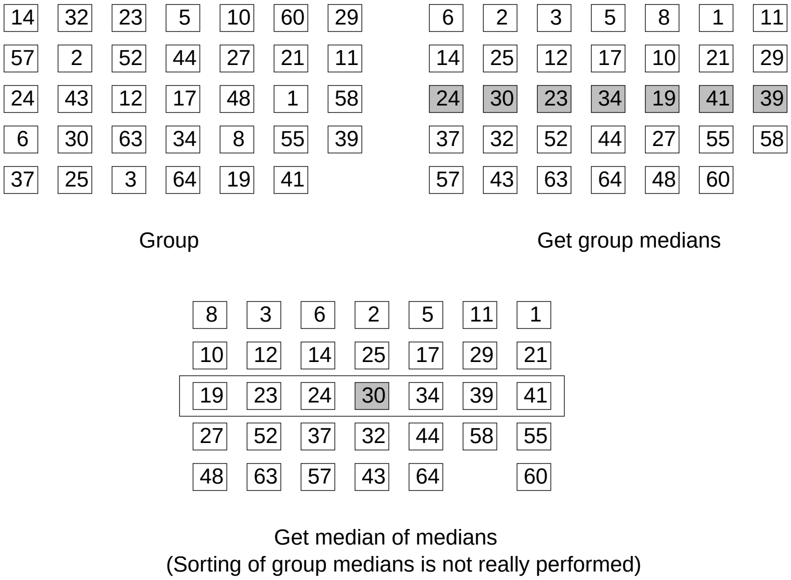


Fig. 84: Choosing the Pivot. 30 is the final pivot.

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

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

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

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 *Var*$[X]$ are

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

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

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

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 5: Long Integer Multiplication

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.

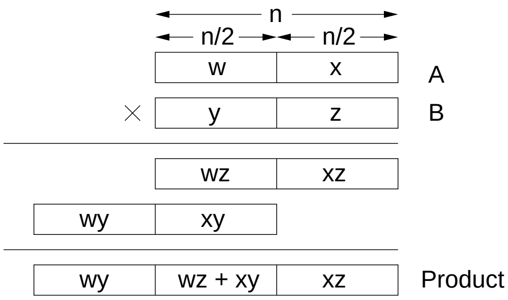


Fig. 85: Long integer multiplication.

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

$$\begin{aligned} w &= A[n-1..m] \qquad & x &= A[m-1..0] \quad \text{and} \\ y &= B[n-1..m] \qquad & z &= B[m-1..0]. \end{aligned}$$

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

$$\textit{mult}(A, B) = \textit{mult}(w, y)10^{2m} + (\textit{mult}(w, z) + \textit{mult}(x, y))10^m + \textit{mult}(x, z).$$

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 a 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).

$$\begin{aligned} C &= \textit{mult}(w, y) \\ D &= \textit{mult}(x, z) \\ E &= \textit{mult}((w + x), (y + z)) - C - D \;=\; (wy + wz + xy + xz) - wy - xz \;=\; (wz + xy). \end{aligned}$$

Finally we have

$$\textit{mult}(A, B) = C \cdot 10^{2m} + E \cdot 10^{m} + D.$$

Altogether we perform 3 multiplications, 4 additions, and 2 subtractions all of numbers with $n/2$ digitis. 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 cryptogrphy applications, encryption keys of this length and longer are quite reasonable.

Supplemental Lecture 6: Dynamic Programming: 0–1 Knapsack Problem

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

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

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

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

subject to

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

Let us assume that the v_i's, w_i's and W are all positive integers. It turns out that this problem is NP-complete, and so we cannot really hope to find an efficient solution. However if we make the same sort of assumption that we made in counting sort, we can come up with an efficient solution.

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

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

To compute the entries of the array V we will imply an inductive approach. As a basis, observe that $V[0,j] = 0$ for $0 \leq j \leq W$ since if we have no items then we have no value. We consider two cases:

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

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

Since these are the only two possibilities, we can see that we have the following rule for constructing the array V. The ranges on i and j are $i \in [0..n]$ and $j \in [0..W]$.

$$\begin{aligned} V[0,j] &= 0 \\ V[i,j] &= \begin{cases} V[i-1,j] & \text{if } w_i > j \\ \max(V[i-1,j], v_i + V[i-1, j-w_i]) & \text{if } w_i \leq j \end{cases} \end{aligned}$$

The first line states that if there are no objects, then there is no value, irrespective of j. The second line implements the rule above.

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

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. They key is to record for each entry $V[i,j]$ in the matrix whether we got this entry by taking the ith item or leaving it. With this information, it is possible to reconstruct the optimum knapsack contents.

Supplemental Lecture 7: Prim's and Baruvka's Algorithms for MSTs

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!)

0-1 Knapsack Problem

```
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	0
1	10	5	0	0	0	0	0	10	10	10	10	10	10
2	40	4	0	0	0	0	40	40	40	40	40	50	50
3	30	6	0	0	0	0	40	40	40	40	40	50	70
4	50	3	0	0	0	50	50	50	50	90	90	90	90

Final result is $V[4, 10] = 90$ (for taking items 2 and 4).

Fig. 86: 0–1 Knapsack Example.

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.

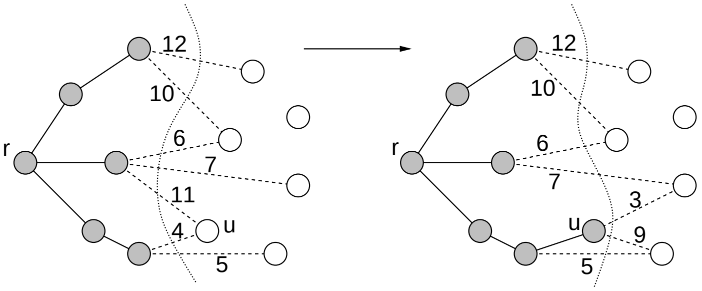


Fig. 87: Prim's Algorithm.

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$(u, \textbf{\textit{key}})$**:** Insert u with the key value *key* in Q.

extractMin$()$**:** Extract the item with the minimum key value in Q.

decreaseKey$(u, \textbf{\textit{new\_key}})$**:** 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.

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

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

```
Prim(G,w,r) {
    for each (u in V) {                    // initialization
        key[u] = +infinity;
        color[u] = white;
    }
    key[r] = 0;                            // start at root
    pred[r] = nil;
    Q = new PriQueue(V);                   // put vertices in Q
    while (Q.nonEmpty()) {                 // until all vertices in MST
        u = Q.extractMin();                // vertex with lightest edge
        for each (v in Adj[u]) {
            if ((color[v] == white) && (w(u,v) < key[v])) {
                key[v] = w(u,v);           // new lighter edge out of v
                Q.decreaseKey(v, key[v]);
                pred[v] = u;
            }
        }
        color[u] = black;
    }
    [The pred pointers define the MST as an inverted tree rooted at r]
}
```

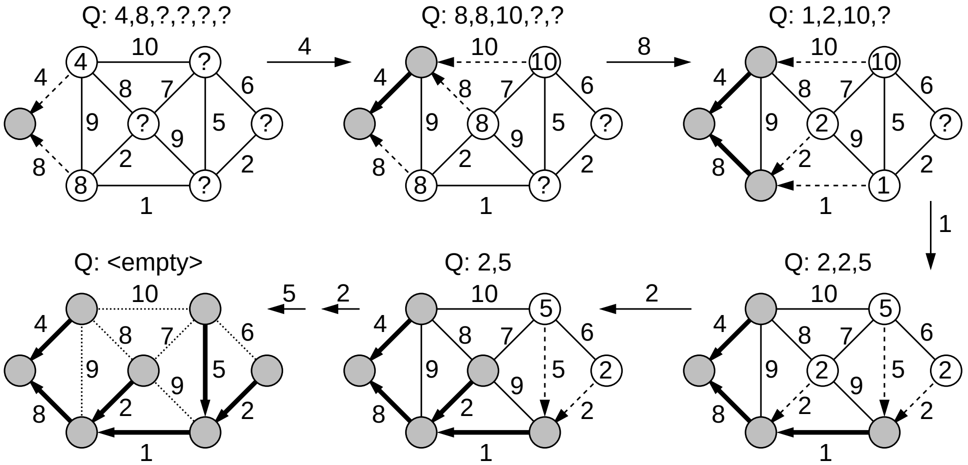


Fig. 88: Prim's Algorithm.

$O(\log V)$ time decreasing the key of the neighboring vertex. Thus the time is $O(\log V + \textit{deg}(u) \log V)$ time. The other steps of the update are constant time. So the overall running time is

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

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.

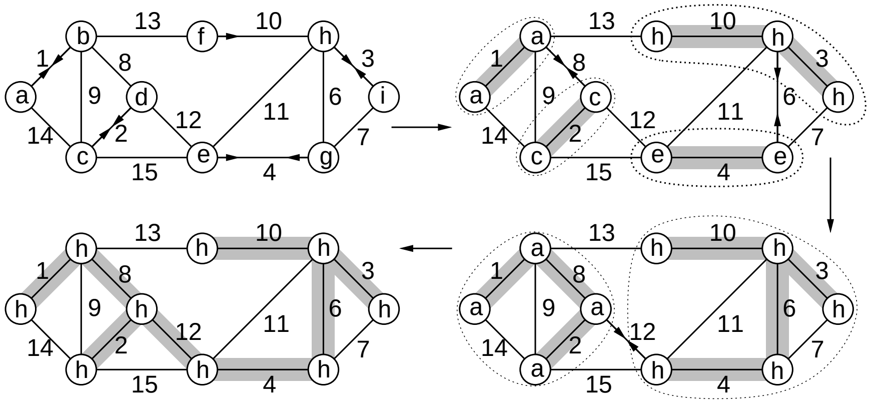


Fig. 89: Baruvka's Algorithm.

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

Supplemental Lecture 8: All-Pairs Shortest Paths and the Floyd-Warshall Algorithm

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

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

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

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

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

Setting $w_{ij} = \infty$ if there is no edge, intuitively means that there is no direct link between these two nodes, and hence the direct cost is infinite. The reason for setting $w_{ii} = 0$ is that 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 $p = \langle v_1, v_2, \ldots, v_\ell \rangle$ we say that the vertices $v_2, v_3, \ldots, v_{\ell-1}$ are the *intermediate vertices* of this path. Note that a path consisting of a single edge has no intermediate vertices.

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

In other words, we consider a path from i to j which either consists of the single edge (i,j), or it visits some intermediate vertices along the way, but these intermediate can only be chosen from among $\{1, 2, \ldots, k\}$. The path is free to visit any subset of these vertices, and to do so in any order. For example, in the digraph shown in the Fig. 90(a), notice how the value of $d_{5,6}^{(k)}$ changes as k varies.

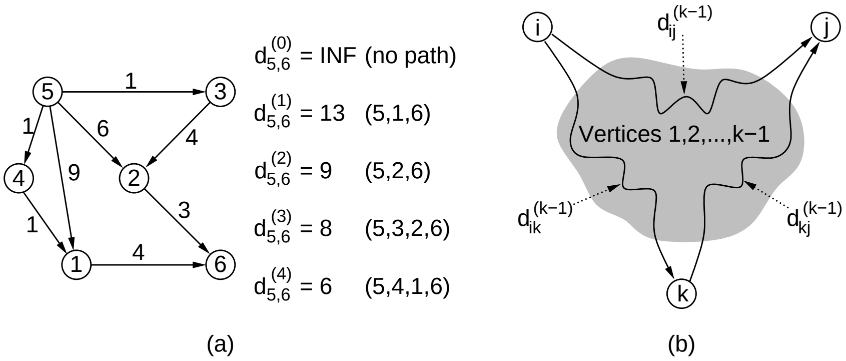


Fig. 90: Limiting intermediate vertices. For example $d_{5,6}^{(3)}$ can go through any combination of the intermediate vertices $\{1, 2, 3\}$, of which $\langle 5, 3, 2, 6 \rangle$ has the lowest cost of 8.

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

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

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

This suggests the following recursive rule (the DP formulation) for computing $d^{(k)}$, which is illustrated in Fig. 90(b).

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

The final answer is $d_{ij}^{(n)}$ because this allows all possible vertices as intermediate vertices. We could write a recursive program to compute $d_{ij}^{(k)}$, but this will be prohibitively slow because the same value may be reevaluated many times. Instead, we compute it by storing the values in a table, and looking the values up as we need them. Here is the complete algorithm. We have also included mid-vertex pointers, *mid*$[i, j]$ for extracting the final shortest paths. We will 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. 91.

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

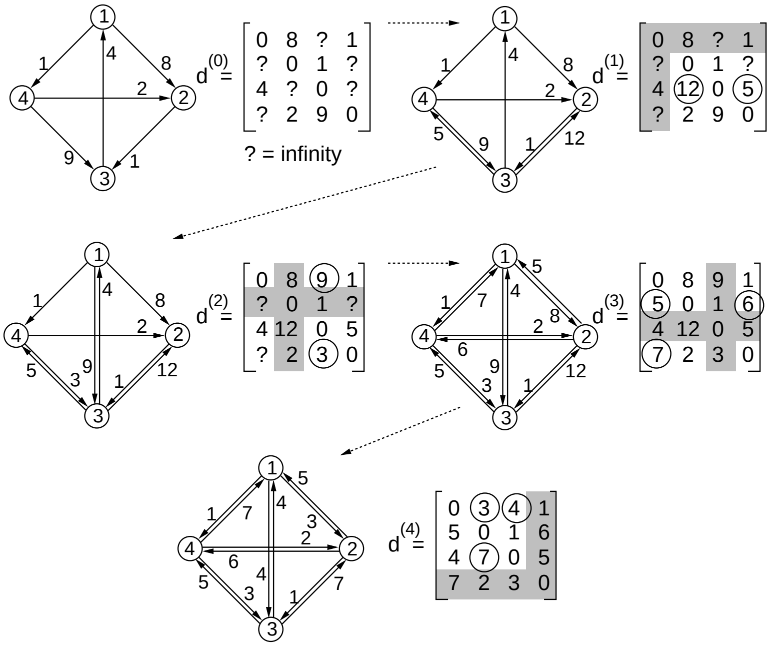


Fig. 91: Floyd-Warshall Example. Newly updates entries are circled.

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

Printing the Shortest Path

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

Supplemental Lecture 9: Dynamic Programming: Minimum Weight Triangulation

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.

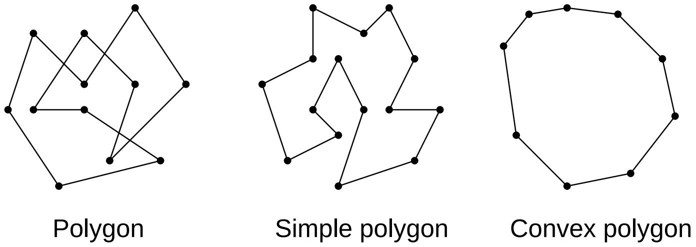


Fig. 92: Polygons.

Given a convex polygon, we assume that its vertices are labeled in counterclockwise order $P = \langle v_1, \ldots, v_n \rangle$. 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 $\overline{v_iv_j}$ is a *chord*. (If the polygon is simple but not convex, we include the additional requirement that the interior of the segment must lie entirely in the interior of P.) Any chord subdivides the polygon into two polygons: $\langle v_i, v_{i+1}, \ldots, v_j \rangle$, and $\langle v_j, v_{j+1}, \ldots, v_i \rangle$. A *triangulation* of a convex polygon 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 of some chord in T.) It is easy to see that such a set of chords subdivides the interior of the polygon into a collection of triangles with pairwise disjoint interiors (and hence the name *triangulation*). It is not hard to prove (by induction) that every triangulation of an n-sided polygon consists of $n - 3$ chords and $n - 2$ triangles. Triangulations are of interest for a number of reasons. Many geometric algorithm operate by first decomposing a complex polygonal shape into triangles.

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. 93.)

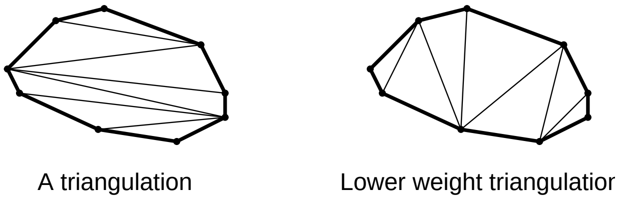


Fig. 93: Triangulations of convex polygons, and the minimum weight triangulation.

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

$$w(v_i, v_j, v_k) = |v_iv_j| + |v_jv_k| + |v_kv_i|,$$

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

Dynamic Programming Solution: Let us consider an $(n + 1)$-sided polygon $P = \langle v_0, v_1, \ldots, v_n \rangle$. 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 $\overline{v_i v_j}$, that is, the polygon with the counterclockwise vertex sequence $\langle v_i, v_{i+1}, \ldots, v_j \rangle$. 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 $\langle v_i, v_{i+1}, \ldots, v_j \rangle$, 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 $\langle v_i, v_{i+1}, \ldots v_k \rangle$ and $\langle v_k, v_{k+1}, \ldots v_j \rangle$ 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. 94

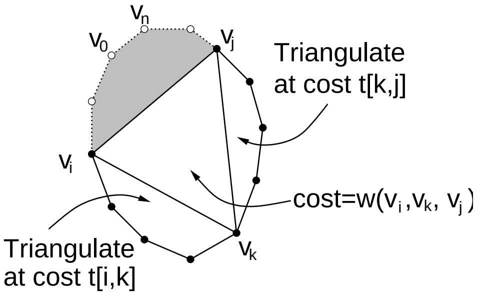


Fig. 94: Triangulations and tree structure.

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 $P = \langle v_0, v_1, \ldots, v_n \rangle$, and fix one side of the polygon (say $\overline{v_0 v_n}$). Now consider a rooted binary tree whose root node is the triangle containing side $\overline{v_0 v_n}$, whose internal nodes are the nodes of the dual tree, and whose leaves correspond to the remaining sides of the tree. Observe that partitioning the polygon into triangles is equivalent to a binary tree with n leaves, and vice versa. This is illustrated in Fig. 95. Note that every triangle is associated with an internal node of the tree and every edge of the original polygon, except for the distinguished starting side $\overline{v_0 v_n}$, is associated with a leaf node of the tree.

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

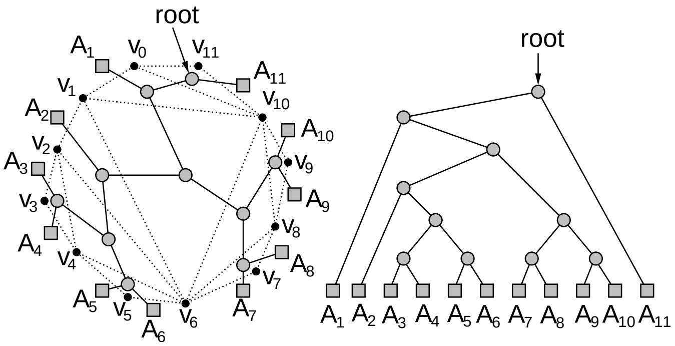


Fig. 95: Triangulations and tree structure.

Supplemental Lecture 10: Subset Sum

Subset Sum: The Subset Sum problem (SS) is the following. Given a finite set S of positive integers $S = \{w_1, w_2, \ldots, w_n\}$ 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.

$$S = \{3, 6, 9, 12, 15, 23, 32\} \qquad \text{and} \qquad t = 33.$$

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.[13]

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 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.

[13] 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 $\{w_1, w_2, \ldots, w_i\}$ that sums to t', and 0 otherwise. The ith row of this table can be computed in $O(t)$ time, given the contents of the $(i-1)$-st row.

(i) SS $\in$ NP.

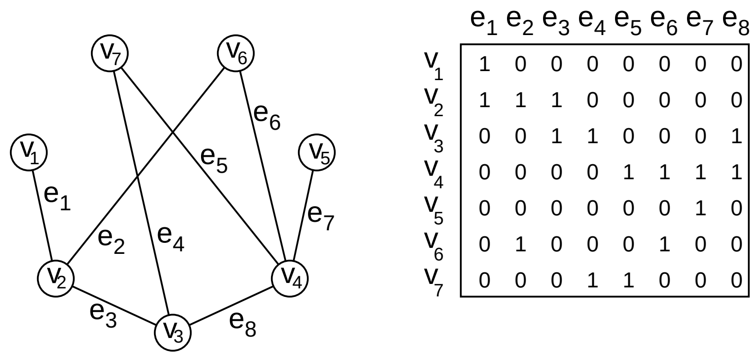
(ii) Some known NP-complete problem is reducible to SS. In particular, we will show that Vertex Cover (VC) is reducible to SS, that is, VC $\leq_P$ SS.

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 $k = 3$.



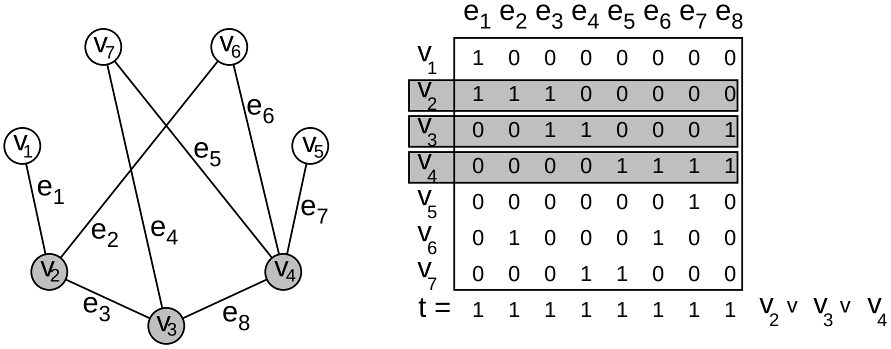
	e1	e2	e3	e4	e5	e6	e7	e8
v1	1	0	0	0	0	0	0	0
v2	1	1	1	0	0	0	0	0
v3	0	0	1	1	0	0	0	1
v4	0	0	0	0	1	1	1	1
v5	0	0	0	0	0	0	1	0
v6	0	1	0	0	0	1	0	0
v7	0	0	0	1	1	0	0	0

Fig. 96: 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, $1111\ldots1$. Conversely, if the logical-or is a bit vector of 1's, then each edge has been covered by some vertex, implying that the vertices form a vertex cover. (Later we will consider how to encode the fact that there only allowed k vertices in the cover.)

Since bit vectors can be thought of as just a way of representing numbers in binary, this is starting to feel more like the subset sum problem. The target would be the number whose bit vector is all 1's. 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



	e1	e2	e3	e4	e5	e6	e7	e8	
v_1	1	0	0	0	0	0	0	0	
v_2	1	1	1	0	0	0	0	0	
v_3	0	0	1	1	0	0	0	1	
v_4	0	0	0	0	1	1	1	1	
v_5	0	0	0	0	0	0	1	0	
v_6	0	1	0	0	0	1	0	0	
v_7	0	0	0	1	1	0	0	0	
t =	1	1	1	1	1	1	1	1	$v_2 \vee v_3 \vee v_4$

Fig. 97: The logical-or of a vertex cover equals $1111\ldots1$.

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. $1211\ldots112$. 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 $1 \le i \le E$, the ith slack value will consist of all 0's, except for a single 1-digit in the ith position, e.g., 00000100000. Our target will be the number $2222\ldots222$ (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 $k \ge 4$). 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.

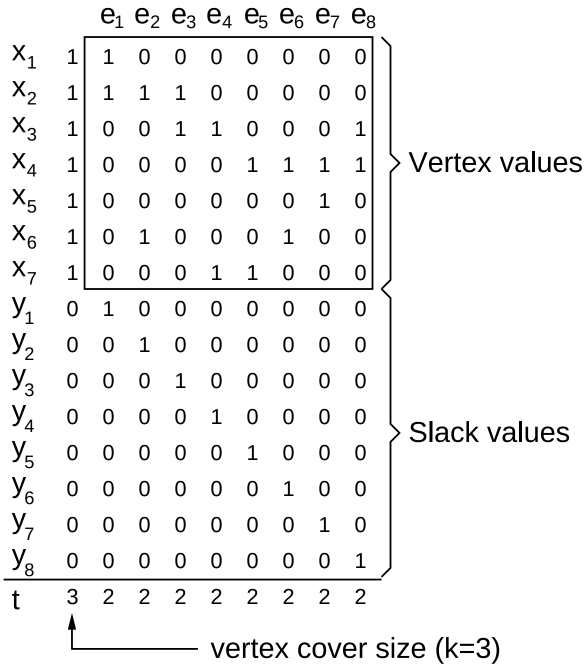
(1) Create a set of n vertex values, $x_1, x_2, \ldots, x_n$ using base-4 notation. The value x_i is equal a 1 followed by a sequence of E base-4 digits. The j-th digit is a 1 if edge e_j is incident to vertex v_i and 0 otherwise.

(2) Create E slack values $y_1, y_2, \ldots, y_E$, where y_i is a 0 followed by E base-4 digits. The i-th digit of y_i is 1 and all others are 0.

(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 $S = \{x_1, \ldots, x_n, y_1, \ldots, y_E\}$ and t.

Observe that this can be done in polynomial time, in $O(E^2)$, in fact. The construction is illustrated in Fig. 98.



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

Vertex values
Slack values
vertex cover size (k=3)

Fig. 98: 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 $222\ldots2$ and because there are k elements in V', the leftmost digit of the sum will be k. Thus, the resulting subset sums to t.

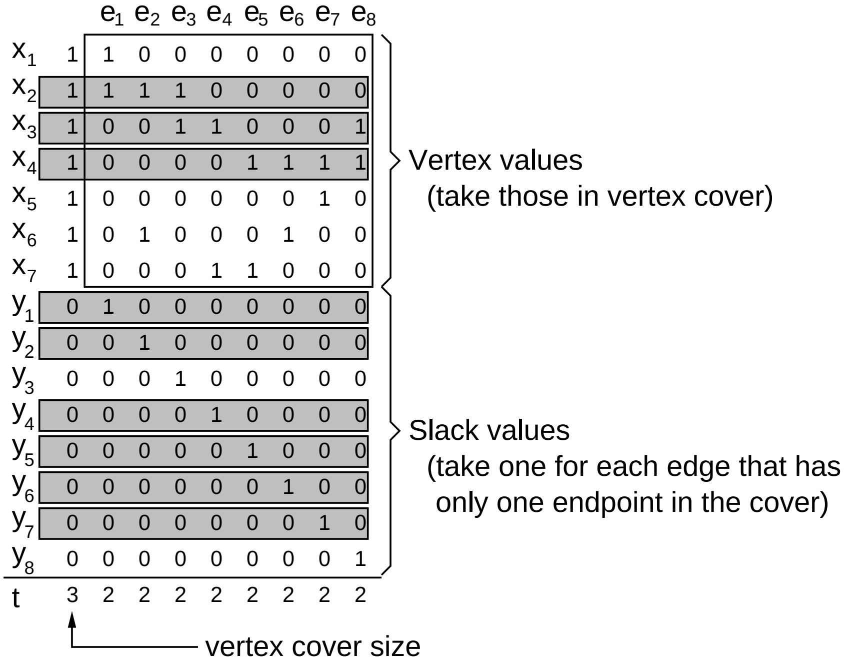
Conversely, if S has a subset S' that sums to t then we assert that it must select exactly k values from among the vertex values, since the first digit must sum to k. We claim that these vertices V' form a vertex cover. In particular, no edge can be left uncovered by V', since (because there are no carries) 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.

Supplemental Lecture 11: Subset Sum Approximation

Polynomial Approximation Schemes: Last time we saw that for some NP-complete problems, it is possible to approximate the problem to within a fixed constant ratio bound. For example, the approximation algorithm produces an answer that is within a factor of 2 of the optimal solution. However, in practice, people would like to the control the precision of the approximation. This is done by specifying a parameter $\epsilon > 0$ as part of the input to the approximation algorithm, and requiring that the algorithm produce an answer that is within a *relative error* of ϵ of the optimal solution. It is understood that as ϵ tends to 0, the running time of the algorithm will increase. Such an algorithm is called a *polynomial approximation scheme*.

For example, the running time of the algorithm might be $O(2^{(1/\epsilon)}n^2)$. It is easy to see that in such cases the user pays a big penalty in running time as a function of ϵ. (For example, to produce a 1% error, the "constant" factor would be 2^{100} which would be around 4 quadrillion centuries on your 100 Mhz Pentium.) A *fully polynomial*



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

Vertex values
(take those in vertex cover)
Slack values
(take one for each edge that has
only one endpoint in the cover)
vertex cover size

Fig. 99: Correctness of the reduction.

approximation scheme is one in which the running time is polynomial in both n and $1/\epsilon$. For example, a running time of $O((n/\epsilon)^2)$ would satisfy this condition. In such cases, reasonably accurate approximations are computationally feasible.

Unfortunately, there are very few NP-complete problems with fully polynomial approximation schemes. In fact, recently there has been strong evidence that many NP-complete problems do not have polynomial approximation schemes (fully or otherwise). Today we will study one that does.

Subset Sum: Recall that in the subset sum problem we are given a set S of positive integers $\{x_1, x_2, \ldots, x_n\}$ and a target value t, and we are asked whether there exists a subset $S' \subseteq S$ that sums exactly to t. The optimization problem is to determine the subset whose sum is as large as possible but not larger than t.

This problem is basic to many packing problems, and is indirectly related to processor scheduling problems that arise in operating systems as well. Suppose we are also given $0 < \epsilon < 1$. Let $z^* \leq t$ denote the optimum sum. The approximation problem is to return a value $z \leq t$ such that

$$z \geq z^*(1 - \epsilon).$$

If we think of this as a knapsack problem, we want our knapsack to be within a factor of $(1 - \epsilon)$ of being as full as possible. So, if $\epsilon = 0.1$, then the knapsack should be at least 90% as full as the best possible.

What do we mean by polynomial time here? Recall that the running time should be polynomial in the size of the input length. Obviously n is part of the input length. But t and the numbers x_i could also be huge binary numbers. Normally we just assume that a binary number can fit into a word of our computer, and do not count their length. In this case we will to be on the safe side. Clearly t requires $O(\log t)$ digits to be store in the input. We will take the input size to be $n + \log t$.

Intuitively it is not hard to believe that it should be possible to determine whether we can fill the knapsack to within 90% of optimal. After all, we are used to solving similar sorts of packing problems all the time in real life. But the mental heuristics that we apply to these problems are not necessarily easy to convert into efficient algorithms. Our intuition tells us that we can afford to be a little “sloppy” in keeping track of exactly full the knapsack is at any point. The value of ϵ tells us just how sloppy we can be. Our approximation will do something similar. First we consider an exponential time algorithm, and then convert it into an approximation algorithm.

Exponential Time Algorithm: This algorithm is a variation of the dynamic programming solution we gave for the knapsack problem. Recall that there we used an 2-dimensional array to keep track of whether we could fill a knapsack of a given capacity with the first i objects. We will do something similar here. As before, we will concentrate on the question of which sums are possible, but determining the subsets that give these sums will not be hard.

Let L_i denote a list of integers that contains the sums of all 2^i subsets of $\{x_1, x_2, \ldots, x_i\}$ (including the empty set whose sum is 0). For example, for the set $\{1, 4, 6\}$ the corresponding list of sums contains $\langle 0, 1, 4, 5(= 1 + 4), 6, 7(= 1 + 6), 10(= 4 + 6), 11(= 1 + 4 + 6)\rangle$. Note that L_i can have as many as 2^i elements, but may have fewer, since some subsets may have the same sum.

There are two things we will want to do for efficiency. (1) Remove any duplicates from L_i, and (2) only keep sums that are less than or equal to t. Let us suppose that we a procedure `MergeLists(L1, L2)` which merges two sorted lists, and returns a sorted lists with all duplicates removed. This is essentially the procedure used in MergeSort but with the added duplicate element test. As a bit of notation, let $L + x$ denote the list resulting by adding the number x to every element of list L. Thus $\langle 1, 4, 6\rangle + 3 = \langle 4, 7, 9\rangle$. This gives the following procedure for the subset sum problem.

Exact Subset Sum

```
Exact_SS(x[1..n], t) {
    L = <0>;
    for i = 1 to n do {
        L = MergeLists(L, L+x[i]);
        remove for L all elements greater than t;
    }
    return largest element in L;
}
```

For example, if $S = \{1, 4, 6\}$ and $t = 8$ then the successive lists would be

$$\begin{aligned}
L_0 &= \langle 0\rangle \\
L_1 &= \langle 0\rangle \cup \langle 0 + 1\rangle = \langle 0, 1\rangle \\
L_2 &= \langle 0, 1\rangle \cup \langle 0 + 4, 1 + 4\rangle = \langle 0, 1, 4, 5\rangle \\
L_3 &= \langle 0, 1, 4, 5\rangle \cup \langle 0 + 6, 1 + 6, 4 + 6, 5 + 6\rangle = \langle 0, 1, 4, 5, 6, 7, 10, 11\rangle.
\end{aligned}$$

The last list would have the elements 10 and 11 removed, and the final answer would be 7. The algorithm runs in $\Omega(2^n)$ time in the worst case, because this is the number of sums that are generated if there are no duplicates, and no items are removed.

Approximation Algorithm: To convert this into an approximation algorithm, we will introduce a "trim" the lists to decrease their sizes. The idea is that if the list L contains two numbers that are very close to one another, e.g. $91,048$ and $91,050$, then we should not need to keep both of these numbers in the list. One of them is good enough for future approximations. This will reduce the size of the lists that the algorithm needs to maintain. But, how much trimming can we allow and still keep our approximation bound? Furthermore, will we be able to reduce the list sizes from exponential to polynomial?

The answer to both these questions is yes, provided you apply a proper way of trimming the lists. We will trim elements whose values are sufficiently close to each other. But we should define close in manner that is relative to the sizes of the numbers involved. The trimming must also depend on ϵ. We select $\delta = \epsilon/n$. (Why? We will see later that this is the value that makes everything work out in the end.) Note that $0 < \delta < 1$. Assume that the elements of L are sorted. We walk through the list. Let z denote the last untrimmed element in L, and let $y \geq z$ be the next element to be considered. If

$$\frac{y-z}{y} \leq \delta$$

then we trim y from the list. Equivalently, this means that the final trimmed list cannot contain two value y and z such that

$$(1-\delta)y \leq z \leq y.$$

We can think of z as *representing* y in the list.

For example, given $\delta = 0.1$ and given the list

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle,$$

the trimmed list L' will consist of

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle.$$

Another way to visualize trimming is to break the interval from $[1, t]$ into a set of *buckets* of exponentially increasing size. Let $d = 1/(1-\delta)$. Note that $d > 1$. Consider the intervals $[1, d], [d, d^2], [d^2, d^3], \ldots, [d^{k-1}, d^k]$ where $d^k \geq t$. If $z \leq y$ are in the same interval $[d^{i-1}, d^i]$ then

$$\frac{y-z}{y} \leq \frac{d^i - d^{i-1}}{d^i} = 1 - \frac{1}{d} = \delta.$$

Thus, we cannot have more than one item within each bucket. We can think of trimming as a way of enforcing the condition that items in our lists are not relatively too close to one another, by enforcing the condition that no bucket has more than one item.

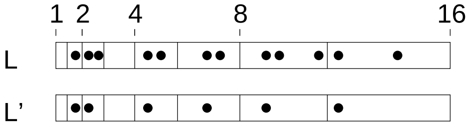


Fig. 100: Trimming Lists for Approximate Subset Sum.

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 $1/\epsilon$.

```
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;
}
```

The approximation algorithm operates as before, but in addition we call the procedure `Trim` given below.

For example, consider the set $S = \{104, 102, 201, 101\}$ and $t = 308$ and $\epsilon = 0.20$. We have $\delta = \epsilon/4 = 0.05$. Here is a summary of the algorithm’s execution.

$$\begin{array}{rrcl}
\text{init:} & L_0 & = & \langle 0\rangle \\
\\
\text{merge:} & L_1 & = & \langle 0, 104\rangle \\
\text{trim:} & L_1 & = & \langle 0, 104\rangle \\
\text{remove:} & L_1 & = & \langle 0, 104\rangle \\
\\
\text{merge:} & L_2 & = & \langle 0, 102, 104, 206\rangle \\
\text{trim:} & L_2 & = & \langle 0, 102, 206\rangle \\
\text{remove:} & L_2 & = & \langle 0, 102, 206\rangle \\
\\
\text{merge:} & L_3 & = & \langle 0, 102, 201, 206, 303, 407\rangle \\
\text{trim:} & L_3 & = & \langle 0, 102, 201, 303, 407\rangle \\
\text{remove:} & L_3 & = & \langle 0, 102, 201, 303\rangle \\
\\
\text{merge:} & L_4 & = & \langle 0, 101, 102, 201, 203, 302, 303, 404\rangle \\
\text{trim:} & L_4 & = & \langle 0, 101, 201, 302, 404\rangle \\
\text{remove:} & L_4 & = & \langle 0, 101, 201, 302\rangle
\end{array}$$

The final output is 302. The optimum is $307 = 104 + 102 + 101$. So our actual relative error in this case is within 2%.

The running time of the procedure is $O(n|L|)$ which is $O(n^2 \ln t/\epsilon)$ by the earlier claim.

Approximation Analysis: The final question is why the algorithm achieves an relative error of at most ϵ over the optimum solution. Let Y^* denote the optimum (largest) subset sum and let Y denote the value returned by the algorithm. We want to show that Y is not too much smaller than Y^*, that is,

$$Y \geq Y^*(1 - \epsilon).$$

Our proof will make use of an important inequality from real analysis.

Lemma: For $n > 0$ and a real numbers,

$$(1 + a) \leq \left(1 + \frac{a}{n}\right)^n \leq e^a.$$

Recall that our intuition was that we would allow a relative error of ϵ/n at each stage of the algorithm. Since the algorithm has n stages, then the total relative error should be (obviously?) $n(\epsilon/n) = \epsilon$. The catch is that these are relative, not absolute errors. These errors to not accumulate additively, but rather by multiplication. So we need to be more careful.

Let L_i^* denote the i-th list in the exponential time (optimal) solution and let L_i denote the i-th list in the approximate algorithm. We claim that for each $y \in L_i^*$ there exists a representative item $z \in L_i$ whose relative error from y that satisfies

$$(1 - \epsilon/n)^i y \leq z \leq y.$$

The proof of the claim is by induction on i. Initially $L_0 = L_0^* = \langle 0 \rangle$, and so there is no error. Suppose by induction that the above equation holds for each item in L_{i-1}^*. Consider an element $y \in L_{i-1}^*$. We know that y will generate two elements in L_i^*: y and $y + x_i$. We want to argue that there will be a representative that is "close" to each of these items.

By our induction hypothesis, there is a representative element z in L_{i-1} such that

$$(1 - \epsilon/n)^{i-1} y \leq z \leq y.$$

When we apply our algorithm, we will form two new items to add (initially) to L_i: z and $z + x_i$. Observe that by adding x_i to the inequality above and a little simplification we get

$$(1 - \epsilon/n)^{i-1}(y + x_i) \leq z + x_i \leq y + x_i.$$

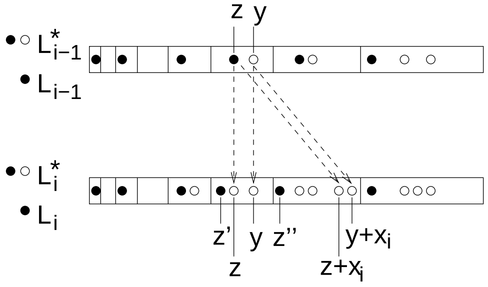


Fig. 101: Subset sum approximation analysis.

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

$$\begin{array}{rcl} (1 - \epsilon/n)z & \leq z' \leq & z \\ (1 - \epsilon/n)(z + x_i) & \leq z'' \leq & z + x_i. \end{array}$$

Combining these with the inequalities above we have

$$\begin{array}{rcl} (1 - \epsilon/n)^{i-1}(1 - \epsilon/n)y \;\leq\; (1 - \epsilon/n)^i y & \leq z' \leq & y \\ (1 - \epsilon/n)^{i-1}(1 - \epsilon/n)(y + x_i) \;\leq\; (1 - \epsilon/n)^i (y + x_i) & \leq z'' \leq & z + y_i. \end{array}$$

Since z and z'' are in L_i this is the desired result. This ends the proof of the claim.

Using our claim, and the fact that Y^* (the optimum answer) is the largest element of L_n^* and Y (the approximate answer) is the largest element of L_n we have

$$(1 - \epsilon/n)^n Y^* \leq Y \leq Y^*.$$

This is not quite what we wanted. We wanted to show that $(1 - \epsilon)Y^* \leq Y$. To complete the proof, we observe from the lemma above (setting $a = -\epsilon$) that

$$(1 - \epsilon) \leq \left(1 - \frac{\epsilon}{n}\right)^n.$$

This completes the approximate analysis.

Supplemental Lecture 12: Hamiltonian Path

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?

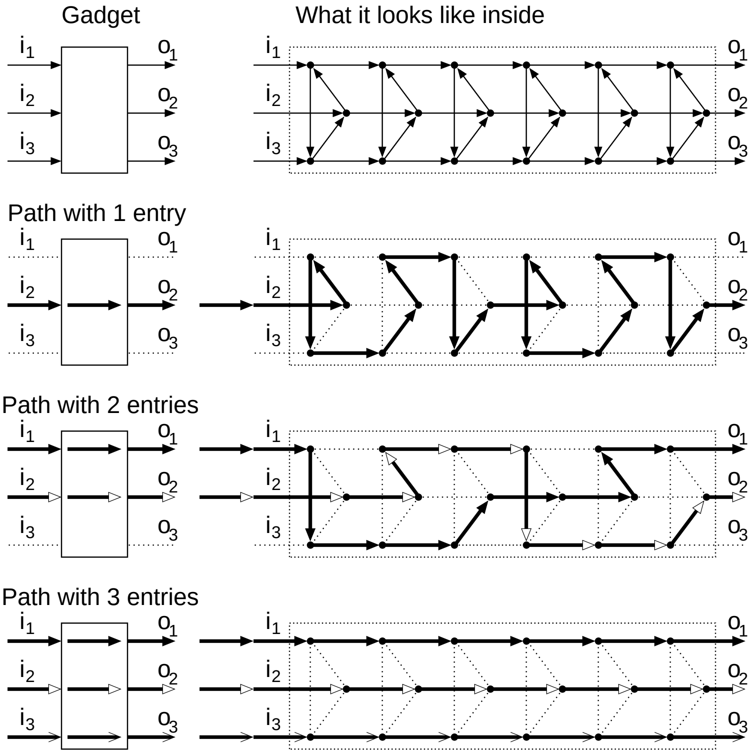


Fig. 102: DHP-Gadget and examples of path traversals.

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

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

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

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

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

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

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

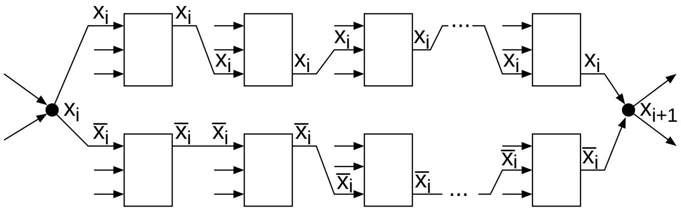


Fig. 103: General structure of reduction from 3SAT to DHP.

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

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

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

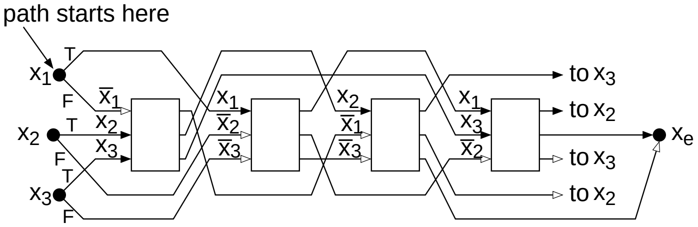
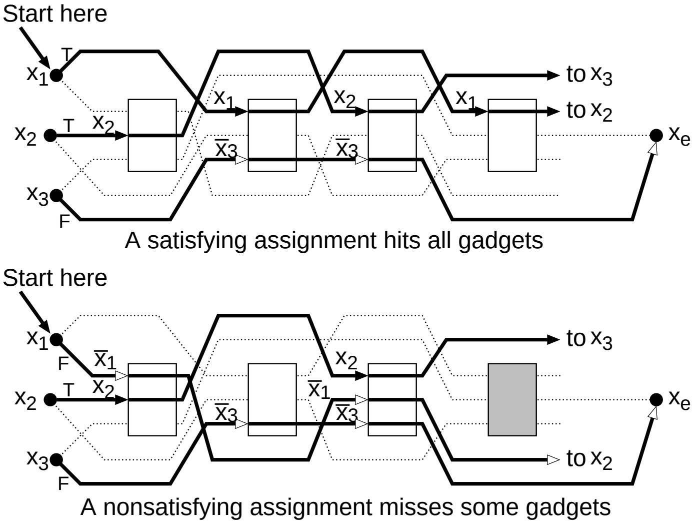


Fig. 104: Example of the 3SAT to DHP reduction.

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

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



A satisfying assignment hits all gadgets

A nonsatisfying assignment misses some gadgets

Fig. 105: 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 $x_1 = 0$, $x_2 = 1$, $x_3 = 0$.

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".

⇒: 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.

⇐: 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.

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

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

Supplemental Lecture 13: Approximations: Set Cover and Bin Packing

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 $U = \{x_1, x_2, \ldots, x_m\}$ is a finite set (a domain of elements), called the *universe*, and $F = \{S_1, S_2, \ldots, S_n\}$ 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. 106.

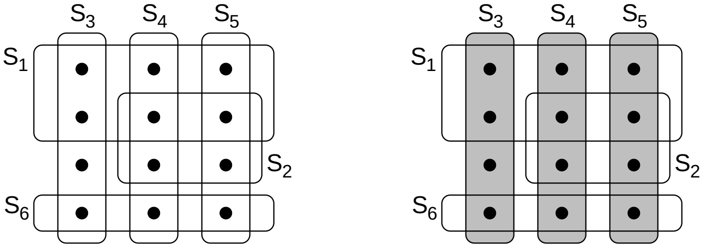


Fig. 106: Set cover. The optimum set cover consists of the three sets $\{S_3, S_4, S_5\}$.

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 than 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. 107.)

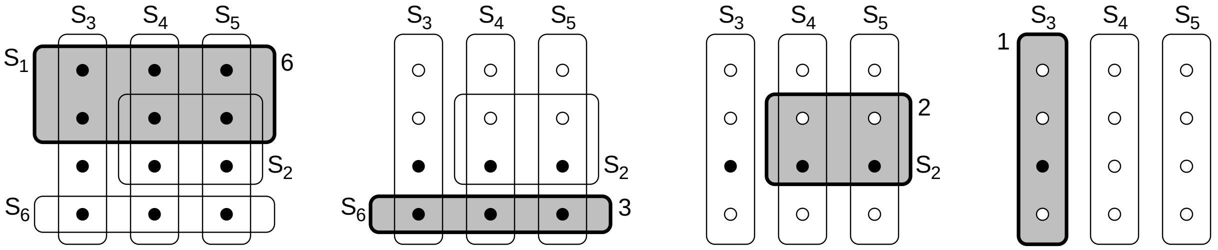


Fig. 107: Example of the greedy algorithm. Final set cover is $\{S_1, S_6, S_2, S_3\}$.

What is the approximation factor? The problem with the greedy set cover algorithm is that it can be "fooled" into picking the wrong set, over and over again. Consider the example shown in Fig. 108. 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.

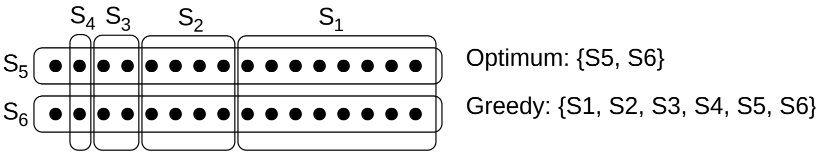


Fig. 108: An example in which the Greedy Set cover performs poorly.

However we will show that the greedy set cover heuristic nevers 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 $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \geq 1 + x$.) Now, if we substitute $-1/c$ for x we have $(1 - 1/c) \leq e^{-1/c}$. By raising both sides to the cth 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 ith 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 pigeonhold 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:

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

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 ith 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. 109. We claim that first-fit uses at most twice as many bins as the optimum. That is, if the optimal solution uses b_{opt} bins, and first-fit uses b_{ff} bins, then we show below that

$$\frac{b_{\text{ff}}}{b_{\text{opt}}} \leq 2.$$

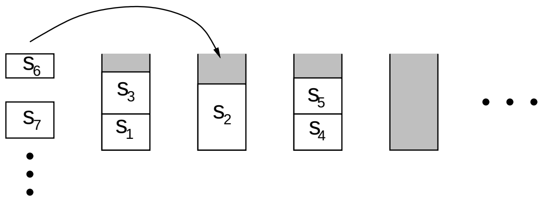


Fig. 109: First-fit Heuristic.

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

Proof: Consider an instance $\{s_1, \ldots, s_n\}$ of the bin packing problem. Let $S = \sum_i s_i$ denote the sum of all the object sizes. Let b_{opt} denote the optimal number of bins, and b_{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 $11/9 \approx 1.222$.

Supplemental Lecture 14: Approximation Algorithms: The k-Center Problem

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 $d(u, v) = d(u, v)$ denote the *distance* between u to v, that is, the length of the shortest path from u to v. Note that the shortest path distance satisfies the triangle inequality. This will be used in our proof.

Consider a subset $C \subseteq V$ of vertices, the *centers*. For each vertex $v \in V$ 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. 110.) More formally, define:

$$V(c_i) \;=\; \{v \in V \mid d(v, c_i) \leq d(v, c_j),\ \text{for } i \neq j\}.$$

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 $V(c_1), V(c_2), \ldots, V(c_k)$ 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,

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

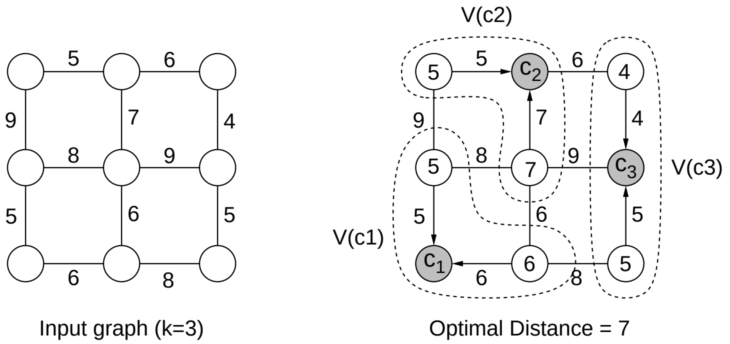


Fig. 110: The k-center problem with optimum centers c_i and neighborhood sets $V(c_i)$.

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.

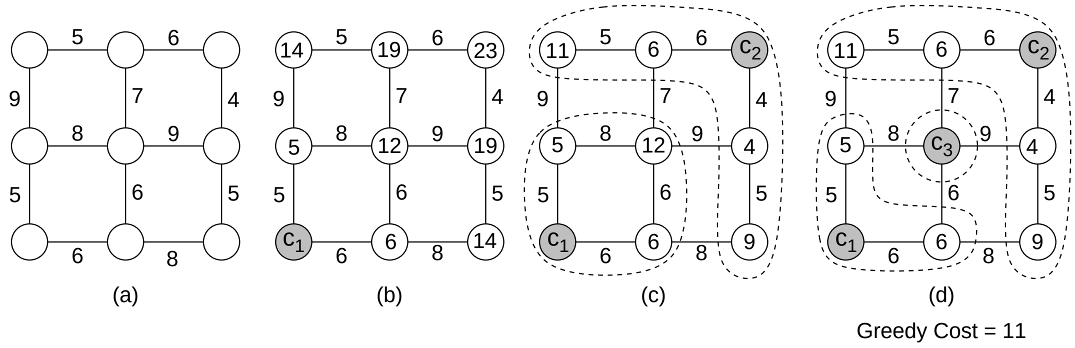


Fig. 111: Greedy approximation to k-center.

We begin by letting the first center c_1 be *any* vertex in the graph (the lower left vertex, say, in Fig. 111(a)). Compute the distances between this vertex and all the other vertices in the graph (Fig. 111(b)). Consider the vertex that is farthest from this center (the upper right vertex at distance 23 in the figure). This the bottleneck vertex for $\{c_1\}$. We would like to select the next center so as to reduce this distance. So let us just make it the next center, called c_2. Then again we compute the distances from each vertex in the graph to the *closer* of c_1 and c_2. (See Fig. 111(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. 111(d)). Again we compute the distances from each vertex to its nearest center. Repeat this until all k centers have been selected. In Fig. 111(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 $pred[s] =$ 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. 112.) 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 $(n/2)/(n/4) = 2$.

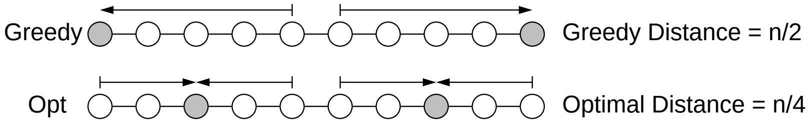


Fig. 112: An example showing that greedy can be a factor 2 from optimal. Here $k = 2$.

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 $O = \{o_1, o_2, \ldots, o_k\}$ denote the centers of the optimal solution (shown as black dots in Fig. 113, and the lines show the partition into the neighborhoods for each of these points). Let $\Delta_* = \Delta(O)$ be the optimal bottleneck distance.

Let $\mathcal{G} = \{g_1, g_2, \ldots, g_k\}$ be the centers found by the greedy approximation (shown as white dots in the figure below). Also, let g_{k+1} denote the next center that *would have* been added next, that is, the bottleneck vertex for $\mathcal{G}$. Let Δ_G denote the bottleneck distance for $\mathcal{G}$. Notice that the distance from g_{k+1} to its nearest center is equal Δ_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 $\mathcal{G}' = \{g_1, g_2, \ldots, g_k, g_{k+1}\}$ be the $(k+1)$-element set consisting of the greedy centers together with the next greedy center g_{k+1} First observe that for $i \neq j$, $d(g_i, g_j) \geq \Delta_G$. 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 Δ_G, all the centers are at least this far apart from one another.

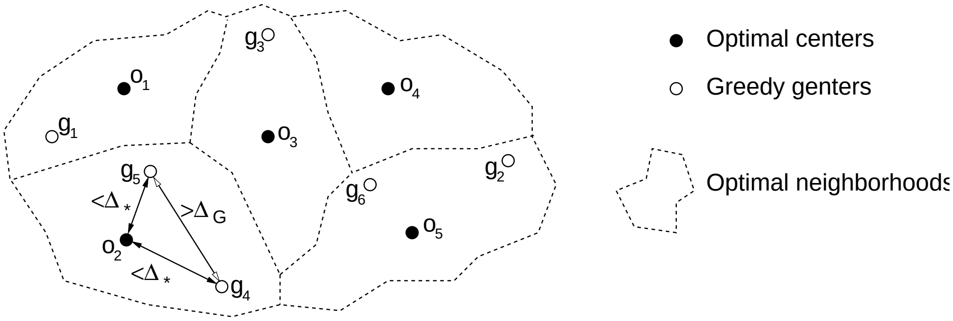


Fig. 113: Analysis of the greedy heuristic for $k = 5$.

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 Δ_* is the bottleneck distance for O, we know that the distance from g_i to o_k is of length at most Δ_* and similarly the distance from o_k to g_j is at most Δ_*. 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.