CMSC 451
Design and Analysis of Computer Algorithms[1]

David M. Mount
Department of Computer Science
University of Maryland

Fall 2015

Lecture 1: Introduction to Algorithm Design

What is an algorithm? This course will focus on the study of the design and analysis of algorithms for discrete (as opposed to numerical) problems. 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.

In complex software systems, a large amount of code is devoted to relatively mundane tasks, such as checking that inputs have the desired format, converting between data representations, handling exceptions, interfacing with the operating system. In contrast, we will focus here on the relatively small fraction of code that performs the most interesting and complex parts of the computation. These complex computations often consume the vast majority of the execution time, so they are very important.

One of the lessons learned from the many years of research in algorithm design is that there are standard ways of approaching the design of algorithms, such as *divide-and-conquer*, *dynamic programming*, *greedy algorithms*, and so on. Another important principle is using high-level tools, such as *worst-case asymptotic analysis*, to obtain a rough idea of an algorithm's running time.

Using these techniques, it is possible to arrive at a good general strategy (or a small number of strategies) for solving these complex computational tasks. Then *prototypes* can be implemented and analyzed to determine their actual efficiency in practice.

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 (particularly for network flow problems).

Most of the emphasis of the first portion of the course will be on problems that can be solved efficiently, in the latter portion we will discuss intractability and NP-hard problems. These are problems for which no efficient solution is known. Finally, we will discuss methods to approximate NP-hard problems, and how to prove how close these approximations are to the optimal solutions.

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. In this class, you will learn how to present clear proofs of an algorithm's correctness (both through seeing examples and developing your own).

Efficiency: 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 *expected* running time? This expectation is computed assuming that the inputs are drawn from some given probability distribution. The choice of distribution can have a significant impact on the final conclusions.

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. For example, some algorithms have the property that they run very fast on typical inputs but might run extremely slowly (perhaps hundreds to thousands of times slower) on a very small fraction of *pathological* inputs. For such algorithms, an average case analysis may be a much more accurate reflection of the algorithm's true performance.

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 plain English prose or pseudo-code, for example). A guiding principal here is to remember that your description will be read by a human, not a compiler. Obvious technical details should be kept to a minimum so that the key computational issues stand out.

Here are a few tips:

- Instead of giving a type declaration (e.g., "double x"), explain a variable's purpose (e.g., "x holds the price of the current commodity")
- When employing standard data structures, explain what your intent (e.g., "append x to the list of nodes"), rather than using the arcane notation expected by a compiler (e.g., "theNodeList.append(x)")
- Replace formal control structures (e.g., "for (int i = 1; i <= 3*n; i += 3)" with more intuitive explanations (e.g., "Let i run from 1 up to $3n$ in steps of 3")
- Employ standard algorithms that would be known to the reader (e.g., "Apply radix sort to sort the node labels in increasing order")

While you should avoid unnecessary formalisms whenever possible, be sure to include enough information that your intent is clear and unambiguous. For example, consider the instruction "Repeatedly remove the highest weight edge from G until the graph is no longer connected." While this is mathematically well defined, the questions of how to (1) find the highest weight edge and (2) determine whether the resulting graph is connected are both nontrivial to implement. These steps would need to be explained in greater detail.

Prove its Correctness: Present a justification (that is, an informal proof) of the algorithm's correctness. This justification may assume that the reader is familiar with the basic background material presented in class. Try to avoid rambling about obvious or trivial elements and focus on the key elements. A good proof provides a high-level 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 and other times it might involve setting up and solving a complex recurrence or a summation. When possible, try to reason based on algorithms that you have seen. For example, the recurrence $T(n) = 2T(n/2) + n$ is common in divide-and-conquer algorithms (like Mergesort) and it is well known that it solves to $O(n \log n)$.

Note that your 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.

Background Information: I will assume that you have familiarity with the information from a basic algorithms course, such as CMSC 351. As is indicated in the syllabus, it is expected that you have knowledge of:

- Basic programming skills (programming with loops, pointers, structures, recursion)
- Discrete mathematics (proof by induction, sets, permutations, combinations, and probability)
- Understanding of basic data structures (lists, stacks, queues, trees, graphs, and heaps)
- Knowledge of sorting algorithms (MergeSort, QuickSort, HeapSort) and basic graph algorithms (minimum spanning trees and Dijkstra's algorithm)

- Basic calculus (manipulation of exponentials, logarithms, differentiation, and integration)

I will provide some handouts reviewing some of this information.

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 have 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" in the sense that matched pairs should not have an obvious incentive to split up in order to form a different partnership.

Following tradition, 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.*[2] 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 women: Anny (A), Betty (B), and Carry (C), and there are three men: Eddy (E), Freddy (F), and Gerry (G). Here are their preferences (highest to lowest).

Men			Women		
Eddy (E)	Freddy (F)	Gerry (G)	Anny (A)	Betty (B)	Carry (C)
B	B	C	G	G	E
A	C	B	F	E	F
C	A	A	E	F	G

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 Eddy and Freddy list Betty first.) There is a very weak condition that we would like to place on our

[2] It is worth noting that in the above applications there is an asymmetrical relationship between the groups. We shall see that the algorithm that we will develop will *not* be gender-neutral. In particular, one gender will play a more active role and the other a more passive role. While this analogy may have made reasonable sense in early 1960's American culture, when the algorithm was first developed, many aspects of the algorithm's description seem out of place with modern culture. If this bothers you, please feel free to swap all references to "men" and "women".

matching. 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 together. That is, there should be no man who can say to another woman, “We each prefer each other to our assigned partners—let’s elope!” 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. (**Beware:** Perfectness in a matching has nothing to do with optimality or stability. It simply means that everyone has a mate.)

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 over their assigned choice.

Assignment I	Assignment II	Assignment III
E [B] $\leftrightarrow$ A [G, F]	E [B, A] $\leftrightarrow$ C [ ]	E [B, A] $\leftrightarrow$ C [ ]
F [B] $\leftrightarrow$ C [E]	F [ ] $\leftrightarrow$ B [G, E]	F [B, C] $\leftrightarrow$ A [G]
G [C] $\leftrightarrow$ B [ ]	G [C, B] $\leftrightarrow$ A [ ]	G [C] $\leftrightarrow$ B [ ]

The answer appears in the footnote below[3] You might wonder whether among all stable matchings, are some better than others? What would “better” mean? (More stable?) 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. The algorithm is based on two basic primitive actions:

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.)

[3] The only unstable one is II. Observe that Eddy would have preferred Betty over his assigned mate Carry, and Betty would have preferred Eddy to her assigned mate Freddy. Thus, the unmarried pair (E, B) is an example of an instability. It is easy to verify that assignments I and III are stable.

The original Gale-Shapley algorithm was presented as occurring over a sequence of *rounds*, during which all the unengaged men make proposals all at once, followed by the women either accepting or rejecting these proposals. However, in our book this is simplified by observing that the loop structure is simpler (and the results no different) if we process one man at a time, repeating the process until every man is either engaged or has exhausted everyone on his preference list.

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 two algorithms are essentially no different, and the order of events and final results are the same for both.) 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 she accepts ((m, w) are now engaged)
    else {
        Let m' be the man w is engaged to currently
        if (w prefers m to m') {
            Break off the engagement (m', w)
            Create the new engagement (m, w) (upgrade)
            Man m' is now unengaged
        }
    }
}
```

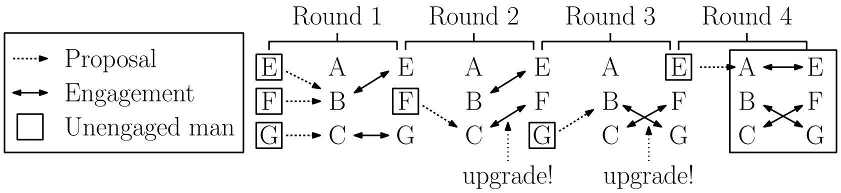


Fig. 1: Example of the round form version of the GS Algorithm on the preference lists given earlier. The final matching is (Eddy ↔ Anny), (Freddy ↔ Betty), (Gerry ↔ Carry).

Correctness of the Gale-Shapley Algorithm: Here are some easy observations regarding the Gale-Shapley (GS) Algorithm.

Lemma 1: Once a woman becomes engaged, she remains engaged for the remainder of the algorithm (although her mate may change), 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.

Lemma 1 follows from the fact that a woman only breaks off an engagement to form a new one to a man of higher preference. Lemma 2 follows from the fact that each man makes offers in decreasing preference order.

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 (by Lemma 2). 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. Recall that 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 m 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 w 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 w has never been asked. But she appears on m'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 is an unmarried pair (m, w) with the following properties. Let w' denote the assigned mate of m and let m' denote the assigned mate to w.

- m prefers w to his assigned mate w', and
- w prefers m to her assigned mate m' (see Fig. 2),

(and hence m and w have the incentive to elope).

Let's see why this cannot happen. Observe that since m prefers w he proposed to his preferred mate w before w'. What went wrong with his plans? Either w was already engaged to someone she preferred over m and rejected the offer outright, or she took his offer initially but later opted for someone whom she preferred and broke off the engagement with m. (Recall from Lemma 1 that once engaged, a woman's assigned

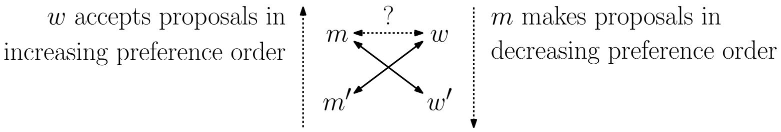


Fig. 2: The proof of Lemma 5.

mate only improves over time with respect to her preferences.) In either case, w ends up with someone she prefers over m. This means that she ends up with someone that she prefers over m', whom she ranked even lower than m. Thus, the pair (m', w) could never have been generated by the algorithm, which yields the desired contradiction.

In summary, we have shown that the algorithm terminates, and it generates a correct result.

Algorithm Efficiency: Is this an efficient algorithm? Observe that this is much more efficient than a brute-force algorithm, which simply enumerates all the possible matchings, testing whether each is stable. This algorithm would take at least $\Omega(n!)$ running time. Given how fast the factorial function grows, such an approach would only be useable for very small input sizes.

As observed earlier in Lemma 3, the GS algorithm runs in $O(n^2)$ time. While normally, we would be inclined to call an algorithm running in $O(n^2)$ time a *quadratic time* algorithm, notice that this is deceptively inaccurate. When we express running time, we do so in terms of the input size. In this case, the input for n men and n women consists of $2n$ preference lists, each consisting of n elements. Thus the input size is $N = 2n^2$. Since the algorithm runs in $O(n^2) = O(N)$ time, this is really a linear-time algorithm!

Note that in the practical applications where the GS algorithm is used, the input size is actually only $O(n)$. The reason is that, when very large input sizes are involved, it may not be practical to ask every many to rank order every woman, and vice versa. Typically, an individual is asked to rank just the top three or top five items in their preference list, and we hope that we can come up with a reasonably stable matching. Of course, if the preference lists are incomplete in this manner, then the algorithm may fail to produce a stable matching.

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.

$$\begin{array}{rcll}
f_1(n) & = & 43n^2\log^4 n + \underline{12n^3\log n} + 52n\log n & \in\ O(n^3\log n) \\
f_2(n) & = & \underline{15n^2} + 7n\log^3 n & \in\ O(n^2) \\
f_3(n) & = & 3n + 4\log_5 n + \underline{91n^2} & \in\ O(n^2) \\
f_4(n) & = & \underline{13\cdot 3^{2n+9}} + 4n^9 & \in\ O(3^{2n})\ \equiv\ O(9^n) \\
f_5(n) & = & \underline{\sum_{i=0}^{\infty} 1/2^i} & \in\ O(1)
\end{array}$$

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 facts about limits are useful:

- For $a, b > 0$, $\displaystyle\lim_{n\to\infty}\frac{(\log n)^a}{n^b} = 0$ (polynomials grow faster than polylogs).
- For $a > 0$ and $b > 1$, $\displaystyle\lim_{n\to\infty}\frac{n^a}{b^n} = 0$ (exponentials grow faster than polynomials).
- For $a, b > 1$, $\displaystyle\lim_{n\to\infty}\frac{\log_a n}{\log_b n} = c \neq 0$ (logarithm bases do not matter).
- For $1 < a < b$, $\displaystyle\lim_{n\to\infty}\frac{a^n}{b^n} = 0$ (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 ∞). Intuitively, each represents a form of "asymptotic relational operator":

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

Note that the aforementioned examples (f_1 through f_5) we could have replaced the O with either Θ or Ω, and they would still hold. Throughout this course, we will not worry about proving these facts, and will instead rely on a fairly intuitive understanding of asymptotic notation.

Do you get it? To see whether you understand this, consider the following functions. In each case, put the two functions increasing order of asymptotic growth rate. That is, indicate whether $f \prec g$ (meaning that $f(n)$ is $o(g(n))$), $g \prec f$ (meaning that $f(n)$ is $\omega(g(n))$) or $f \approx g$ (meaning that $f(n)$ is $\Theta(g(n))$).

(a) $f(n) = 3^{(n/2)}$, $g(n) = 2^{(n/3)}$.

(b) $f(n) = \lg(n^2)$, $g(n) = (\lg n)^2$.

(c) $f(n) = n^{\lg 4}$, $g(n) = 2^{(2 \lg n)}$.

(d) $f(n) = \max(n^2, n^3)$, $g(n) = n^2 + n^3$.

(e) $f(n) = \min(2^n, 2^{1000}n)$, $g(n) = n^{1000}$.

Answers appear in the footnote below.[4]

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.

[4] (a): $f(n) \succ g(n)$: $f(n) = (3^{1/2})^n \approx 1.73^n$ and $g(n) = (2^{1/3})^n \approx 1.26^n$. Thus, $f(n)/g(n) \approx 1.37^n$, whose limit tends to ∞. (b): $f(n) \prec g(n)$: $f(n) = 2 \lg n$ and so $f(n)/g(n) = 2/\lg n$, which tends to 0. (c): $f(n) \approx g(n)$: $f(n) = n^2$ and $g(n) = (2^{\lg n})^2 = n^2$. (d): $f(n) \approx g(n)$: Generally, if $x, y \geq 0$, then $(x+y)/2 \leq \max(x, y) \leq (x+y)$, and so the ratio $f(n)/g(n)$ lies between $1/2$ and 1. (e): $f(n) \prec g(n)$: For all sufficiently large n, $2^n > 2^{1000}n$, so $f(n)$ is asymptotically bounded by $2^{1000}n \approx n$, whereas $g(n) \approx n^{1000}$.

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 $n \geq 0$,

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

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

$$\sum_{i=0}^{n} c^i = 1+c+c^2+\cdots+c^n \;=\; \frac{c^{n+1}-1}{c-1} \;\in\; \begin{cases} \Theta(1) & \text{if } c<1 \\ \Theta(c^n) & \text{if } c>1 \end{cases}$$

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 $n \geq 0$,

$$\sum_{i=1}^{n} i^2 = 1^2+2^2+\cdots+n^2 \;=\; \frac{2n^3+3n^2+n}{6} \;\in\; \Theta(n^3).$$

In general, for any constant $p \geq 1$, it is not hard to show that $\sum_{i=1}^{n} i^p \in \Theta(n^{p+1})$. These are called *polynomial series*. The constant factor hidden in the O-notation depends on the value of p.

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

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

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 $n \geq 0$,

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

Aside: It is a bit of a headache to remember all these formulas. Typically we are only interested in the asymptotic results. Consider the series of the form $\sum_{i=1}^{n} f(i)$, where $f(i) \geq 1$. The last term of the summation is $f(n)$. Observe that asymptotic bounds are either $\Theta(n \cdot f(n))$ (as is the case for the arithmetic and quadratic series) or $\Theta(f(n))$ (as in the case of the geometric and linear-geometric series). Indeed, a simple rule to remember is that series that grow at a polynomial rate or lower are of the former form and series that grow at an exponential rate or higher are of the latter form.

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.

Approximation 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 $a_0 = \infty$. If we visualize a sequence of vertical poles, where the ith pole is of height a_i, this problem asks which pole would be hit if we shoot a bullet to the left of the top of each pole (see Fig. 3).

The code block below shows a very simple $O(n^2)$ time algorithm to solve this problem. The code is so simple that my pseudo-code is almost raw C or Java code. The correctness of this algorithm is easy to see. The inner while loop has two ways of terminating:

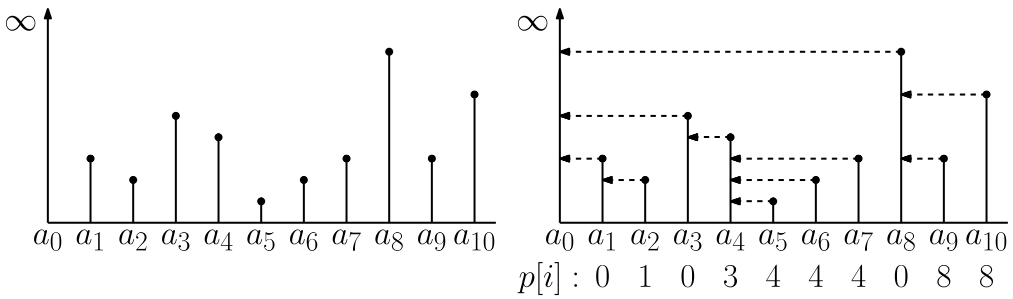

∞
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
∞
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
$p[i]$: 0 1 0 3 4 4 4 0 8 8

Fig. 3: Example of the previous larger element problem.

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.
naivePrevLarger(a, n) {
    for (i = 1 to n) {
        j = i-1
        while (j > 0 and a[j] <= a[i]) j--
        p[i] = j
    }
    return p
}
```

(1) if $a[j] > a[i]$, we have found a larger element at $a[j]$, and we set $p[i] = j$,

(2) if no prior larger elements exists then eventually $j = 0$, and we set $p[i] = j = 0$.

Worst-Case Analysis: 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:

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

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

$$T(n) \;=\; \frac{(n-1)n}{2} \in \Theta(n^2).$$

Average-Case Analysis? An interesting question to consider at this point is, what would the average-case running time be if the elements of the array are given in random order. Note that if i is large, it seems that it would be quite unlikely to go through most, much less all i iterations of the inner while loop, before finding a larger element. But exactly how many iterations would be expected? We will leave this as an exercise for the interested reader.

Faster Algorithm: The above algorithm is certainly simple, but can we do better? A key observation is that in the process of computing $p[i]$ we might exploit the fact that we already know the values of $p[1]$ up to $p[i-1]$. Can we exploit this fact?

The modified algorithm involves a very minor modification to the naive solution. In particular, we simply replace the statement "`j--`" in the while loop with "`j = p[j]`". To see why this is correct, observe that in order to get into the body of the while loop, it must be that $a[j] \leq a[i]$. Since $a[p[j]]$ is the previous element of the array to be larger than $a[j]$, all the elements from $a[p[j]]$ to $a[j-1]$ are not greater than $a[j]$, and hence they are not greater than $a[i]$. Thus, we may skip over all of them in one step.

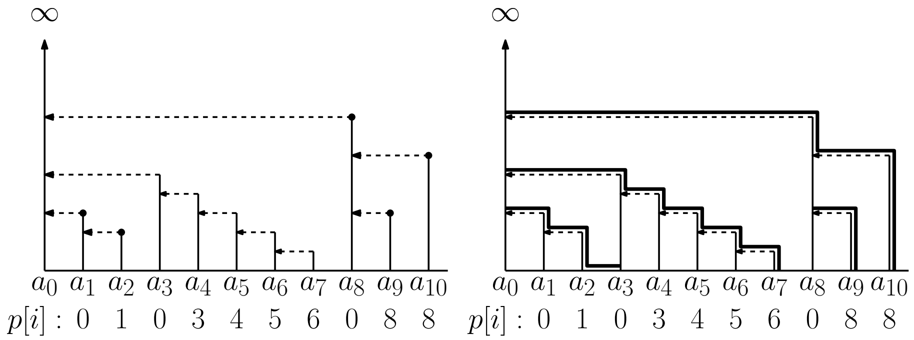

∞
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
$p[i]$: 0 1 0 3 4 5 6 0 8 8
∞
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10}
$p[i]$: 0 1 0 3 4 5 6 0 8 8

Fig. 4: Improved algorithm for the previous larger element problem, which operates by using the p-values to "leap-frog" over previously discovered smaller elements.

Fig. 4 illustrates this idea. The left side shows the output of the algorithm and the right side shows (in heavy lines) where jumps were made using the p-values.

By the comments made above this algorithm is correct, but is it asymptotically more efficient? You might be tricked into thinking not. Consider the following example, the elements $a[1..i-1]$ are in decreasing order, and thus, for $1 \leq j < i$, $p[j] = j-1$. Suppose that $a[i]$ is larger than all these elements. Using the p-values is no help here, since we would have to go element-by-element back to the start of the array. This takes i steps, just as in the naive algorithm. If each iteration of the for-loop can take $\Theta(i)$ time in the worst-case, then it would seem that the overall worst-case running time is $\sum_i i = \Theta(n^2)$. We will show that this not the case.

To see why the algorithm is much better, observe that the above worst-case scenario cannot be repeated. Once $p[i]$ is set, we will never again visit all these elements again, because $p[i]$ will jump over all of them. To make this insight clearer, we say that a pointer $p[j]$ is *touched* if the statement "`j = p[j]`" is executed for this value of j. We assert that each $p[j]$ can only be touched only once. The reason is that if $p[j]$ is touched in the process of computing $p[i]$ (for some $i > j$), then we know $p[i] \leq p[j]$, and therefore all subsequent searches (for $i' > i$) will use $p[i]$ to "leap-frog" over $p[j]$.

Now, since each $p[j]$ value can be touched only once, and there are n possible choices for j, it follows that the total number of touches is n. Of course, it is possible that the body of the while loop is not executed at all, meaning that no element of p is touched. This happens if and only if $a[i-1] > a[i]$. But this can happen at most n times, once for each iteration of the for-loop. Combining these two facts, it follows that the total number of iterations of the while loop is $O(n)$. (Pretty neat!)

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: Introduction to Graphs: Definitions, Representations, and BFS

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 *vertices* or *nodes*, and a set of pairwise relations E between these objects, called *edges.* Edges may be *directed* from one vertex to another or may be *undirected.* The term "graph" means an undirected graph, and directed graphs are often called *digraphs* (see Fig. 5). 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.

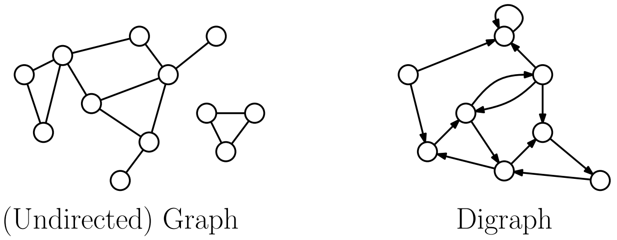


Fig. 5: Graphs and digraphs.

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.

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 $\text{out-deg}(v)$, and the number of edges coming in is called its *in-degree*, denoted $\text{in-deg}(v)$. 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: $0 \leq m \leq n^2$.

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

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. 6). (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. 6).

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. 6). 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*

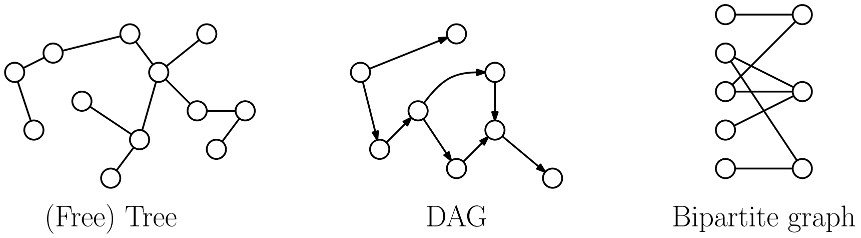


Fig. 6: Illustration of common graph terms.

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. In digraphs the notion of reachability is a bit different, because it is possible for u to reach w but not vice versa. A digraph is said to be *strongly connected* if for each u and w, there is a path from u to w and a path from w to u.

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 $1 \leq v, w \leq n$.

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

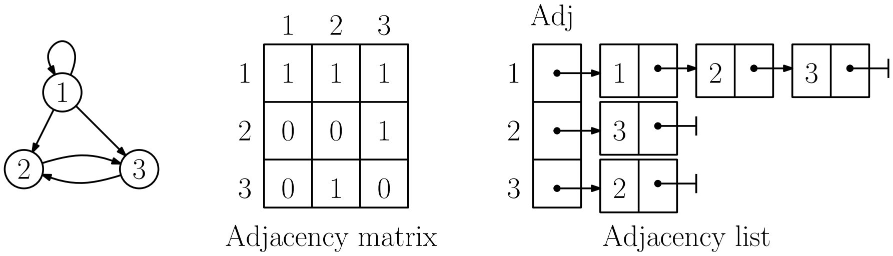
(See Fig. 7.) If the digraph has weights we can store the weights in the matrix. For example if $(v, w) \in E$ then $A[v, w] = W(v, w)$ (the weight on edge (v, w)). If $(v, w) \notin E$ then generally $W(v, w)$ need not be defined, but often we set it to some “special” value, e.g. $A(v, w) = -1$, or ∞. (By ∞ we mean some number which is larger than any allowable weight.)

It might come as a surprise, but there are a number of interesting relationships between the use of matrices to represent graphs and the matrices that arise in linear algebra to represent linear transformations. For example, the eigenvalues of the adjacency matrix of a graph provide a lot of information about the structure of the graph.

Adjacency List: An array $Adj[1 \ldots n]$ of pointers where for $1 \leq v \leq n$, $Adj[v]$ points to a list (e.g., a singly or doubly linked list) containing the vertices that 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. 7).

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. 8). 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.

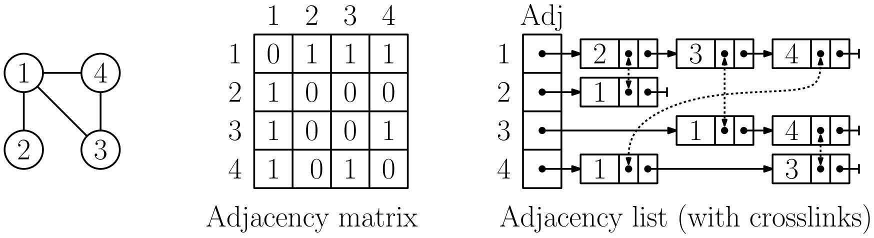
This can cause some complications. For example, suppose you write an algorithm that operates by marking edges of a graph. You need to be careful when you mark edge (v, w) in the



$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 \end{array}$$

Fig. 7: Adjacency matrix and adjacency list for digraphs.

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.



$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 1 \\ 4 & 1 & 0 & 1 & 0 \end{array}$$

Fig. 8: 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. 9.

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 vertex 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*.

Running-Time 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

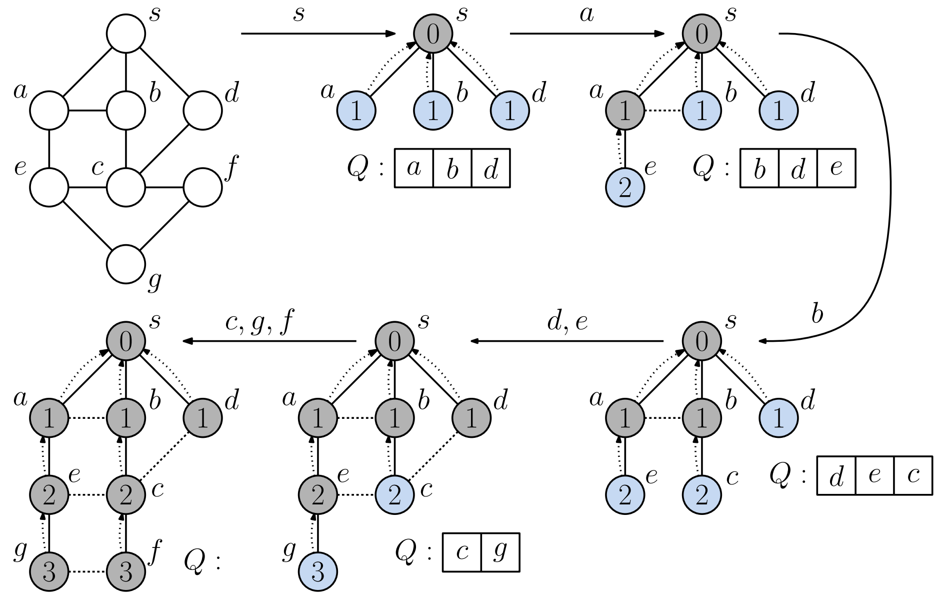


Fig. 9: Breadth-first search. (Tree edges are shown as solid lines, cross edges as dashed lines, and predecessor pointers as arrowed dotted lines.)

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.

Distances: It is useful to observe that BFS visits vertices in increasing order of d-values. This follows from the fact that we use a FIFO queue, so before we complete the processing all the vertices of a given d-value before we move on to process the vertices of the next higher d-value. Using this fact, we can show that the $d[u]$ is equal to the length of the shortest path from s to u.

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: For $i = 0, 1, \ldots$, define V_i denote the subset of vertices whose d-values are equal to i. It suffices to show that a vertex v is in V_i if and only it is at distance i from s, that is, $\delta(s, v) = i$. The proof is by induction on the distance from s.

For the basis case we have $d[s] = 0 = \delta(s, s)$. Clearly, $s \in V_0$ and all other vertices are given strictly larger d-values, so no other vertex is in this set.

Let us assume by induction that a vertex u is in V_{i-1} if and only if $\delta(s, u) = i - 1$, and we will then show that this holds for V_i. As observed above, vertices are processed in increasing order of d-values. Therefore, all the vertices of V_{i-1} are processed before any of the vertices of V_i are processed. By definition, each vertex $v \in V_i$ must be discovered by some vertex $u \in V_{i-1}$. When this discovery occurs, the algorithm sets $d[v] = d[u]+1$. Clearly, $\delta(s,v)$ cannot be smaller than i (or else by the induction hypothesis it would belong to $V_{i'}$ for some $i' < i$). Because v is connected to a vertex u at distance $i-1$ from s, we have

$$\delta(s,v) \;=\; \delta(s,u)+1 \;=\; (i-1)+1 \;=\; i,$$

as desired.

To prove the converse, observe that if any vertex v is at distance i from s, then the vertex u immediately prior to v on the shortest path from s to v is at distance $i-1$ from s. By the induction hypothesis $u \in V_{i-1}$. Therefore, v will be discovered by u (or some other vertex of V_{i-1}) and its d-value will be set to i. Therefore, $v \in V_i$ if and only if $\delta(s,v) = i$.

Cross-Edge Structure: Because of the nature of the algorithm, cross edges are not arbitrary. The following lemma shows that the d-values associated with each cross edge are restricted.

Lemma: If (x,y) is a cross edge in the execution of BFS (for either a directed or undirected graph), then $d[y] \leq d[x]+1$.

Proof: Suppose to the contrary that there is a cross edge (x,y) where $d[y] > d[x]+1$. We will show that this cannot happen. Since d-values are integers, we have $d[y] \geq d[x]+2$. Let z denote the vertex that discovered y. Since $d[y] = d[z]+1$, we can infer that $d[z] \geq d[x]+1$. Since BFS processes vertices in increasing d-value, z is processed after x. This implies that when x was visiting its neighbor y in the inner-for loop, y has not yet been discovered. Therefore, x would have discovered y, not z, which yields the contradiction.

If G is an undirected graph, we observe that if (x,y) is an edge, so is (y,x). Applying the above lemma to both endpoints, we have the following:

Lemma: If (x,y) is a cross edge in the execution of BFS for an undirected graph, then $d[x]-1 \leq d[y] \leq d[x]+1$.

Odd-Length Cycle: As an exercise to see whether you understand how BFS works, see whether you can use BFS to solve the following problem. Given an undirected graph $G=(V,E)$, does it contain a cycle of odd length? As a hint, observe that the tree edges of the BFS do not contain cycles. So, you should consider the structure of the cross edges.

The solution is based on BFS. As in the above skeleton algorithm, we maintain the d-values for each vertex. Define the *level* of a vertex in the BFS tree as the number of tree edges between it and the root. Clearly, $d[u]$ equals u's level. The algorithm is as follows. For each cross edge (u,v), we check whether it connects two vertices that are at the same level of the BFS tree. If this ever occurs, we announce that the graph contains an odd cycle, and otherwise we claim it does not.

To see why this is correct, observe first off that if there is a cross edge (u,v) connecting two vertices at the same level, then we can form an odd-length cycle as follows. Let a be the least common ancestor of u and v in the BFS tree. Let ℓ denote the number of levels that a lies above both u and v. We can construct a cycle by joining the edge (u, v) with the path from u to a and the path from a to v. The result is of length $2\ell + 1$, which is clearly odd.

Conversely, if there is no such cross-edge, then by our earlier observation, each cross-edge joins two vertices on consecutive levels. If we label all the vertices on even levels of the tree with 0 and all the vertices on the odd levels of the tree with 1, we see that every edge in the graph, whether tree or cross, joins a 0-vertex and 1-vertex. Therefore, there can be no cycle of odd length (because such a cycle must contain a 0-0 edge or a 1-1 edge.)

There is one technicality to implementing this algorithm. We need to detect whether an edge is a cross edge. To do this, we can add an else-clause to the if-statement of the BFS code. If we arrive at the else clause, then the vertex v must have already been discovered (and may even be finished). Such an edge is a good candidate for being a cross edge, but we should be careful because it might be the other half of a tree edge. (Recall that every edge of an undirected graph is represented twice.) So, to check whether the edge (u, v) is a cross edge, we check both (1) that v is already discovered and (2) that $v \neq \mathsf{pred}[u]$. If so, we add the additional test whether $d[u] = d[v]$. If all these conditions are ever satisfied, we report that the graph contains an odd-length cycle. If not, we report that the graph has no odd-length cycle.

Lecture 5: More on Graph Traversals: DFS

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 a graph $G = (V, E)$, which may be directed or undirected. We employ 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 time: $d[u]$ indicates the time when vertex u was discovered, which coincides with the moment that the DFS process is started at this vertex. (Don't confuse this with the distance value, $d[u]$, used in BFS. The two are very different.)

Finish time: $f[u]$ indicates the time when vertex u is finished processing. At this point, all of u's neighboring nodes have been visited, and indeed, everything reachable from u has been discovered and possibly finished.

Predecessor pointer: $p[u]$ indicates the vertex that discovered u. Each edge of the form $(p[u], u)$ is a tree edge in the DFS recursion tree.

As with BFS, DFS induces a tree structure. In order to handle instances where not all vertices are reachable from the starting vertex, we include a main program that invokes DFS whenever an undiscovered vertex is encountered. The main program is shown in code block below and the recursive DFSvisit function is shown in the next code block. (Fig. 10 illustrates the execution on an undirected graph, and Fig. 11 shows an example on a directed graph.)

Depth-First Search (Main Program)

```
DFS(G) {                                    // main program
    time = 0
    for each (u in V)                       // initialization
        mark[u] = undiscovered

    for each (u in V)
        if (mark[u] == undiscovered)        // undiscovered vertex?
            DFSVisit(u)                     // ...start a new search here
}
```

DFS Visit (Process a single node)

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

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.

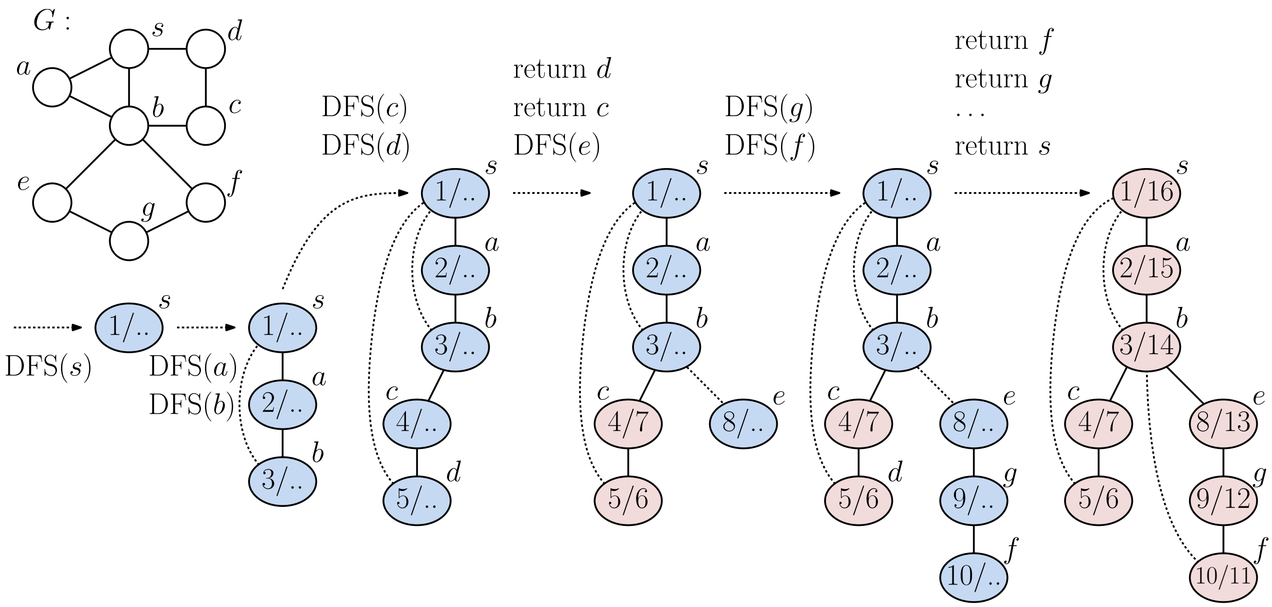


Fig. 10: Depth-first search on an undirected graph. (Each node u is labeled with the values $d[u]/f[u]$.)

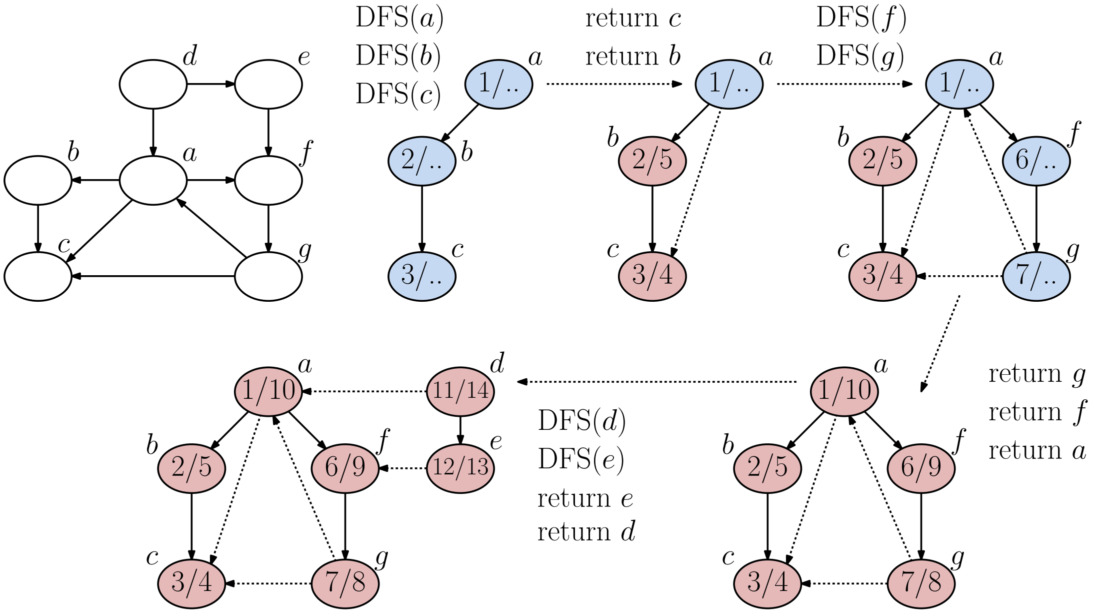
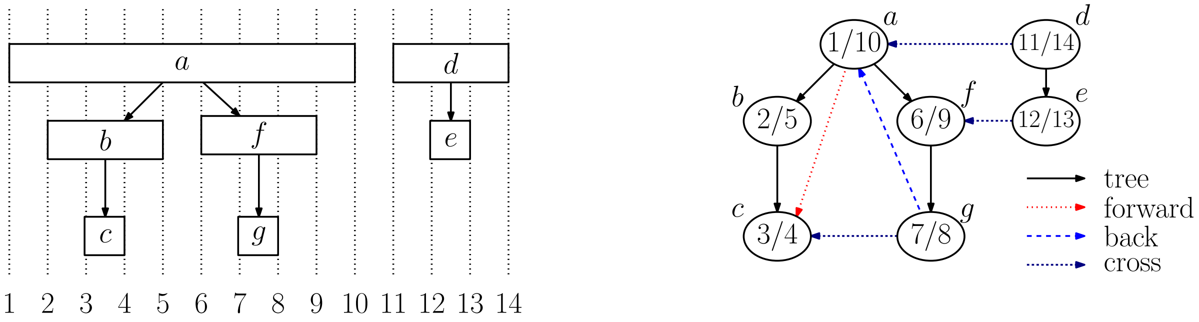


Fig. 11: Depth-first search on a directed graph.

Parenthesis Lemma and Edge Types: 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. The hierarchical structure naturally imposes a nesting structure on the discovery-finish time intervals. This is described in the following lemma (and illustrated in Fig. 12(a)).

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

- u is a descendant of v iff $[d[u], f[u]] \subseteq [d[v], f[v]]$.
- u is an ancestor of v iff $[d[u], f[u]] \supseteq [d[v], f[v]]$.
- u and v are unrelated (in terms of ancestor/descendant) iff $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint.



(a)

(b)

Fig. 12: (a) the Parenthesis Lemma and (b) the DFS edge types.

The structure of the remaining (non-tree) edges of the graph depend on the type of graph involved. For **undirected graphs**, the remaining edges are called *back edges*. An important observation is that for each back edge (u, v), u is either a proper ancestor or a proper descendant of v. To see why, consider any non-tree edge (u, v). Since the graph is undirected, we may assume without loss of generality that u was discovered before v. By the parenthesis lemma, this means either that u is an ancestor of v (and we are done) or that their discovery-finish intervals are disjoint. If they are disjoint, u must finish before v is discovered. However, this is impossible, because as we are processing u, we will see the edge (u, v) and thus discover v.

For **directed graphs** the non-tree edges of the graph can be classified as follows (See Fig. 12(b)):

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

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

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

It is not difficult to classify the edges of a DFS tree on-the-fly by analyzing the vertex status (undiscovered, discovered, finished) and/or considering the time stamps. (This is left as an exercise.)[5]

Cycles: The time stamps given by DFS allow us to determine a number of things about a graph or digraph. For example, it is easy to determine whether a graph is acyclic. Can you see how?

We do this with the help of the following two lemmas.

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

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

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

Proof:

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

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

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

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

Lecture 6: Applications of DFS: Topological Sort and Cut Vertices

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

[5] Be careful, however. Remember that in an undirected graph, every edge is represented twice. When classifying back edges, you should be sure that you are not seeing the other half of a tree edge.

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

Topological Sorting: A *topological sorting* (or *topological ordering*) of a DAG is a linear ordering of the vertices of the DAG such that for each edge (u, v), u appears before v in the ordering. Note that in general, there may be many valid orderings for a given DAG. We will present a simple algorithm based on DFS. (Kleinberg and Tardos present a different algorithm. We have elected this approach as an illustration of DFS.)

Recall our earlier comments on the nature of DFS edge types and discover/finish times. After running any DFS on a graph, if (u, v) is a tree, forward, or cross edge, then the finish time of u is greater than the finish time of v. Since a DAG is acyclic, there can be no back edges, which implies that *every* edge goes from node a higher finish time to one of lower finish times. Thus, in order to produce a topological ordering of the vertices it suffices to output the vertices in *reverse* order of finish times. To do this we run a (stripped down) DFS. As each vertex is finished, we push it onto a stack. (Thus, the later a vertex finishes, the closer it is to the top of the stack.) Popping the elements off the stack yields the final topological order.

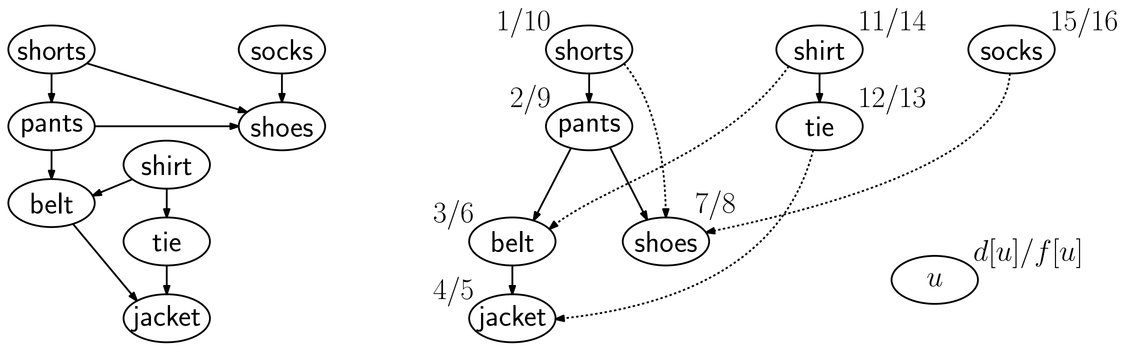
Topological Sort via DFS

```
topSort(G) {
    for each (u in V) mark[u] = undiscovered    // initialize
    S = empty stack
    for each (u in V)
        if (mark[u] == undiscovered) topVisit(u)
    while (S is nonempty) output pop(S)         // pop stack for final topol ordering
}

topVisit(u) {                                   // start a search at u
    mark[u] = discovered                        // mark u visited
    for each (v in Adj(u))
        if (mark[v] == undiscovered) topVisit(v)
    push u onto S                               // last to finish is top of stack
}
```

Observe that the structure is essentially the same as the generic DFS procedure given in the previous lecture, but we only include the elements of DFS that are needed for this application. As with standard DFS, the running time is $O(n + m)$ (recalling that $n = |V|$ and $m = |E|$).

As an example we consider the DAG showed in the Fig. 13(a), which shows the precedence constraints for a professor (who is obviously better dressed than me!) for putting on his clothes. In the example we show the discovery/finish times, but the algorithm does not need them. Note that there are many different possible DFS's of the same graph, and each one corresponds to a potentially different, but still valid, topological ordering. (A question worth pondering is whether every possible topological ordering arises from some DFS search.)



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

(a) (b)

Fig. 13: Topological ordering example.

Longest Path in a DAG: Here is a short exercise to test your understanding of DFS. Suppose that you are given a DAG $G = (V, E)$, where each vertex $u \in V$ is to be thought of as a task that takes $\mathsf{time}[u]$ time units to perform. Each edge (u, v) of the DAG represent precedence constraints, meaning that u must be completed before v is started. The question is, assuming the maximum degree of parallelism is allowed, what is the minimum amount of time needed to complete all the tasks? This is equivalent to computing the maximum cost of any path in the DAG, where *cost* is defined to be the sum of values $\mathsf{time}[u]$ of the vertices along the path.

We can solve this in $O(n+m)$ time through DFS. The trick is to associate each vertex u of the DAG with the maximum cost any path that starts at this vertex, which we denote by $\mathsf{cost}[u]$. When we first encounter a vertex u in the DFS visit procedure, which we rename LongPathVisit, we initialize $\mathsf{cost}[u] = 0$. For each adjacent vertex v, we invoke $\mathsf{LongPathVisit}(v)$ if v has not yet been discovered. We let max_cost to be the maximum cost of all u's neighbors and we set $\mathsf{cost}[u] = \mathsf{max\_cost} + \mathsf{time}[u]$. As in standard DFS, the main program invokes LongPathVisit(u) for all undiscovered vertices u. LongPathVisit(u) is given in the code-block below.

Longest Path via DFS

```
LongPathVisit(u) {                                  // start a search at u
   mark[u] = discovered                             // mark u visited
   max_cost = 0                                     // initialize max outgoing cost
   for each (v in Adj(u)) {
       if (mark[v] == undiscovered) LongPathVisit(v) // process v if undiscovered
       max_cost = max(max_cost, cost[v])            // update maximum cost
   }
   cost[u] = max_cost + time[u]                     // save final cost
}
```

Because the graph is acyclic, every edge (u, v) goes from u to a vertex v whose finish time is greater than u's. Therefore, $\mathsf{cost}[v]$ is fully defined before it is accessed by u. The longest path in the entire DAG is the largest value of $\mathsf{cost}[u]$ among all vertices u.

Strong Components: A digraph $G = (V, E)$ is said to be *strongly connected* if for every vertex u and v there is a path from u to v and from v to u. It is easy that this *mutual reachabilitys* relation between vertices is an equivalence relation. This implies that it partitions V into equivalence class, called the *strong components* (or *strongly-connected components*) of G (see Fig. 14(a) and (b)).

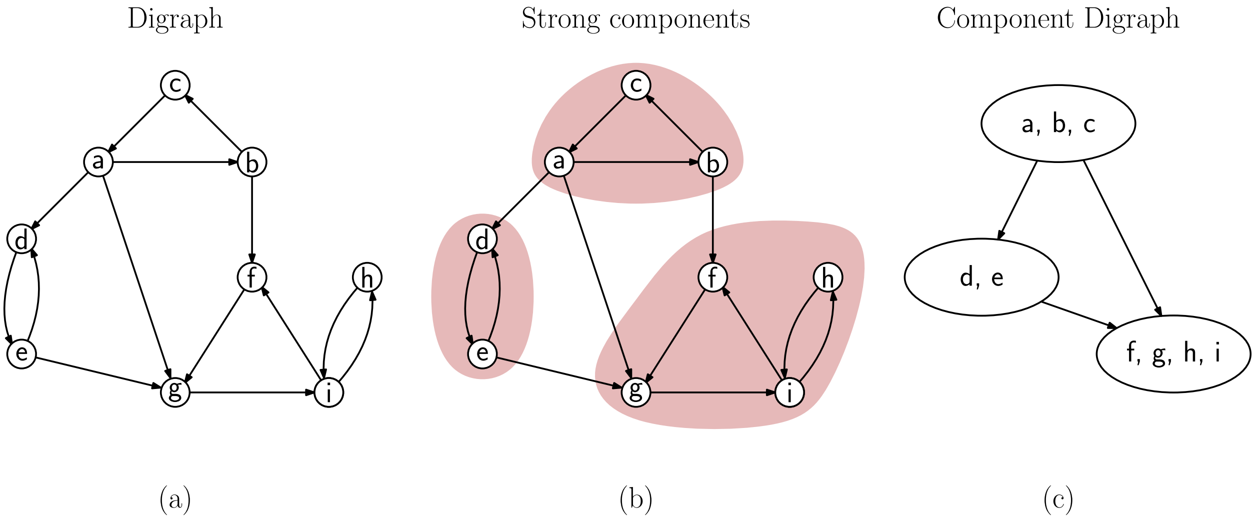


Fig. 14: Strong components of a digraph and component digraph.

If the vertices within each strong component are collapsed into a single vertex, the resulting digraph is called the *component digraph* (see Fig. 14(c)). You should consider for a moment what notable property does the component digraph possess?[6]

There exists an $O(n+m)$-time DFS algorithm for computing strong components. It is based on the following lemma.

Lemma: Let $G = (V, E)$ be a digraph and let $G' = (V', E')$ be its component digraph. Let $\langle u'_1, \ldots, u'_k \rangle$ be an enumeration of the vertices of G' in *reverse* topological order. For each $u'_i \in V'$, let $u_i \in V$ be any node in the corresponding strong component of G. If we apply a DFS to G by starting a DFS at each of the vertices $\langle u_1, \ldots, u_k \rangle$ then the subtrees of the DFS forest are the strong components of G (see Fig. 15).

Unfortunately, this lemma does not provide us with an implementable algorithm because it is hopelessly circular. (In order to compute the component graph we need to know the strong components, but that is our final objective!) Nonetheless, there is a breathtakingly clever algorithm that effectively achieves this. We will not discuss it, but it is presented in CLRS.

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

[6] Answer: It is acyclic, that is, a DAG. We will not prove it, but it is not hard to see that if two vertices of the component digraph were mutually reachable, then their associated strong components could be merged into an even larger strong component in the original digraph.

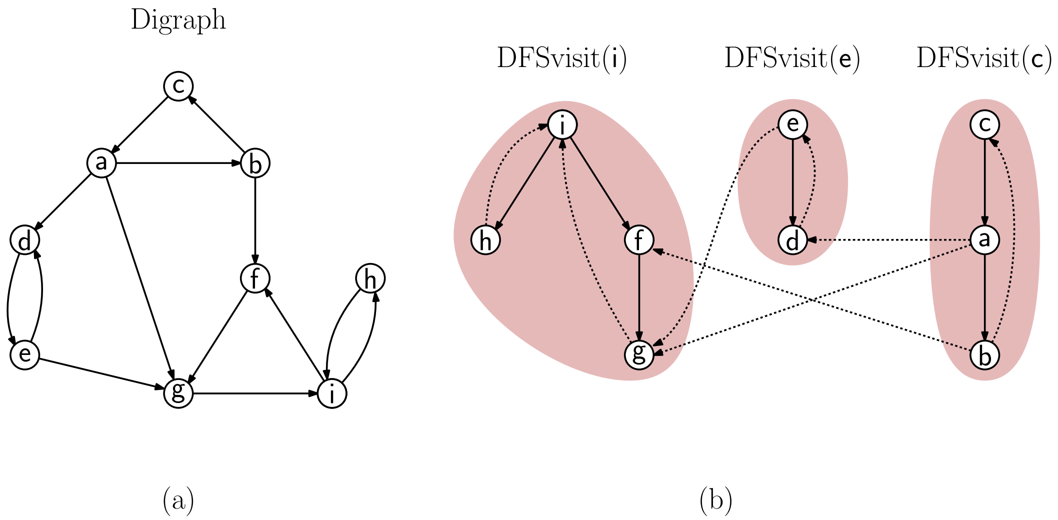


Fig. 15: Strong components and DFS.

Cut Vertex: (also called an *articulation point*) any vertex whose removal (together with the removal of any incident edges) results in a disconnected graph (see Fig. 16(a)).

Bridge: an edge whose removal results in a disconnected graph (see Fig. 16(a)).

Biconnected: A graph is *biconnected* if it contains no cut vertices (see Fig. 16(b)). In general a graph is *k-connected* if the removal of any $k-1$ vertices results in a connected graph.

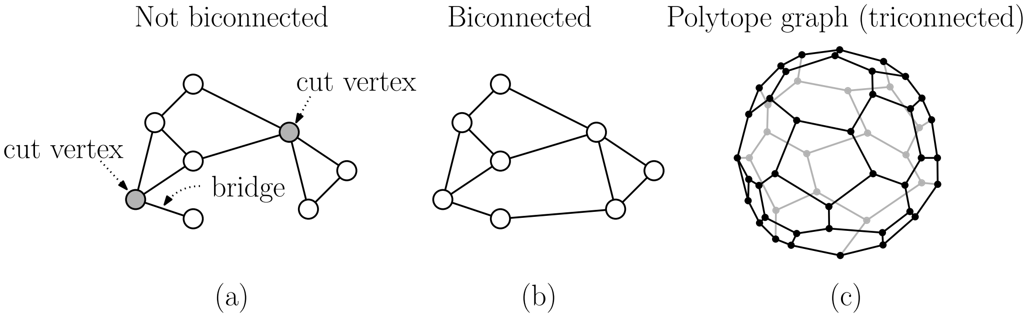


Fig. 16: Biconnected graphs, cut Vertices and Bridges

Biconnected graphs and cut vertices are of great interest in the design of network algorithms, because these are the "critical" points, whose failure will result in the network becoming disconnected. Graphs of various degrees of connectivity are of interest in many other areas. For example, it can be proved that the edge graph of any convex polytope in 3-dimensional space (e.g., a cube, tetrahedron, octahedron) is 3-connected (see Fig. 16(c)).

Intuitively, if a graph is not biconnected, it is useful to break it up into maximal subgraphs that are biconnected. A natural way in which to do this is based on partitioning the edges. Formally, we say two edges e_1 and e_2 are *cocyclic* if either $e_1 = e_2$ or if there is a simple cycle that contains both edges. It is not hard to verify that this defines an equivalence relation on the edges of a graph. Notice that if two edges are cocyclic, then there are essentially two different ways of getting from one edge to the other (by going around the cycle each way). The equivalence classes of this cocyclicity defines a partition of the edge set of the graph. These are called the *biconnected components* of the graph (see Fig. 17). Observe that a graph is biconnected if and only if it consists of a single biconnected component.

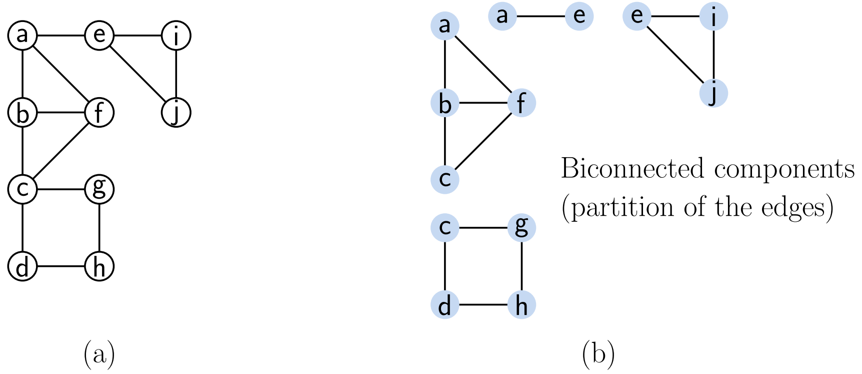


Fig. 17: (a) A graph and (b) its biconnected components.

We will give an $O(n + m)$-time algorithm for computing cut vertices. The identification of biconnected components and bridges are both straightforward modifications of this algorithm.

Cut Vertices and DFS: An obvious algorithm for computing cut vertices would be to delete each vertex one-by-one, and then apply DFS to determine whether the resulting graph remains connected. However, this would take $O(n(n + m))$ time. We will see that we can determine all the cut vertices with a single application of DFS in $O(n + m)$ time.

We assume that G is connected, which implies that there is a single DFS tree. Recall that the DFS tree consists of two types of edges: *tree edges*, which connect a parent with its child in the DFS tree, and *back edges*, which connect a (non-parent) ancestor with a (non-child) descendant.

For now, let us consider the typical case of a vertex u, where u is not a leaf, and u is not the root. Let $v_1, v_2, \ldots, v_k$ be the children of u. For each child v_i there is a subtree of the DFS tree rooted at this child. If there is no back edge going from this subtree to a proper ancestor of u, then if we were to remove u, this subtree would become disconnected from the rest of the graph, and hence u would be an cut vertex (see Fig. 18(a)). On the other hand, if every one of the subtrees rooted at the children of u have back edges to proper ancestors of u, then if u is removed, the graph remains connected because the back edges hold everything together (see Fig. 18(b)). This yields the following lemma.

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

Please check this condition carefully to see that you understand it. In particular, notice that the condition for whether u is an cut vertex depends on a test applied to its children. (This is

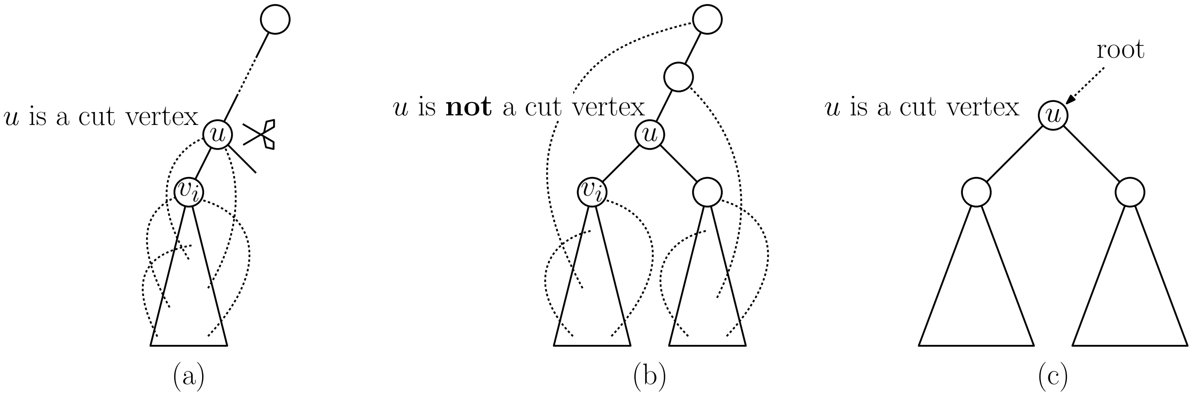


Fig. 18: Conditions for a vertex to be a cut vertex.

the most common source of confusion for this algorithm.) What about the leaves and root? First, let's consider whether leaf u can be cut vertex? The answer is no. The reason is that when you delete a leaf from the DFS tree, the rest of the tree remains connected, thus even ignoring the back edges, the graph is connected after the deletion of a leaf from the DFS tree. This yields the next

Lemma 2: A leaf of the DFS tree is not a cut vertex.

Finally, what about the root? Clearly we cannot apply Lemma 1, since the root has no ancestor. Instead observe that, because are no cross edges between the subtrees of the root, if the root has two or more children then it is an cut vertex (since its removal separates these two subtrees). On the other hand, if the root has only a single child, then (as in the case of leaves) its removal does not disconnect the DFS tree. Therefore, we have the following.

Lemma 3: The root of the DFS is an cut vertex if and only if it has two or more children (see Fig. 18(c)).

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

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

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

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

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

The term "descendant" is used in the nonstrict sense, that is, v may be equal to u (see Fig. 19 right).

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

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

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

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

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

Observe that once Low[u] is computed for all vertices u, we can test whether a given non-root vertex u is an cut vertex by Lemma 1 as follows: u is an cut vertex if and only if it has a child v in the DFS tree for which Low[v] $\geq d[u]$ (see Fig. 19(a)). To see why, observe that if there were a back edge from either v or one of its descendants to an ancestor of v then we would have Low[v] $< d[u]$ (see Fig. 19(b)).

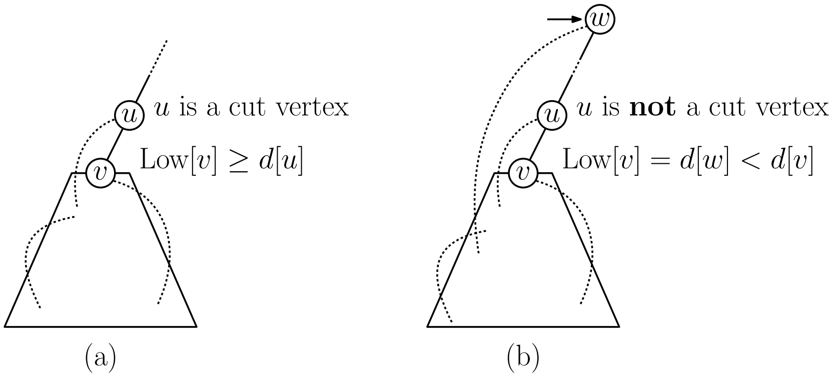


Fig. 19: Cut Vertices and the definition of Low[u].

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

The complete algorithm for computing cut vertices is given below. The main procedure for DFS is the same as before, except that it calls the following routine rather than `DFSvisit()`. An example is shown in Fig. 20. As with all DFS-based algorithms, the running time is $\Theta(n + m)$.

Cut Vertices

```
findCutVertices(u) {
    mark[u] = discovered
    Low[u] = d[u] = ++time                      // set discovery time and init Low
    for each (v in Adj(u)) {
        if (mark[v] == undiscovered) {          // (u,v) is a tree edge
            pred[v] = u                         // v's parent is u
            findCutVertices(v)
            Low[u] = min(Low[u], Low[v])        // update Low[u]
            if (pred[u] == null) {              // u is root: apply Lemma 3
                if (this is u's second child)
                    label u as a cut vertex
            }
            else if (Low[v] >= d[u]) {          // internal: apply Lemma 1
                label u as a cut vertex
            }
        }
        else if (v != pred[u]) {                // (u,v) is a back edge
            Low[u] = min(Low[u], d[v])          // update Low[u]
        }
    }
}
```

There are some interesting problems that we still have not discussed. We did not discuss how to compute the bridges of a graph. This can be done by a small modification of the algorithm above. We'll leave it as an exercise. (It is tempting to conjecture that the edge (u, v) is a bridge if and only if u and v are both cut vertices. However, as seen in Fig. 17(a) this is not always true.) Another question is how to determine which edges are in the biconnected components. A hint here is to store the edges in a stack as you go through the DFS search. When you come to an cut vertex, you can show that all the edges in the biconnected component will be consecutive in the stack.

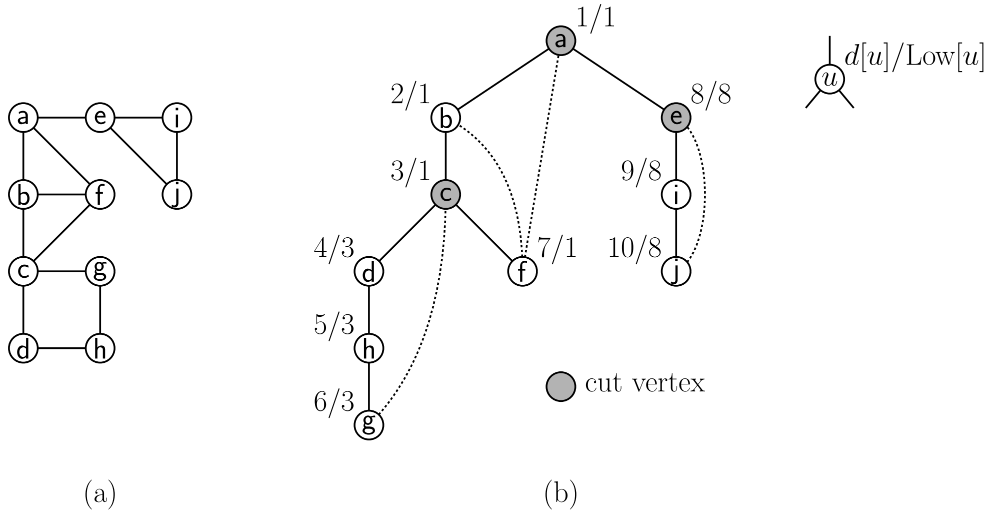


Fig. 20: Computing cut vertices via DFS.

Lecture 7: 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. (Of course, we would also need to consider the cost of transmitting the code book itself, but typically the code book is much smaller than the text being transmitted.) 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 critical. Observe that if two codewords did share a common prefix, e.g. a → 001 and b → 00101, then when we see 00101 . . . 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: Mapping of codewords to characters so that no codeword is a prefix of another.

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. 21.

Character	a	b	c	d
Codeword	0	110	10	111

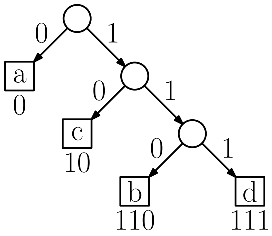


Fig. 21: A tree-representation of a prefix code.

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, Huffman coding was used for many years by the Unix utility pack for file compression. Later it was discovered that there are better compression methods. For example, gzip is based on a more sophisticated method called the *Lempel-Ziv coding* (in the form of an algorithm called *LZ77*), and bzip2 is based on combining the *Burrows-Wheeler transformation* (an extremely cool invention!) with run-length encoding, and Huffman 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. 22.

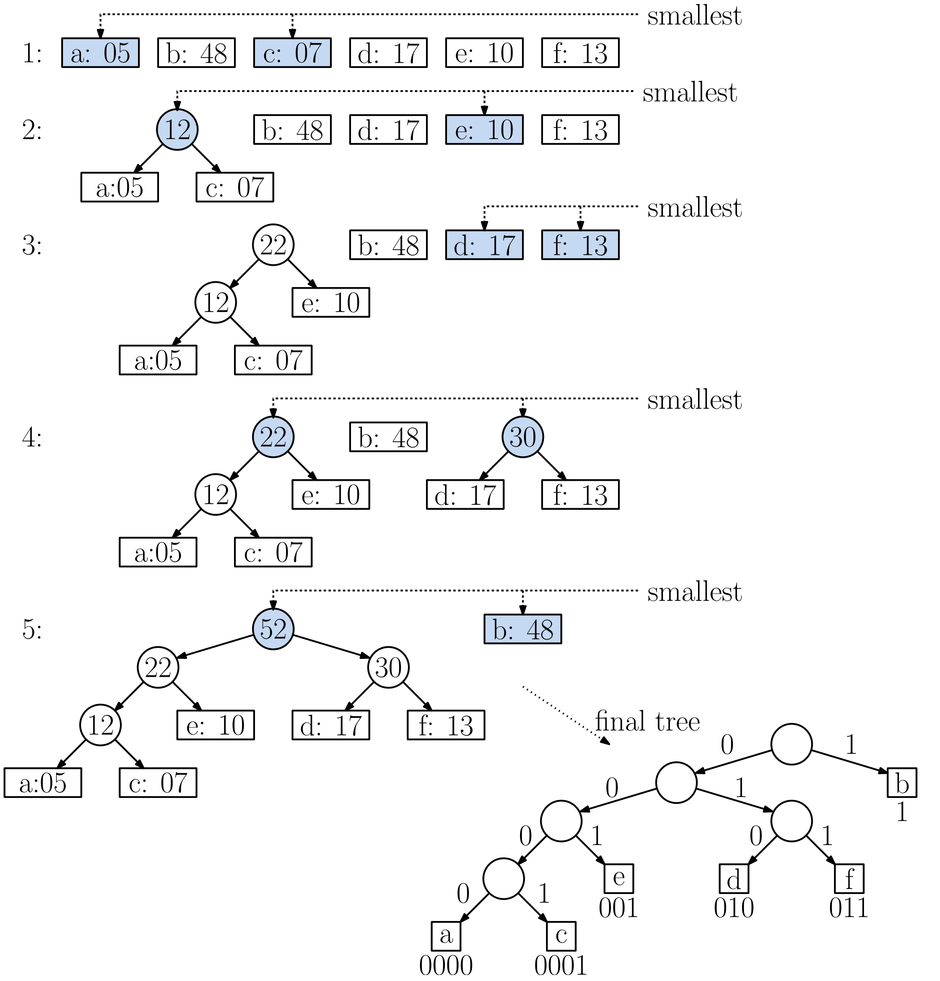


Fig. 22: Huffman's Algorithm.

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 prob$[x]$. 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(char C[], float prob[]) {
    for each (x in C) {
        add x to Q sorted by prob[x]      // add all to priority queue
    }
    n = size of C
    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 min 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. As with many other greedy algorithms we have seen, the key is showing that the greedy choice is always the proper one to make (or at least it is a good as any other choice). In many previous cases (scheduling, for example), the solution involved a sequential list of choices, and so we can convert an arbitrary solution to the greedy solution by *swapping elements*. However, Huffman's algorithm produces a tree. The question is then is how to structure an induction proof? (Note that we could try swapping leaf nodes, and while this will change the placement of the leaves, it cannot alter the tree's overall structure.) Hence, we need to find a new technique on which to base our induction proof.

Our approach is based a few observations. 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 we could replace this node with its child without increasing the tree's cost.) So we may safely limit consideration to full binary trees. Our next observation is that in any optimal code tree, the two characters with the lowest probabilities will be siblings at the maximum depth in the tree. Once we have this fact, we will merge these two characters into a single *super character* whose probability is the sum of their individual probabilities. As a result, we will now have one less character in our alphabet. This will allow us to apply induction to the remaining $n - 1$ characters.

Let's first prove the above assertion that the two characters of lowest probability may be assumed to be siblings at the lowest level of the tree.

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

Proof: Let T be any optimal prefix code tree, and let b and c be two siblings at the maximum depth of the tree. Assume without loss of generality that $p(b) \le p(c)$ and $p(x) \le p(y)$ (if this is not true, then swap b with c and/or swap x with y). Now, since x and y have the two smallest probabilities it follows that $p(x) \le p(b)$ and $p(y) \le p(c)$. (In both cases they may be equal.) Because b and c are at the deepest level of the tree we know that $d_T(b) \ge d_T(x)$ and $d_T(c) \ge d_T(y)$. (Again, they may be equal.) Thus, we have $p(b) - p(x) \ge 0$ and $d_T(b) - d_T(x) \ge 0$, and hence their product is nonnegative. Now switch the positions of x and b in the tree, resulting in a new tree T'(see Fig. 23).

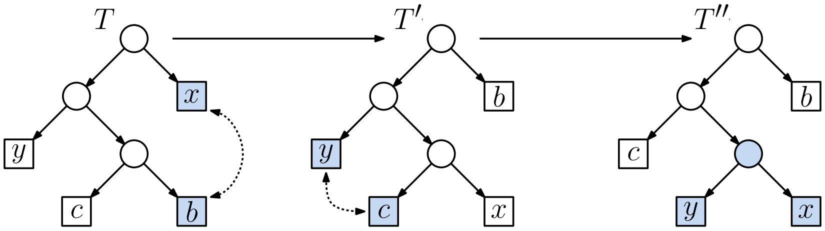


Fig. 23: Showing that the lowest probability nodes are siblings at the tree's lowest level.

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

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

Thus the cost does not increase. (Given our assumption that T was already optimal, it certainly cannot decrease either, since otherwise we would have a contradiction.) Since T was an optimal tree, T' is also an optimal tree.

By a similar argument, we can switch y with c to obtain a new tree T''. Again, the same sort of argument implies that T'' is also optimal. The final tree T'' satisfies the statement of the claim.

Although we have presented Huffman's algorithm in sequential form, we can think of it as a recursive algorithm. First merge the two lowest probability characters x and y into a single "super character" z whose probability is the sum of their individual probabilities, and then recursively apply the same algorithm to the $n-1$ element character set with x and y replaced by z. In order to complete the proof of the correctness of Huffman's algorithm, we will show that this recursive view of Huffman's algorithm is optimal.

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. Let us assume inductively that when given $n-1$ characters, Huffman's algorithm produces an optimal tree. We want to show this is true when the alphabet has exactly n characters.

Suppose we have exactly n characters. Let x and y be the two characters with the lowest probabilities. By the previous claim, we may assume that x and y are siblings at the lowest level of the optimal tree. Let us remove x and y from the alphabet, and replace them with a new "super character" z whose probability is $p(z) = p(x) + p(y)$. Thus, we now have $n-1$ characters in our alphabet.

Consider any prefix code tree T made with this new set of $n-1$ characters. We know that z will appear as a leaf node somewhere in this tree. Let us convert T into a prefix code tree for the original set of characters by replacing the leaf node for z with an internal node whose left child is x and whose right child is y (see Fig. 24).

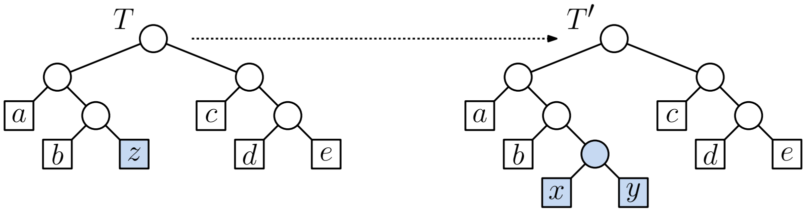


Fig. 24: Proving the correctness of Huffman's algorithm.

Let T' denote the resulting tree. Because z is removed and x and y have now been added at depth $d(z) + 1$, the new cost is

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

Observe that the change in cost does not depend on the structure of the tree T. Therefore, in order to minimize $B(T')$, we should build T to minimize $B(T)$. By the induction hypothesis (and the fact that T involves $n-1$ characters), we may assume that T results from running Huffman's algorithm. Thus the final tree is optimal. However, as observed earlier, this is exactly the same tree that is produced by (the recursive view of) Huffman's algorithm.

Lecture 8: 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 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 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. 27.) We should also remember that the shortest path to v passes through u, which we do by setting pred$[v]$ to u (see the code block below).

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

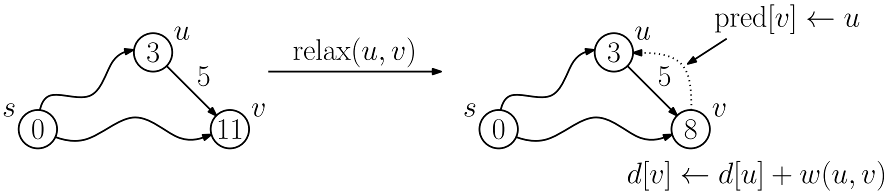


Fig. 25: Relaxation.

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. Assuming that the edge weights are nonnegative, Dijkstra's algorithm achieves this objective.[7]

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

[7] Note, by the way that while this objective is optimal in the worst case, there are instances where you might hope for much better performance. For example, given a cartographic road map of the entire United States, computing the shortest path between two locations near Washington DC should not require relaxing every edge of this road map. A better approach to this problem is provided by another greedy algorithm, called A^*-search.

$V \setminus S$ to add to S. (If you haven't seen it before, the notation "$A \setminus B$" means the set A excluding 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. 25 for an example.

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

```
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]
}
```

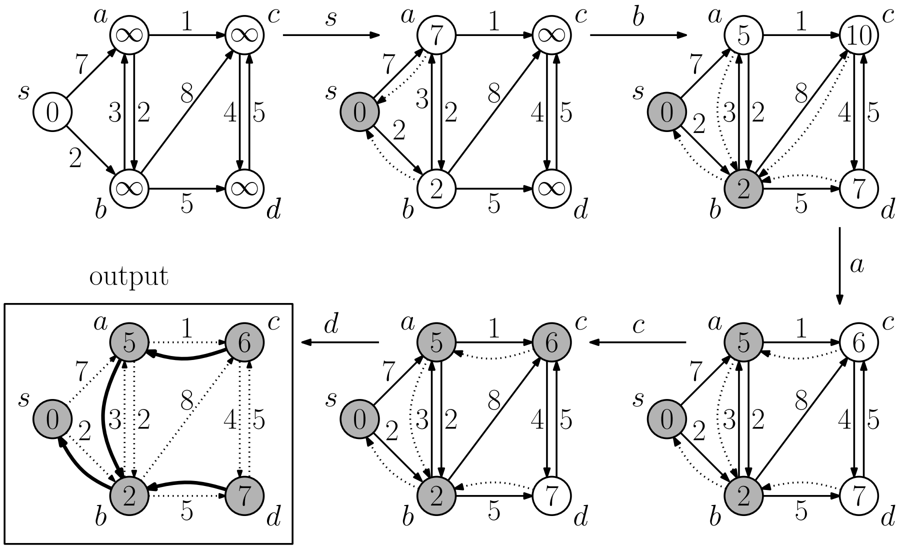


Fig. 26: Dijkstra's Algorithm example.

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

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

Proof: Suppose to the contrary that at some point Dijkstra's algorithm *first* attempts to add a vertex u to S for which $d[u] \neq \delta(s,u)$. By our observations about relaxation, $d[u]$ is never less than $\delta(s,u)$, thus we have $d[u] > \delta(s,u)$. Consider the situation just prior to the insertion of u, and consider the true shortest path from s to u. Because $s \in S$ and $u \in V \setminus S$, 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. 27). (Note that it may be that $x = s$ and/or $y = u$).

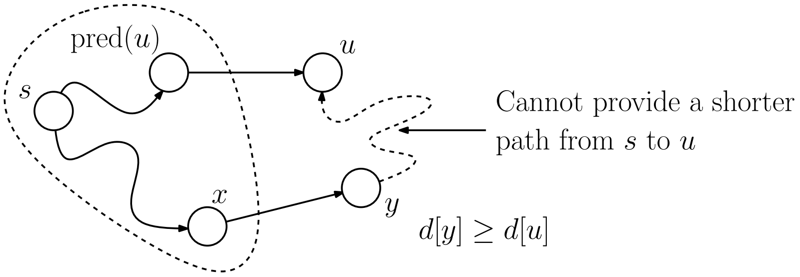


Fig. 27: Correctness of Dijkstra's Algorithm.

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.

Variants: Dijkstra's algorithm is robust to a number of variations. Here are some variants of the single-source shortest path problem that can be solved either by making a small modification to Dijkstra's algorithm and/or by modifying the underlying graph. (I'll leave their solutions as an exercise.)

Vertex weights: There is a cost associated with each vertex. The overall cost is the sum of vertex and/or edge weights on the path.

Single-Sink Shortest Path: Find the shortest path from each vertex to a sink vertex t.

Multi-Source/Multi-Sink: You are given a collection of source vertices $\{s_1, \ldots, s_k\}$. For each vertex find the shortest path from its nearest source. (Analogous for multi-sink.)

Multiplicative Cost: Define the cost of a path to be the product of the edge weights (rather than the sum.) If all the edge weights are at least 1, find the single-source shortest path.

Lecture 9: Greedy Algorithms for Minimum Spanning Trees

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. 28 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. 28(a) is not a minimum spanning tree (in fact, it is a maximum weight spanning tree), while the other two are MSTs.

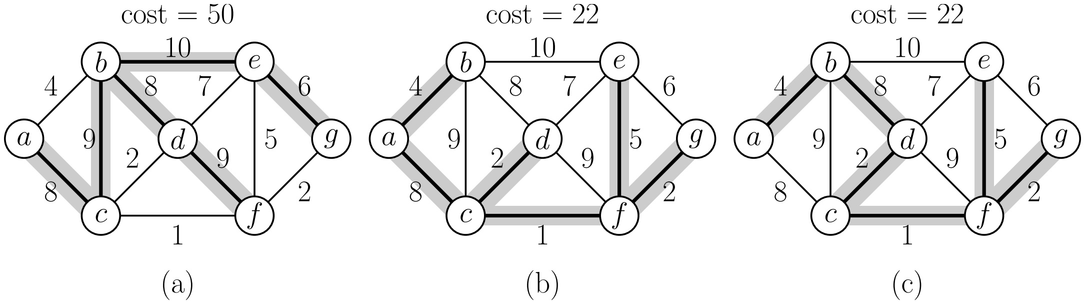


Fig. 28: Spanning trees (the middle and right are minimum spanning trees).

Generic approach: We will present three *greedy* algorithms (Kruskal's, Prim's, and Boruvka's) 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 say that an edge $(u, v) \in E \setminus A$ is *safe* if $A \cup \{(u, v)\}$ is viable. (Recall that $E \setminus A$ means the edges of E that are *not* in A.) 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? WLet S be a subset of the vertices $S \subseteq V$. Here are a few useful definitions (see Fig. 29):

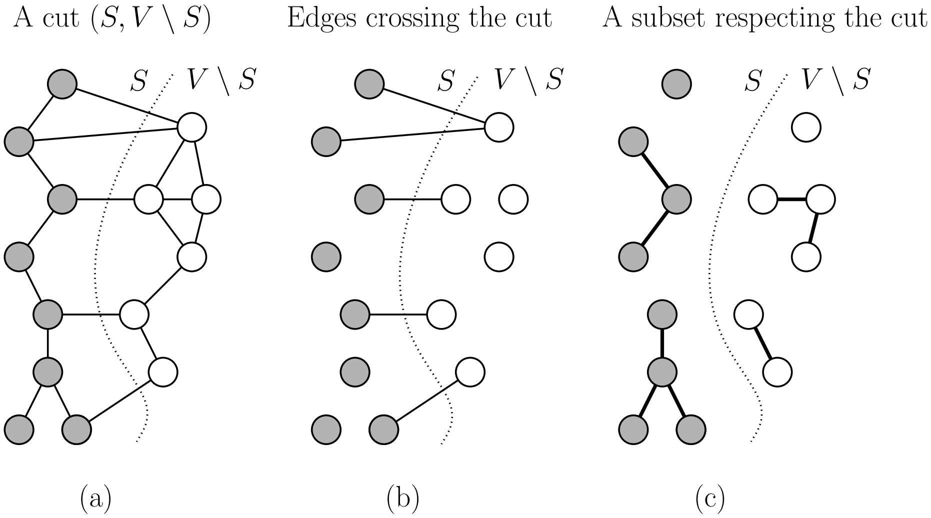


Fig. 29: MST-related terminology.

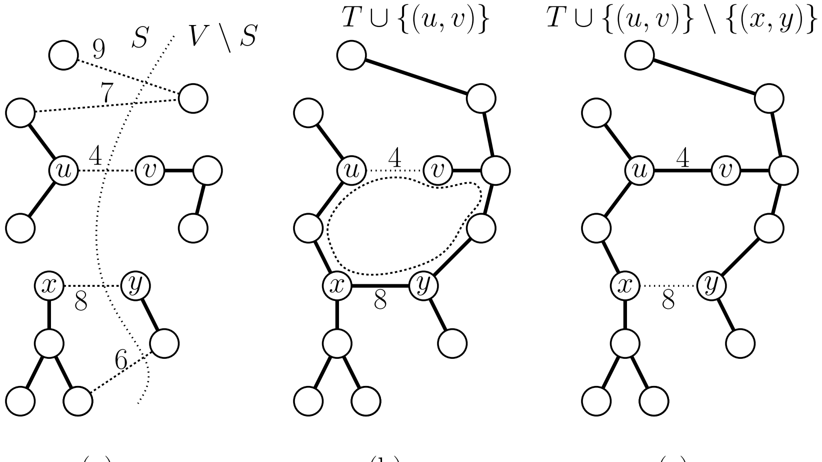
- A *cut* $(S, V \setminus S)$ is a partition of the vertices into two disjoint subsets (see Fig. 29(a)).
- An edge (u, v) *crosses* the cut if $u \in S$ and $v \notin S$ (see Fig. 29(b)).
- Given a subset of edges A, we say that a cut *respects* A if no edge in A crosses the cut (see Fig. 29(c)).

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. 30). If T contains (u, v) then we are done. Suppose that no MST contains (u, v). We will derive a contradiction.



$T \cup \{(u, v)\}$ $\quad$ $T \cup \{(u, v)\} \setminus \{(x, y)\}$

(a) (b) (c)

Fig. 30: Proof of the MST Lemma. Edge (u, v) is the light edge crossing cut $(S, V \setminus S)$.

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

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

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

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

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

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

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

find(u)**:** Find the set that contains a given item u.

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

Theorem: Given a collection of n elements, each initially in its own set, the union-find data structure can perform any sequence of up to n union and find operations in total $O(n \cdot \alpha(n))$ time, where $\alpha(n)$ is the (*extremely slow growing*) inverse Ackerman's function.

You are not responsible for knowing how this data structure works, since we will use it as a "black-box". In Kruskal's algorithm, the vertices of the graph will be the elements to be stored in the sets, and the sets will be vertices in each tree of A. The set A can be stored as a simple list of edges. The algorithm is shown in the code fragment below, and an example is shown in Fig. 31.

Kruskal's Algorithm

```
KruskalMST(G=(V,E), w) {
    A = {}                              // initially A is empty
    Place each vertex u in a set by itself
    Sort E in increasing order by weight w
    for each ((u, v) in this order) {
        if (find(u) != find(v)) {       // u and v in different trees
            add (u, v) to A             // join subtrees together
            union(u, v)                 // merge these two components
        }
    }
    return A
}
```

Analysis: How long does Kruskal's algorithm take? As usual, let n be the number of vertices and m be the number of edges. Since the graph is connected, we may assume that $m \geq n - 1$. Observe that it takes $\Theta(m \log m)$ time to sort the edges. The for-loop is iterated m times, and each iteration involves a constant number of accesses to the Union-Find data structure

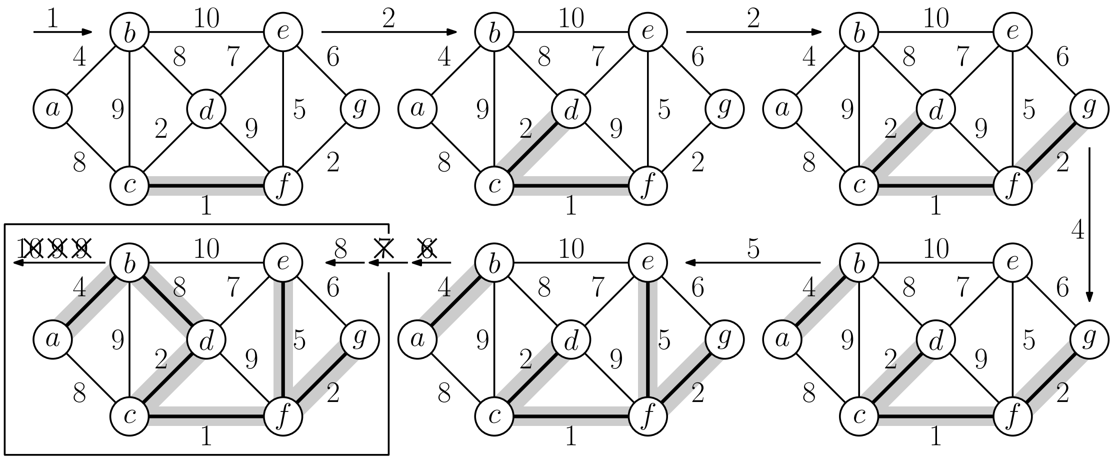


Fig. 31: Kruskal's Algorithm. Each vertex is labeled according to the set that contains it.

on a collection of n items. Thus each access is $\Theta(n)$ time, for a total of $\Theta(m \log n)$. Thus the total running time is the sum of these, which is $\Theta((n+m)\log n)$. Since n is asymptotically no larger than m, we could write this more simply as $\Theta(m \log n)$.

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

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

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

Observe that if we consider the set of vertices S currently part of the tree, and its complement $(V \setminus S)$, 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

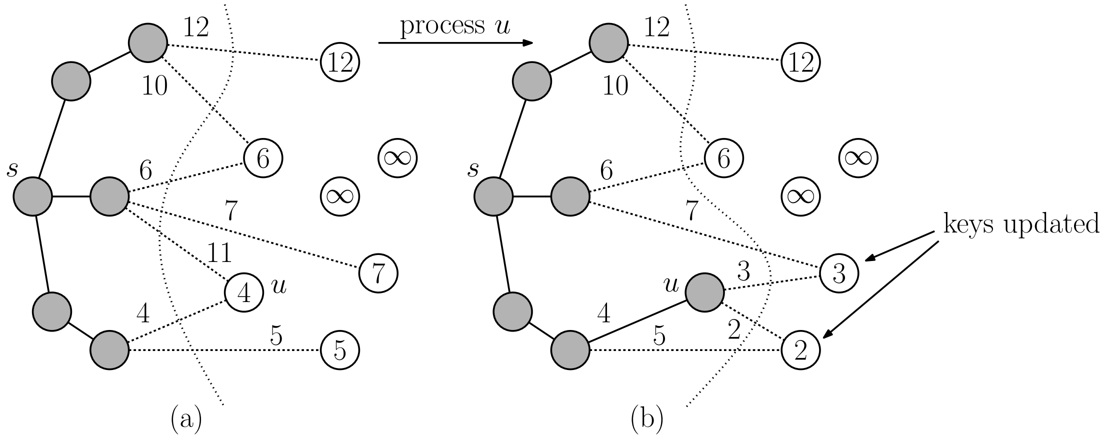


Fig. 32: Prim's Algorithm.

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

extract-min()**:** Extract the item with the minimum key value in Q.

decrease-key(u, k')**:** Decrease the value of u's key value to k'.

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.

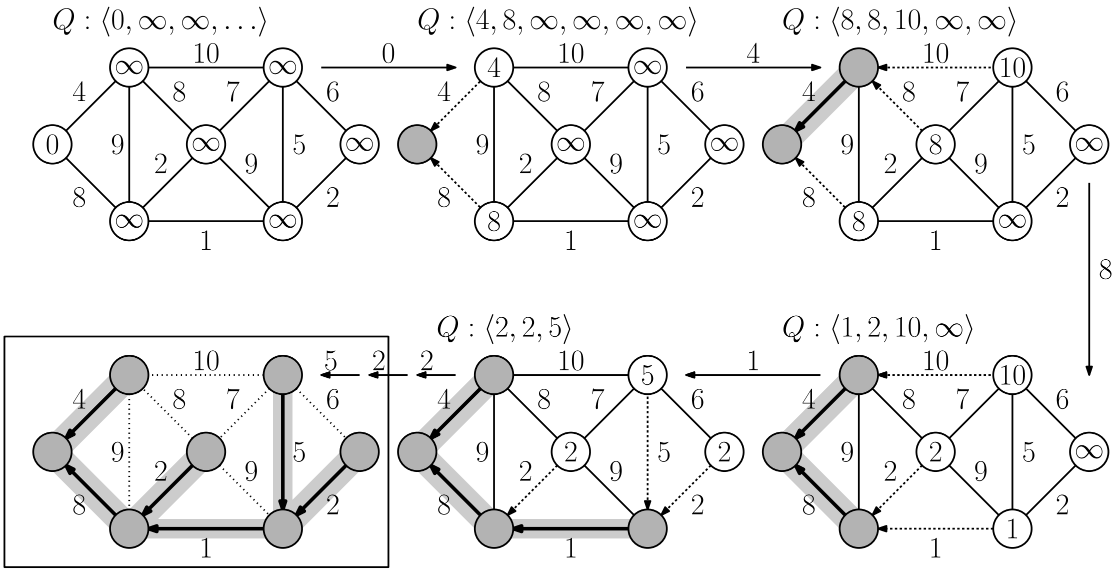
There is a much more elegant solution, and this is what makes Prim's algorithm so nice. For each vertex in $u \in V \setminus S$ (not part of the current spanning tree) we associate u with a key value *key*$[u]$, which is the weight of the lightest edge going from u to any vertex in S. We also store in *pred*$[u]$ the end vertex of this edge in S. If there is not edge from u to a vertex in $V \setminus S$, 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 s 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 $O(\log n)$ to extract this vertex from the queue. For each incident edge, we spend potentially $O(\log n)$ time decreasing the key of the neighboring vertex.

```
PrimMST(G=(V,E), w, s) {
    for each (u in V) {                     // initialization
        key[u] = +infinity
        color[u] = undiscovered
    }
    key[s] = 0                              // start at root
    pred[s] = null
    add all vertices to priority queue Q
    while (Q is nonEmpty) {                 // until all vertices in MST
        u = extract-min from Q              // vertex with lightest edge
        for each (v in Adj[u]) {
            if ((color[v] == undiscovered) && (w(u,v) < key[v])) {
                key[v] = w(u,v)             // new lighter edge out of v
                decrease key value of v to key[v]
                pred[v] = u
            }
        }
        color[u] = finished
    }
    [The pred pointers define the MST as an inverted tree rooted at s]
}
```



$Q : \langle 0, \infty, \infty, \ldots \rangle$
$Q : \langle 4, 8, \infty, \infty, \infty, \infty \rangle$
$Q : \langle 8, 8, 10, \infty, \infty \rangle$
$Q : \langle 2, 2, 5 \rangle$
$Q : \langle 1, 2, 10, \infty \rangle$
0
4
8
1
2
5
2
2
10
4
8
7
6
9
2
9
5
8
2
1
∞
10

Fig. 33: Prim's algorithm example.

Thus the time is $O(\log n + \text{degree}(u) \log n)$ time. The other steps of the update are constant time. So the overall running time is

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

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

Boruvka's Algorithm: Given that we have seen two algorithms (Kruskal's and Prim's) for solving the MST problem, it may seem like complete overkill to consider yet another algorithm. This one is called Boruvka's algorithm. It is actually the oldest of the three algorithms (invented in 1926 by the Czech mathematician Otakar Bo̊ruvka, well before the first digital 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, Boruvka's algorithm adds a whole set of edges all at once to the MST.

Boruvka'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 (globally) lightest (minimum 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 lightest edge exiting *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. If there are edges of equal weight, the algorithm might attempt to add two such edges between the same two components, which would result in the generation of a cycle. (Each edge individually is safe, but adding both simultaneously generates a cycle.) We make the assumption that edge weights are distinct (or at least, there is some uniform rule for breaking ties so the above problem cannot arise).

Note that in a single step of Boruvka's algorithm many components can 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 Boruvka's algorithm is given below.

There are a number of unspecified details in Boruvka's algorithm, which we will not spell out in detail, except to note that they can be solved in $\Theta(n + m)$ 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

```
BoruvkaMST(G=(V,E), w) {
    initialize each vertex to be its own component
    A = {}                                // A holds edges of the MST
    while (there are two or more components) {
        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 (V, A), to compute the new components
    }
    return A                              // return final MST edges
```

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.

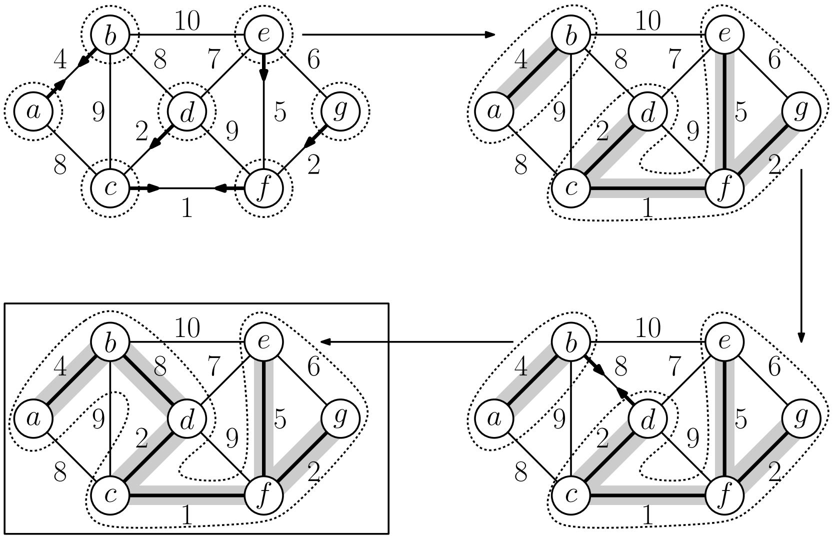


Fig. 34: Boruvka's Algorithm.

Analysis: How long does Boruvka's algorithm take? Observe that because each iteration involves doing a DFS, each iteration (of the outer do-while loop) can be performed in $\Theta(n + m)$ 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 n components, this can happen at most $\lg n$ time, until only one component remains. Thus, the total running time is $\Theta((n+m)\log n)$ time. Again, since G is connected, n is asymptotically no larger than m, so we can write this more succinctly as $\Theta(m \log n)$. Thus all three algorithms have the same asymptotic running time.

Lecture 10: 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 $R = \{1, \ldots, n\}$ 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. 35. 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.

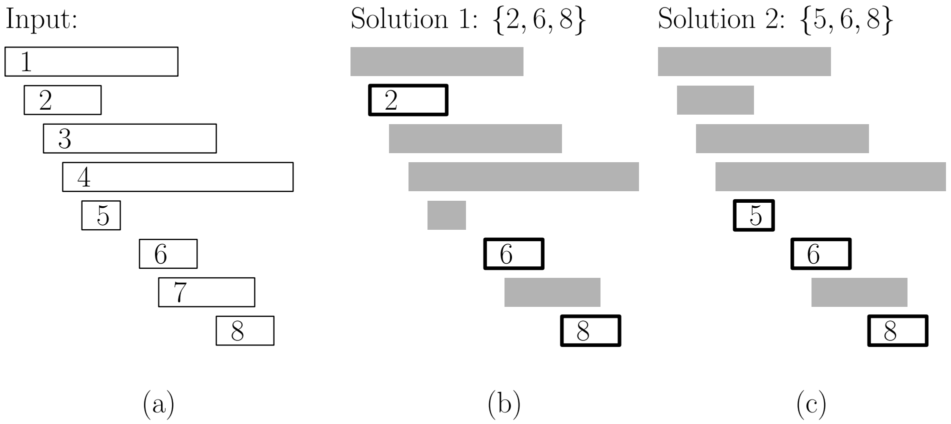


Fig. 35: An input and two possible solutions to the interval scheduling problem.

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. Try to construct a counterexample. (If you get stuck, there is an example given in our textbook.)

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.* The pseudo-code is presented in the code-block below. It returns the set S of scheduled activities.

Greedy Interval Scheduling

```
greedyIntervalSchedule(R=(s,f)) {   // schedule tasks with given start/finish times
    sort tasks by increasing order of finish times
    S = empty                       // S holds the sequence of scheduled activities
    prev_finish = -infinity         // finish time of previous task
    for (i = 1 to n) {
        if (s[i] > prev_finish) {   // task i doesn't conflict with previous?
            append task i to S      // ...add it to the schedule
            prev_finish = f[i]      // ...and update the previous finish time
        }
    }
    return S
}
```

An example is given in Fig. 36. The start-finish intervals are given in increasing order of 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.

The algorithm's correctness will be shown below. The running time is dominated by the $O(n \log n)$ time needed to sort the jobs by their finish times. After sorting, the remaining steps can be performed in $O(n)$ time.

Correctness: Let us consider the algorithm's correctness. First, observe that the output is a valid schedule in the sense that no two conflicting tasks appear in the final schedule. This is because we only add a task if its start time exceeds the previous finish time, and the previous finish time increases monotonically as the algorithm runs.

Second, we consider optimality. The proof's structure is worth noting, because it is common to many correctness proofs for greedy algorithms. It begins by considering an arbitrary solution, which may assume to be an optimal solution. If it is equal to the greedy solution, then the greedy solution is optimal. Otherwise, we consider the first instance where these two solutions differ. We replace the alternate choice with the greedy choice and show that things can only get better. Thus, by applying this argument inductively, it follows that the greedy solution is as good as an optimal solution, thus it is optimal.

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

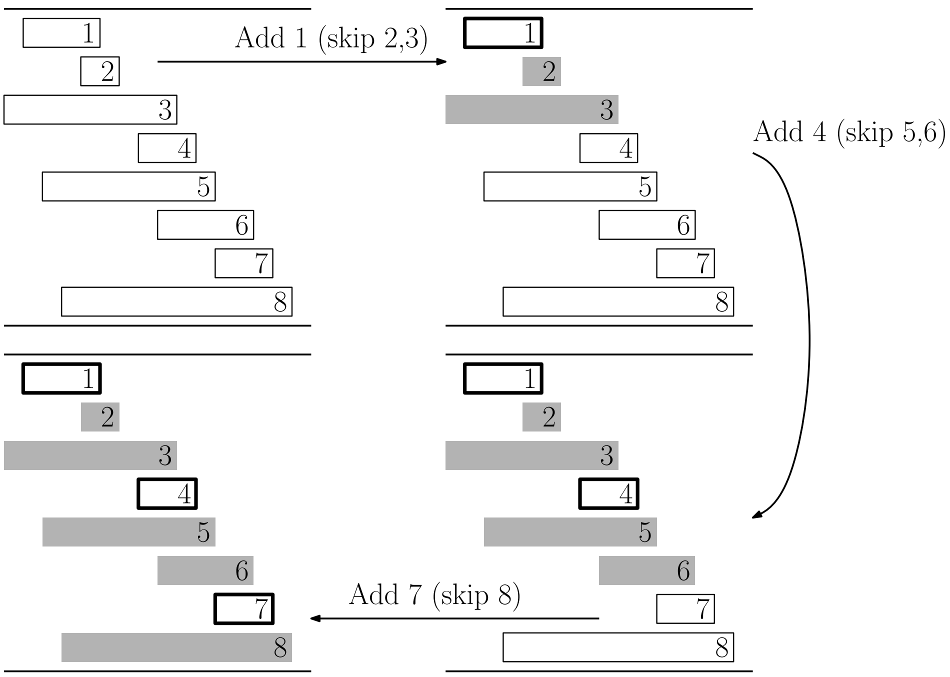


Fig. 36: An example of the greedy algorithm for interval scheduling. The final schedule is $\{1, 4, 7\}$.

Proof: Consider an optimal schedule O, and let G be the greedy schedule. Let $O = \langle x_1, \ldots, x_k \rangle$ be the activities of O listed in increasing order of finish time, and let $G = \langle g_1, \ldots, g_{k'} \rangle$ be the corresponding greedy schedule. If $G = O$, then G is optimal, and we are done. Otherwise, observe that since O is optimal, it must contain at least as many activities as the greedy schedule, and hence there is a first index 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 no later than x_j finishes.

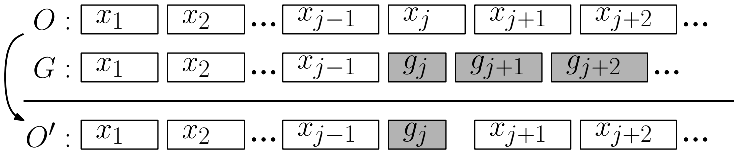


Fig. 37: Proof of optimality for the greedy schedule.

Consider the modified "greedier" schedule O' that results by replacing x_j with g_j in the schedule O (see Fig. 37). That is, $O' = \langle x_1, \ldots, x_{j-1}, g_j, x_{j+1}, \ldots, x_k \rangle$. Clearly, O' is a valid schedule, because g_j finishes no later than x_j, and therefore it cannot create any new conflicts. This new schedule has the same number of activities as O, and so it is at least as good with respect to our optimization criterion.

By repeating this process, we will eventually convert O into G without ever decreasing the number of activities. It follows that G is 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 $R = \{1, \ldots, n\}$ 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 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.

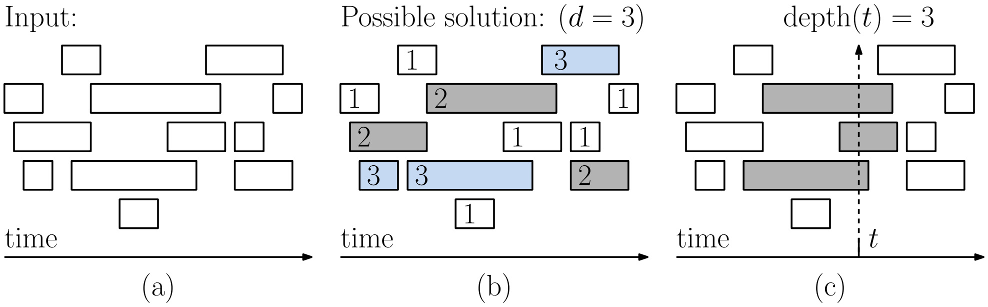


Fig. 38: Interval partitioning: (a) input, (b) possible solution, and (c) depth.

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. 38(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 (in the sense of being NP-hard). However, due to the simple nature of intervals, it is possible to solve the interval partitioning problem quite efficiently by a simple greedy approach. First, we sort the requests by increasing order of start times. We then assign each request the smallest color (possibly a new color) such that it conflicts with no other requests of this color class. The algorithm is presented in the following code block.

Greedy Interval Partitioning

```
greedyIntervalPartition(R=(s,f)) {  // schedule tasks with given start/finish times
    sort requests by increasing start times -- (x[1], ..., x[n])
    for (i = 1 to n) do {           // classify the ith request
        E = emptyset                // E stores set of excluded colors for xi
        for (j = 1 to i-1) do {
          if ([s[j],f[j]] overlaps [s[i],f[i]]) add color[x[j]] to E
        }
        Let c be the smallest color not in E
        color[x[i]] = c
    }
    return color[1...n]
}
```

(The solution given in Fig. 38(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.[8]

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.

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 $\text{depth}(t)$ to be the number of activities whose start-finish interval contains t (see Fig. 38(c)). Given an set $R = \{x_1, \ldots, x_n\}$ of activity requests, define $\text{depth}(R)$ to be the maximum depth over all possible values of t. Since the activities that contribute to $\text{depth}(t)$ conflict with one another, clearly we need at least this many resources to schedule these activities. Therefore, we have the following:

[8] 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 $O(n \log n)$.

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

This claim states that, if d denotes the minimum number of colors in any schedule, we have $d \geq \text{depth}(R)$. 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 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 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.

Scheduling to Minimize Lateness: Finally, let us discuss a problem of scheduling a set of n tasks where each task is associated with a duration time and a deadline. (Consider, for example, the assignments from your various classes and their due dates.) Here, the objective is to determine the start times for each task so that they are all completed before their associated deadline. As in interval scheduling, we assume that there is a single resource, and no more than one task can be scheduled at a given time.

More formally, let $T = \{x_1, \ldots, x_n\}$ be the tasks, where d_i denotes the *deadline* of the ith task and t_i denotes the time needed to perform this task, that is, its *duration*. 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* of task i to be the amount by which its finish time exceeds its deadline. We define the *lateness* of the entire schedule to be the maximum lateness over all tasks. 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 defined to be $\ell_i = \max(0, f_i - d_i)$.

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. 39. The input is given in Fig. 39(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. 39(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.

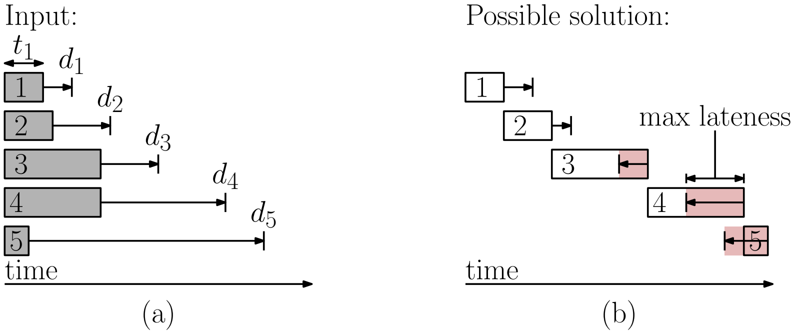


Fig. 39: Scheduling to minimize lateness: (a) input, (b) possible solution.

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,d) {        // schedule tasks with given durations and deadlines
    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  // start next task
        f = f[i] = s[i] + t[i]              // its finish time
        lateness[i] = max(0, f[i] - d[i])   // its lateness
    }
    return arrays s and lateness            // return the start times and latenesses
}
```

The solution shown in Fig. 39(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. 40. 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.

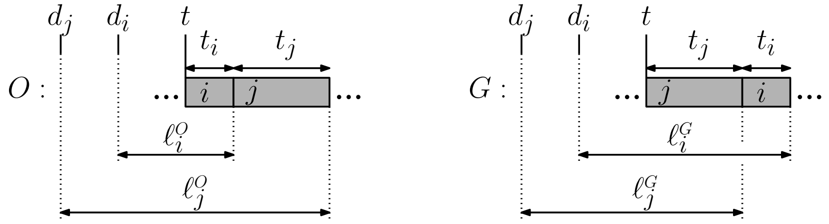


Fig. 40: Optimality of the greedy scheduling algorithm for minimizing lateness.

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. 40. 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. 40). 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 $L^G \leq L^O$. 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 $L^G \leq L^O$, it suffices to show that $\ell_i^G \leq L^O$ and $\ell_j^G \leq L^O$. 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

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

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

Claim: The greedy scheduling algorithm minimizes maximum lateness.

Lecture 11: Greedy Approximation Algorithms: The k-Center Problem

Greedy Approximation for NP-Hard Problems: As algorithm design methods go, the greedy approach is not very powerful. Nonetheless, one of the common applications of greedy algorithms is for producing approximation solutions to NP-hard problems.

As we shall see later this semester, NP-hard optimization problems represent very challenging computational problems in the sense that there is no known exact solution that has worst-case polynomial-time running time. Given an NP-hard problem, there are no ideal algorithmic solutions. One has to compromise between optimality or running time. Nonetheless, there are are number of examples of NP-hard problems where simple greedy heuristics produce solutions that are not far from optimal.

Clustering and Facility Location: Clustering is among the most widely studied problems in computer science. In a nutshell, clustering problems involve taking a very large set of objects, called *points*, and selecting a much smaller set of representative points. For example, companies like Netflix want to develop profiles of their users in order to recommend good programs and films to show them. Based on the available data about a user (demographic, geographic, language, and past preferences), they would like to classify each user as belonging to one (or generally a few) of a relatively small set of representative categories that will reflect the films that they like.

One way to approach this problem is to represent the information about each user as a long vector of numerical properties that encode this user's individual characteristics. Thus, each user is mapped to a point in a multi-dimensional space of *user characteristics*. It is possible to compute the distance between any pair of points within this space. The hope is that if two users have similar characteristics, that is, if the distance between their property vectors is small, then these users are expected to have similar interests.

Suppose that Netflix decides that it is going to partition its tens of millions of users into a small number k (perhaps tens to hundreds) of clusters. We will identify each cluster by specifying a central point of the cluster, its *cluster center*. To define the problem formally, we need some *objective function* by which to measure the quality of our solution. A simple way is based on minimizing the distance of each point to its closest cluster center. This gives rise to the k-center problem, which we define next.

The k-Center Problem: Let us suppose that we are given a set P of n points in space. For each pair of points $u, v \in P$, let $\delta(u, v)$ denote the distance between u and v. We will assume that the distance satisfies the typical properties of any natural distance function:

Positivity: $\delta(u, v) \geq 0$ and $\delta(u, v) = 0$ if and only if $u = v$.

Symmetry: $\delta(u, v) = \delta(v, u)$

Triangle inequality: $\delta(u, w) \leq \delta(u, v) + \delta(v, w)$

These properties hold both for geometric distances, like the Euclidean distance, as well graph-based distances, like the shortest path between two nodes of a weighted graph.

We are also given a number k, which is presumably much much smaller than n. In the *k-center problem* the objective is to identify a subset C of k points of P, called *cluster centers*, so as to minimize the maximum distance of every point of P to its closest point in C.

A Geometric Perspective: Given a set $C = \{c_1, \ldots, c_k\}$ of cluster centers and $c_i \in C$ we define its *neighborhood*, denoted $N(c_i)$, to be the set of points u of P such that c_i is the closest center to u, that is

$$N(c_i) \;=\; \{v \in P \mid \delta(v, c_i) \leq \delta(v, c_j), \text{ for } i \neq j\}.$$

Observe that every point of $N(c_i)$ lies within distance $\Delta(C)$ of c_i. This suggests a more visual way of interpretting the k-center problem. In particular, if the distance is the Euclidean distance and P are points in the plane, then this means that every point of $N(c_i)$ lies within a circular disk of radius $\Delta(C)$ centered at c_i. Thus, the k-center problem is equivalent to determining the smallest radius r such that all the points of P are contained within k disks of radius r, where each disk is centered at some point of P. (The radius r is equal to $\Delta(C)$ and the centers of the disks are the cluster centers. See Fig. 41.)

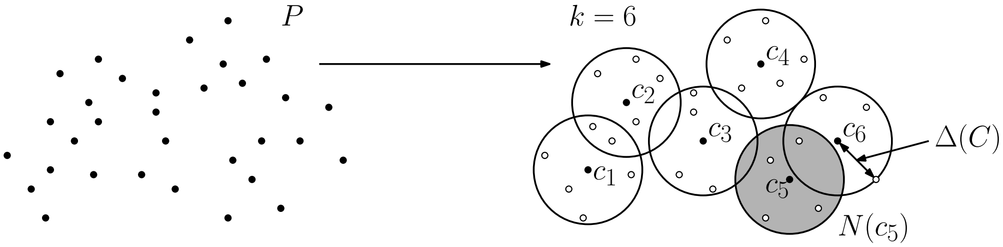


Fig. 41: The k-center problem in the Euclidean plane.

You might wonder why we constrain the cluster centers to be points of P. In fact there is no compelling reason. The version of k-center where centers must be chosen from P is sometimes referred to as the *discrete k-center problem*, and the greedy approach assumes this version of the problem. An more general form is one in which we have two sets P and Q, where the points of P are to be covered by the disks, and the cluster centers must be chosen from Q. (Thus, in our version $P = Q$.)

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 $N(c_1), \ldots, N(c_k)$ defines a *partition* of P. The *bottleneck distance* associated with each center is defined to be the distance to its farthest point in $N(c_i)$, that is,

$$\Delta(c_i) = \max_{v \in N(c_i)} \delta(v, c_i).$$

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.

Intuitively, if we think of the cluster centers as the locations of Starbucks (or your favorite retailer), then each customer (point in P) is associated with the closest Starbucks. The set $N(c_i)$ are the customers that go to the ith Starbucks location. The value $\Delta(c_i)$ is the maximum distance any of these customers needs to travel, and $\Delta(C)$ is the maximum distance that *anyone* needs to travel to their nearest Starbucks.

Greedy Approximation Algorithm: Later in the semester we will show that the k-center problem is NP-hard. (This assumes that k is an arbitrary value between 1 and n, and hence it can grow asymptotically with n. When k is a fixed constant, the problem can be solved in $O(n^{k+1})$ time, which is a polynomial.) We will consider a simple approximation algorithm for this problem. Given P and k, this algorithm produces a set of centers such that the final bottleneck distance is at most twice the optimum value.

The algorithm begins by selecting any point of P to be the initial center g_1. We then repeat the following process until we have k centers. Let $G_i = \{g_1, \ldots, g_i\}$ denote the current set of centers. Recall that $\Delta(G_i)$ is the maximum distance of any point of P from its nearest center. Let u be the point achieving this distance. The greediest way to satisfy u is to put a center directly at u, that is, set $g_{i+1} \leftarrow u$. In other words, set $G_{i+1} \leftarrow G_i \cup \{u\}$.

Following our Starbucks analogy, the point u selected in each step is the most dissatisfied customer, because she has to drive the farthest to get to the nearest Starbucks. However, she is rewarded by having a new Starbucks plopped down on top of her house.

The pseudocode is presented in the code block below. The value $d[u]$ denotes the distance from u to its closest center. (We make one simplification, by starting with G being empty. When we select the first center, all the points of P have infinite distances, so the initial choice is arbitrary.)

It is easy to see that the algorithm's running time is $O(kn)$. In general $k \leq n$, so this is $O(n^2)$ in the worst case. For the sake of the analysis, define $\Delta_i = \Delta(G_i)$. The algorithm is illustrated (for the Euclidean case) in Fig. 42. In each step we show the current set of centers $G_i = \{g_1, \ldots, g_i\}$ and the value Δ_i, which is the farthest from its closest center. (Note that the closest center can be any point of G_i.) The point that achieves this distance is labeled as g_{i+1}, since it will be the next center chosen.

Approximation Bound: Let $G = \{g_1, \ldots, g_k\}$ denote some set of centers computed by the greedy algorithm, and let $\Delta(G)$ denote its bottleneck distance. Let $O = \{o_1, \ldots, o_k\}$ denote any

```
GreedyKCenter(P, k) {
    G = empty
    for each u in P do                // initialize distances
        d[u] = INFINITY

    for (i = 1 to k) {
        Let u be the point of P such that d[u] is maximum
        Add u to G                    // u is the next cluster center
        for (each v in P) {           // update distance to nearest center
            d[v] = min(d[v], delta(v,u))
    }
    return G                          // final centers
}
```

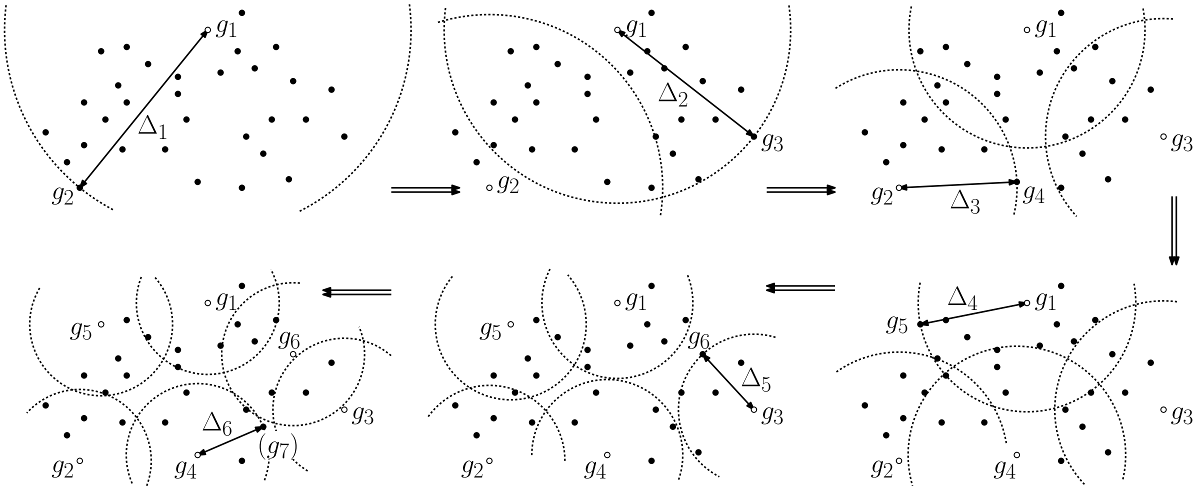


Fig. 42: Greedy approximation to k-center (for $k = 6$).

optimum set of centers, and let $\Delta(O)$ denote the optimum bottleneck distance. Our main result is

$$\Delta(G) \ \leq \ 2\Delta(O),$$

which implies that the greedy approximation algorithm is within a factor of 2 of the optimum.

Our approach will be similar to one that we have used in other greedy proofs. We will present a lower bound on the optimum bottleneck distance, and we will show that our algorithm produces a value that is at most twice this optimum value. The analysis is based on the following three lemmas, each of which is quite straightforward to prove. The greedy algorithm stops with g_k, but for the sake of the analysis it is convenient to consider the next center to be added if we ran it for one more iteration. That is, define g_{k+1} to be the point of P that is maximizes the distance to its closest center in G_k. (This distance is $\Delta(G_k)$.) Also, define $G_{k+1} = \{g_1, \ldots, g_{k+1}\}$.

Lemma 1: The sequence of bottleneck distances $\langle \Delta_1, \ldots, \Delta_k \rangle$ in the greedy algorithm is monotonically nonincreasing, that is $\Delta_i \geq \Delta_{i+1}$. (In Fig. 42 this is represented by the fact that the radii of the disks decreases with each stage.)

Lemma 2: Every pair of greedy centers is separated by at least the greedy bottleneck distance, that is, for $1 \leq i < j \leq k+1$, $\delta(g_i, g_j) \geq \Delta(G)$. (In Fig. 42 this is represented by the fact that no disk in the final picture contains the center of any other.)

Lemma 3: Let $\Delta_{\min} = \Delta(G)/2$. Then for any set C of k cluster centers, we have $\Delta(C) \geq \Delta_{\min}$. (In Fig. 42 this means that it is *not* possible to cover all the points of P using k disks whose radii are smaller than $\Delta_{\min}$.)

Before getting to the proofs, let's see why this implies the approximation bound. By Lemma 3, the bottleneck cost of any k-center solution C must be at least as large as $\Delta_{\min} = \Delta(G)/2$. Thus, $\Delta_{\min}$ plays the role of the lower bound, which we mentioned earlier. Since this applies to the optimal solution as well, we have $\Delta(O) \geq \Delta(G)/2$, or equivalently $\Delta(G) \leq 2\Delta(O)$. Thus, G is a factor-2 approximation to the optimal bottleneck distance.

Here are the proofs of the lemmas:

Proof of Lemma 1: As more centers are added, the distance of any point to its closest center cannot increase (it can only stay the same or decrease). Therefore, since $G_i \subseteq G_{i+1}$, we have $\Delta_i \geq \Delta_{i+1}$.

Proof of Lemma 2: When g_j is added, it is at distance Δ_{i-1} from its closest center in G_{j-1}. Since $i < j$, $g_i \in G_{j-1}$, and therefore $\delta(g_i, g_j) \geq \Delta_{i-1}$, and by Lemma 1, this is at least as large as $\Delta_k = \Delta(G)$.

Proof of Lemma 3: Consider any set C of k centers whose bottleneck distance is smaller than $\Delta_{\min}$. By the pigeonhole principal, there exists at least two centers $g_i, g_j \in G_{k+1}$, where $1 \leq i < j \leq k+1$ that are in the same neighborhood of some center $c \in C$. Since $\Delta(c) \leq \Delta_{\min}$, by combining Lemma 2 with the triangle inequality we have

$$\Delta(G) \ \leq \ \delta(g_i, g_j) \ \leq \ \delta(g_i, c) + \delta(c, g_j) \ < \ \Delta(C) + \Delta(C) \ = \ 2\Delta(C).$$

Therefore, $\Delta(C) > \Delta(G)/2 = \Delta_{\min}$. Since this holds for any set of centers, it holds for the optimum set O.

Lecture 12: Greedy Approximation: Set Cover

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.* We are given a pair (X, S), called a *set system*, where $X = \{x_1, \ldots, x_m\}$ is a finite set of objects, called the *universe*, and $S = \{s_1, \ldots, s_n\}$ is a collection of subsets of X, such that every element of X belongs to at least one set of S. A *cover* of S is a subcollection $C \subseteq S$ such that every element of X belongs to at least one set of C, that is,

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

Given a set system, *set cover* is the problem of computing the smallest sized cover C of S (Fig. 43(a) and (b)).

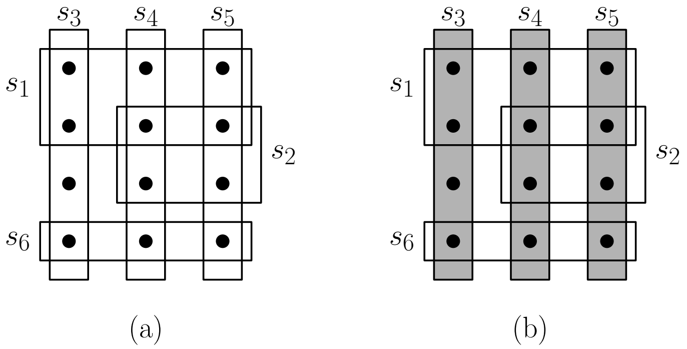


Fig. 43: Set cover. The optimum cover consists of the three sets $\{s_3, s_4, s_5\}$.

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 (which should be thought of as the cost of including set s_i). The problem is to compute the set cover of *minimum total weight.* Our simpler version is equivalent to setting $w_i = 1$ for all i. The approximation algorithm that we will present can be generalized to this case.

The set cover problem is a very important and powerful optimization problem. It arises in a vast 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.

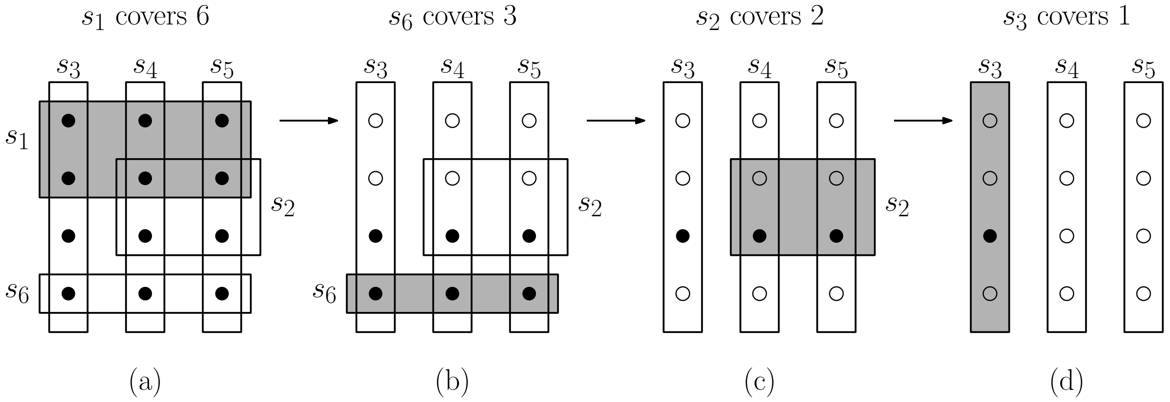
Unfortunately, the set cover problem is known to be NP-hard. We will present a simple *greedy heuristic* for this problem. Given an input instance (X, S), let $k_{\text{opt}}(X, S)$ denote the optimum cover size and let $k_{\text{apx}}(X, S)$ denote the set cover size of some approximation algorithm. Of course, $k_{\text{apx}}(X, S) \geq k_{\text{opt}}(X, S)$. Given $\alpha \geq 1$, we say that the approximation algorithm achieves an *approximation factor* of α, if $k_{\text{apx}}(X, S) \leq \alpha k_{\text{opt}}(X, S)$. There is no known polynomial-time approximation algorithm for set cover that achieves a constant approximation factor. We will show that our greedy heuristic achieves an approximation factor of at most $\ln m$, where $m = |X|$ (the number of items in the universe). Our book proves an even stronger bound, namely that the greedy heuristic achieves an approximation bound of $\ln d$, where d is the maximum cardinality of any set of S.

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. The algorithm is presented in the code block below. The set C contains the indices of the sets of the cover, and the set U stores the elements of X that are still uncovered. Initially, C is empty and $U \leftarrow X$. We repeatedly select the set of S that covers the greatest number of elements of U and add it to the cover.

Greedy Set Cover

```
Greedy-Set-Cover(X, S) {
    U = X                              // U stores the uncovered items
    C = empty                          // C stores the sets of the cover
    while (U is nonempty) {
        select s[i] in S that covers the most elements of U
        add i to C
        remove the elements of s[i] from U
    }
    return C
}
```

We will not worry about implementing this algorithm in the most efficient manner. If we assume that U and the sets s_i are each represented as a simple list of elements of X (each of length at most m), then we can perform each iteration of the main while loop in time $O(mn)$, for a total running time of $O(mn^2)$.



s_1 covers 6

s_6 covers 3

s_2 covers 2

s_3 covers 1

(a) (b) (c) (d)

Fig. 44: The greedy heuristic. Final cover is $\{s_1, s_6, s_2, s_3\}$.

For the example given earlier the greedy-set cover algorithm would select s_1 (see Fig. 44(a)), then s_6 (see Fig. 44(b)), then s_2 (see Fig. 44(c)) and finally s_3 (see Fig. 44(d)). Thus, it would return a set cover of size four, whereas the optimal set cover has size three.

What is the approximation factor? The problem with the greedy heuristic is that it can be "fooled" into picking the wrong set, over and over again. Consider the example shown in Fig. 45. The optimal set cover consists of sets s_7 and s_8, each of size 16. Initially all three sets s_1, s_7, and s_8 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_7 and s_8 all cover eight of the remaining elements. Again, if we choose poorly, s_2 is chosen. The pattern repeats, choosing s_3 (covering four of the remainder), s_4 (covering two) and finally s_5 and s_6 (each covering one). Although there are ties for the greedy choice in this example, it is easy to modify the example so that the greedy choice is unique.

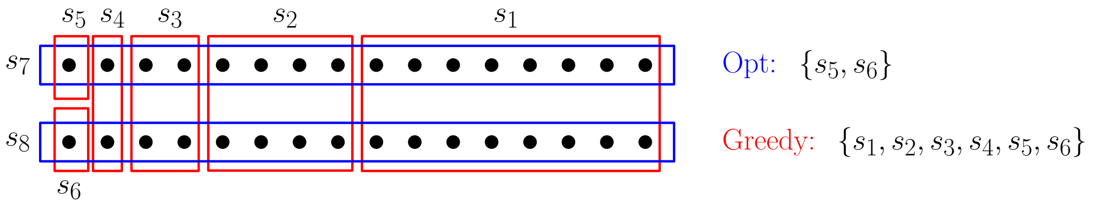


Fig. 45: Repeatedly fooling the greedy heuristic.

From the pattern, you can see that we can generalize this to any number of elements that is a power of 2. While there is a optimal solution with 2 sets, the greedy algorithm will select roughly $\lg m$ sets, where $m = |X|$. (Recall that "lg" denotes logarithm base 2, and "ln" denotes the natural logarithm.) Thus, on this example the greedy heuristic achieves an approximation factor of roughly $(\lg m)/2$. 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.

We will show that the greedy set cover heuristic never performs worse than a factor of $\ln m$. Before giving the proof, we need one useful technical 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 z (positive, zero, or negative), $1 + z \leq e^z$. (This follows from the Taylor's expansion $e^z = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \ldots \geq 1 + z$.) Now, if we substitute $-1/c$ for z 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: The greedy set-cover heuristic achieves an approximation factor 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 $g \leq c \cdot \ln m$. (Note that we should really show that $g + 1 \leq c \cdot \ln m$, but this is close enough and saves us some messy details.)

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 (namely, the optimal cover), by the pigeonhole principal there exists some set that covers at least m_0/c elements. (To see this, observe that 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

$$m_1 \;\leq\; m_0 - \frac{m_0}{c} \;=\; m_0\left(1 - \frac{1}{c}\right) \;=\; m\left(1 - \frac{1}{c}\right).$$

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 by the same logic as above there exists a subset that covers at least m_1/c elements, leaving at most

$$m_2 \;\leq\; m_1\left(1 - \frac{1}{c}\right) \;\leq\; m\left(1 - \frac{1}{c}\right)^2$$

uncovered elements. Thus, $m_2 \leq m(1 - \frac{1}{c})^2$.

The pattern should now be evident. If we apply this argument i 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 i sets have been chosen by the greedy algorithm is at most $m_i \leq m(1 - \frac{1}{c})^i$.

How long can this go on? Since the greedy heuristic 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

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

(In the last step, we just rewrote the expression in a manner that makes it easier to apply the above technical lemma.) By the above lemma we have

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

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

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

Therefore (ignoring the missing "+1" as mentioned above) the greedy set cover is larger than the optimum set cover by a factor of at most $\ln m$.

There is anecdotal evidence that, even though the greedy heuristic has this relatively high approximation factor, 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. You might also wonder whether there is a more sophisticated approximation algorithm that has a better approximation factor. The answer is that such algorithms exist for various special cases of set cover, but for the general problem no significantly better approximation bound is believed to be computable in polynomial time. (Formally, such an algorithm would imply that NP has quasi-polynomial time algorithms, and the experts do not believe that this is the case.)

Lecture 13: Dynamic Programming: Weighted Interval Scheduling

Dynamic Programming: In this lecture we begin our coverage 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 powerful technique for solving optimization problems that have certain well-defined 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: This property (sometimes called the *principle of optimality*) 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 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.

An important issue 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*. Recall that the objective in the unweighted problem is to select a set of mutually compatible request of maximum size (see Fig. 46(a)).

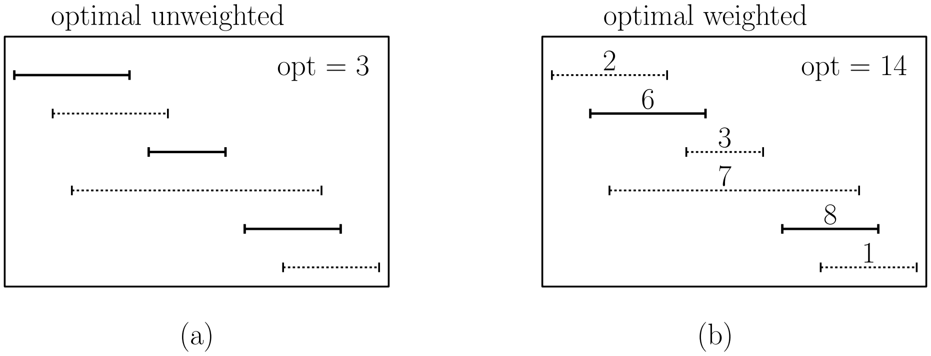


Fig. 46: Weighted and unweighted interval scheduling.

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. 46(b)). The unweighted version can be thought of as a special case 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 demonstrate 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 in nondecreasing order of finish time, so that $f_1 \leq \ldots \leq f_n$. Given any request j, define $p(j)$ to be the largest $i < j$ such that the ith and jth requests are compatible, that is, $p(j) = \max\{i : f_i < s_j\}$. If no such i exists, let $p(j) = 0$ (see Fig. 47).

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 this value. A natural idea would be to define a function that gives the optimum value for just the first i requests.

Definition: For $0 \leq i \leq n$, $\text{opt}(i)$ denotes the maximum possible value achievable if we consider just tasks $\{1, \ldots, i\}$ (assuming that tasks are given in order of finish time).

As a basis case, observe that if we have no tasks, then we have no value. Therefore, $\text{opt}(0) = 0$. If we can compute $\text{opt}(i)$ for each value of i, then clearly, the final desired result will be the maximum value using *all* the requests, that is, $\text{opt}(n)$.

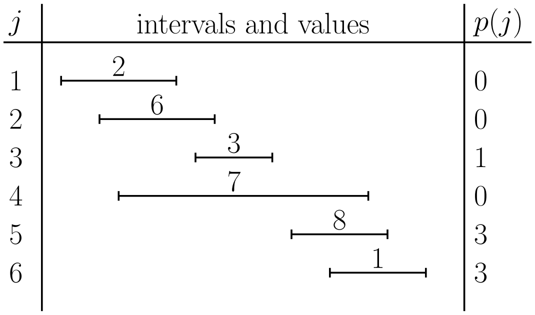


Fig. 47: Weighted interval scheduling input and p-values.

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, $\text{opt}(j) = \text{opt}(j-1)$.

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 $\text{opt}(j) = v_j + \text{opt}(p(j))$.

How do we know which of the these two options to select? The danger in trying to be too smart (e.g., trying a greedy choice) is that the choice may not be correct. Instead, the best approach is often simple brute force:

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

In this cases there are two options, which suggests the following recursive rule:

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

We could express this in pseudocode as shown in the following code block:

Recursive Weighted Interval Scheduling

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

I have left it as self-evident that this simple recursive procedure is correct. Indeed the only subtlety is the inductive observation that, in order to compute 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.[9]

Memoized Formulation: The principal 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 $j \geq 2$, $T(j) = T(j-1)+T(j-2)$. If you start expanding this recurrence, you will find that the resulting series is essentially a Fibonacci series:

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

It is well known that the Fibonacci series $F(j)$ grows exponentially as a function of j. 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 48). 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.

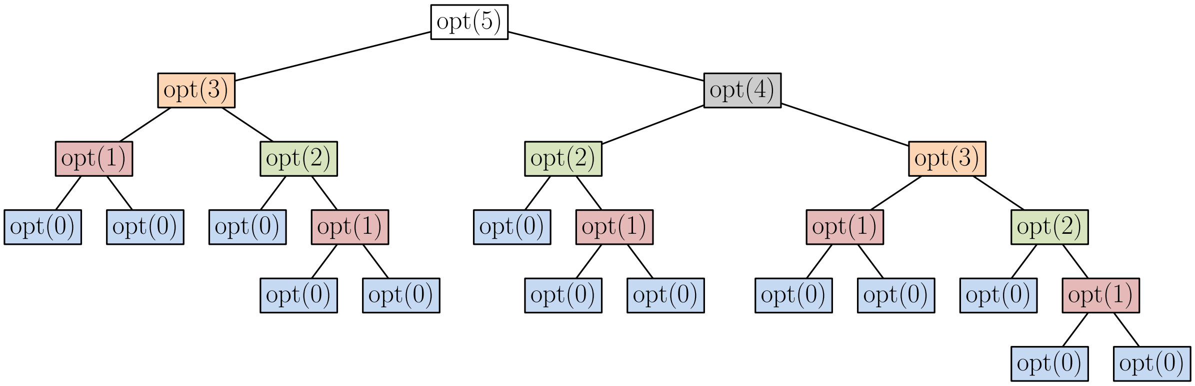


Fig. 48: The exponential nature of recursive-WIS.

This suggests a smarter version of the algorithm. Once a value has been computed for recursive-WIS(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 memoized-WIS 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.

[9] Why would you want to solve a subproblem suboptimally? Suppose, that you had the additional constraint that the final schedule can only contain 23 intervals. Now, it might be advantageous to solve a subproblem suboptimally, so that you have a few extra intervals left over to use at a later time.

Memoized Weighted Interval Scheduling

```
memoized-WIS(j) {
    if (j == 0) return 0
    else if (M[j] has been computed) return M[j]
    else {
        M[j] = max( memoized-WIS(j-1), v[j] + memoized-WIS(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*, 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. 49. Clearly the running time is $O(n)$.

Bottom-Up Weighted Interval Scheduling

```
bottom-up-WIS() {
    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];
        }
    }
}
```

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 bottom-up-WIS() 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

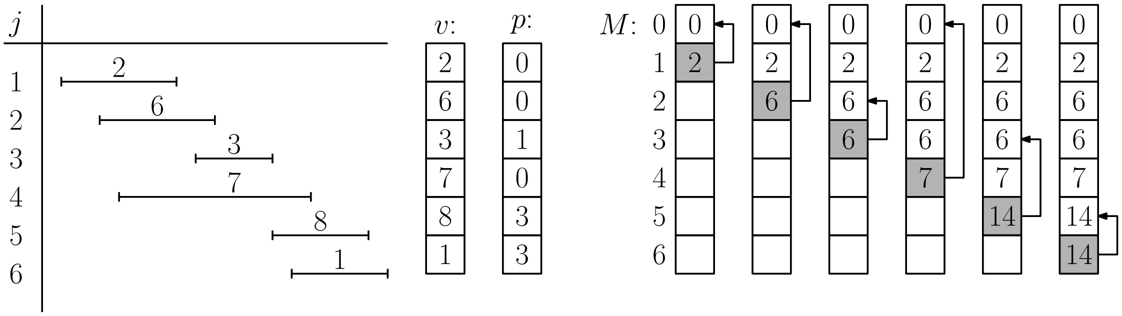


Fig. 49: Example of iterative construction and predecessor values. The final optimal value is 14. By following the predecessor pointers back from $M[6]$ we see that the requests that are in the schedule are 5 and 2.

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. 49 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 14: 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 $X = \langle \text{ABRACADABRA} \rangle$ and let $Z = \langle \text{AADAA} \rangle$, then Z is a subsequence of X.

Given two strings X and Y, the *longest common subsequence* of X and Y is a longest sequence Z that is a subsequence of both X and Y. For example, let $X = \langle \text{ABRACADABRA} \rangle$ and let $Y = \langle \text{YABBADABBADOO} \rangle$. Then the longest common subsequence is $Z = \langle \text{ABADABA} \rangle$ (see Fig. 50).

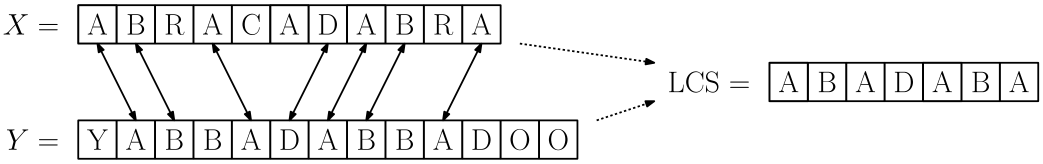


Fig. 50: An example of the LCS of two strings X and Y.

The *Longest Common Subsequence Problem* (LCS) is the following. Given two sequences $X = \langle x_1, \ldots, x_m \rangle$ 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 $\langle \text{ABC} \rangle$ and $\langle \text{BAC} \rangle$ is either $\langle \text{AC} \rangle$ or $\langle \text{BC} \rangle$.

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

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

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

Let us start by deriving a recursive formulation for computing $\text{lcs}(i, j)$. 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, $\text{lcs}(i, 0) = \text{lcs}(j, 0) = 0$.

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, but intuitively this is proved by contradiction. If the LCS did not end in 'A', then we could make it longer by adding 'A' to its end.) Also, 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 instance of 'A' of the other can only limit our future 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 $\text{lcs}(X_{i-1}, Y_{j-1}) + 1$ (see Fig. 51), which provides us with the following rule:

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

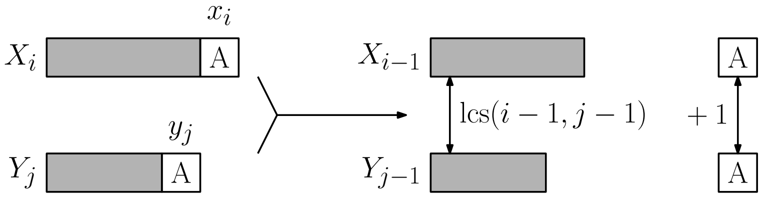


Fig. 51: LCS of two strings whose last characters are equal.

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 $\text{lcs}(i-1, j)$.

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 $\text{lcs}(i, j-1)$.

We compute both options and take the one that gives us the longer LCS (see Fig. 52).

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

Combining these observations we have the following recursive formulation:

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

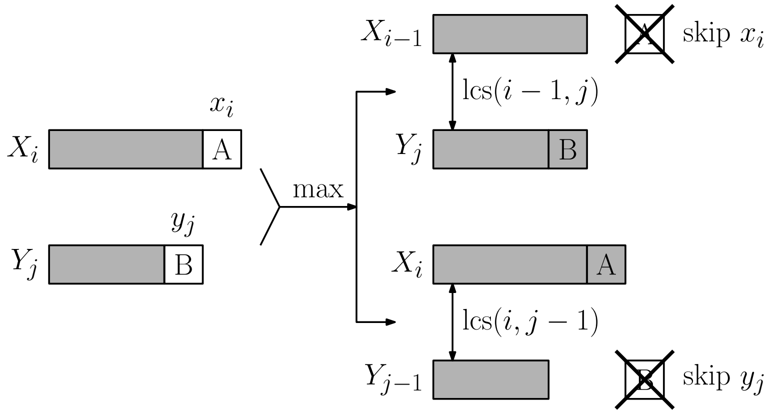


Fig. 52: The possibe cases in the DP formulation of LCS.

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 $\mathrm{lcs}(i,j)$ for the same values of i and j. To avoid this, it creates a 2-dimensional array $\mathrm{lcs}[0..m, 0..n]$, 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).

Memoized Longest Common Subsequence

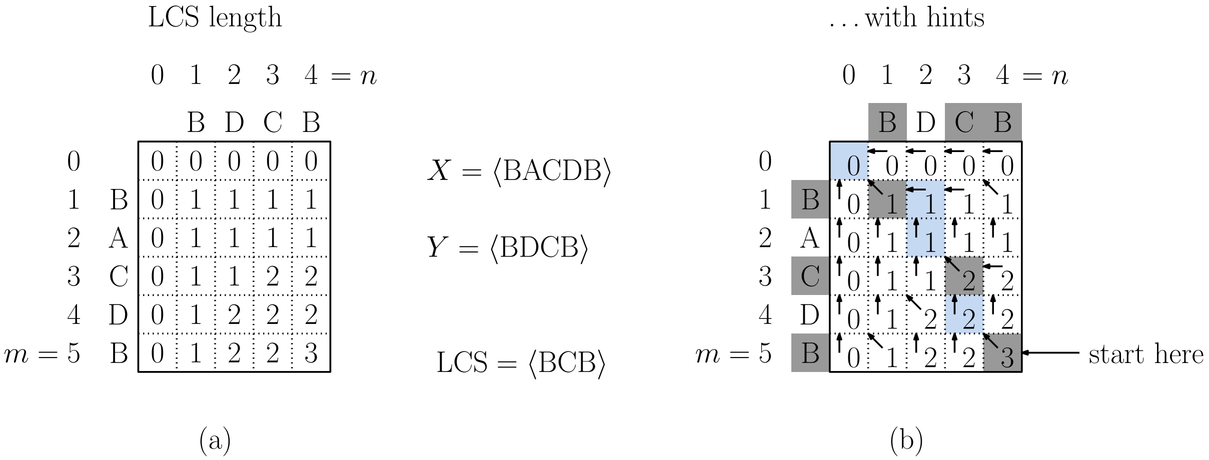
```
memoized-lcs(i,j) {
    if (lcs[i,j] has not yet been computed) {
        if (i == 0 || j == 0)                 // basis case
            lcs[i,j] = 0
        else if (x[i] == y[j])                // last characters match
            lcs[i,j] = memoized-lcs(i-1, j-1) + 1
        else            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

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. 53(a). The running time and space used by the algorithm are both clearly $O(mn)$.

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
}
```



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

(a)

$X = \langle \text{BACDB} \rangle$

$Y = \langle \text{BDCB} \rangle$

LCS = $\langle \text{BCB} \rangle$

. . . with hints

(b)

Fig. 53: 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. 53(b)) and continue with lcs$[i-1, j-1]$

skip$_X$: Do not include x_i to the LCS ('↑' in Fig. 53(b)) and continue with lcs$[i-1, j]$

skip$_Y$: Do not include y_j to the LCS ('←' in Fig. 53(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. 53(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 $h[i, j] = \text{skip}_X$ 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 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 15: 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. 54). The number of columns of A must equal the number of rows of B. In particular for $1 \le i \le p$ and $1 \le j \le r$, we have

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

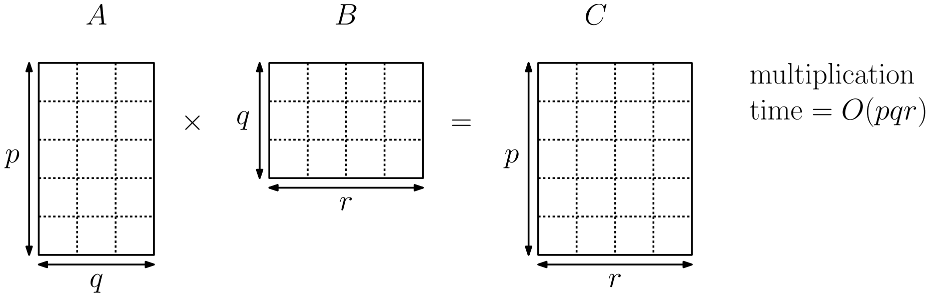


Fig. 54: Matrix Multiplication.

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,

$$A_{1..n} = A_{1..k} \cdot A_{k+1..n} \qquad \text{for } 1 \leq k \leq n-1.$$

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

- What is the best place to split the chain? (what is k?)
- How do we parenthesize each of 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 $1 \leq i \leq j \leq n$, 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, $i \leq k < j$ (see Fig. 55).

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 $m(i,j)$.

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

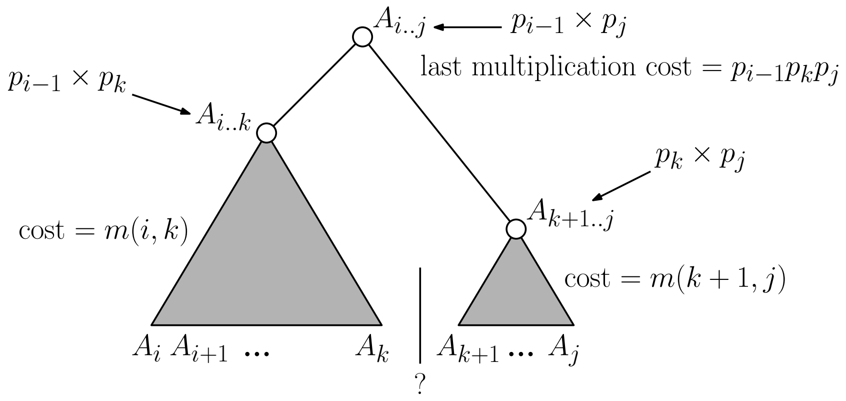


Fig. 55: Dynamic programming decision.

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. 56 for an example.) We will explain below the purpose of the s array.

Chain Matrix Multiplication

```
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)
}
```

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

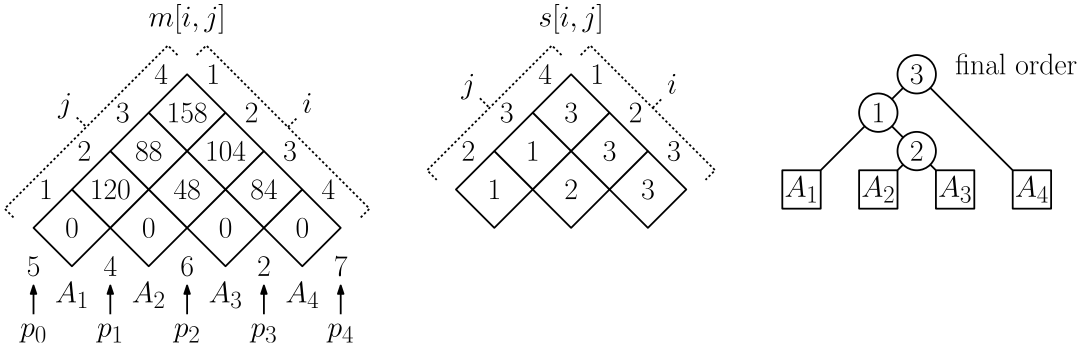


Fig. 56: Chain matrix multiplication for the product $A_1 \cdots A_4$, where A_i is of dimension $p_{i-1} \times p_i$.

first multiply the subchain $A_{i..k}$ and then multiply the subchain $A_{k+1..j}$, and finally multiply these together. Intuitively, $s[i,j]$ tells us what multiplication to perform *last*. Note that we only need to store $s[i,j]$ when we have at least two matrices, that is, if $j > i$.

The actual multiplication algorithm uses the $s[i,j]$ value to determine how to split the current sequence. Assume that the matrices are stored in an array of matrices $A[1..n]$, and that $s[i,j]$ is global to this recursive procedure. The recursive procedure do-mult does this computation and below returns a matrix (see Fig. 56).

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 16: 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.)

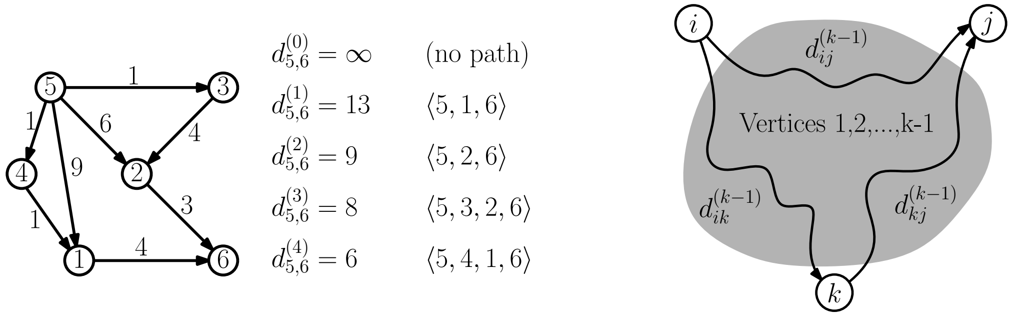
The output will be an $n \times n$ distance matrix $D = d_{ij}$ where $d_{ij} = \delta(i,j)$, the shortest path cost from vertex i to j. Recovering the shortest paths will also be an issue. To help us do this, we will also compute an auxiliary matrix $\mathit{mid}[i,j]$. The value of $\mathit{mid}[i,j]$ 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 $\mathit{mid}[i,j]$ 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. 57(a), notice how the value of $d_{5,6}^{(k)}$ changes as k varies.



$$d_{5,6}^{(0)} = \infty \qquad \text{(no path)}$$
$$d_{5,6}^{(1)} = 13 \qquad \langle 5, 1, 6\rangle$$
$$d_{5,6}^{(2)} = 9 \qquad \langle 5, 2, 6\rangle$$
$$d_{5,6}^{(3)} = 8 \qquad \langle 5, 3, 2, 6\rangle$$
$$d_{5,6}^{(4)} = 6 \qquad \langle 5, 4, 1, 6\rangle$$

(a) (b)

Fig. 57: 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: The shortest path from node i to node j uses 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. 57(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
}
```

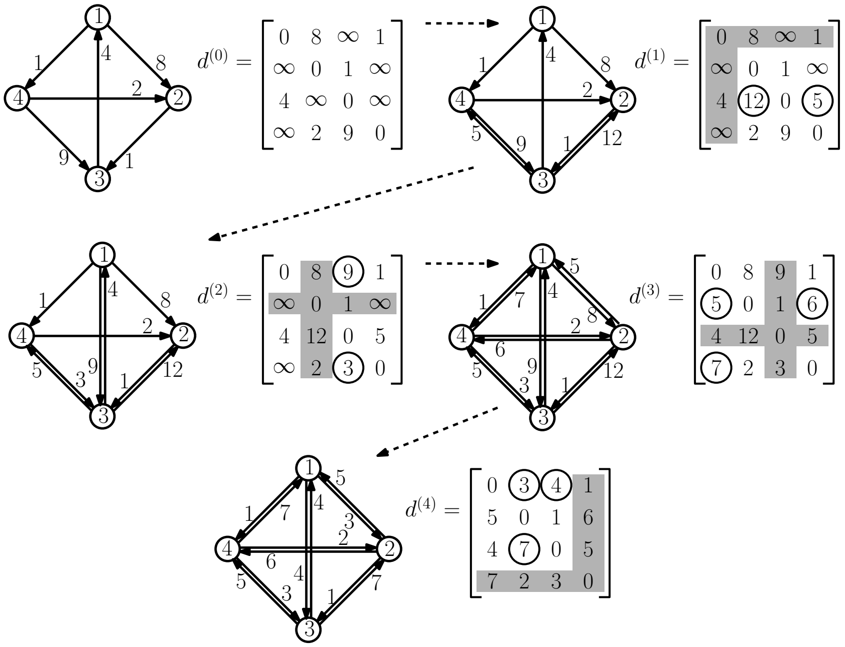
An example of the algorithm's execution is shown in Fig. 58.

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

Extracting Shortest Paths: The mid-vertex pointers *mid*$[i, j]$ can be used to extract the final path. Here is the idea, whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set *mid*$[i, j] = k$. If the shortest path does not pass through any intermediate vertex, then *mid*$[i, j] =$ *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.

Lecture 17: 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



$$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$

$$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$$

$$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$

$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

$$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Fig. 58: Floyd-Warshall Example. Newly updates entries are circled.

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
    }
}
```

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 nonnegative *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. 59).

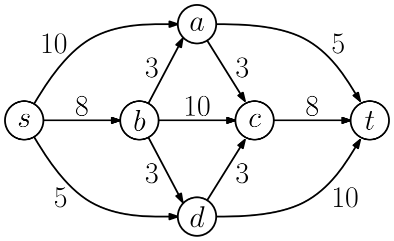


Fig. 59: A flow network.

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.[10] 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

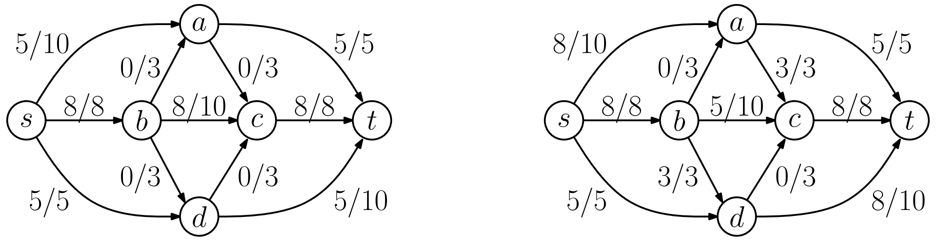
[10] Neither of these is an essential requirement. Given a network that fails to satisfy these assumptions, we can easily generate an equivalent one that satisfies both.

total flow out of v as:

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

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. 60, where we use the notation f/c on each edge to denote the flow f and capacity c for this edge.



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

(a)

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

(b)

Fig. 60: A valid flow and a maximum flow.

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,

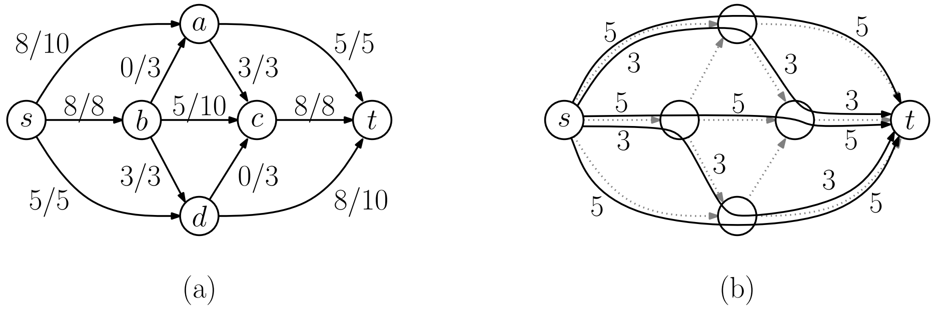
$$|f| \;=\; f^{\text{out}}(s) \;=\; \sum_{w\in V} f(s,w),$$

(For example, the value of the flow shown in Fig. 60(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.

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. 60(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. 59, $\langle s,a,t\rangle$, $\langle s,b,a,c,t\rangle$ 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. 61(b) we show a path-based flow that is equivalent to the edge-based flow of Fig. 61(a). The paths carrying zero flow are not shown.



(a)

(b)

Fig. 61: (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. 62). 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.

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 18: 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

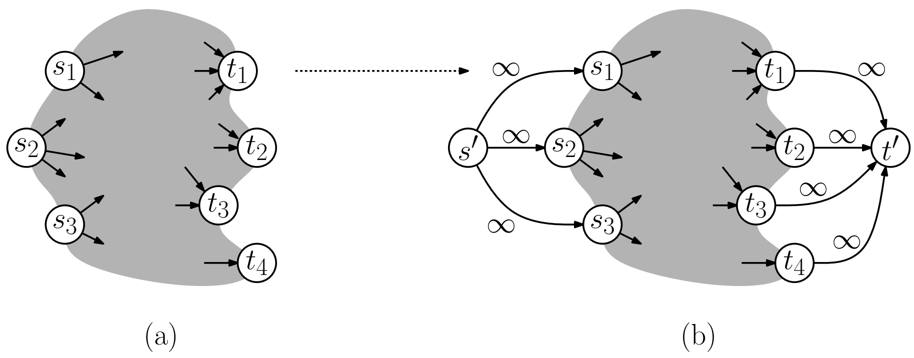


Fig. 62: Reduction from (a) multi-source/multi-sink to (b) single-source/single-sink.

Max Flow algorithm, cuts, and the relationship between flows and cuts.

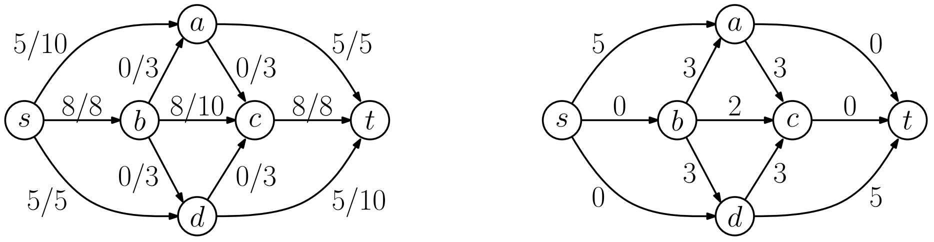
Recall that a *flow network* is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative *capacity* $c(u, v) \geq 0$, with a designated source vertex s and sink vertex t. We assume that there are no edges entering s or exiting t. A *flow* is a function f that maps each edge to 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. 63(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. 63(b)) we see that there is no path of positive capacity from s to t. Thus, greedy gets stuck.

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



(a) (b)

Fig. 63: The greedy flow algorithm can get stuck before finding the maximum flow.

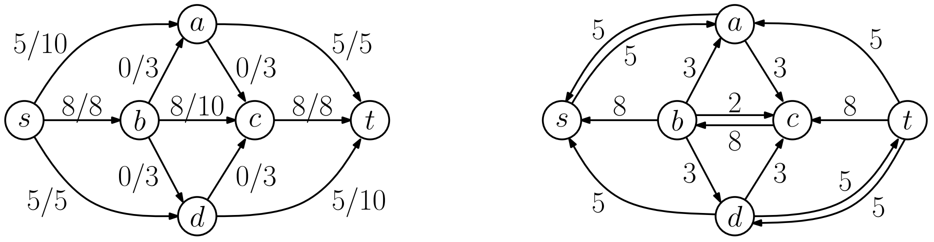
augmenting paths. Given a flow network G and a flow f, define the *residual network*, denoted G_f, to be a network having the same vertex set and same source and sink, and whose edges are defined as follows:

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. 64(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. 64: 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 $|f| + |f'|$.

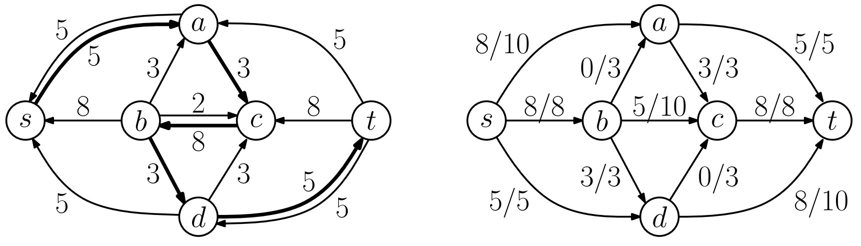
Lemma: Let f be a flow in G and let f' be a flow in G_f. Then $(f + f')$ is a flow in G.

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. 65(a) we show an augmenting path of capacity 3 in the residual network for the flow given earlier in Fig. 64. 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. 65: 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.

Ford-Fulkerson Network Flow

```
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 augmentations left
        P = any-augmenting-path of G’           // augmenting path
        c = minimum capacity edge of P          // augmentation amount
        augment f by adding c to the flow on every edge of P
    }
    return f
}
```

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 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.

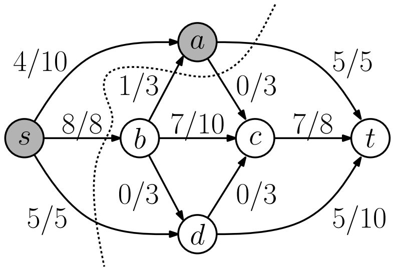
We want to make this concept mathematically formal. Since such a set of edges lie on every path from s from t, their removal defines a partition separating the vertices that s can reach from the vertices that s cannot reach. This suggests the following concept.

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



$X = \{s, a\}$

$Y = \{b, c, d, t\}$

$f(X, Y) = 5 + 0 - 1 + 8 + 5 = 17$

Fig. 66: Flow of value 17 across the cut $(\{s, a\}, \{b, c, d, t\})$.

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 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 19: 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. 67. 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 (see Fig. 67(b)). 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 (see Fig. 67(c)). Proceeding in this way, it will take 200 augmentations until we terminate with the final maximum flow (see Fig. 67(d)). 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.

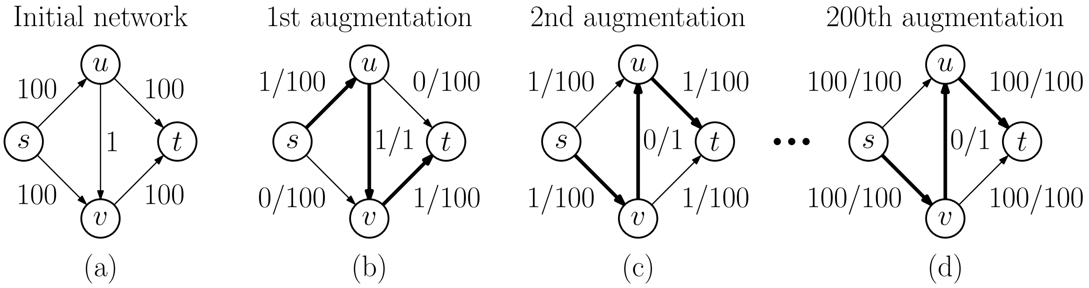


Fig. 67: Bad example for Ford-Fulkerson.

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 $\Delta \leftarrow \lceil \Delta/2 \rceil$, 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.

For example, consider the same network as in Fig. 67. We start with the upper bound of $C = 401$. (Let's just round this to 400 for simplicity.) As usual, let us start with f being the zero flow (see Fig. 68(a)).

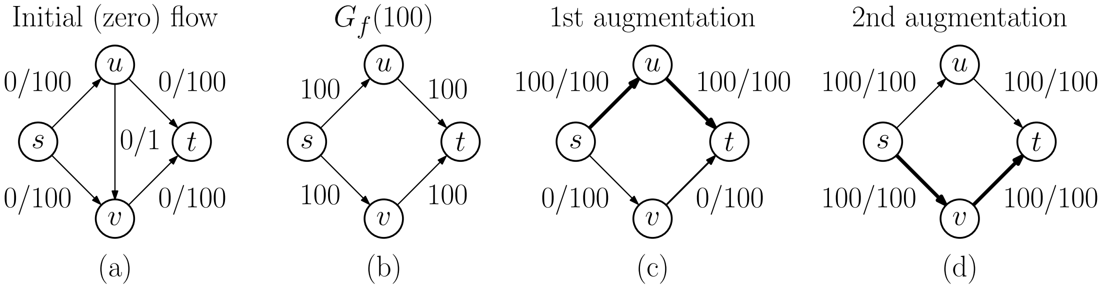


Fig. 68: Scaling algorithm for network flow.

The residual networks $G_f(400)$, $G_f(200)$ are empty, since there are no edges whose capacity exceeds these values. However, $G_f(100)$ includes the edges of capcity 100 (see Fig. 68(b)). They key is that is *does not* contain the edge of capacity 1, and therefore we cannot push flow through this edge at this time. After two augmentations, we obtain the flow shown in Fig. 68(d). The algorithm will then try all the remaining residual networks $G_f(50), G_f(25), \ldots, G_f(1)$. The edge of capacity 1 will appear only in the final residual network, but by this time it serves no purpose because we have already found the 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 *pseudo-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 $O(nm^2)$.

Even Faster 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. The fastest known algorithm for network flow is a hybrid of two algorithms (depending on the ratio between the number of edges and the number of vertices). Its running time is $O(nm)$. The two algorithms are due to King, Rao, and Tarjan (1994) and Orlin (2013).

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 in Bipartite Graphs: 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. 69(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. 69(b)).

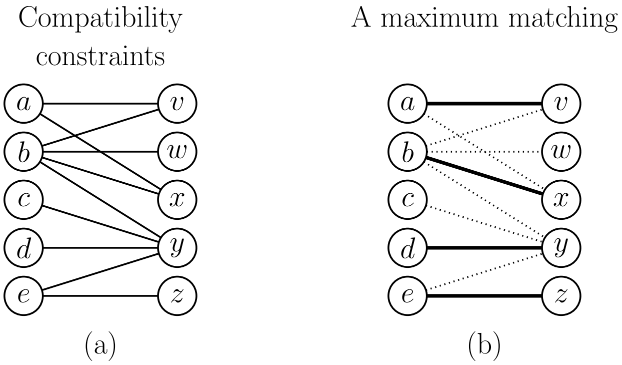


Fig. 69: A bipartite graph G and a maximum matching in G.

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 (see Fig. 70(a)) for 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 $V' = V \cup \{s, t\}$.

$$E' = \begin{cases} \{(s,u) \mid u \in X\} \;\cup & \text{(connect source to left-side vertices)} \\ \{(v,t) \mid v \in Y\} \;\cup & \text{(connect right-side vertices to sink)} \\ \{(u,v) \mid (u,v) \in E\} & \text{(direct } G\text{'s edges from left to right).} \end{cases}$$

Set the capacity of all edges in this network to 1 (see Fig. 70(b)).

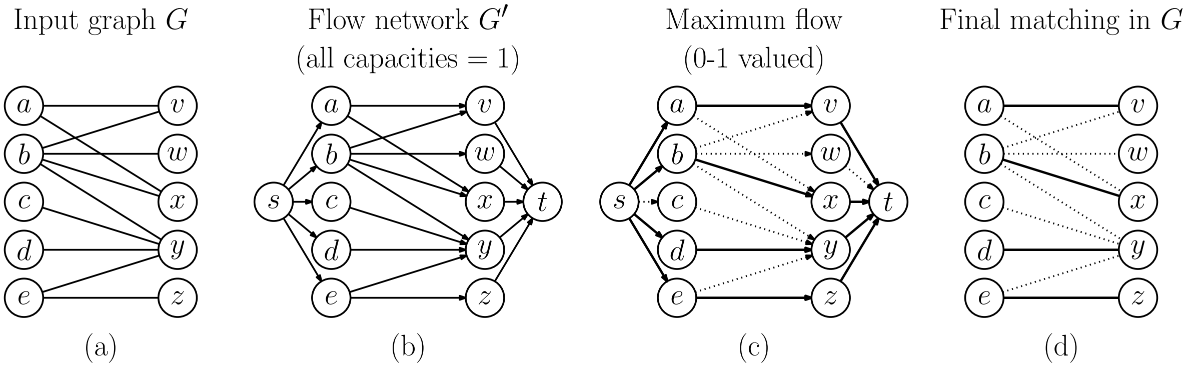


Fig. 70: Reducing bipartite matching to network flow.

Now, compute the maximum flow in G' (see Fig. 70(c)). Although in general it can be that flows are real numbers, observe that the Ford-Fulkerson method will only assign integer value flows to the edges (and this is true of all existing network flow algorithms).

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

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

(see Fig. 70(d)).

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

Because the capacities are so low, we do not need to use a fancy implementation. 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 20: 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.[11] 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. 71(a), we show a network in which each node is each labeled with its demand.)

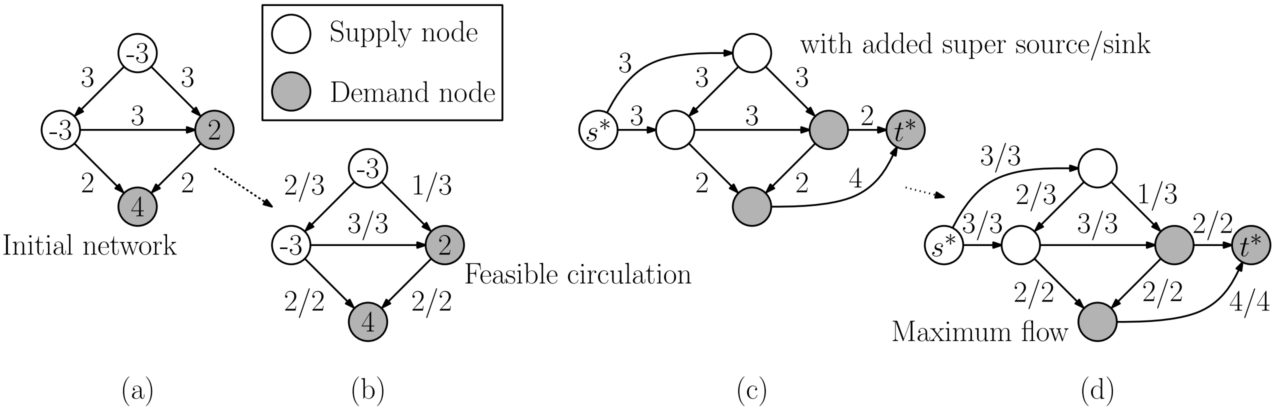


Fig. 71: Reducing the circulation problem to network flow.

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.

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. 71(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

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

Define $D = \sum_{v \in T} d_v$ denote the *total demand.* (Note that this is equal to the negation of the total supply, $\sum_{v \in S} d_v$.)

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

[11] 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.

can simply answer "no" immediately.) The reduction is similar in spirit to what we did for bipartite matching. Given the network $G = (V, E)$ as input (with vertex demands d_v and edges capacities $c(u, v)$):

- Create a new network $G' = (V', E')$ that has all the same vertices and edges as G (that is, $V' \leftarrow V$ and $E' \leftarrow E$)
- Add to V' a *super-souce* s^* and a *super-sink* t^*
- For each supply node $v \in S$, we add a new edge (s^*, v) of capacity $-d_v$
- For each demand node $u \in T$, we add a new adge (u, t^*) of capacity d_v

The entire construction is illustrated in Fig. 71(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' (by whichever maximum flow algorithm you like). If the flow value is at least D, then intuitively, we have managed to push enough flow into the network and (by flow balance) enough flow out of the network to satisfy all the demand constraints (see Fig. 71(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.

Note that we have not presented a new algorithm. Rather, we have reduced the problem of computing circulations to the known problem of computing maximum flows. The running time of the overall solution is equal to the time to compute the reduction, which is $O(n + m)$ plus the time to compute max flows (which is $O(mn)$ time by the current best technology).

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

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

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

Henceforth, we will use the term *upper flow bound* in place of *capacity* (since it doesn't make sense to talk about a lower bound as a capacity constraint). An example of such a network is shown in Fig. 72(a), and a valid circulation is shown in Fig. 72(b).

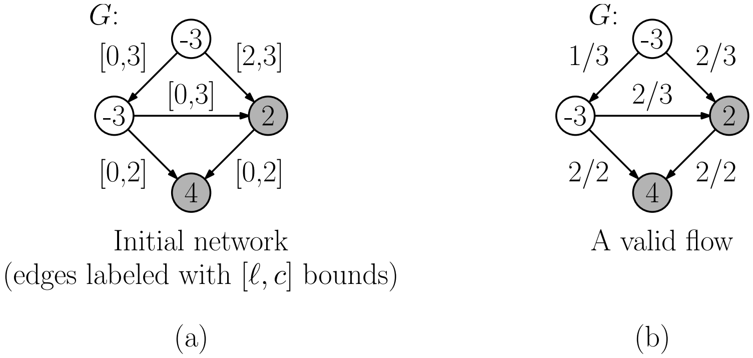


Fig. 72: (a) A network with both upper and lower flow bounds and (b) a valid circulation.

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 (possibly invalid) circulation f_0 that exactly satisfies all the lower flow bounds. In particular, we let $f_0(u, v) = \ell(u, v)$ (see Fig. 73(a)). This circulation may be invalid because f_0 need not satisfy the demand constraints (which, recall, provide for flow balance as well). We will modify the supply/demand values to compensate for this imbalance. Since the lower-bound constraints are all satisfied, it will be possible to apply a standard circulation algorithm (without lower flow bounds) to solve the problem.

To make this formal, for each $v \in V$, let L_v denote the *excess flow* coming into v in f_0, 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 v's supply/demand needs are already met. Otherwise, we will adjust the supply and demand values so that by (1) computing any valid circulation f_1 for the adjusted values and (2) combining this with f_0, we will obtain a flow that satisfies all the requirements.

How do we adjust the supply/demand values? We want to generate a net flow of d_v units coming into v and cancel out the excess L_v coming in. That is, we want f_1 to satisfy:

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

(In particular, this means that if we combine f_0 and f_1 by summing their flows for each edge, then the final flow into v will be d_v, as desired.)

How do we determine whether there exists such a circulation f_1? Let's consider how to set the edge capacities. 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?)

We are now ready to put the pieces together. Given the network $G = (V,E)$ as input (with vertex demands d_v and lower and upper flow bounds $\ell(u,v)$ and $c(u,v)$):

1. Create an initial pseudo-circulation f_0 by setting $f(u,v) = \ell(u,v)$ (see Fig. 73(a))
2. Create a new network $G' = (V', E')$ that has all the same vertices and edges as G (that is, $V' \leftarrow V$ and $E' \leftarrow E$)
3. For each $(u,v) \in E'$, set its adjusted capacity to $c'(u,v) \leftarrow c(u,v) - \ell(u,v)$
4. For each $v \in V'$, set its adjusted demand to $d'_v \leftarrow d_v - L_v$, where, $L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v)$ (see Fig. 73(b))

Now, invoke any (standard) circulation algorithm on G' (see Fig. 73(c)). Note that there is no need to consider lower flow bounds with G', because f_0 has already taken care of those. If the algorithm reports that there is no valid circulation for G', then we declare that there is no valid circulation for the original network G. If the algorithm returns a valid circulation f_1 for G', then the final circulation for G is the combination $f_0 + f_1$ (see Fig. 73(d)).

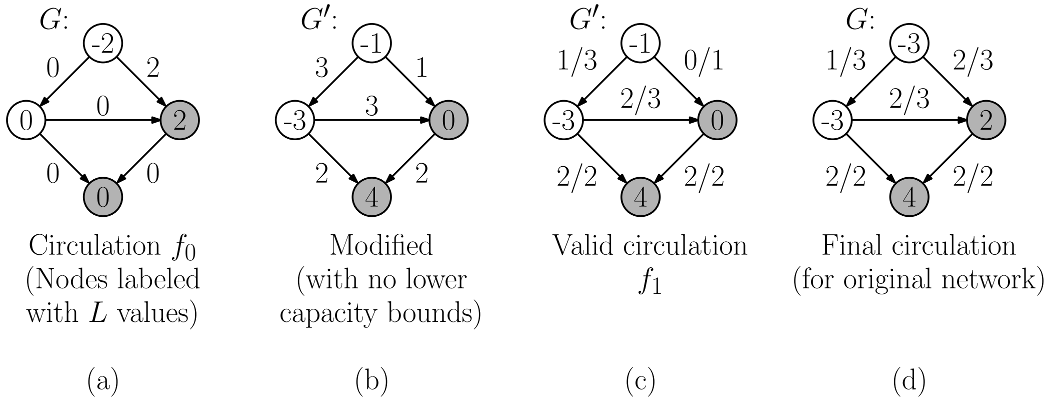


Fig. 73: Reducing the circulation problem with upper and lower flow bounds to a standard circulation problem.

To establish the correctness of our reduction, we prove below that its output $f_0 + f_1$ 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 $f_0 + f_1$.)

Note that (as in the original circulation problem) we have not presented a new algorithm. Instead, we have shown how to *reduce* the current problem (circulation with lower and upper flow bounds) to a problem we have already solved (circulation with only upper bounds). Again, the running time will be the sum of the time to perform the reduction, which is easily seen to be $O(n+m)$ plus the time to compute the circulation, which as we have seen reduces to the time to compute a maximum flow, which according to the current best technology is $O(nm)$ time.

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. 74).

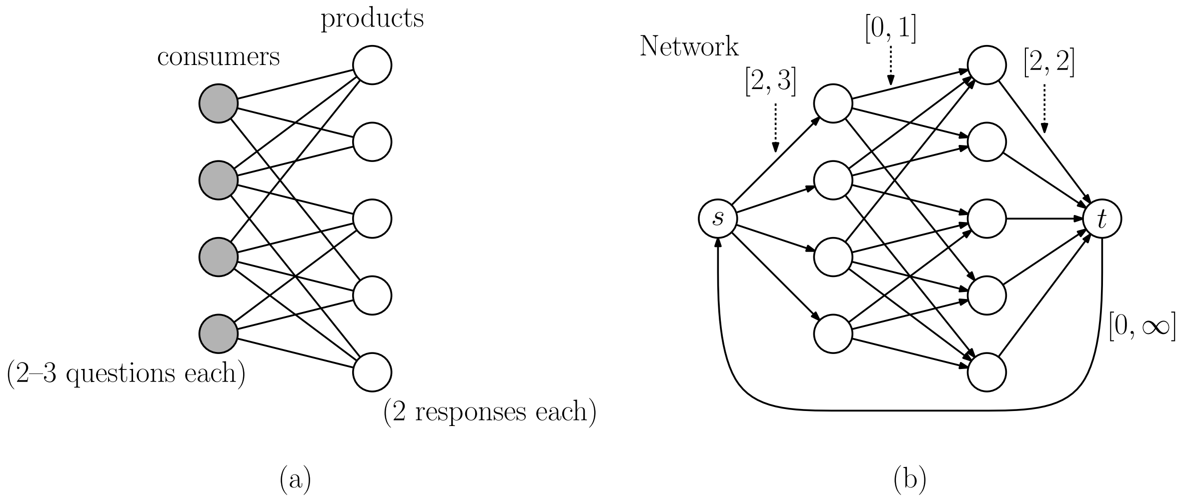


Fig. 74: Reducing the survey design problem to circulation with lower and upper flow bounds.

- 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. 75).

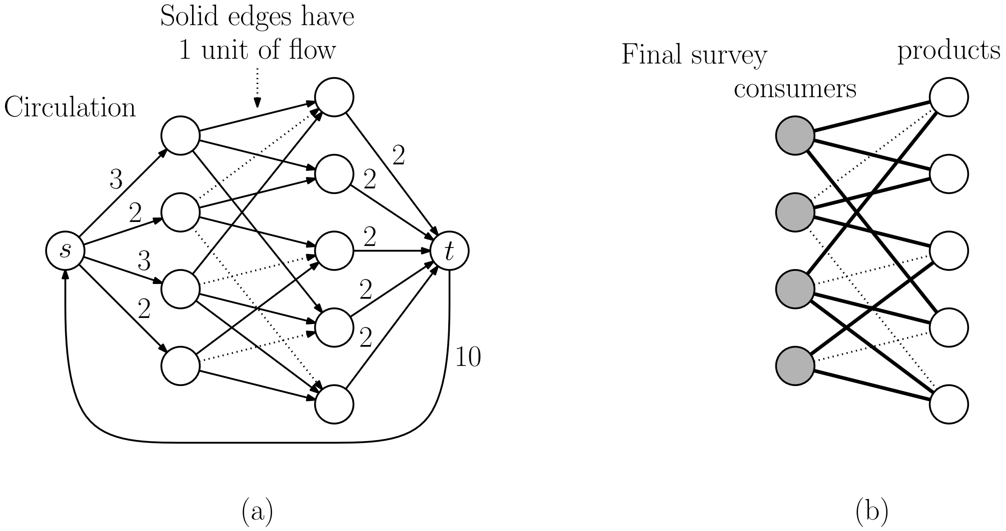


Fig. 75: Reducing the survey design problem to circulation with lower and upper flow bounds.

Lecture 21: 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[12]), 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.

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

[12] 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.

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. 76(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. 76(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.)

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

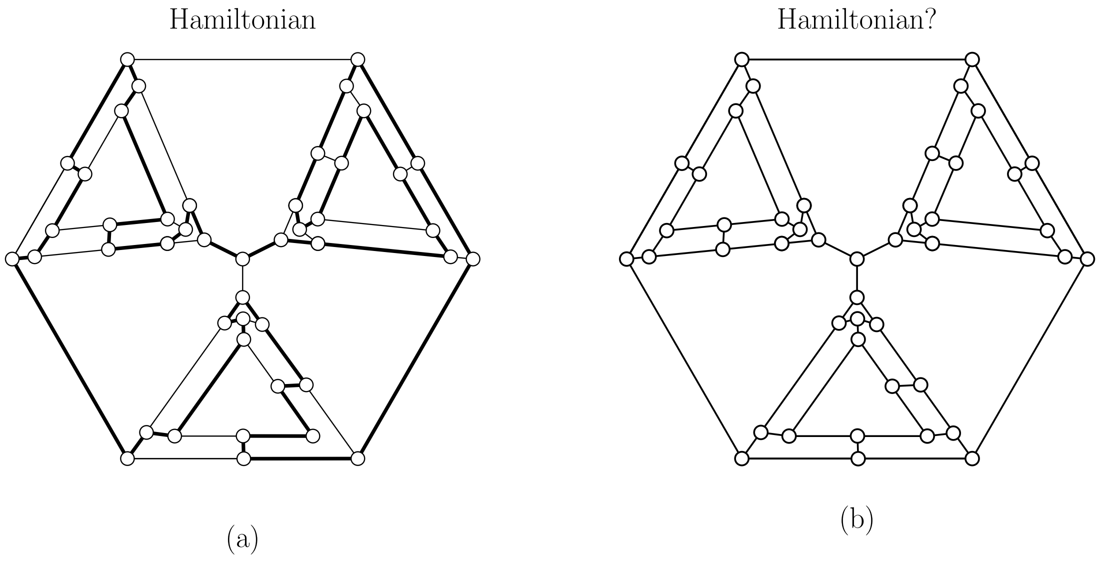


Fig. 76: The Hamiltonian cycle (HC) problem.

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, $\mathrm{P} \subseteq \mathrm{NP}$. However, it is not known whether $\mathrm{P} = \mathrm{NP}$. 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 $\mathrm{P} \neq \mathrm{NP}$, 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 22: NP-Completeness: Reductions

Recap: 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:

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

P: is the class of all decision problems which can be solved in polynomial time. While $\mathrm{MST} \in \mathrm{P}$, we do not know whether $\mathrm{HC} \in \mathrm{P}$ (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, $\mathrm{P} \subseteq \mathrm{NP}$. It is widely believed that $\mathrm{P} \neq \mathrm{NP}$.

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

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

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[13] 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. 77). Then we run our subroutine on x' and output whatever it outputs.

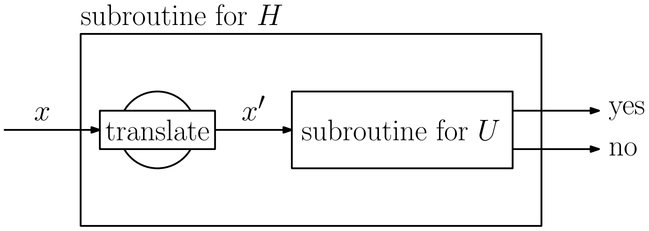


Fig. 77: Reducing H to U.

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.

[13] 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.

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. 78(a) and (b)).

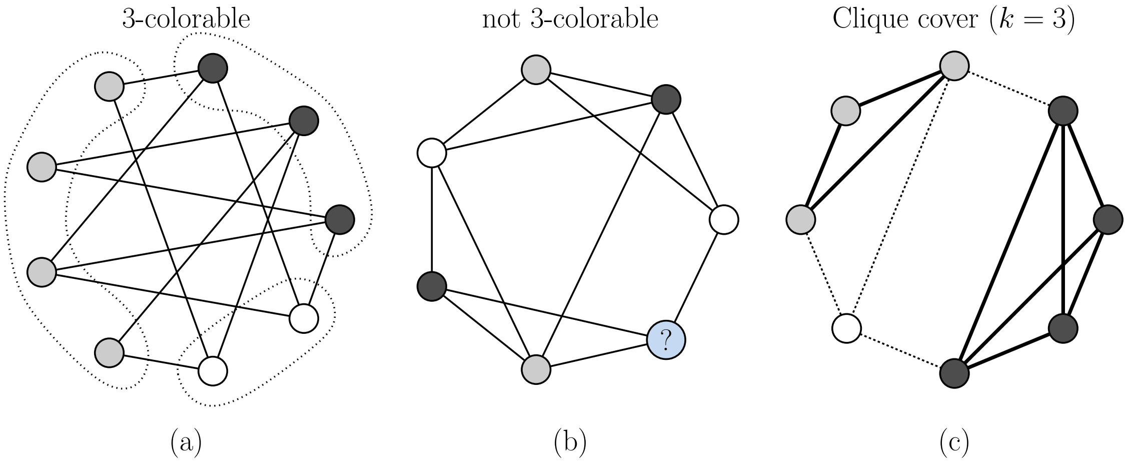


Fig. 78: 3-coloring and Clique Cover.

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. 78(c)).

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. 79).

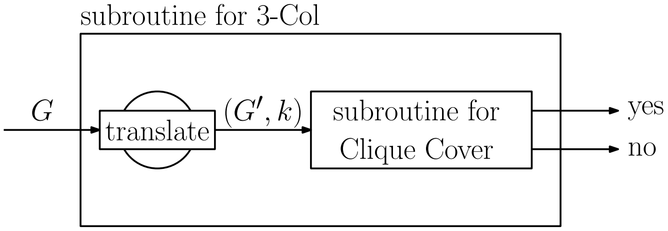


Fig. 79: Reducing 3Col to CCov.

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 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. 80).

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,

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

Proof: $(\Rightarrow)$ If G 3-colorable, then let V_1, V_2, V_3 be the three color classes. We claim that this is a clique cover of size 3 for $\overline{G}$, since if u and v are distinct vertices in V_i, then $\{u, v\} \notin E$ (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 $(\overline{G}, 3)$.

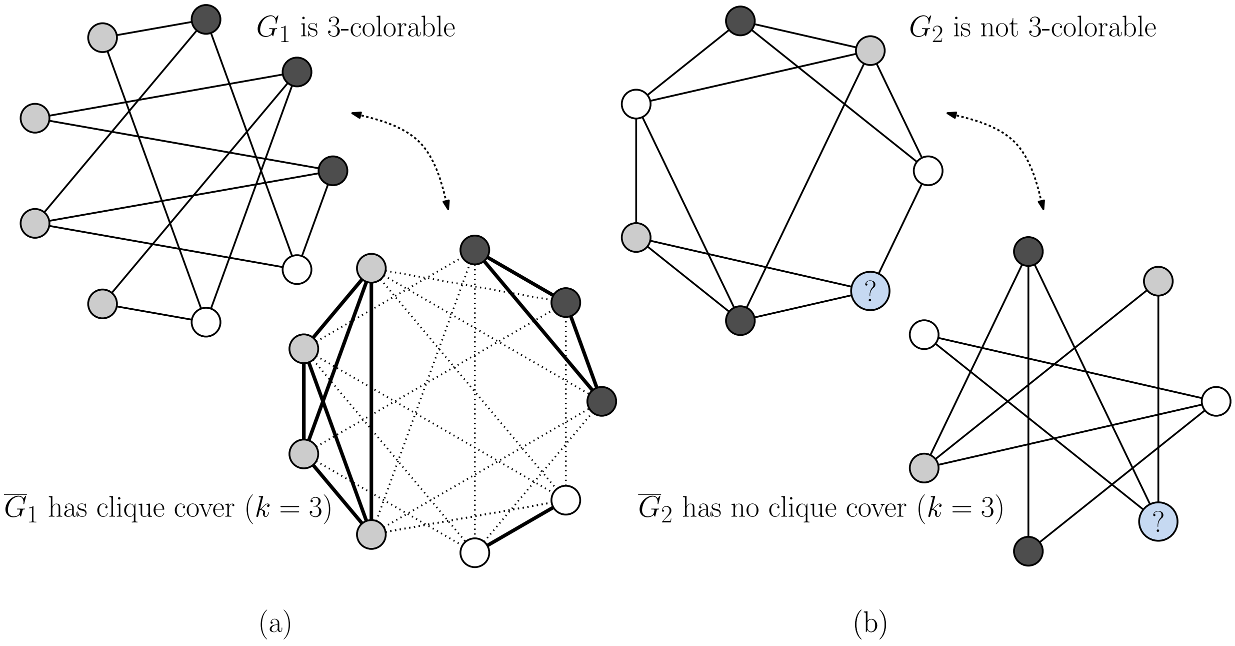

G_1 is 3-colorable
$\overline{G}_1$ has clique cover ($k = 3$)
(a)
G_2 is not 3-colorable
?
$\overline{G}_2$ has no clique cover ($k = 3$)
?
(b)

Fig. 80: Clique covers in the complement.

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. 81 below.

Lecture 23: 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.

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

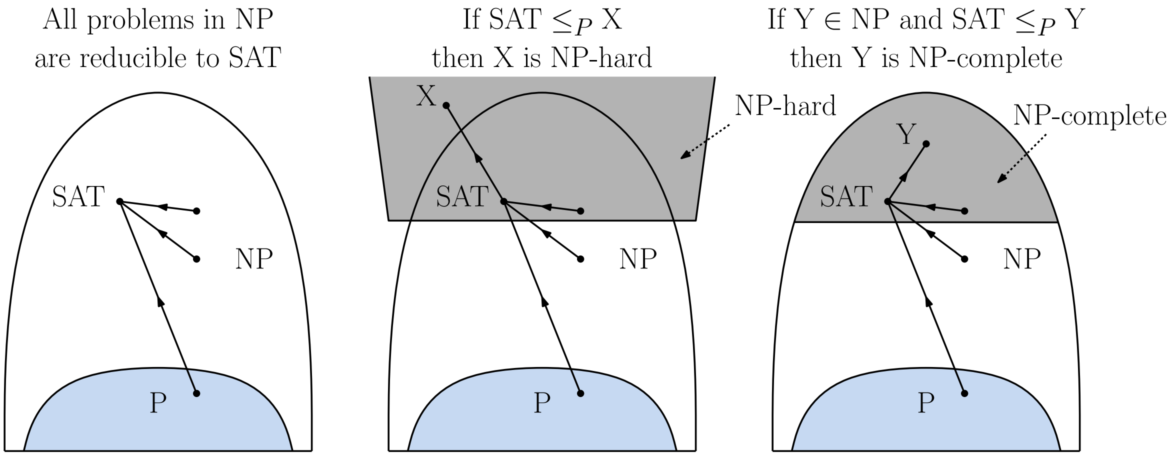


Fig. 81: Structure of NPC and reductions.

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[14] 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.

[14] 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 $k \geq 3$.

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. 82 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.)

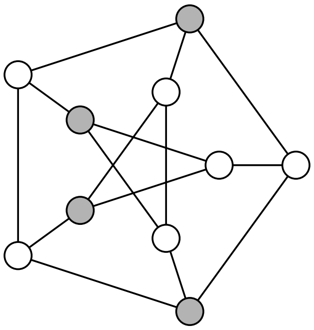


Fig. 82: A graph with an independent set of size $k = 4$.

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. 83(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.

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.

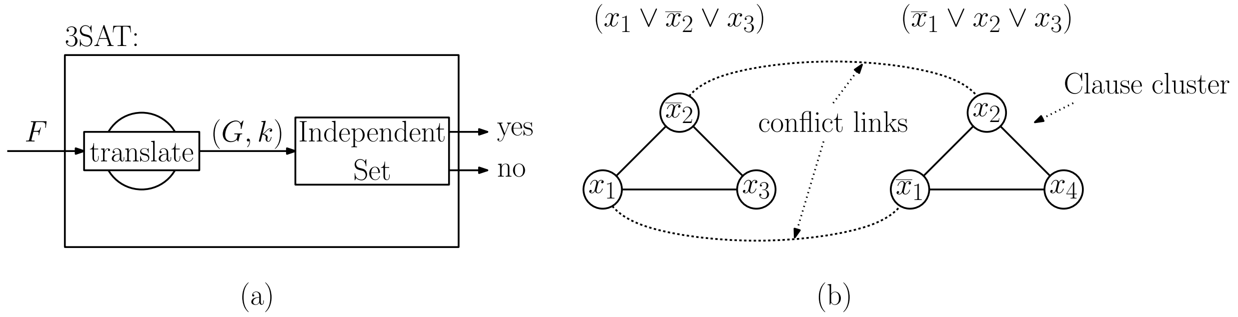


Fig. 83: (a) Reduction of 3-SAT to IS and (b) Clause clusters for the clauses $(x_1 \vee \overline{x}_2 \vee x_3)$ and $(\overline{x}_1 \vee x_2 \vee x_5)$.

- **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. 83(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.

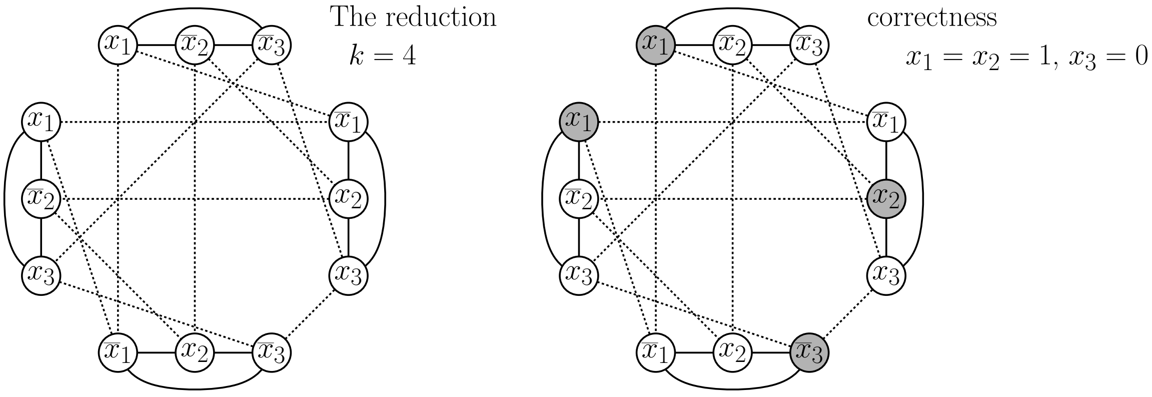
To enforce the restriction that only one of x and $\overline{x}$ can be set to 1, we create edges between all vertices associated with x to all vertices associated with $\overline{x}$. We call these *conflict links.* A formal description of the reduction is given below. The input is a boolean formula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

```
$k \leftarrow$ 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. 84. The clauses clusters appear in clockwise order starting from the top.



(a) (b)

Fig. 84: 3SAT to IS reduction.

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.

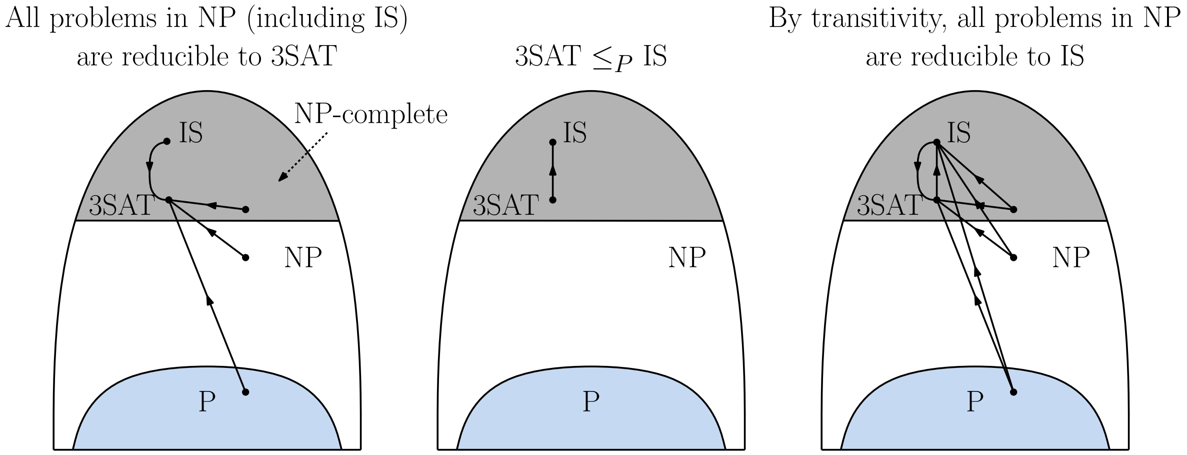
($\Rightarrow$) : If F is satisfiable, then each of the k clauses of F must have at least one true literal. Select such a literal from each clause. Let V' denote the corresponding vertices from each of the clause clusters (one from each cluster). We claim that V' is an independent set of size k. Since 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.

($\Leftarrow$) : 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. 85).


All problems in NP (including IS) are reducible to 3SAT

3SAT $\leq_P$ IS

By transitivity, all problems in NP are reducible to IS

Fig. 85: Our updated picture of NP-completeness.

Lecture 24: 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).

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?

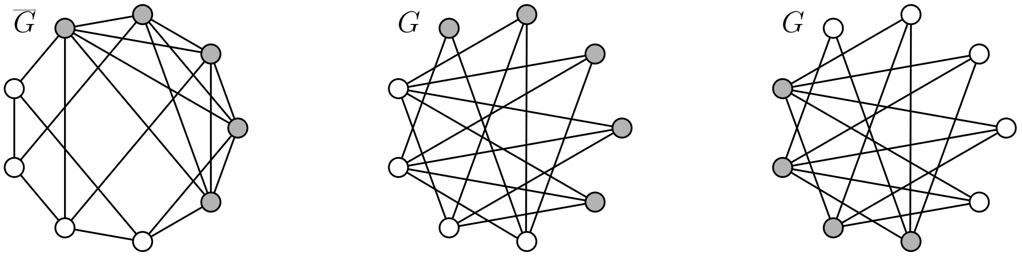
Dominating Set (DS): A *dominating set* in a graph $G = (V, E)$ is a subset of vertices V' such that every vertex in the graph is either in V' or is adjacent to some vertex in V'. The *dominating set problem* (DS) is: given a graph $G = (V, E)$ and an integer k, does G have a dominating set of size k?

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. 86). 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:



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

Fig. 86: Clique, Independent set, and Vertex Cover.

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

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

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

Proof:

(i) ⇒ (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) ⇒ (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) ⇒ (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 ∈ 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 ∈ 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'.

Vertex Cover to Dominating Set: Next, we show that a known 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. 87. Note that every step can be performed in polynomial time.

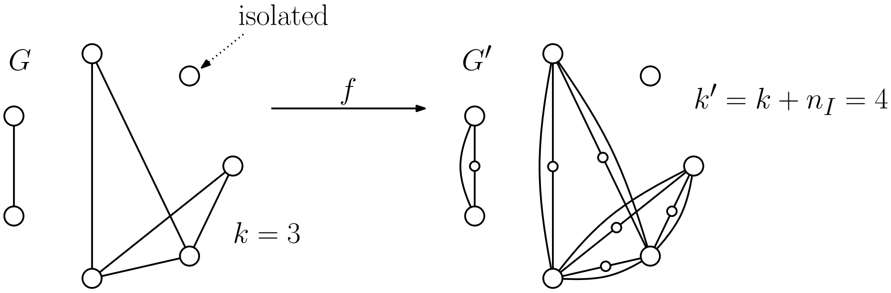


Fig. 87: Dominating set reduction with $k = 3$ and one isolated vertex.

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

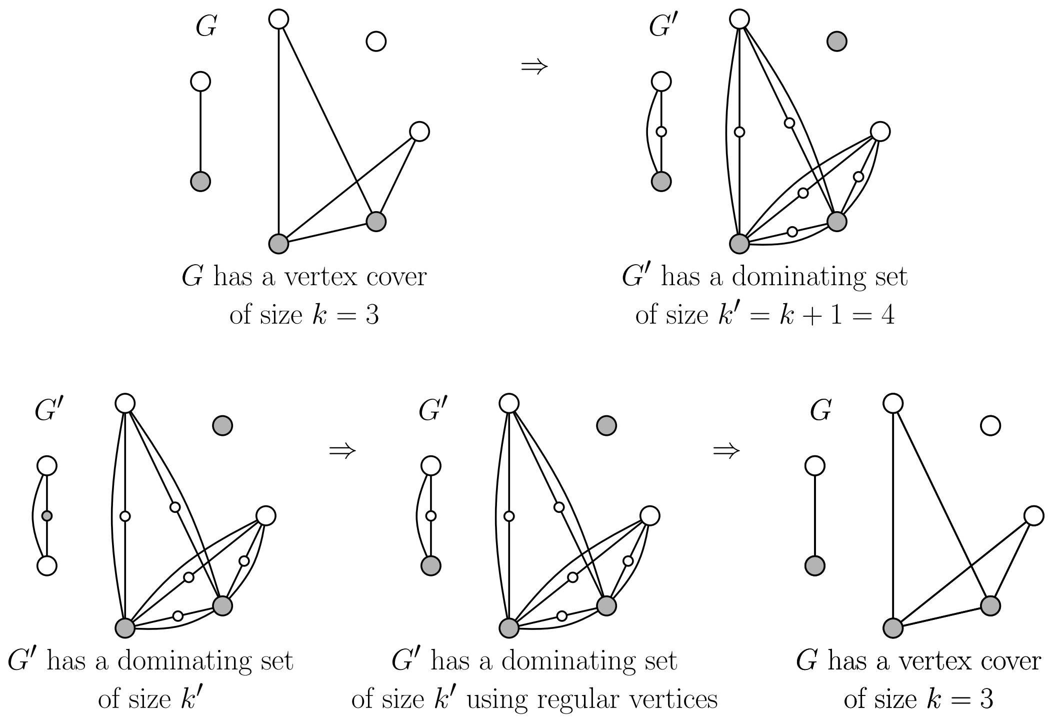
$$|V''| = |V' \cup V_I| \leq k + n_I = k'.$$

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. 88.

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 88.)



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

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

Lecture 25: 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. 89).

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.

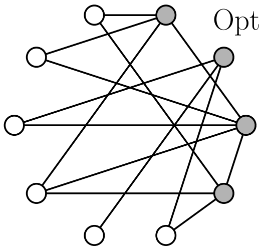


Fig. 89: Vertex cover (optimal solution).

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. 90.)

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: The 2-for-1 approximation for VC 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.

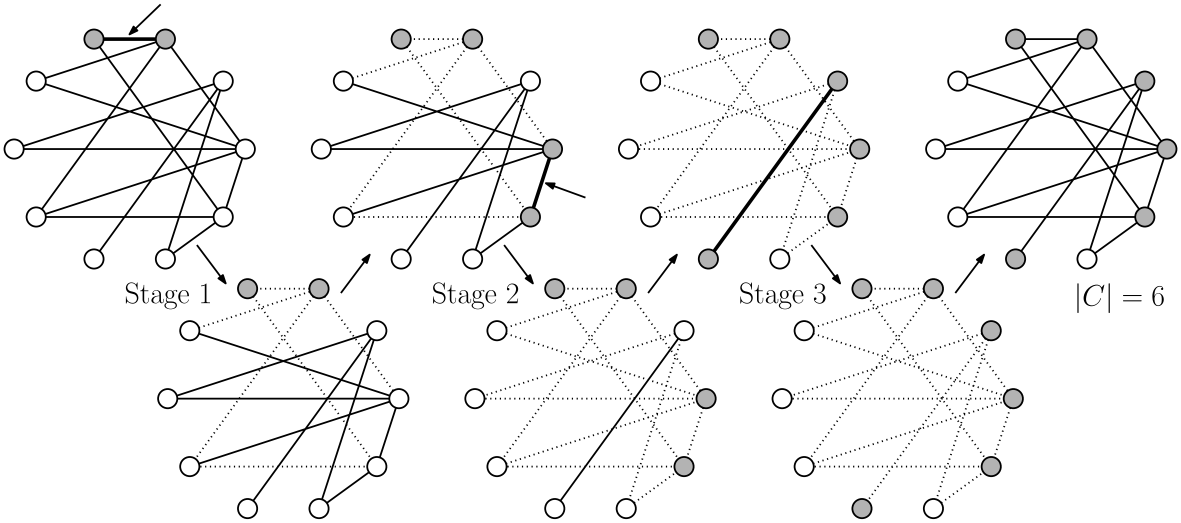


Fig. 90: The 2-for-1 heuristic for vertex cover.

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

The Greedy Heuristic: It seems that there is a very simple way to improve the 2-for-1 heuristic. This algorithm simply selects any edge, and adds both vertices to the cover. Instead, why not concentrate instead on vertices of high degree, since a vertex of high degree covers the maximum number of edges. This is greedy strategy. We saw in the minimum spanning tree and shortest path problems that greedy strategies were optimal.

Here is the greedy heuristic. Select the vertex with the maximum degree. Put this vertex in the cover. Then delete all the edges that are incident to this vertex (since they have been covered). Repeat the algorithm on the remaining graph, until no more edges remain. (This algorithm is given in the code fragment below and is illustrated in Fig. 91.)

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. 91, the greedy heuristic actually

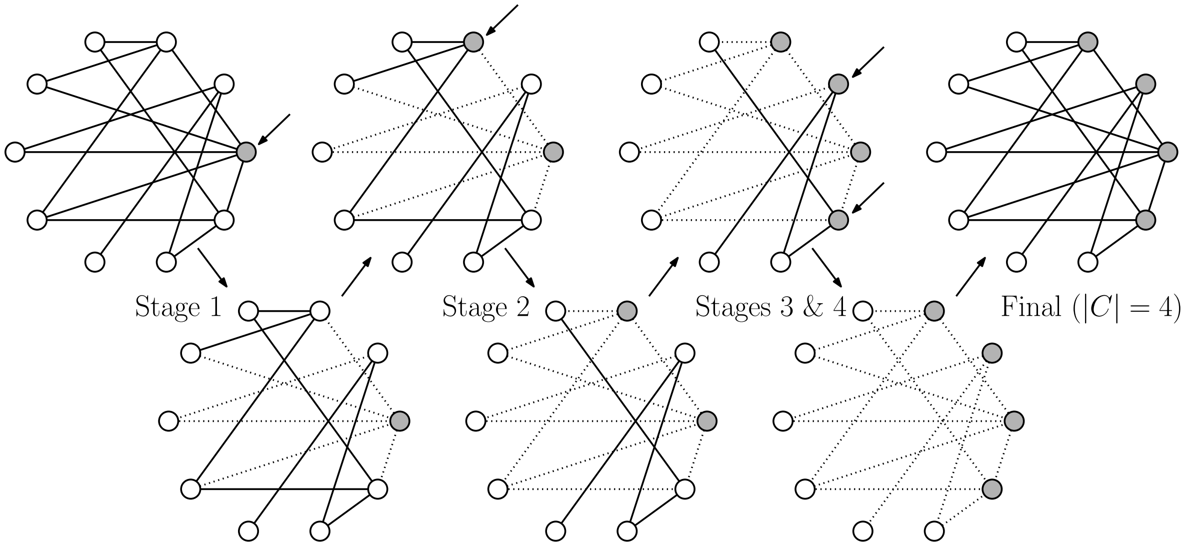


Fig. 91: The greedy heuristic for vertex cover.

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 the complement vertex set, $V \setminus V'$, is an independent set for G. Suppose that G has n vertices, and a minimum vertex cover V' of size k. Then our heuristic is guaranteed to produce a vertex cover V'' that is of size at most $2k$. If we consider the complement set $V \setminus V'$, we know that G has a maximum independent set of size $n-k$. By complementing our approximation $V \setminus V''$ we have an "approximate" independent set of size $n-2k$. How good is this? We achieve a performance ratio 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!

In summary, just because you can approximate one NP-complete problem, it does not follow that you approximate another.

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. 92).

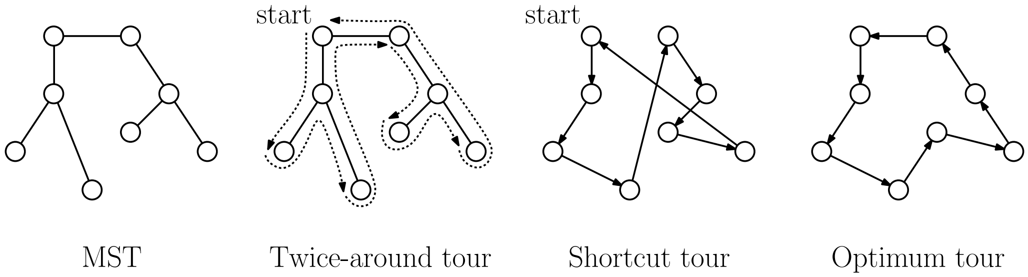


Fig. 92: TSP Approximation.

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

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

as desired.

Supplemental Topics

Lecture 26: Max Dominance

Max-Dominance: Let us consider a natural problem, that arises in a number of financial applications. Suppose you are considering buying a new car. You would like a sporty car with fast acceleration, but you are frugal and also want a car that has good gas mileage. You study various models of cars and for each you record the acceleration and mileage as points on an (x, y) plot. It is not surprising that some cars have excellent mileage but poor acceleration and vice versa. One thing you can determine is that if model A has both lower mileage and lower acceleration than model B, you are not interested in model A. Formally, we say that a point $A = (A.x, A.y)$ is *dominated* by $B = (B.x, B.y)$ if $A.x < B.x$ and $A.y < B.y$. The points that are not dominated by any other point are said to be the *dominant points.* (In economics, this is related to the concept of *Pareto optima.*) The *max dominance problem* is, given a set P of n points (see Fig. (a)), compute the set of dominant points from the set (see Fig. (b)).

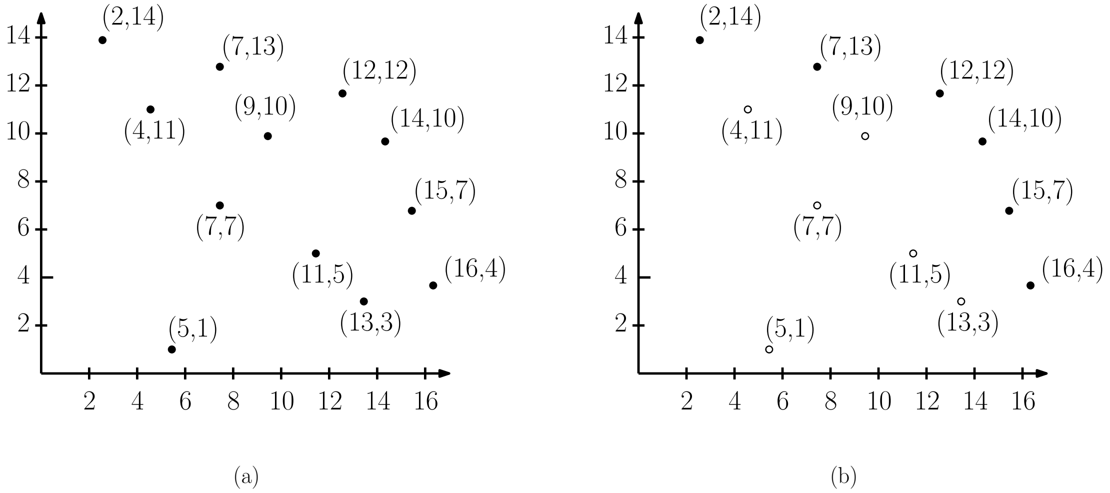


Fig. 93: Max-Dominance.

There is a obvious brute-force algorithm that runs in $O(n^2)$ time, which operates by comparing all pairs of points. The question we consider here 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.

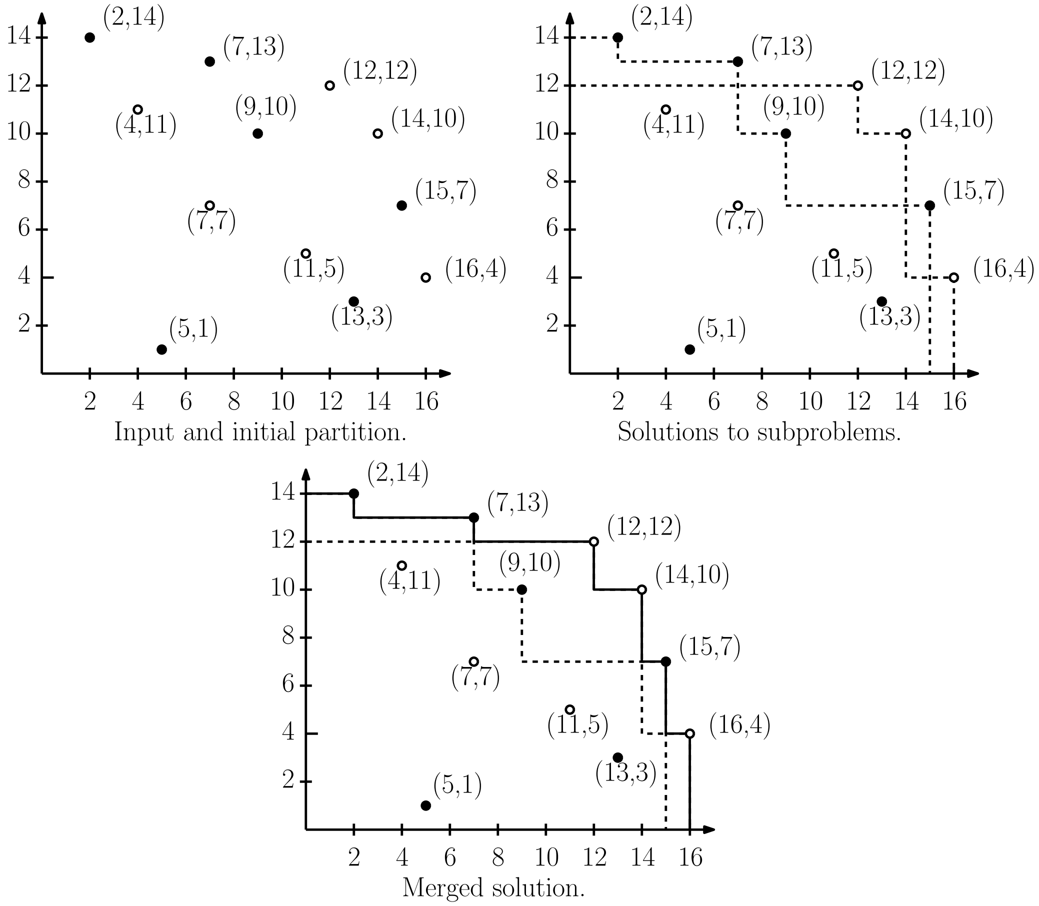
Max Dominance: Divide-and-Conquer

```
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 in Fig. .

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



Input and initial partition.

Solutions to subproblems.

Merged solution.

Fig. 94: Divide and conquer approach.

unprocessed item in each list. Think of these as *fingers*. Take the finger pointing to the point with the smaller x-coordinate. If its y-coordinate is larger than the y-coordinate of the point under the other finger, then this point is maximal, and is copied to the next position of the result list. Otherwise it is not copied. In either case, we move to the next point in the same list, and repeat the process. The result list is returned.

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

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

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

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

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 $L(i) = F_{i+1}$.

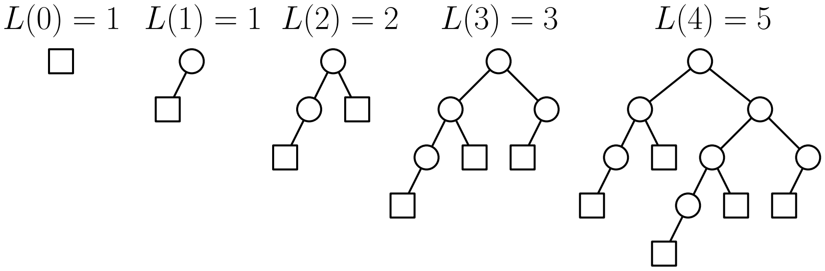


Fig. 95: Minimum-sized AVL trees.

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

Lemma: For all integers $n \ge 1$, $F_n \le \phi^{n-1}$ 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 $F_1 = 1 \le \phi^0$, as desired.

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

Since $n \ge 2$, 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

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

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 $(1+\phi) \le \phi^2$. This is not true for all values of ϕ (for example it is not true when $\phi = 1$ but it is true when $\phi = 2$.)

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

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

By the quadratic formula we have

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

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

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

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

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

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

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

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

Lecture 27: 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:

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

Combining terms we have

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

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

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

This is an exact result, and no guesswork was needed. 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.

Lecture 28: 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, $1 \le k \le n$, 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 (see Fig. 96).

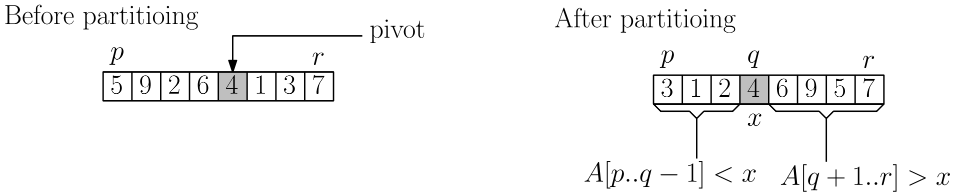


Fig. 96: Partitioning

It is easy to see that the rank of the pivot x is $q - p + 1$ in $A[p..r]$. Let $\text{xRank} = q - p + 1$. If $k = \text{xRank}$, then the pivot is the kth smallest, and we may just return it. If $k < \text{xRank}$, then we know that we need to recursively search in $A[p..q-1]$ and if $k > \text{xRank}$ 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 (see Fig. 97).

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)       // <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
        else
            return Select(A, q+1, r, k-xRank)// select from right
    }
}
```

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 $k = \text{xRank}$ then we get lucky and eliminate everything. Otherwise we either eliminate the pivot and the right subarray or the pivot and the left subarray.

We will discuss the details of choosing the pivot and partitioning later, but assume for now that they both take $\Theta(n)$ time. The question that remains is how many elements did we

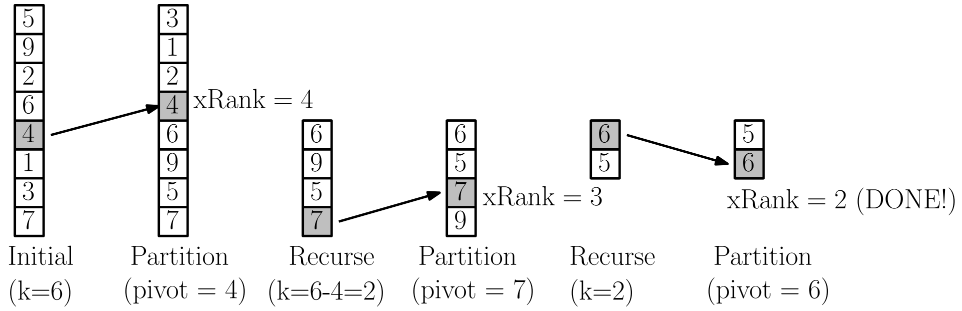


Fig. 97: Selection Algorithm.

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.

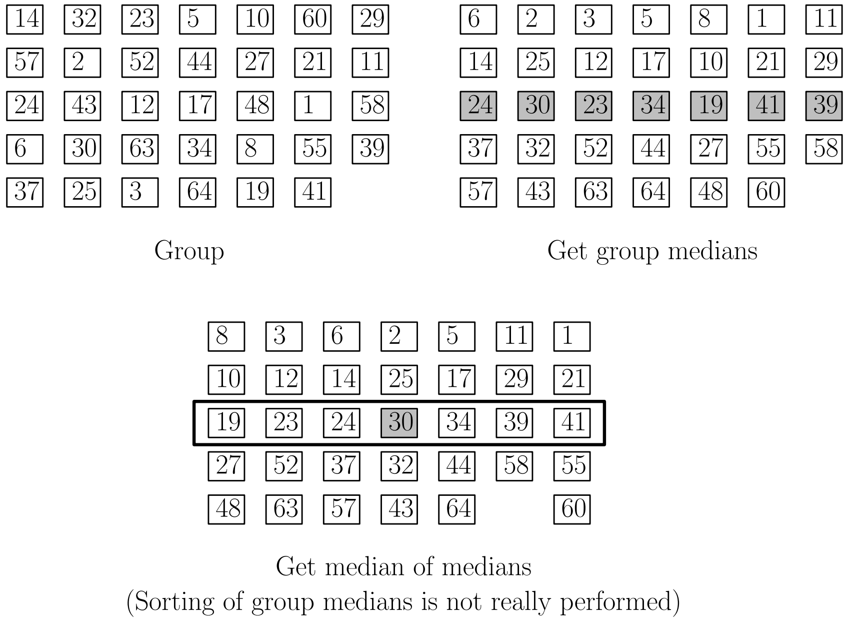
Instead, we will describe a rather complicated method for computing a pivot element that achieves the desired properties. Recall that we are given an array $A[1..n]$, and we want to compute an element x whose rank is (roughly) between $n/4$ and $3n/4$. We will have to describe this algorithm at a very 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.



Group

Get group medians

Get median of medians
(Sorting of group medians is not really performed)

Fig. 98: 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.)

Lecture 29: 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.

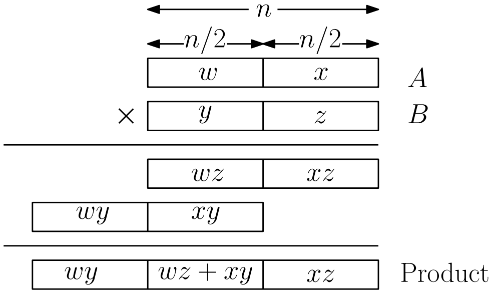


Fig. 99: Long integer multiplication.

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

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

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

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

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 this 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).

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

Finally we have

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

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.

Lecture 30: 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 size(A) denote the number of elements of A.

Basis case: If 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. 100.)

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

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

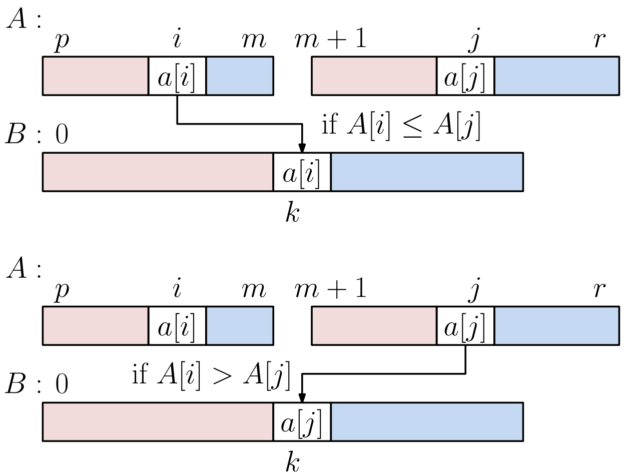


Fig. 100: Merging two sorted lists.

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) {                           // merge A[p..m] and 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 are nonempty
        if (A[i] <= A[j]) B[k++] = A[i++]     // next item from left
        else              B[k++] = A[j++]     // next item from right
    }
    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
}
```

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. 101 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.

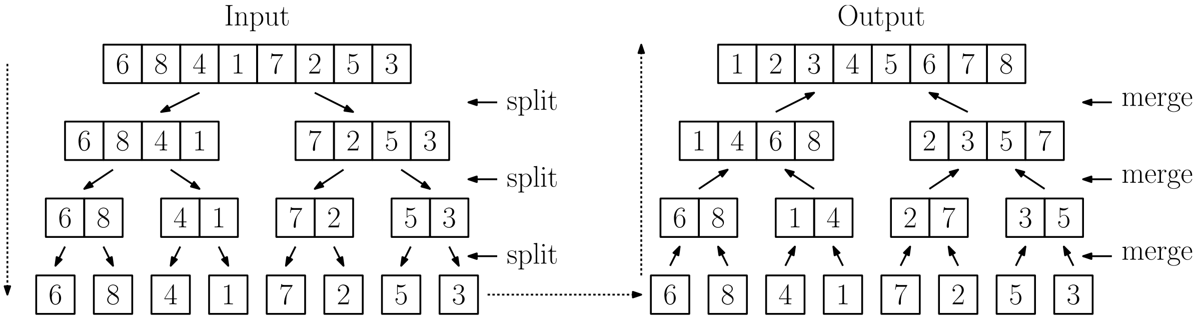


Fig. 101: MergeSort example.

Analysis: Next, let us analyze the running time of MergeSort. First observe that the running time of the procedure 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. 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 $k = \lg n$.[15] By substituting this value for k, we have $T(n/2^k) = T(1) = 1$ and plugging this into the above formula, we obtain

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

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 $T(n) \in \Theta(n^{\log_b a})$

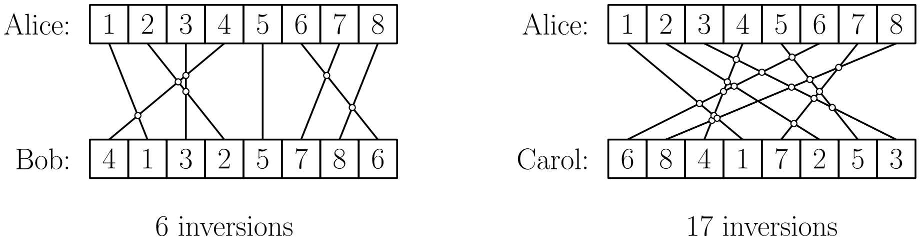
Case 2: if $a = b^k$, then $T(n) \in \Theta(n^k \log n)$

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



6 inversions

17 inversions

Fig. 102: Movie preferences and inversions.

one possible measure of similarity between two lists of n numbers. (An example is shown in in Fig. 102.)

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 $i + 1 \le j \le n$, 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 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. 103).

Whenever $A[i] > A[j]$ the algorithm advances j. It follows, therefore, that if $A[i] \le A[j]$, then every element of the subarray $A[m + 1..j - 1]$ is strictly smaller than $A[i]$. Since the

[15] 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 $k = \lceil \lg n \rceil$.

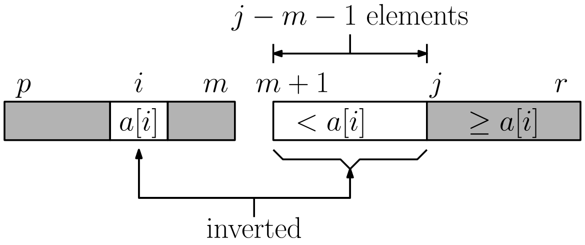


Fig. 103: Counting inversions when $A[i] \leq A[j]$.

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 InvCount$(A, 1, n)$.

This approach is illustrated in Fig. 104. Observe that inversions are counted in the merging process (shown as small white circles in the figure).

Lecture 31: 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. 105(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.

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

```
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 to B
        ct += r - m                 // increment inversion counter
    }
    while (j <= r) B[k++] = A[j++]  // copy extras from right to B

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

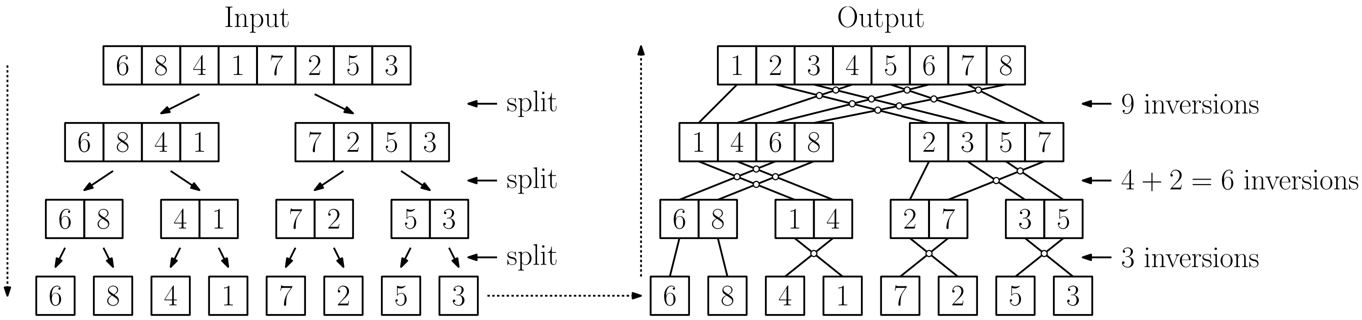


Fig. 104: Inversion counting by divide and conquer.

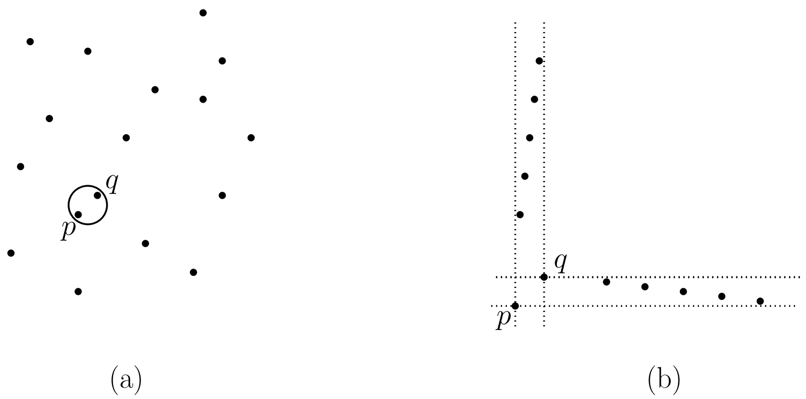


Fig. 105: (a) The closest pair problem and (b) why sorting on x- or y-alone doesn't work.

the point set of Fig. 105(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.[16]

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. 106). Let P_L be the points to the left of ℓ and P_R be the points to the right of ℓ.

[16] While the above example shows that sorting along any one coordinate axis may fail, there is a variant of this strategy that can be used for computing nearest neighbors approximately. This approach is based on the observation that if two points are close together, their projections onto a *randomly oriented vector* will be close, and if they are far apart, their projections onto a randomly oriented vector will be far apart in expectation. This observation underlies a popular nearest neighbor algorithm called *locality sensitive hashing.*

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 each.

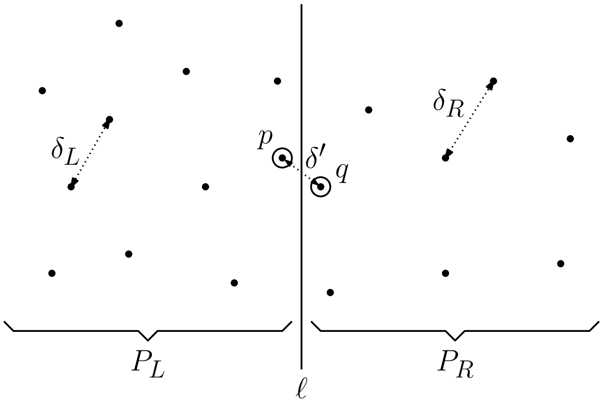


Fig. 106: Divide-and-conquer closest pair algorithm.

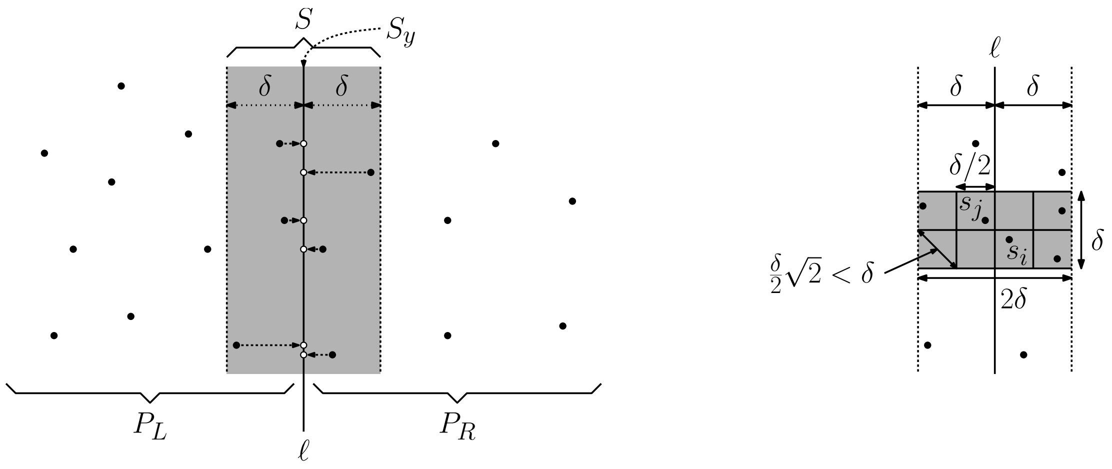
Conquer: Compute the closest pair *within* each of the subsets P_L and P_R each, by invoking the algorithm recursively. Let δ_L and δ_R be the closest pair distances in each case (see Fig. 106). Let $\delta = \min(\delta_L, \delta_R)$.

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. 106). 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, because we already know of the existence of two points within distance δ of each other, this algorithm is allowed to fail, 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. 107(a)).[17]



(a)

(b)

Fig. 107: Closest pair in the strip.

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. 107(a)). At the start of the lecture, we asserted that considering the 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 *must* 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 each 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. 107(b)). Split this rectangle into eight identical squares each 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

[17] 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 generally 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.

pair seen so far. Thus, there can be at most eight points of S in this rectangle, one for each square. Therefore, $|j-i| \leq 7$.

Avoiding Repeated Sorting: 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[18] 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 the value "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.

Lecture 32: 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.

[18] If we were to pay the full sorting cost with each recursive call, the running time would be given by the recurrence $T(n) = 2T(n/2) + n \log n$. Solving this recurrence leads to the solution $T(n) = O(n \log^2 n)$, thus we would miss our target running time by an $O(\log n)$ factor.

Closest Pair in P

```
closestPair(P = (Px, Py)) {
    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’
}
```

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.

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];             // value if we leave i
            if (j >= w[i])                     // enough capacity for i
                take_val = v[i] + V[i-1, j - w[i]]; // 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. 108: 0–1 Knapsack Example.

Lecture 33: 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.

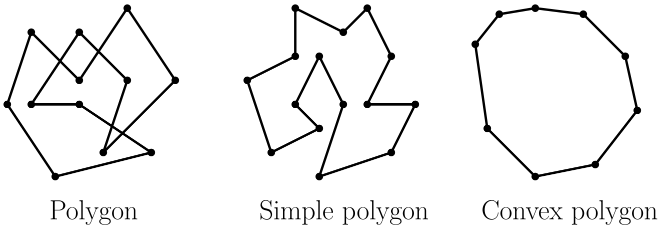


Fig. 109: Polygons.

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. 110.)

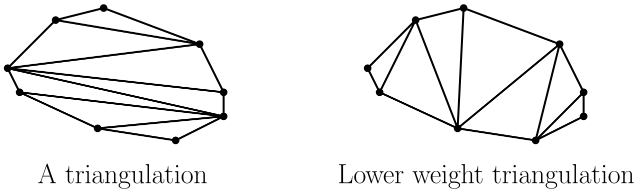


Fig. 110: Triangulations of convex polygons, and the minimum weight triangulation.

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

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

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

Dynamic Programming Solution: Let us consider an $(n+1)$-sided polygon $P = \langle v_0, v_1, \ldots, v_n \rangle$. 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. 111

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.

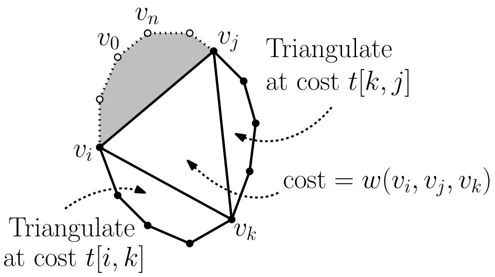

v_n
v_0
v_j
Triangulate at cost $t[k, j]$
v_i
cost $= w(v_i, v_j, v_k)$
Triangulate at cost $t[i, k]$
v_k

Fig. 111: Triangulations and tree structure.

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. 112. Note that every triangle is associated with an internal node of the tree and every edge of the original polygon, except for the distinguished starting side $\overline{v_0 v_n}$, is associated with a leaf node of the tree.

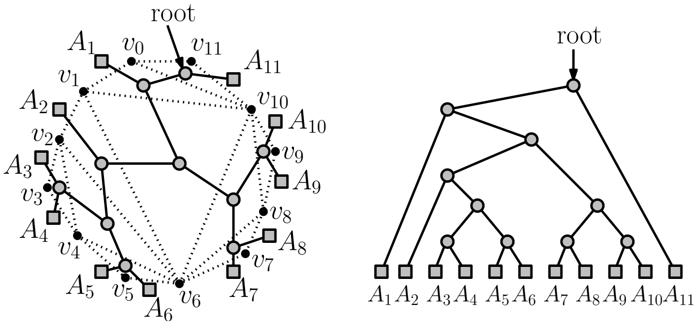


Fig. 112: Triangulations and tree structure.

Once you see this connection. Then the following two observations follow easily. Observe that the associated binary tree has n leaves, and hence (by standard results on binary trees) $n-1$ internal nodes. Since each internal node other than the root has one edge entering it, there are $n-2$ edges between the internal nodes. Each internal node corresponds to one triangle, and each edge between internal nodes corresponds to one chord of the triangulation.

Lecture 34: 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?

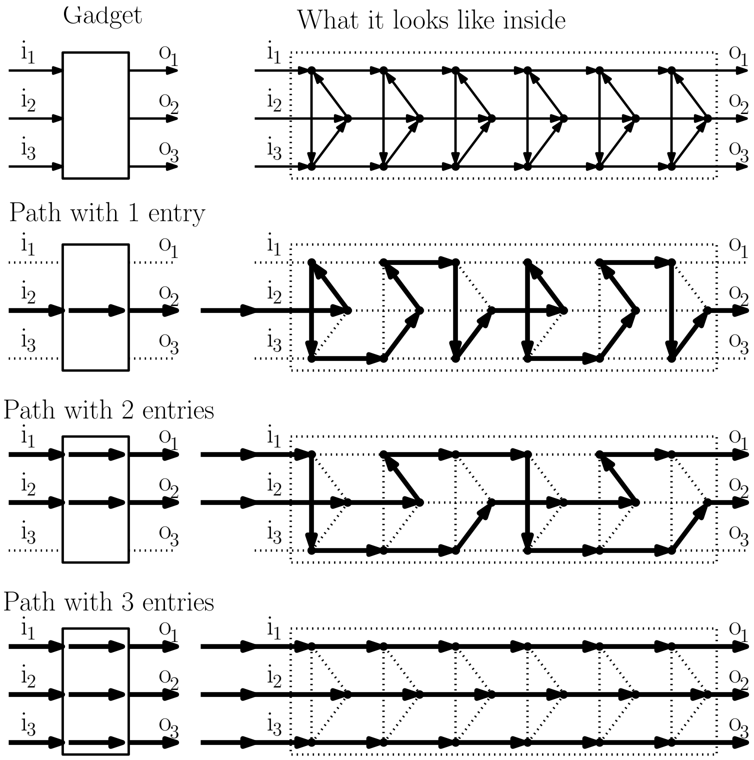


Fig. 113: DHP-Gadget and examples of path traversals.

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.)

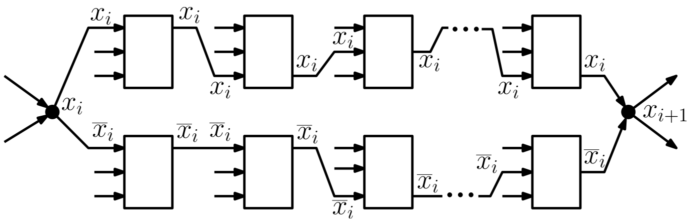


Fig. 114: General structure of reduction from 3SAT to DHP.

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.

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

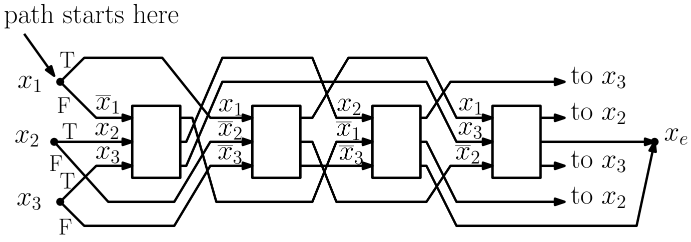
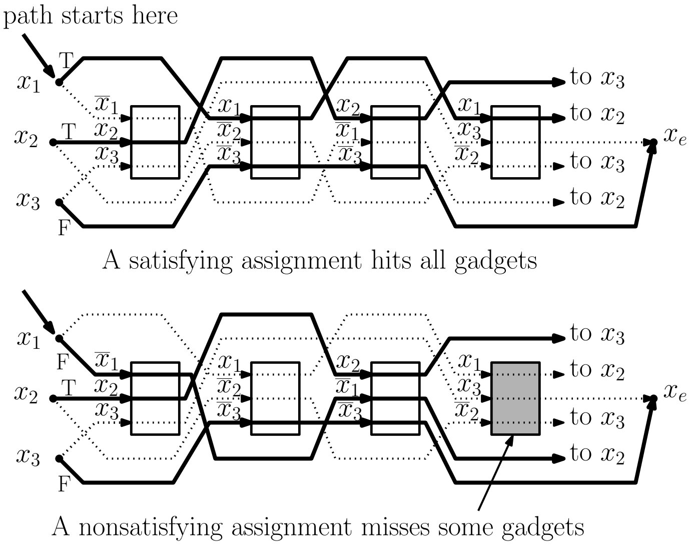


Fig. 115: Example of the 3SAT to DHP reduction.

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. 116: 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".

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

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

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

Also observe that for each variable vertex, the path will proceed along either the true path or the false path. If it proceeds along the true path, set the corresponding variable to 1 and otherwise set it to 0. We will show that the resulting assignment is a satisfying assignment for F.

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

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

Lecture 35: 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.[19]

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.

(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.

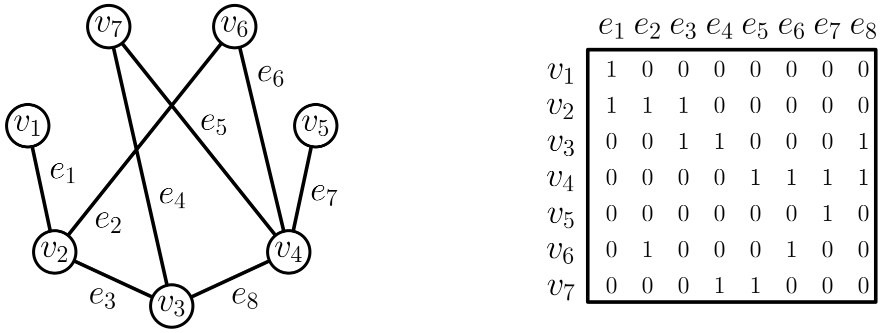
To show that SS is in NP, we need to give a verification procedure. Given S and t, the certificate is just the indices of the numbers that form the subset S'. We can add two b-bit numbers together in $O(b)$ time. So, in polynomial time we can compute the sum of elements in S', and verify that this sum 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.

[19] 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.

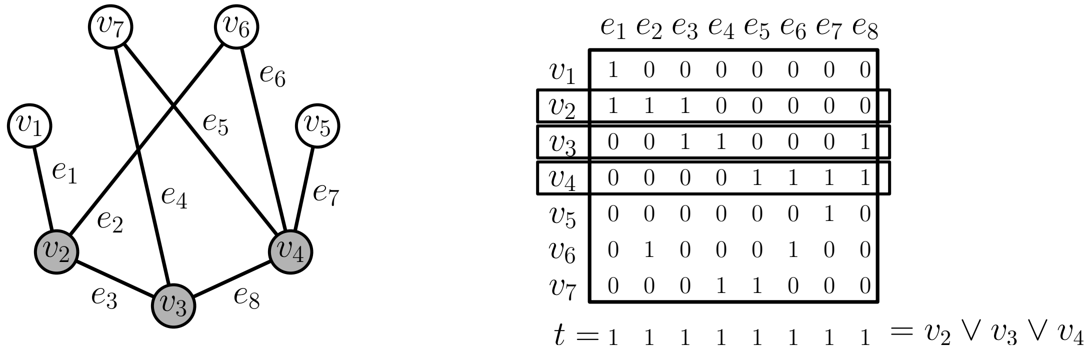
An Initial Approach: Here is an idea, which does not work, but gives a sense of how to proceed. Let E denote the number of edges in the graph. First number the edges of the graph from 1 through E. Then represent each vertex v_i as an E-element bit vector, where the j-th bit from the left is set to 1 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 $k = 3$.



$$\begin{array}{c|cccccccc|}
 & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \\
\hline
v_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
v_3 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
v_4 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
v_5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
v_6 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
v_7 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline
\end{array}$$

Fig. 117: 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.)



$$\begin{array}{c|cccccccc|l}
 & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & \\
\hline
v_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
\boxed{v_2} & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \\
\boxed{v_3} & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & \\
\boxed{v_4} & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \\
v_5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \\
v_6 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & \\
v_7 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & \\
\hline
t = & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & = v_2 \vee v_3 \vee v_4
\end{array}$$

Fig. 118: The logical-or of a vertex cover equals $1111\ldots1$.

Since bit vectors can be thought of as just a way of representing numbers in binary, this is starting to feel more like the subset sum problem. The target would be the number whose bit vector is all 1's. There are a number of problems, however. First, logical-or is not the same as addition. For example, if both of the endpoints of some edge are in the vertex cover, then its value in the corresponding column would be 2, not 1. Second, we have no way of controlling how many vertices go into the vertex cover. (We could just take the logical-or of all the vertices, and then the logical-or would certainly be a bit vectors of 1's.)

There are two ways in which addition differs significantly from logical-or. The first is the issue of carries. For example, the $1101 \vee 0011 = 1111$, but in binary $1101 + 0011 = 1000$. To fix this, we recognize that we do not have to use a binary (base-2) representation. In fact, we can assume any base system we want. Observe that each column of the incidence matrix has at most two 1's in any column, because each edge is incident to at most two vertices. Thus, if use any base that is at least as large as base 3, we will never generate a carry to the next position. In fact we will use base 4 (for reasons to be seen below). Note that the base of the number system is just for own convenience of notation. Once the numbers have been formed, they will be converted into whatever form our machine assumes for its input representation, e.g. decimal or binary.

The second difference between logical-or and addition is that an edge may generally be covered either once or twice in the vertex cover. So, the final sum of these numbers will be a number consisting of 1 and 2 digits, e.g. $1211\ldots112$. 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 $1 \leq i \leq E$, 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 $k \geq 4$). But since this is the last column, it will not affect any of the other aspects of the construction.

The Final Reduction: Here is the final reduction, given the graph $G = (V, E)$ and integer k for the vertex cover problem.

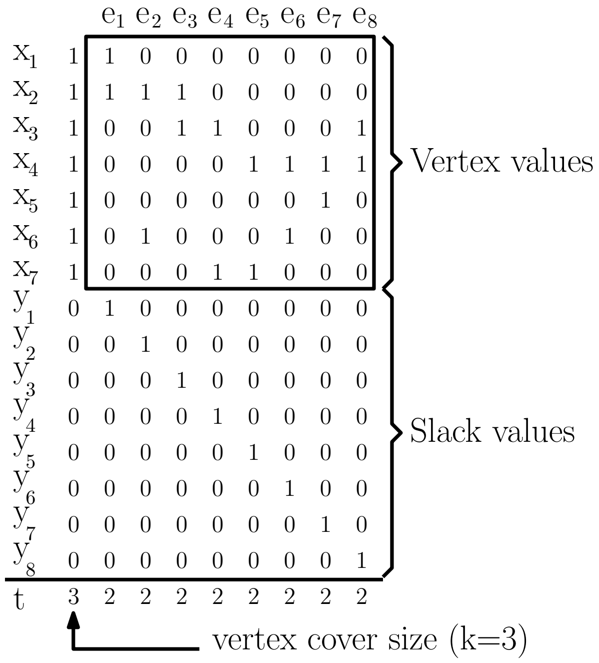
(1) Create a set of n vertex values, $x_1, x_2, \ldots, x_n$ using base-4 notation. The value x_i is equal a 1 followed by a sequence of E base-4 digits. The j-th digit is a 1 if edge e_j is incident to vertex v_i and 0 otherwise.

(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. 119.



		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

Fig. 119: Vertex cover to subset sum reduction.

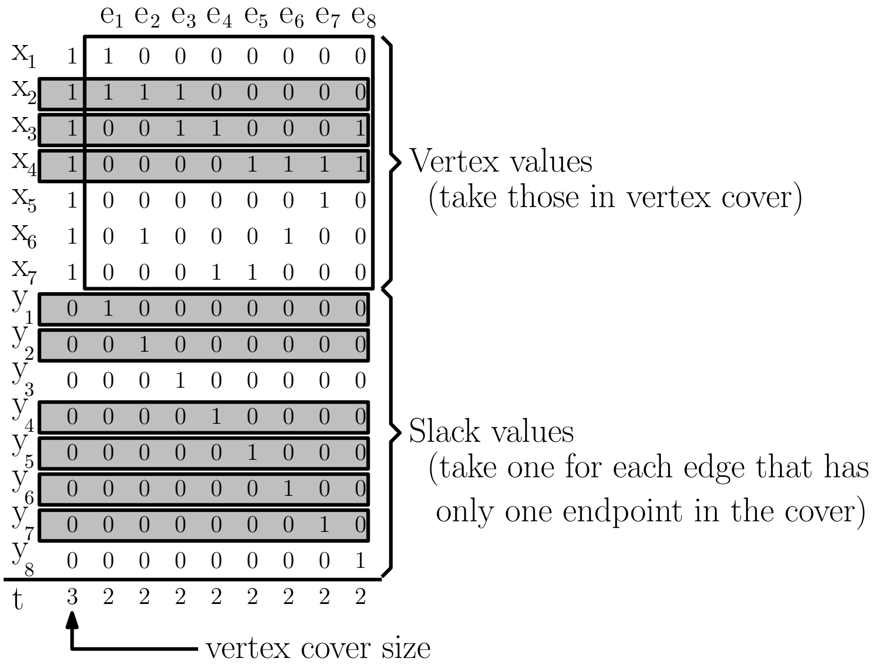
Correctness: We claim that G has a vertex cover of size k if and only if S has a subset that sums to t. If G has a vertex cover V' of size k, then we take the vertex values x_i corresponding to the vertices of V', and for each edge that is covered only once in V', we take the corresponding slack variable. 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.

Lecture 36: 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



		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. 120: Correctness of the reduction.

approximation. This is done by specifying a parameter $\epsilon > 0$ as part of the input to the approximation algorithm, and requiring that the algorithm produce an answer that is within a *relative error* of ϵ of the optimal solution. It is understood that as ϵ tends to 0, the running time of the algorithm will increase. Such an algorithm is called a *polynomial approximation scheme*.

For example, the running time of the algorithm might be $O(2^{(1/\epsilon)}n^2)$. It is easy to see that in such cases the user pays a big penalty in running time as a function of ϵ. (For example, to produce a 1% error, the "constant" factor would be 2^{100} which would be around 4 quadrillion centuries on your 100 Mhz Pentium.) A *fully polynomial approximation scheme* is one in which the running time is polynomial in both n and $1/\epsilon$. For example, a running time of $O((n/\epsilon)^2)$ would satisfy this condition. In such cases, reasonably accurate approximations are computationally feasible.

Unfortunately, there are very few NP-complete problems with fully polynomial approximation schemes. In fact, recently there has been strong evidence that many NP-complete problems do not have polynomial approximation schemes (fully or otherwise). Today we will study one that does.

Subset Sum: Recall that in the subset sum problem we are given a set S of positive integers $\{x_1, x_2, \ldots, x_n\}$ and a target value t, and we are asked whether there exists a subset $S' \subseteq S$ that sums exactly to t. The optimization problem is to determine the subset whose sum is as large as possible but not larger than t.

This problem is basic to many packing problems, and is indirectly related to processor scheduling problems that arise in operating systems as well. Suppose we are also given $0 < \epsilon < 1$.

Let $z^* \leq t$ denote the optimum sum. The approximation problem is to return a value $z \leq t$ such that

$$z \geq z^*(1 - \epsilon).$$

If we think of this as a knapsack problem, we want our knapsack to be within a factor of $(1 - \epsilon)$ of being as full as possible. So, if $\epsilon = 0.1$, then the knapsack should be at least 90% as full as the best possible.

What do we mean by polynomial time here? Recall that the running time should be polynomial in the size of the input length. Obviously n is part of the input length. But t and the numbers x_i could also be huge binary numbers. Normally we just assume that a binary number can fit into a word of our computer, and do not count their length. In this case we will to be on the safe side. Clearly t requires $O(\log t)$ digits to be store in the input. We will take the input size to be $n + \log t$.

Intuitively it is not hard to believe that it should be possible to determine whether we can fill the knapsack to within 90% of optimal. After all, we are used to solving similar sorts of packing problems all the time in real life. But the mental heuristics that we apply to these problems are not necessarily easy to convert into efficient algorithms. Our intuition tells us that we can afford to be a little "sloppy" in keeping track of exactly full the knapsack is at any point. The value of ϵ tells us just how sloppy we can be. Our approximation will do something similar. First we consider an exponential time algorithm, and then convert it into an approximation algorithm.

Exponential Time Algorithm: This algorithm is a variation of the dynamic programming solution we gave for the knapsack problem. Recall that there we used an 2-dimensional array to keep track of whether we could fill a knapsack of a given capacity with the first i objects. We will do something similar here. As before, we will concentrate on the question of which sums are possible, but determining the subsets that give these sums will not be hard.

Let L_i denote a list of integers that contains the sums of all 2^i subsets of $\{x_1, x_2, \ldots, x_i\}$ (including the empty set whose sum is 0). For example, for the set $\{1, 4, 6\}$ the corresponding list of sums contains $\langle 0, 1, 4, 5(= 1 + 4), 6, 7(= 1 + 6), 10(= 4 + 6), 11(= 1 + 4 + 6) \rangle$. Note that L_i can have as many as 2^i elements, but may have fewer, since some subsets may have the same sum.

There are two things we will want to do for efficiency. (1) Remove any duplicates from L_i, and (2) only keep sums that are less than or equal to t. Let us suppose that we a procedure `MergeLists(L1, L2)` which merges two sorted lists, and returns a sorted lists with all duplicates removed. This is essentially the procedure used in MergeSort but with the added duplicate element test. As a bit of notation, let $L + x$ denote the list resulting by adding the number x to every element of list L. Thus $\langle 1, 4, 6 \rangle + 3 = \langle 4, 7, 9 \rangle$. This gives the following procedure for the subset sum problem.

For example, if $S = \{1, 4, 6\}$ and $t = 8$ then the successive lists would be

$$\begin{aligned}
L_0 &= \langle 0 \rangle \\
L_1 &= \langle 0 \rangle \cup \langle 0 + 1 \rangle = \langle 0, 1 \rangle \\
L_2 &= \langle 0, 1 \rangle \cup \langle 0 + 4, 1 + 4 \rangle = \langle 0, 1, 4, 5 \rangle \\
L_3 &= \langle 0, 1, 4, 5 \rangle \cup \langle 0 + 6, 1 + 6, 4 + 6, 5 + 6 \rangle = \langle 0, 1, 4, 5, 6, 7, 10, 11 \rangle.
\end{aligned}$$

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;
}
```

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.

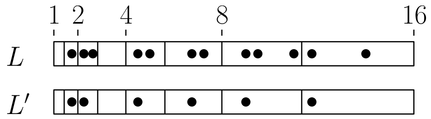


Fig. 121: Trimming Lists for Approximate Subset Sum.

Claim: The number of distinct items in a trimmed list is $O((n \log t)/\epsilon)$, which is polynomial in input size and $1/\epsilon$.

Proof: We know that each pair of consecutive elements in a trimmed list differ by a ratio of at least $d = 1/(1-\delta) > 1$. Let k denote the number of elements in the trimmed list, ignoring the element of value 0. Thus, the smallest nonzero value and maximum value in the trimmed list differ by a ratio of at least d^{k-1}. Since the smallest (nonzero) element is at least as large as 1, and the largest is no larger than t, then it follows that $d^{k-1} \leq t/1 = t$. Taking the natural log of both sides we have $(k-1)\ln d \leq \ln t$. Using the facts that $\delta = \epsilon/n$ and the log identity that $\ln(1+x) \leq x$, we have

$$\begin{aligned} k - 1 &\leq \frac{\ln t}{\ln d} = \frac{\ln t}{-\ln(1-\delta)} \\ &\leq \frac{\ln t}{\delta} = \frac{n \ln t}{\epsilon} \\ k &= O\left(\frac{n \log t}{\epsilon}\right). \end{aligned}$$

Observe that the input size is at least as large as n (since there are n numbers) and at least as large as $\log t$ (since it takes $\log t$ digits to write down t on the input). Thus, this function is polynomial in the input size and $1/\epsilon$.

The approximation algorithm operates as before, but in addition we call the procedure `Trim` given below.

For example, consider the set $S = \{104, 102, 201, 101\}$ and $t = 308$ and $\epsilon = 0.20$. We have $\delta = \epsilon/4 = 0.05$. An example of the algorithm's execution is shown in Fig. 122.

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

```
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;
}
```

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

$$
\begin{array}{rrcl}
\text{init:} & L_0 & = & \langle 0 \rangle \\
\\
\text{merge:} & L_1 & = & \langle 0, 104 \rangle \\
\text{trim:} & L_1 & = & \langle 0, 104 \rangle \\
\text{remove:} & L_1 & = & \langle 0, 104 \rangle \\
\\
\text{merge:} & L_2 & = & \langle 0, 102, 104, 206 \rangle \\
\text{trim:} & L_2 & = & \langle 0, 102, 206 \rangle \\
\text{remove:} & L_2 & = & \langle 0, 102, 206 \rangle \\
\\
\text{merge:} & L_3 & = & \langle 0, 102, 201, 206, 303, 407 \rangle \\
\text{trim:} & L_3 & = & \langle 0, 102, 201, 303, 407 \rangle \\
\text{remove:} & L_3 & = & \langle 0, 102, 201, 303 \rangle \\
\\
\text{merge:} & L_4 & = & \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle \\
\text{trim:} & L_4 & = & \langle 0, 101, 201, 302, 404 \rangle \\
\text{remove:} & L_4 & = & \langle 0, 101, 201, 302 \rangle
\end{array}
$$

Fig. 122: Subset-sum approximation example.

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

$$(1 - \epsilon/n)^{i-1}(y + x_i) \leq z + x_i \leq y + x_i.$$

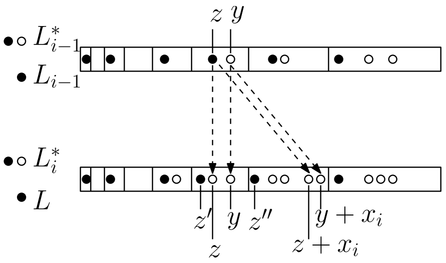


Fig. 123: Subset sum approximation analysis.

The items z and $z + x_i$ might not appear in L_i because they may be trimmed. Let z' and z'' be their respective representatives. Thus, z' and z'' are elements of L_i. We have

$$\begin{aligned} (1 - \epsilon/n) z &\leq z' \leq z \\ (1 - \epsilon/n)(z + x_i) &\leq z'' \leq z + x_i. \end{aligned}$$

Combining these with the inequalities above we have

$$\begin{aligned} (1 - \epsilon/n)^{i-1}(1 - \epsilon/n) y \;\leq\; (1 - \epsilon/n)^i y &\leq z' \leq y \\ (1 - \epsilon/n)^{i-1}(1 - \epsilon/n)(y + x_i) \;\leq\; (1 - \epsilon/n)^i (y + x_i) &\leq z'' \leq z + y_i. \end{aligned}$$

Since z and z'' are in L_i this is the desired result. This ends the proof of the claim.

Using our claim, and the fact that Y^* (the optimum answer) is the largest element of L^*_n and Y (the approximate answer) is the largest element of L_n we have

$$(1 - \epsilon/n)^n Y^* \leq Y \leq Y^*.$$

This is not quite what we wanted. We wanted to show that $(1 - \epsilon) Y^* \leq Y$. To complete the proof, we observe from the lemma above (setting $a = -\epsilon$) that

$$(1 - \epsilon) \leq \left(1 - \frac{\epsilon}{n}\right)^n.$$

This completes the approximate analysis.

Lecture 37: Approximations: Bin Packing

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. 124. 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

$$\frac{b_{\text{ff}}}{b_{\text{opt}}} \leq 2.$$

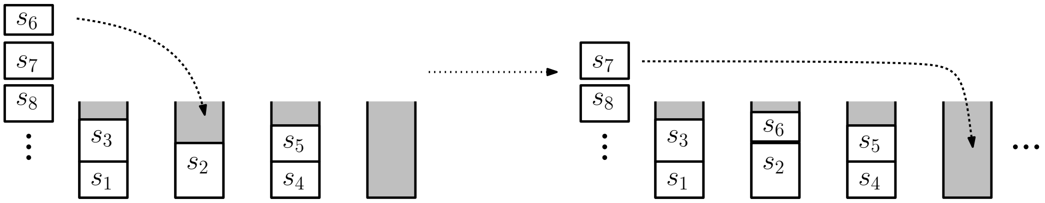


Fig. 124: First-fit Heuristic.

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 $11/9 \approx 1.222$.