CMSC 451
Design and Analysis of Computer Algorithms[1]

David M. Mount
Department of Computer Science
University of Maryland
Fall 2025

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. We can define *algorithm* to be:

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

The concept of a "well-defined computational procedure" dates back to ancient times. In fact, the word "algorithm" is derived from the Latin form of the Persian scholar Muhammad ibn Musa al-Khwarizmi, who lived in ninth century A.D. Al-Khwarizmi codified procedures for numerous arithmetic operations (such as addition, multiplication, and division with Arabic numerals) and algebraic and trigonometric operations (such as computing square roots and computing the digits of π).

Why study algorithm design? While the study of algorithms predates digital computers, the field really took off with the advent of computers. The use of asymptotic (big-Oh) notation became popular in the 1960's and 1970's as a means to provide a rigorous mathematical measure of an algorithm's running time. This evolved into the field of *computational complexity*, which seeks to categorize computational problems according to their complexity. This gave rise to the study of NP-Hard problems.

The field has also led to the development of general techniques for the design of efficient algorithms, such as divide-and-conquer, greedy algorithms, dynamic programming, and so on.

From a more practical perspective, algorithm design and analysis is often the first step the development of software for tricky combinatorial problems. Asymptotic analysis is used to identify promising solutions, which can then be prototyped in order to determine which methods perform best.

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 we need to be concerned both with its *correctness* and *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*, which is usually represented by the symbol n. For example, in a sorting algorithm, this might be the number of items to be sorted. In a graph algorithm, this might be the number of vertices and/or edges in the graph. In a numerical algorithm like factoring, this might be the number of digits in a number. Since there are many inputs of the same input size, there are two common ways to aggregate these into a single quantity.

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.

Asymptotic 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. Suppose that we analyze two algorithms, and find that they have running times of:

$$T_1(n) \;=\; 3.9n + 4.17\log n + 3.5n^2 \quad \text{and} \quad T_2(n) \;=\; \max(4.6n(\log n)^4, 6.4n^3 - 3n\log_3 n).$$

Which of these algorithms is better? Asymptotic analysis is based on (1) focusing on the growth rate by considering the performance as the value of n increases to infinity and (2) ignoring constant factors, which tend to rely on secondary issues such as programming style and machine architecture. We'll give the formal definition later, but intuitively we can say that T_1 grows on the order of n^2 and T_2 grows on the order of n^3, that is

$$T_1(n) \;=\; O(n^2) \quad \text{and} \quad T_2(n) \;=\; O(n^3).$$

Here's a formal definition:

Asymptotic noation: A function $f(n)$ 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 $n \geq n_0$.

Intuitively, 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.

Another (and often easier) way to think about asymptotics is in terms of limits. An alternative definition is that $f(n)$ is $O(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c, \quad \text{for some constant } c \geq 0.$$

For example, we can say that $T_1(n)$ is $O(n^2)$ since

$$\lim_{n\to\infty} \frac{3.9n + 4.17\log n + 3.5n^2}{n^2} \;=\; \lim_{n\to\infty} \left(3.9\frac{1}{n} + 4.17\frac{\log n}{n^2} + 3.5\right) \;=\; 3.5,$$

since in the limit $1/n$ and $\log n/n^2$ both tend to zero in the limit.

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, not ∞, and not depending on n). Intuitively, each represents a form of "asymptotic relational operator":

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

By far, the most commonly arising functions in algorithm analysis are of one of three forms. They are:

Polylogarithmic: Of the form $(\log n)^a = \log^a n$, for some constant a,

Polynomial: Of the form n^a, for some constant a, and

Exponential: Of the form a^n, for some constant a.

(There are, of course, many functions that do not fit into any of these categories, such as $O(n^{\log n})$.)

For any a, b, c, such that $a, b > 0$ and $c > 1$ we have the following relative order:

$$(\log n)^a \prec n^b \prec c^n.$$

To keep matters simple, we will focus almost exclusively on **worst-case analysis** measured using **asymptotic 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. Also, sometimes one algorithm will have a better asymptotic complexity, but the constant factors are so large that there is no practical value of n where its running time is better, such as $9999n$ versus $2n \log n$.

Describing Algorithms: Throughout out this course, when you will be asked to present an algorithm. This means that you need to do three things:

Present the Algorithm: Give 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, and **not** a compiler. Obvious technical details should be kept to a minimum so that the key computational issues stand out.

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: We will assume that you have familiarity with the information from an introductory algorithms course. 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, CountingSort, and RadixSort) and basic graph algorithms (minimum spanning trees and depth-first search).
- Basic calculus (manipulation of exponentials, logarithms, differentiation, and integration).

Topics to be Covered: Here is a tentative list of topics to be covered in this course.

Introduction: Review of algorithm design and analysis, review of basic graph theory and graph representations.

Graph Exploration: DFS and BFS, topological sorting, strong components, shortest paths.

Greedy Algorithms: Greedy scheduling algorithms, clustering and Gonzalez's algorithm, greedy set cover.

Dynamic Programming: Weighted interval scheduling, longest common subsequences and edit distance, chain matrix multiplication, all pairs shortest paths in graphs (Floyd-Warshall).

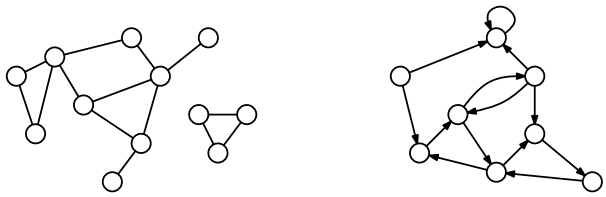
Network Flow: Network flow algorithms, bipartite matching, circulations and applications.

NP-Hardness and Intractability: Basic definitions, polynomial-time reductions, examples of NP-complete problems (3SAT, Independent Set, Clique, Vertex Cover, Dominating Set, and Hamiltonian Cycle).

Approximation Algorithms: Approximation ratio. Examples of approximation algorithms (vertex cover, travelling salesman, subset sum).

Lecture 2: Graph Basics and Depth-First Search

Graphs and Digraphs: 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. 1). 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.



(Undirected) Graph Digraph

Fig. 1: 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 out-deg(v), and the number of edges coming in is called its *in-degree*, denoted in-deg(v). In an undirected graph we just talk about the *degree* of a vertex as the number of incident edges, denoted deg(v).

When discussing the size of a graph, we typically consider both the number of vertices and the number of edges. The number of vertices is typically written as n, and the number of edges is written as m. (**Beware:** There are many different conventions. The number of vertices may be expressed as n, v, V, or $|V|$, and the number of edges may be expressed as m, e, E, or $|E|$.

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: $0 \leq m \leq \binom{n}{2} = n(n-1)/2 = O(n^2)$.
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 may revisit the same vertex multiple times. 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. In an undirected graph, the sequence $\langle v_0, v_1, v_0 \rangle$ is *not* a simple cycle (even if this edge exists), but in a directed graph, it is considered to be a simple cycle (assuming both edges exist) since these are separate edges.

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. 2). (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. 2).

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. 2). Note that every cycle in a bipartite graph contains an even number of edges.

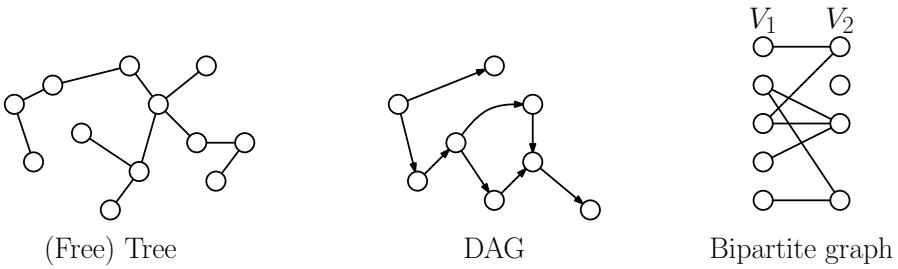


Fig. 2: Illustration of common graph terms.

We say that w is *reachable* from u if there is a path from u to w. Note that every vertex is reachable from itself by a trivial path that uses zero edges. An undirected graph is *connected* if every vertex can reach every other vertex. (Connectivity is a bit messier for digraphs, and we will define it later.) The subsets of mutually reachable vertices partition the vertices of the graph into disjoint subsets, called the *connected components* of the graph. 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. 3.) 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. 3).

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. 4). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

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

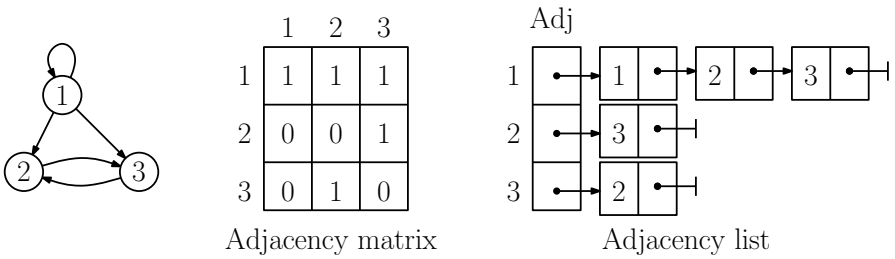


Fig. 3: Adjacency matrix and adjacency list for digraphs.

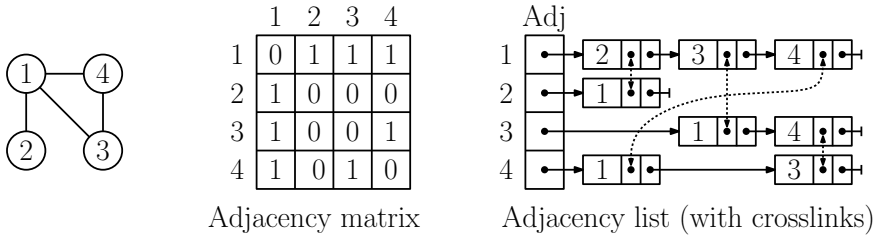


Fig. 4: Adjacency matrix and adjacency list for graphs.

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.

Depth-First Search: One of the most important basic operations on a graph is to systematically visit all its vertices. These traversals natually 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.

We are given a graph $G = (V, E)$, which may be directed or undirected. We employ four auxiliary arrays. To avoid revisiting the same vertex, 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.

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.

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. 5 illustrates the execution on an undirected graph, and Fig. 6 shows an example on a directed graph.)

Analysis: The running time of DFS is $O(n+m)$. We'll do the analysis for undirected graphs. 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

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
}
```

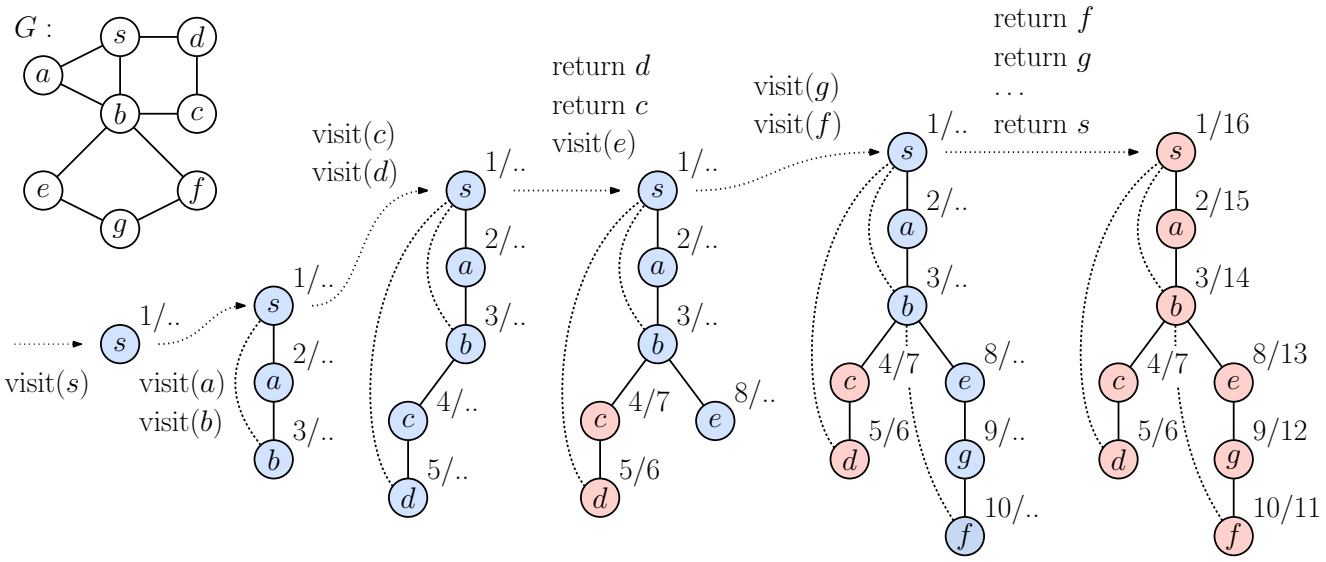


Fig. 5: Depth-first search on an undirected graph. (Blue nodes are discovered, and pink nodes are finished. Each node u is labeled with the values $d[u]/f[u]$.)

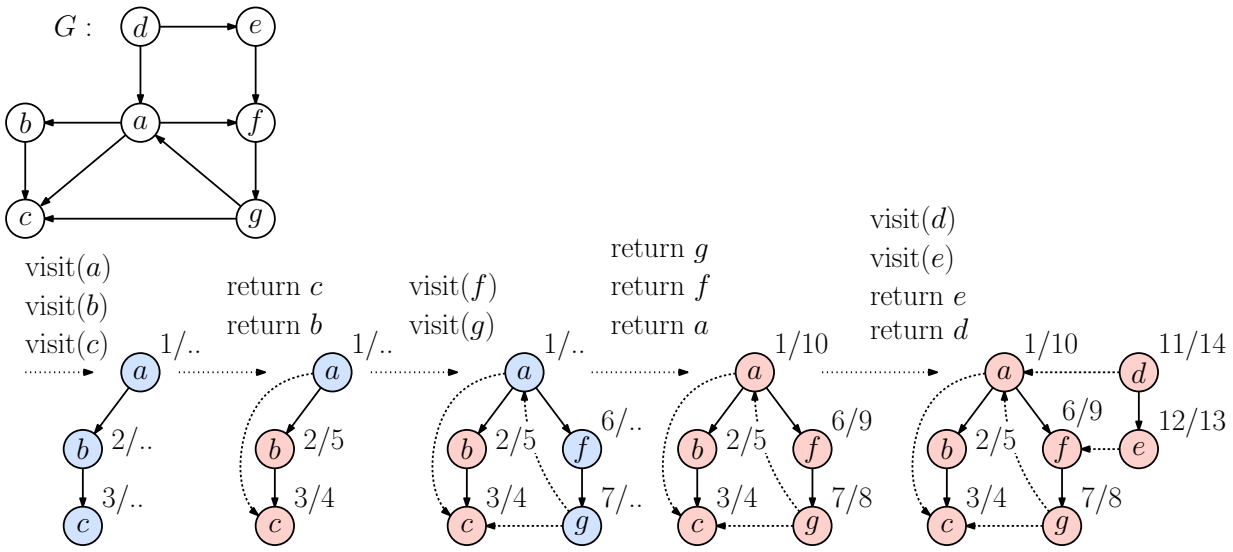


Fig. 6: Depth-first search on a directed graph. (Blue nodes are discovered, and pink nodes are finished. Each node u is labeled with the values $d[u]/f[u]$.)

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

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

A similar analysis holds if we consider DFS for digraphs.

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. 7(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.

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. 7(b)):

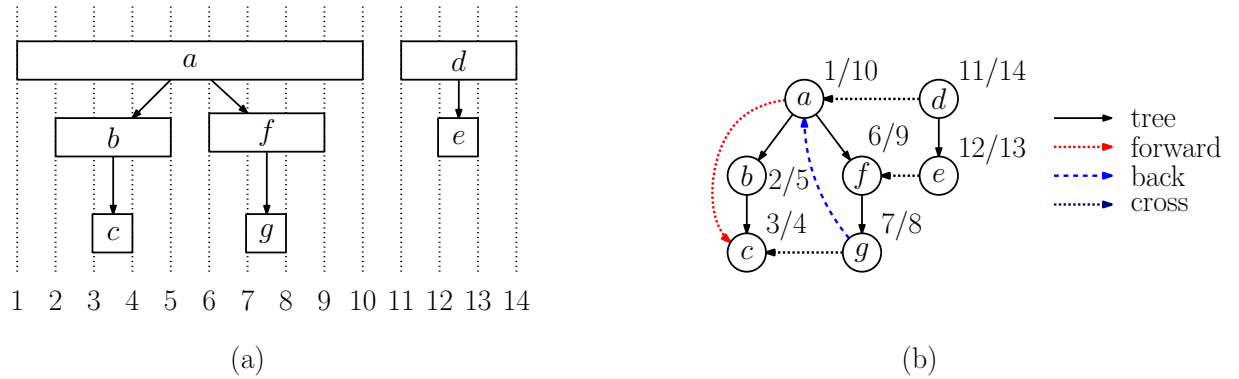


Fig. 7: (a) The Parenthesis Lemma and (b) the DFS edge types.

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 non-child, 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.)[2]

Lecture 3: Cycles and Strong Components

Directed Acyclic Graphs: A *directed acyclic graph*, or *DAG*, is a directed graph that has no cycles (see Fig. 8). 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.

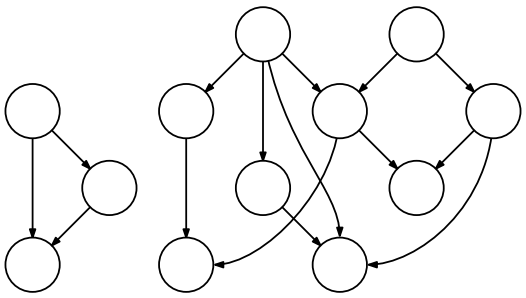


Fig. 8: Directed acyclic graph (DAG).

It is easy to see that every DAG must have at least one vertex with no incoming edges, and at least one vertex with no outgoing edges. A vertex with no incoming edges (only outgoing) is called a *source* and a vertex with no outgoing edges (only incoming) is called a *sink*.

[2]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.

Acylicity Testing: Let us consider the problem of determining whether a digraph is acyclic. We are given a directed graph $G = (V, E)$, and we with to determine whether G contains a cycle. If so, G is not a DAG.

We will present a simple algorithm based on DFS. Recall that in addition to tree edges, a DFS forest contains three other types of edges, back edges (which go to a vertex's ancestor), forward edges (which go to a vertex's descendant), and cross edges (everything else). Observe that if the DFS forest of G contains at least one back edge, then G has a cycle. This is easy to see. If (u, v) is a back edge, then there is a path in the tree from the ancestor v to the descendant u, and the back edge from u to v completes the cycle. The following lemma shows that this condition is not only sufficient, but necessary.

Claim: If a digraph G has a cycle, then *any* DFS forest of G (i.e., no matter what order the vertices are visited) has a *back edge*.

Proof: The proof is based on a very simple observation about the various edge types and finish times. Recall from the Parenthesis Lemma (from the previous lecture) that if u is an ancestor of v then we have $[d[v], f[v]] \subset [d[u], f[u]]$. It follows that if (u, v) is a tree edge or forward edge then $f[u] > f[v]$. Also, observe that (u, v) is a cross edge, it must be u was discovered after v was finished (for otherwise, u would have made a DFSvisit call on v, implying that this would be a tree edge). Therefore $d[u] > f[v]$. Since a vertex cannot finish until after it was discovered, we have $f[u] > f[v]$.

In summary, for all these three edge types (tree, forward, and cross), the finish time of the origin is strictly greater than the finish time of the destination. It follows directly that it is impossible to complete a cycle from any combination of just these three edge types. Only for back edges do we have $f[u] < f[v]$, and therefore in order to form a cycle we need to have *at least one* back edge. Therefore, if a graph G has a cycle, in any DFS forest of G there must be at least one back edge.

The above theorem implies that in order to determine whether a graph G has a cycle, it suffices to test whether it has a back edge. How do we know whether an edge is a back edge. The proof of the above theorem provides an easy way. We can first apply DFS to G, and we then run through the edges, checking whether $d[u] > d[v]$. Can we do this on the fly as DFS is running? The answer is yes. Observe that a back edge goes from a vertex u to an ancestor v. Such an ancestor must have been discovered, but not yet finished. For the other non-tree edge types, the destination v will have already finished. The main DFS function is the same, only DFSvisit needs to be updated.

Determining whether a graph has a cycle

```
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
        }
        else if (mark[v] != finished) {      // equivalent to f[u] <= f[v]
            output "cycle found!" and terminate DFS
        }
    }
    mark[u] = finished                       // we're done with u
    f[u] = ++time
}
```

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.

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 vertex 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

```
topolSort(G) {
    for each (u in V) mark[u] = undiscovered    // initialize
    S = empty stack
    for each (u in V)                           // be sure to visite every vertex
        if (mark[u] == undiscovered) topolVisit(u)
    while (S is nonempty) output pop(S)         // pop stack for final ordering
}

topolVisit(u) {                                 // start a search at u
    mark[u] = discovered                        // mark u visited
    for each (v in Adj(u))
        if (mark[v] == undiscovered)
            topolVisit(v)                       // visit v (pushing it on stack)
    push u onto S                               // push u when finished
}
```

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 $m = |E|$).

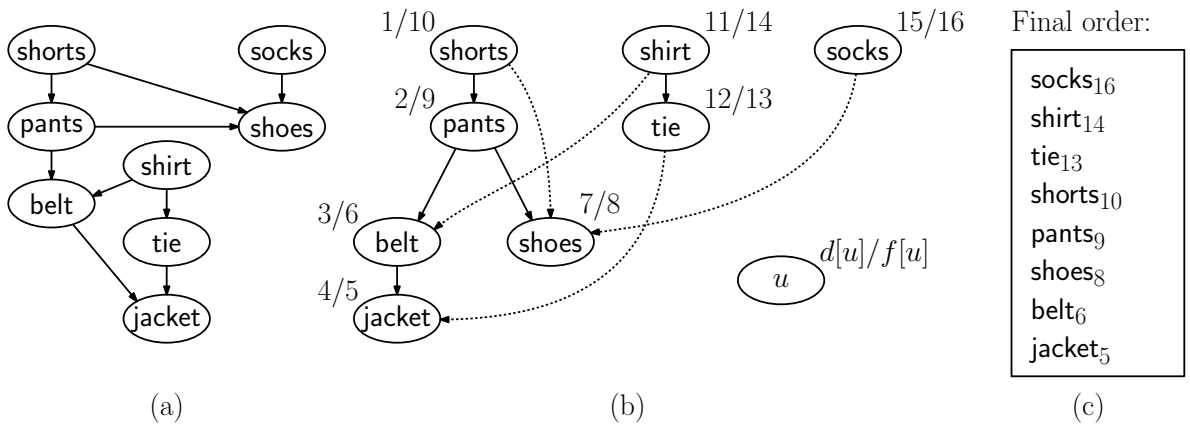


Fig. 9: Topological sorting example.

As an example we consider the DAG showed in the Fig. 9(a), which shows the precedence constraints for a person deciding in what order to get dressed. 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.)

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 task u must be completed before task 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 just maximum sum of the time values along any path in the DAG.

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 length any path that emanates from this vertex. We denote this quantity by $\mathsf{maxTime}[u]$. When we first encounter a vertex u in the DFS visit procedure, which we rename LongPathVisit, we initialize $\mathsf{maxTime}[u] = \mathsf{time}[u]$. For each adjacent vertex v, we invoke $\mathsf{LongPathVisit}(v)$ if v has not yet been discovered. We let maxLength to be the maximum maxTime of all u's neighbors, and we set $\mathsf{maxTime}[u] = \mathsf{maxTime} + \mathsf{time}[u]$. As in standard DFS, the main program invokes $\mathsf{LongPathVisit}(u)$ for all undiscovered vertices u. $\mathsf{LongPathVisit}(u)$ is given in the code-block below.

Longest Path via DFS

```
LongPathVisit(u) {                                          // start a search at u
   mark[u] = discovered                                     // mark u visited
   maxTime[u] = time[u]                                     // initialize max time for u
   for each (v in Adj(u)) {
       if (mark[v] == undiscovered) LongPathVisit(v)        // process v if undiscovered
       maxTime[u] = max(maxTime[u], maxTime[v] + time[u]) // update our max time
   }
}
```

An example is shown in Fig. 10. Each vertex's maxTime value is the sum of its own time and the maxTime values of its neighbors.

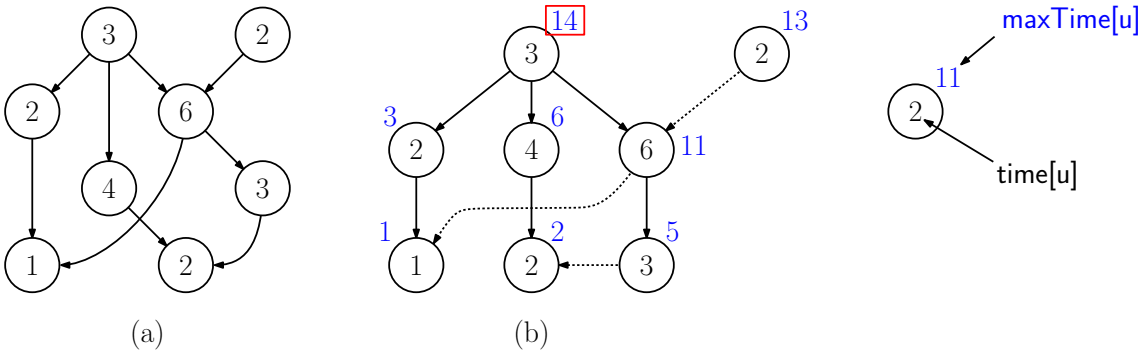


Fig. 10: Longest path in a DAG. Each vertex u is labeled with $\mathsf{time}[u]$ and $\mathsf{maxTime}[u]$.

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, $\mathsf{maxTime}[v]$ is fixed before it is accessed by u. The longest path in the entire DAG is the largest value of $\mathsf{maxTime}[u]$ among all vertices u. The *critical tasks* are those that lie on the longest path. How would you compute these?

Can this be used to compute the longest simple path in a digraph with cycles? The answer is no, but you should think about why it does not work.

Strong Components: (The following material applies *only* to directed graphs!)

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 reachability*s 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. 11(a) and (b)).

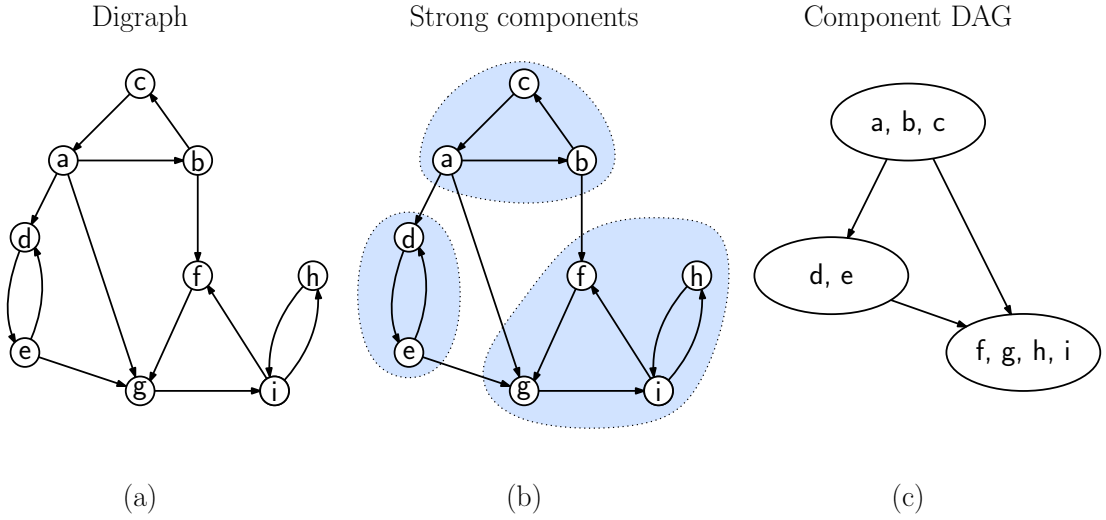


Fig. 11: Strong components and the component DAG.

If the vertices within each strong component are collapsed into a single vertex, the resulting digraph is called the *component digraph* (see Fig. 11(c)). It is easy to see that the component digraph must be acyclic (since if a number of components could be joined in a cycle, they would collapse into a single larger strong component). Therefore, this graph is usually called the *component DAG*.

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

Claim 1: If DFSvisit is started at a vertex u, it will terminate precisely when all the vertices reachable from u have been visited.

Proof: This follows from the exhaustive nature of DFS. (Note that some of these vertices may have been reached by earlier calls to DFSvisit.)

Claim 2: If C and C' are two strong components, and there is an edge from a vertex in C to a vertex in C', then the highest finish time in C is bigger than the highest finish time in C'.

Proof: There are two cases depending on whether the DFS first encounters a vertex from C or C'. If it first encounters a vertex u in C, then by Claim 1 the DFS will visit all the vertices of both C and C' before returning to u. Therefore, u will have the highest finish time of every vertex in $C \cup C'$. If it first visits a vertex in C', then the DFS will get stuck in C' (since it is not possible to reach anything in C). It follows that all the vertices of C will have higher discovery times than those of C', which further implies that they will have higher finish times as well.

Claim 3: The vertex that receives the highest finish time in a DFS must lie in a source vertex of the component DAG. (Recall that a vertex in a DAG is called a source if it has no incoming edges.)

Claim 3 is equivalent to saying that the strong components can be linearly arranged in decreasing order of their highest finish times. By doing so, every edge in the component DAG will go from an earlier

component in the linear order to a later one. How can we exploit this to obtain an efficient algorithm to find the strong components.

Claim 3 allows us to identify a vertex in some *source* of the component DAG. Unfortunately, this is not all that useful. What *would* be useful is to identify a vertex in a *sink* of the component DAG. If we could do this, we could start a DFS at this vertex, with the knowledge (by Claim 2) that no other strong components would be visited. We could then delete all these vertices (or equivalently, mark them as visited), and repeat the process. Eventually, all the strong components will be identified, each one arising as a separate subtree of the DFS forest.

So how to we convert an algorithm that identifies sources to one that identifies sinks in the component DAG? The trick is to reverse all the edges of G. Let G^R denote the directed graph that has the same vertex set as G, but every edge (u, v) is replaced by its reverse (v, u). Note that the strong components of G^R are the same as G, but the direction of edges in the component DAG have all be *reversed*. Thus, the sources in the component DAG of G^R are sinks in the component DAG of G. This leads to the following (insanely clever) algorithm for computing strong components.

(1) Given G, compute G^R (see Fig. 12(a)). (Note: This is a small programming exercise, which involves a simple traversal of G's adjacency list. It can be done in $O(n + m)$ time.)

(2) Run DFS(G^R) and label each vertex of G with the finish time of the corresponding vertex of G^R (see Fig. 12(b)).

(3) Sort the vertices of G in decreasing order of finish times. (Note: Since finish times are integers in the range $[1, 2n]$, this can be done in $O(n)$ time through Bucket Sort.)

(4) Run DFS(G), but in the outermost loop, whenever we need to find a new vertex to start DFSvisit, take the vertex with the highest finish time (using the above sorted order).

(5) Each subtree of the DFS forest will be a strong component (see Fig. 12(c)).

The correctness of the above algorithm follows from the remarks made earlier. The running time is $O(n + m)$, dominated by the time to compute G^R and the times for the two DFS's.

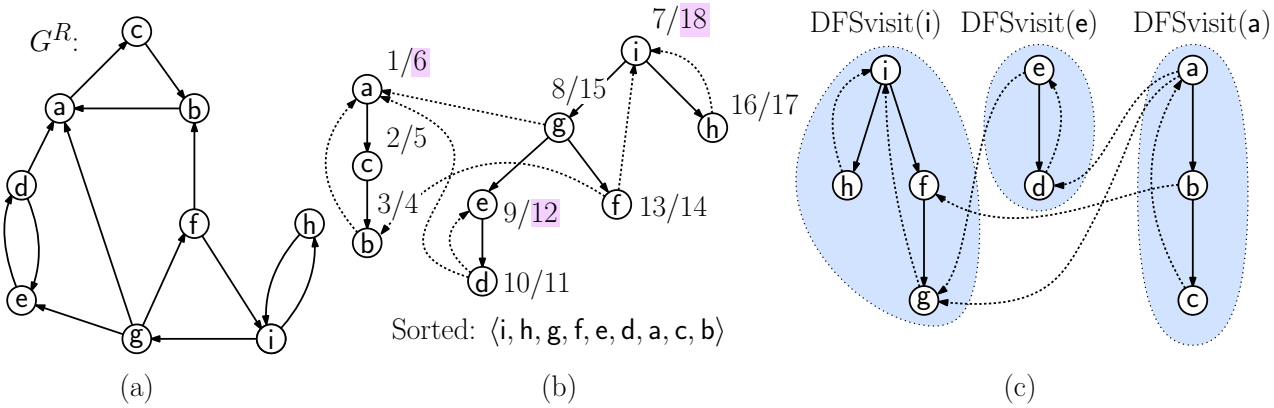


Fig. 12: Strong components and DFS.

Lecture 4: Graph Shortest Paths: Bellman-Ford and Dijkstra

Shortest Paths: Today we consider the problem of computing shortest paths in a directed graph. We are given a digraph $G = (V, E)$ and a source vertex $s \in V$, and we want to compute the shortest path from

s to every other vertex in G. This is called the *single source shortest path problem*. The algorithms we will present work for undirected graphs as well, by simply assuming that each undirected edge consists of two directed edges going in opposite directions.

We assume that each edge $(u, v) \in E$ is associated with a numerical edge weight $w(u, v)$ (see Fig. 13(a)). (If the graph is not weighted in this manner, then we assume that each edge has a weight of 1.) The *cost* of a path is defined to be the sum of edge weights along the path. Define the *distance* from any vertex u to any vertex v to be the minimum cost over all the paths from u to v. We denote this by $\delta(u, v)$. Thus, we are interested in computing $\delta(s, v)$ for all $v \in V$ (see Fig. 13(b)). We assume that every vertex has a trivial path of cost zero to itself, and hence $\delta(v, v) = 0$, for all $v \in V$. The collection of shortest paths defines a directed tree rooted at s, called the *shortest-path tree* (see Fig. 13(c)).

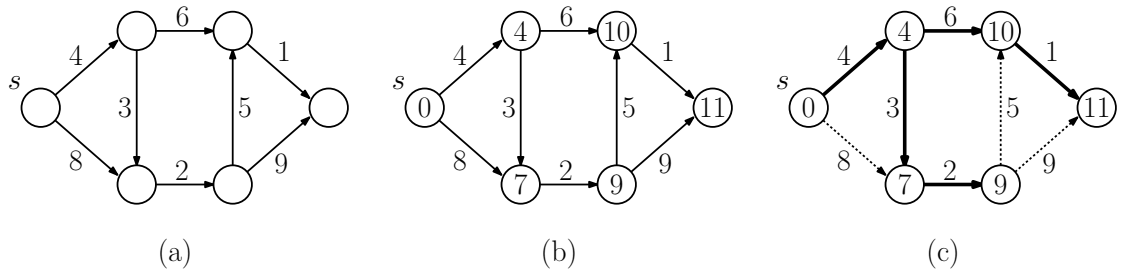


Fig. 13: (a) A weighted digraph G, (b) with vertices labeled with their distance $\delta(s, v)$, and (c) the shortest-path tree.

Preliminaries: If the edges of the graph are not weighted, there is an efficient algorithm, called *breadth-first search*, which solves the single-source shortest path problem. This algorithm runs in $O(n + m)$ time, where $n = |V|$ and $m = |E|$. This algorithm uses a first-in, first-out (FIFO) queue to process the vertices. The algorithm is very simple. It starts with a queue containing just the source vertex s, setting $d[s] \leftarrow 0$. It repeatedly dequeues the next vertex u. It then enqueues all of its neighbors v that have not already been visited and sets $d[v] \leftarrow d[u] + 1$. It stops when the queue is empty, and $d[u]$ is the distance from s to u.

In this lecture, we will focus on graphs with weighted edges. Since edge weights usually correspond to distances or travel times, it is common to assume that edges weights are positive, or at least, they are nonnegative. However, there are applications where negative edge weights make sense (see Fig. 14(a)). For example, if an edge denotes a financial trade (buying or selling commodities), the cost may either be positive (a loss) or negative (a gain). In this context, the shortest path corresponds to a sequence of transactions that minimizes loss (or equivalently, maximizes gain).

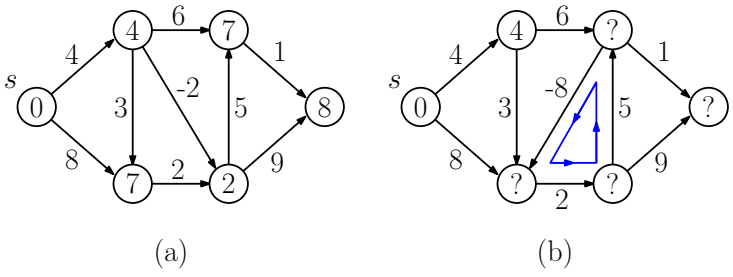


Fig. 14: (a) A weighted digraph with negative-cost edges and (b) a negative cost cycle. Observe that the shortest path from s to any of the vertices labeled "?" is undefined, because the distance can be made arbitrarily low by repeatedly looping through this cycle.

In general it is possible to define shortest paths even for graphs with negative edge weights, but it should be noted that shortest paths may not be well defined if the graph has *negative-cost cycles*. The reason is that the distance could be made arbitrarily small by traversing the cycle over and over (see Fig. 14(b)). You might wonder whether we could solve this by adding the constraint that it is not allowed to repeat vertices (that is, the path must be *simple*). This, however, makes the problem much harder solve.

We will discuss two algorithms. One that allows negative edge weights, but no negative-cost cycles (Bellman-Ford), and the other requires nonnegative edge weights (Dijkstra).

Shortest Paths and Relaxation: Before presenting these algorithms, let's explain some common elements. Both maintain an *estimate* of the cost of the shortest path from the source to each vertex v, which is denoted $d[v]$. 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 we initialize $d[s] \leftarrow 0$, and for all other vertices v, we set $d[v] \leftarrow \infty$. As the algorithm proceeds and sees ever more edges, it updates the d-values. Eventually, they will converge to the true shortest distances.

The process by which an estimate is updated is sometimes called *relaxation*. Intuitively, if you can see that the distance estimate can be improved along an edge, then do so. Consider any edge (u, v). We know that there exists a path from s to u of weight $d[u]$. By taking this path and following it with the edge (u, v) we obtain a path to v of length $d[u] + w(u, v)$. If this path is better than the current estimate $d[v]$, then we use it instead (see Fig. 144 and the code presented in the following code block.)

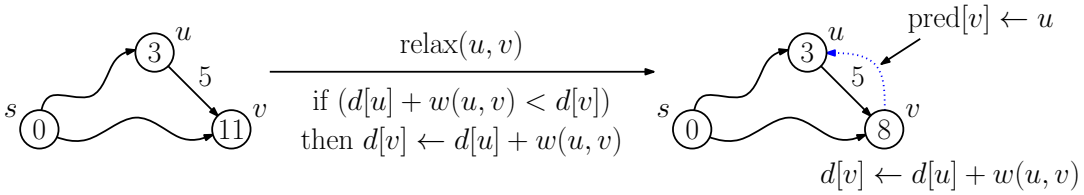


Fig. 15: Relaxation.

Propagating distances through relaxation

```
relax(u, v) {                       // relax along edge (u,v)
    if (d[u] + w(u, v) < d[v]) {    // is going through u better?
        d[v] = d[u] + w(u,v)        // update the distance estimate
        pred[v] = u                 // save our predecessor
    }
}
```

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 $d[v] = \delta(s, v)$, 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. (This remark will reemerge when we discuss the Bellman-Ford algorithm later.) The cleverness of any shortest path algorithm is to perform the updates in a judicious manner, so the convergence is as fast as possible. Dijkstra's algorithm does this in a simple, greedy manner and is optimal in the worst case. There are more practical variants, such as A^*-search, which can do better for typical instances.

Bellman-Ford Algorithm: Let us first consider the question of how to solve the single-source shortest path problem in an edge-weighted digraph when negative edge costs are allowed, but there are no

negative-cost cycles. (As an exercise, you are encouraged to think about how to detect whether the graph has negative cost cycles.)

We shall present the *Bellman-Ford algorithm*, which solves this problem. The algorithm was originally due to Alfonso Shimbel in 1955. (This slightly predates Dijkstra's algorithm, which dates to 1956.) It was rediscovered by Ford in 1956, and then by Bellman in 1958. (And for some reason, poor Shimbel was not credited with the name!) The algorithm runs in time worst-case $O(nm)$. Note, however, that the running time depends on the maximum number of edges in any shortest path and the order in which edges are visited. It can run as fast as $O(n+m)$ time if you are extremely lucky.

The idea behind the Bellman-Ford algorithm is quite simple. We initialize $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all other vertices. We know that for each edge $(u, v) \in E$, the operation relax(u, v) propagates shorest-path information outwards from s. So, let's simply apply this operation repeatedly along each edge of E until the d-values converge. The algorithm is shown in the following code block, and a worked example is shown in Fig. 145.

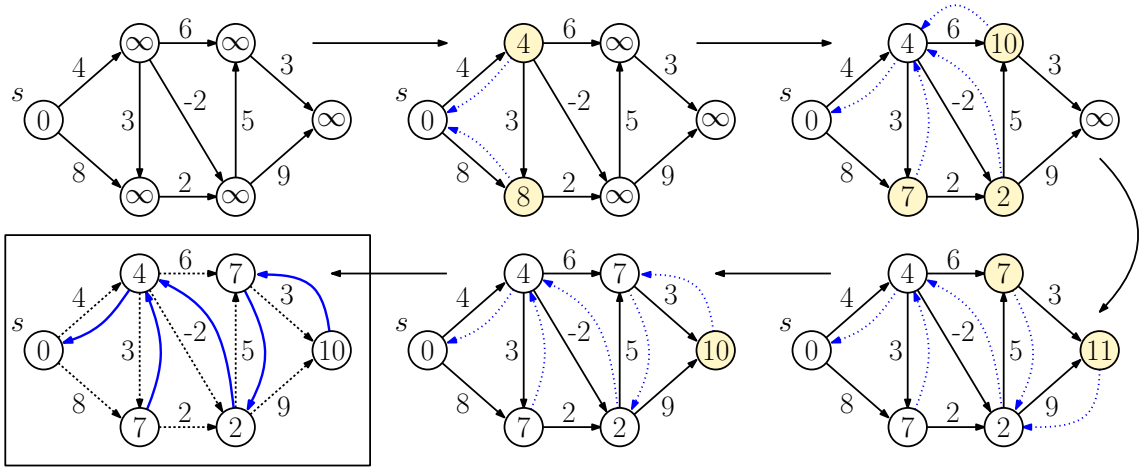


Fig. 16: Bellman-Ford Algorithm. In each stage, relaxations are performed on all edges.

Bellman-Ford Algorithm

```
bellman-ford(G=(V,E), w, s) {
    for each (u in V) {                 // initialization
        d[u] = +infinity; pred[u] = null
    }
    d[s] = 0
    repeat {                            // repeat until convergence
        converged = true
        for each ((u, v) in E) {        // relax along each edge
            relax(u, v)
            if (d[v] changed) converged = false
        }
    } until (converged)
    [The pred pointers define an inverted shortest-path tree]
}
```

Correctness: The following lemma establishes the correctness of the Bellman-Ford algorithm.

Lemma: If G has no negative-cost cycles, Bellman-Ford will terminate, and on termination of the Bellman-Ford algorithm, for all $v \in V$, $d[v]$ contains the correct distance from s to v.

Proof: We assert first that *if* the algorithm converges, then $d[v] = \delta(s, v)$ for all $v \in V$. As observed earlier, $d[v]$ contains the cost of *some* path from s to v, so we have $d[v] \geq \delta(s, v)$. We will prove that $d[v] \leq \delta(s, v)$ by induction on the length of the shortest path from s to v. In particular, if the shortest path from s to v consists of i edges, then after the ith iteration of the repeat-loop, $d[v] = \delta(s, v)$.

For the basis case ($i = 0$) the only vertex whose shortest path is of length zero is s itself. By our initialization code, $d[s] = 0$, and by definition of shortest paths $\delta(s, s) = 0$, so we're done.

For the general case ($i \geq 1$), consider any vertex v be any vertex such that the length of the shortest path from s to v consists of i edges. Let u be the vertex that immediately precedes v along this shortest path. It follows that (1) the shortest path from s to u is of length $i - 1$, and (2) $\delta(s, v) = \delta(s, u) + w(u, v)$. By the induction hypothesis, we know that after $i - 1$ iterations of the repeat-loop, we have $d[u] = \delta(s, u)$. After the ith iteration of the repeat-loop, when we consider the edge (u, v) we will update the value of $d[v]$ to

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

In conclusion, we have shown that $\delta(s, v) \leq d[v] \leq \delta(s, v)$, which implies that $d[v] = \delta(s, v)$.

To see that the algorithm converges, observe that if a vertex is not reachable from s, then its d-value never changes, and it does not affect convergence. Otherwise, when we first reach a vertex on a path from s, the relax process sets its distance to a finite value. After this, each time the algorithm fails to converge, at least one vertex's d-value has strictly decreased. This can happen only a finite number of times, so the algorithm eventually converges.

Running time: Observe that time through the for-loop of the algorithm visits each edge once and spends constant time, so the for-loop takes $O(m)$ time, where $m = |E|$. To show that the algorithm runs in $O(nm)$ time, it suffices to show that we go through the outer while loop at most $n = |V|$ times.

Lemma: If G has no negative-cost cycles, then there is a shortest path from s to any vertex v that does not repeat any vertex.

Proof: Suppose to the contrary that the shortest path from s to v did repeat some vertex u. Thus, the path has the form $s \rightsquigarrow u \rightsquigarrow u \rightsquigarrow v$. Since G has no negative cost cycles, we can remove the cycle $u \rightsquigarrow u$ from the path without increasing the path's total cost. If we repeat this for every repeated vertex, we will eventually have a path that contains no repetitions.

Corollary: For each $v \in V$, there is a shortest path from s to v consisting of at most $n - 1$ edges (where $n = |V|$).

Now, go back and review the induction from the correctness proof for Bellman-Ford. It shows that after i iterations of the outer while-loop, all the vertices whose shortest paths consist of i or fewer edges have the correct d-values. By the corollary, the algorithm terminates in at most $n-1$ iterations, so the overall running time is $O((n - 1)m) = O(nm)$. As observed earlier, this the worst case. The running time can be smaller.

Dijkstra's Algorithm: Next, we present a more efficient algorithm for the single-source problem in digraphs with weighted edges. This algorithm requires that the edge weights are nonnegative. The algorithm, called *Dijkstra's algorithm*, was invented by the famous Dutch computer scientist, Edsger Dijkstra in 1956 (and published later in 1959). It is among the most famous graph algorithms.

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). We will store a *predecessor link* with each vertex, which points to way back to the source (see Fig. 17(b)). Thus, the actual path can be found by traversing the predecessor links and reversing the resulting path. Since we store one predecessor link per vertex, the total space needed to store all the shortest paths is just $O(n)$.

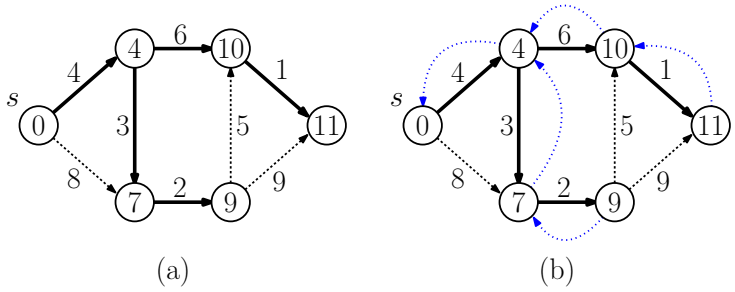


Fig. 17: (a) The shortest-path tree and (b) showing predecessor links (in blue).

As with Bellman-Ford, Dijkstra's algorithm is based on the relax operation. Rather than simply apply relax on all edges over and over, it achieves its efficiency by ordering the relaxations in a manner so that each edge is relaxed on only once. It does this in a simple, greedy manner, processing vertices in increasing order of distance from s. Note that there are more practical variants, such as A^*-search, which can do better for typical instances.

Dijkstra's algorithm operates by maintaining a subset of vertices, $S \subseteq V$, for which we claim we "know" the true distance, that is $d[v] = \delta(s, v)$. Initially $S = \emptyset$, the empty set, and we set $d[s] = 0$ and all others to $+\infty$. One by one, we select vertices from $V \setminus S$ to add to S. (If you haven't seen it before, the notation "$A \setminus B$" means the set A excluding the elements of set B.)

Paradoxically, the algorithm does not explicitly store the set S. Rather, we store all the vertices that are *not* in S in a priority queue, sorted by d-values. We greedily extract the "closest" vertex from the queue to be processed next, that is, the vertex with the smallest d-value. (Later we will justify why this is the proper choice.) We then propagate distance information forward by applying the relax operation on all of the outgoing edges from this selected vertex. We assume that our priority queue supports the following operations efficiently:

Build: Create a priority queue from a set 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 build operation can be performed in $O(n)$ time, and the other two can be done in $O(\log n)$ time each. The algorithm is given in the code block below. An example is shown in Fig. 142.

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 establish correctness, we need to show that on termination, $d[v] = \delta(s, v)$. This is a consequence of the following lemma, which states that when we finish processing a vertex, its distance value is correct.

Lemma: At all times and for all vertices $u \in V$, if $d[u] \neq \infty$, there exists a path of this cost. After u has been processed, $d[u] = \delta(s, u)$.

Proof: The first assertion follows by induction and the fact that $d[u]$ values change through the relax operation, which propagates a d-value (which represents an actual path cost) through one more edge. Thus, there is a path of cost $d[u]$. This implies that $d[u] \geq \delta(s, u)$.

For the second assertion, suppose to the contrary that this is not true. Consider the *first* finished vertex u for which this fails to hold, that is, $d[u] \neq \delta(s, u)$.Since $d[u] \geq \delta(s, u)$, we have $d[u] > \delta(s, u)$. Let S denote the set of processed vertices, just prior to processing u. The true shortest

```
dijkstra(G=(V,E), w, s) {
    for each (u in V) {                    // initialization
        d[u] = +infinity;  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]) {           // relax all outgoing edges from u
            if (d[u] + w(u,v) < d[v]) {    // relax(u,v)
                d[v] = d[u] + w(u,v)
                pred[v] = u
                decrease v's key in Q to d[v]
            }
        }
    }
    [The pred pointers define an inverted shortest-path tree]
}
```

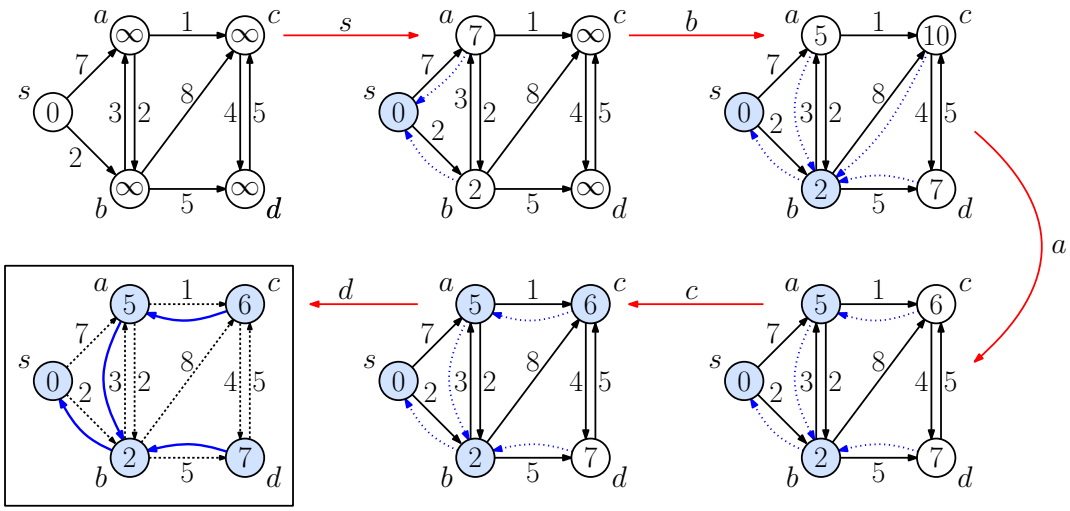


Fig. 18: Dijkstra's Algorithm example. Processed vertices are shaded in blue and broken blue links are predecessor links. The red arrows indicate the vertex being processed, which is always the unprocessed (white) vertex having the smallest d-value.

path from s to u must exit S along some edge (x, y), where $x \in S$ and $y \notin S$ (see Fig. 144). (Note that it may be that $x = s$ and/or $y = u$).

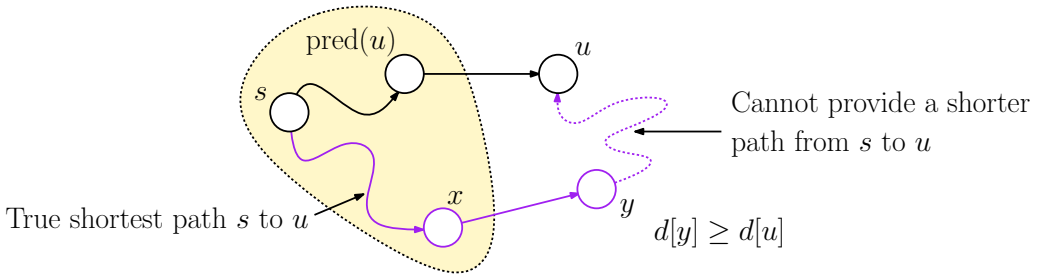


Fig. 19: Correctness of Dijkstra's Algorithm.

Because u is the first vertex where we made a mistake, and since x was already processed, we have $d[x] = \delta(s, x)$. When the algorithm processes x, it performs relax(x, y), which implies that

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

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

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

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

Running Time: 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 $O(\log n + \text{out-deg}(u) \cdot \log n)$ 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

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

Since G is connected, n is asymptotically no greater than m, so this is $O(m \log n)$. If you use a "smarter" heap (in particular, a Fibonacci heap), which supports the decrease-key operation in $O(1)$ amortized time, the running time decreases to only $O(n \log n + m)$.

Summary: We have presented two algorithms for computing shortest paths in directed graphs with edge weights. Bellman-Ford works on general graphs (assuming no negative cost cycles) and runs in $O(nm)$ time. Dijkstra's algorithm requires that edge weights are nonnegative, and runs more efficiently in $O(n \log n + m)$ time. If the edges have no weights (or equivalently all edge weights are equal), breadth-first search solves the problem more efficiently, in $O(n + m)$ time.

Lecture 5: Greedy Algorithms for Scheduling

Greedy Algorithms: Before discussing greedy algorithms in this lecture, let us explore the general concept of greedy optimization algorithms. In an *optimization problem*, we are given an *input* and asked to

compute a *discrete structure*, subject to various *constraints*, in a manner that either minimizes cost or maximizes some *objective function*. Such problems are extremely common in computation. Given an optimization problem, a fundamental question is whether it can be solved efficiently (as opposed to a brute-force enumeration of all possible solutions). If so, what approach should be used?

In most optimization algorithms the final structure is based on a series of selections. A simple design technique for optimization problems is based on a *greedy* approach, which builds up a solution by repeatedly selecting the *best alternative* in each step. (In particular, each choice is made without regard as to how this choice may interfere with later choices, and once a choice is made, it is never revoked.) When applicable, this method can lead to very simple and efficient algorithms. Even if a greedy algorithm does not yield the optimal solution, it sometimes produces a good approximation or a starting point for more sophisticated search algorithms.

Interval Scheduling: In this lecture, we discuss a number of problems motivated by applications in resource scheduling. In all instances we have one or more resources and a collection of requests to use these resources. We want to schedule all or some of these requests, subject to the limitations of our resources. As an example, suppose you work for the Parks and Recreation division of your community, and people want to reserve time at picnic tables at a local park. You want to write an algorithm to assign picnic tables to requests.

Our first problem is called *interval scheduling*. We are given a set $R = \{r_1, \ldots, r_n\}$ of n *activity requests* that are to be scheduled to use some resource. Each request r_i is associated with a given *start time* s_i and a given *finish time* f_i. For example, request indicates the desire to use the one picnic table in your park. (You work for a really cheap city!) Given that two groups cannot use the same picnic table at the same time, the objective is to grant as many of the requests as possible, but only one group can use the picnic area at a time.

We say that two requests r_i and r_j *conflict* if their start-finish intervals overlap, that is,

$$[s_i, f_i] \cap [s_j, f_j] \neq \emptyset.$$

(Note that by this criteria, we do not allow finish time of one request to overlap the start time of another one, but this is easily remedied in practice by shaving a tiny time period off the finish times.) Here is a formal problem definition.

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

An example of an input and a possible optimal solution is shown in Fig. 20 (another optimal solution is $\{5, 6, 8\}$). Notice that goal here is to maximize the *number* of requests that are scheduled. There are many other alternatives, such as maximizing *utilization*, the total time that the resource is being used.

What is the best way to greedily schedule the largest number of requests? Here are a number of seemingly reasonable approaches that *do not* guarantee an optimal solution:

Earliest Activity First: Repeatedly select the request with the earliest start time, provided that it does not overlap any of the previously scheduled requests.

Shortest Activity First: Repeatedly select the request with the smallest duration ($f_i - s_i$), provided that it does not conflict with any previously scheduled requests.

Lowest Conflict Activity First: Repeatedly select the request that conflicts with the smallest number of remaining requests, provided that it does not conflict with of the previously scheduled requests. (Note that once an request is selected, all the conflicting requests can be effectively deleted, and this affects the conflict counts for the remaining requests.)

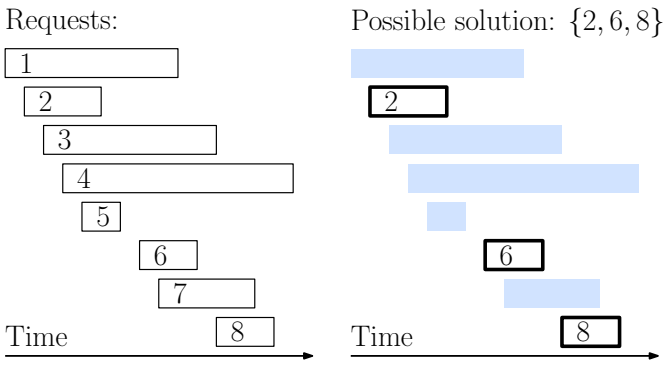


Fig. 20: An input and possible solution to the interval scheduling problem.

As an exercise, show (by producing a counterexample) that each of the above strategies may fail to generate an optimal solution for a given set of requests.

Earliest Finish First: If at first you don't succeed, keep trying. Here, finally, is a greedy strategy that does work. Intuitively, whenever we start a request, we want it to end as soon as possible to allow other requests to begin. This suggests that, among the non-conflicting requests, we repeatedly select the request that finishes first. Call this strategy *Earliest Finish First* (EFF). The pseudo-code is presented in the code-block below. It returns the set S of scheduled requests.

Interval Scheduling by Earliest Finish First

```
greedy-interval-schedule(s, f) {    // schedule requests using earliest-finish-first
    sort requests by increasing order of finish times
    S = empty                       // S holds the sequence of scheduled requests
    prevFinish = -infinity          // finish time of previous request
    for (i = 1 to n) {
        if (s[i] > prevFinish) {    // request i doesn't conflict with previous?
            append request i to S      // ...add it to the schedule
            prevFinish = f[i]       // ...and update the previous finish time
        }
    }
    return S
}
```

An example is illustrated in Fig. 21, where the requests are numbered in finish-order. Activity 1 is scheduled first. It conflicts with requests 2 and 3. Then request 4 is scheduled. It conflicts with requests 5 and 6. Finally, request 7 is scheduled, and it interferes with the remaining requests. 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 because we only schedule a request if it finishes after the previous one, the final schedule has no conflicting requests and hence is *feasible*.

To establish *optimality*, we use a common approach for greedy algorithms. Suppose towards a contradiction that the final schedule is not optimal. Then there must be some *first decision* where the algorithm differs from the optimal solution. We show that optimal solution can be modified to match

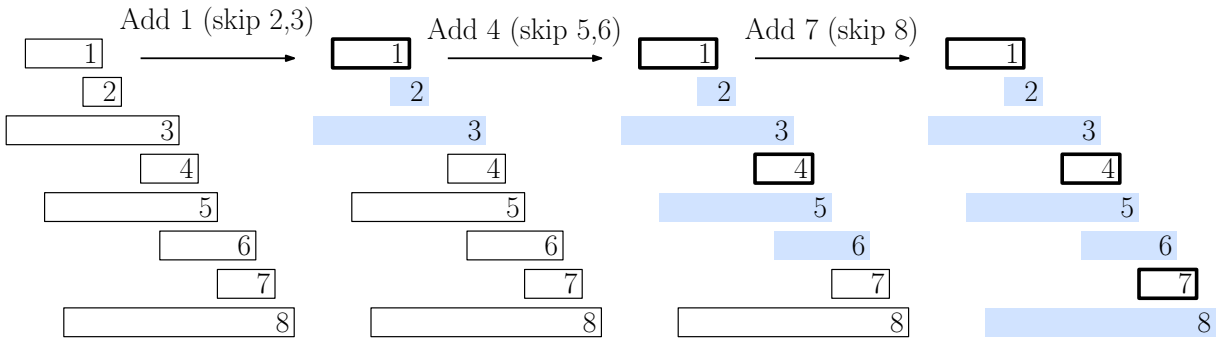


Fig. 21: An example of the greedy algorithm for interval scheduling. The final schedule is $\{1, 4, 7\}$.

the greedy solution at this decision point, without affecting its global quality. By applying this argument inductively, it follows that the greedy solution is as good as any optimal solution, thus it is optimal.

Claim: The EFF strategy yields an optimal solution to interval scheduling.

Proof: Let $O = \langle x_1, \ldots, x_k \rangle$ be the requests of an *optimal solution* listed in increasing order of finish time. (There may be many such solutions, and we may take O to be any of them.) Let $G = \langle g_1, \ldots, g_{k'} \rangle$ be the requests of the EFF solution similarly sorted. If $G = O$, then we are done. Otherwise, observe that since O is optimal, it must contain at least as many requests as G. Hence, there must be a first index j, $1 \leq j \leq k'$, 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 $g_j \neq x_j$.

The greedy algorithm repeatedly selects the request with the earliest finish time that does not conflict with any earlier request. Thus, we know that g_j does not conflict with any earlier request, and it finishes no later than x_j finishes (see Fig. 22).

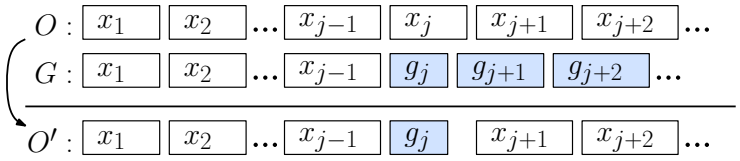


Fig. 22: Proof of optimality for the greedy schedule.

Consider the modified "greedier" schedule O' that results by replacing x_j with g_j in the schedule O (see Fig. 22). 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 requests 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 requests. Therefore, G is optimal.

Interval Partitioning: One shortcoming with interval scheduling is that some requests are not satisfied. Next, let us consider a variant where all the requests can be satisfied. Instead of a single resource, we

have an infinite number of possible exclusive resources to use, and we want to schedule *all* the requests using the *smallest* number resources. (The Department of Parks and Recreation can truck in as many picnic tables as it likes, but there is a cost for each table.)

As before, we are given a collection $R = \{r_1, \ldots, r_n\}$ of n requests, each with a start and finish time $[s_i, f_i]$. The 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 requests of R_j are mutually nonconflicting, for each j, $1 \leq j \leq d$.

We can view this as a *coloring problem*. In particular, we want to assign "colors" to the requests such that two overlapping requests must have different colors. (In our example, each picnic table has its own color. Two overlapping requests must be assigned to different tables, that is, different colors.) If this is done using at most d colors, the result is called a *d-coloring* of the requests. Scheduling all the requests to the minimum number of resources is equivalent to finding the smallest d, such that it is possible to d-color the activity requests.

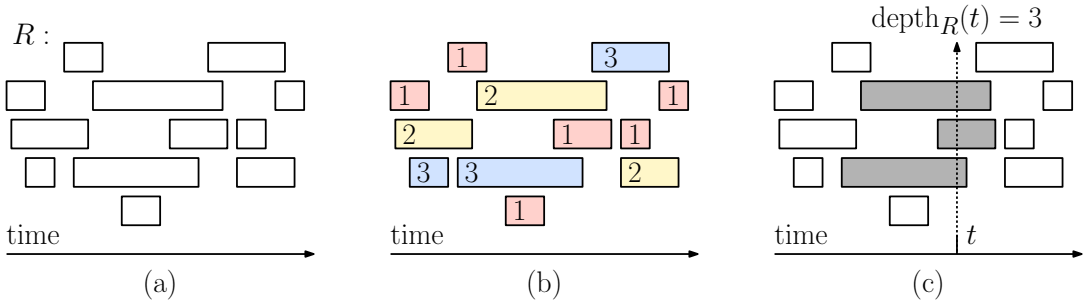


Fig. 23: Interval partitioning: (a) the requests R, (b) a possible 3-coloring, and (c) depth$_R(t)$.

We refer to the subset of requests that share the same color as a *color class*. The requests of each color class are assigned to the same resource. (For example, in Fig. 23(a) we give an example with $n = 12$ requests and in (b) show a 3-coloring. Thus, the six requests labeled 1 can be scheduled in one picnic table, the three requests labeled 2 can be put in a second table, and the three requests labeled 3 can be put in a third table.)

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 in increasing order of start times. Whenever we encounter a new request, we assign it to the smallest color (possibly a new color) such that this color has not been assigned to any overlapping request. The algorithm is presented in the following code block.

The solution given in Fig. 23(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 $O(n \log n)$ time with a bit of added cleverness.[3]

Correctness: Let us now establish the correctness of the greedy interval partitioning algorithm. We first observe that the algorithm generates a *feasible* output, since it takes care to never assign the same color to two overlapping requests.

[3]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 entry (start or finish), the index of the request (a number $1 \leq i \leq n$), and the time of the request (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 $2n$ entries can be processed in $O(1)$ time. We leave the implementation details as an exercise. Given $O(n \log n)$ time to sort the start and finish times, the total processing time is $2n \cdot O(1) = O(n)$. Thus, the overall running time is $O(n \log n)$.

```
greedy-interval-partition(s, f) {   // interval partition schedule
    sort requests by increasing start times
    for (i = 1 to n) do {           // classify the ith request
        X = emptyset                // X stores excluded colors for request i
        for (j = 1 to i-1) do {
          if ([s[j],f[j]] overlaps [s[i],f[i]]) add color[j] to X
        }
        Let c be the smallest color NOT in X
        color[i] = c
    }
    return color[1...n]
}
```

To establish that the algorithm is *optimal*, we will employ another widely used technique in greedy algorithm analysis. We will identify a statistic based on the given instance. We will show (a) that this statistic is a *lower bound* on the size of *any* feasible solution, and (b) that our algorithm's output is *upper bounded* by this statistic. It follows that our algorithm's output is optimal.

In our case, this statistic, called *depth*, is defined as follows. Given a set R of time intervals, and letting t denote any time instant, define $\text{depth}_R(t)$ to be the number of intervals of R that contain t (see Fig. 23(c)). Next, define $\text{depth}(R)$ to be the maximum depth over all possible values of t:

$$\text{depth}(R) \;=\; \max_{t \geq 0} \text{depth}_R(t).$$

Since the requests that contribute to $\text{depth}_R(t)$ conflict with one another, clearly we need at least this many resources to schedule these requests. Therefore, we have the following lower bound on the number of colors.

Claim: Given any set R of intervals, for any d-coloring, we have $d \geq \text{depth}(R)$.

Next, we show that our algorithm achieves this as an upper bound.

Claim: Given any set R of intervals, the greedy partitioning algorithm generates a d-coloring, where $d \leq \text{depth}(R)$.

Proof: It will simplify the proof to assume that all start and finish times are distinct. (We can always perturb them infinitesimally to guarantee this.) We will prove a stronger result, namely that at any time t, the number of colors in use at time t is at most $\text{depth}_R(t)$. The claim follows by taking the maximum over all times t.

Assume that requests are sorted by start times. Suppose towards a contradiction that there is some time t such that greedy uses more than $\text{depth}_R(t)$ colors. The first such time must be the start time s_i of some request. Consider the time t immediately prior to s_i. Let d denote the number of colors used by greedy at time t. Since s_i is the first violation, we have $d \leq \text{depth}_R(t)$. When s_i is seen, the depth increases by $+1$, so $\text{depth}_R(s_i) = \text{depth}_R(t) + 1$. Since the excluded set X used in the algorithm has d elements, there is an unused color among the first $d+1$ colors, which implies that the algorithm uses at most $d+1$ colors at time s_i. Thus, we are using at most $d + 1 \leq \text{depth}_R(s_i)$ colors, which contradicts the hypothesis that more than $\text{depth}_R(s_i)$ colors were needed at this point.

The above algorithm may seem utterly obvious. But, to see whether you really understand it, consider the following question. If the algorithm was identical, but the intervals were sorted according to some different criterion (not start times), would the result still be feasible? still be optimal?

Scheduling to Minimize Lateness: Finally, let us discuss a problem of scheduling a set of n tasks T where each task is associated with an *execution time* t_i and a *deadline* d_i. The objective is to schedule the tasks, no two overlapping in time, such that they are all completed before their deadline. If this is not possible, define the *lateness* of the ith task to be amount by which its finish time exceeds its deadline. The objective is to minimize the maximum lateness over all the tasks. (As an exmple, consider the assignments given to you from your various classes. You know how long each assignment takes, and you know when the deadline is. You lose points whenever assignments are late.)

More formally, given the execution times t_i and deadlines d_i, the output is a set of n starting times, $S = \{s_1, \ldots, s_n\}$, for the various tasks. Define the the finish time of the ith task to be $f_i = s_i + t_i$ (its start time plus its execution time). The intervals $[s_i, f_i]$ must be pairwise disjoint. The *lateness* of the ith task is the amount of time by which it exceeds its deadline, that is, $\ell_i = \max(0, f_i - d_i)$. The *maximum lateness* of S is defined to be

$$L(S) \;=\; \max_{1 \le i \le n} \max(0, f_i - d_i) \;=\; \max_{1 \le i \le n} \ell_i.$$

The overall objective is to compute S that minimizes $L(S)$.

Consider the instance shown in Fig. 24(a), where the execution time is shown by the length of the rectangle and the deadline is indicated by an arrow pointing to a vertical line segment. A suboptimal solution is shown in Fig. 24(b), and the optimal solution is shown in Fig. 24(c). 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.

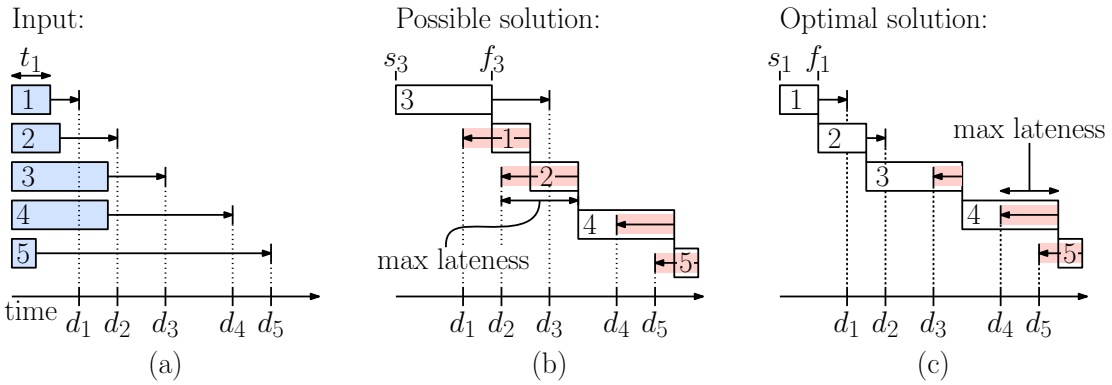


Fig. 24: Scheduling to minimize lateness.

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 *do not* guarantee an optimal solution.

Smallest duration first: Sort tasks by increasing order of execution times t_i and schedule them in this order.

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. Schedule the tasks in increasing order of slack-time.

As before, see if you can generate a counterexample showing that each of the above strategies may fail to give the optimal solution.

Earliest Deadline First: So what is the right solution? The best strategy turns out to find the task that needs to finish first and get it out of the way. Define the *Earliest Deadline First* (EDF) strategy work

by sorting the tasks by their deadline, and then schedule them in this order. (This is counterintuitive, because 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

```
greedy-lateness-schedule(t, d) { // schedule to minimize lateness
    sort tasks by increasing deadline (d[1] <= ... <= d[n])
    prevFinish = 0               // f is the finish time of previous task
    for (i = 1 to n) do {
        assign task i to start at s[i] = prevFinish  // start next task
        prevFinish = f[i] = s[i] + t[i]              // its finish time
        lateness[i] = max(0, f[i] - d[i])            // its lateness
    }
    return array s              // return array of start times
}
```

The solution shown in Fig. 24(c) is the result of this algorithm. 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: 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. We will show that this greedy algorithm produces an optimal schedule, that is, one that minimizes the maximum lateness. 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.

Claim: The EDF scheduling algorithm yields an optimal schedule for maximum lateness.

Proof: 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. Henceforth, let us consider schedules that are "idle-free." Let G be the schedule produced by the greedy algorithm, and let O be any optimal idle-free 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

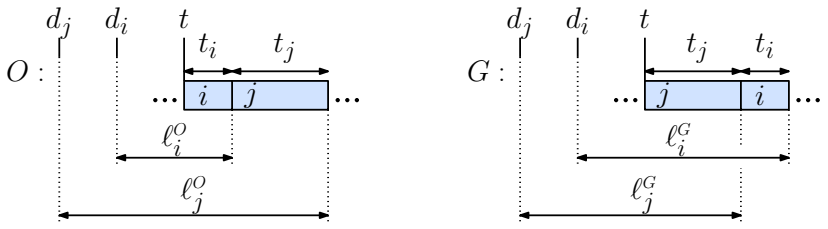


Fig. 25: Optimality of the greedy scheduling algorithm for minimizing lateness.

We will show that by swapping x_i and x_j in O, the maximum lateness cannot increase. The reason that swapping x_i and x_j in O does not increase lateness can be seen intuitively from Fig. 25. The lateness is reflected in the length of the horizontal arrowed line segments in 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. 25). In schedule O, task i finishes at time $t + t_i$ and (because it needs to wait for task i to finish) task j finishes as time $t + (t_i + t_j)$. The lateness of each of these tasks is the maximum of 0 and the difference between the finish time and the deadline. Therefore, we have

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

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

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

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

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

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

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

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

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

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

and

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

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

Lecture 6: k-Center Clustering and Gonzalez's Algorithm

Greedy Approximation for NP-Hard Problems: 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 Center-Based Clustering: Clustering is a widely studied problem with applications in statistics, pattern recognition, and machine learning. In a nutshell, it involves partition a large set of objects, called *points*, into a small number of subsets whose members are all mutually close to one another. In machine learning, clustering is often performed in high-dimensional spaces. Each data object is associated with a vector designating its *properties* or *features*. This is a numeric vector whose length may range from tens up to thousands that describes the salient properties of this object. Then clustering is performed to group similar objects together.

In this geometric view of clustering, we think of the data set as a set P of n points in a multi-dimensional space (see Fig. 26(a)). The output of the clustering algorithm is a partition of the point set into subsets, called *clusters*, denoted $\{C_1, \ldots, C_k\}$ (see Fig. 26(b)). In *center-based clustering*, the output is a set of cluster *center points*, $\mathcal{C} = \{c_1, \ldots, c_k\}$, and the clusters themselves are implicitly defined by the closest center (see Fig. 26(c)). In particular, a point lies in the ith cluster if its closest cluster center is c_i. Let $N(c_i)$ denote this *neighborhood* of nearest points. Given a set of k centers, we can assign points to their closest center in $O(nk)$ time.

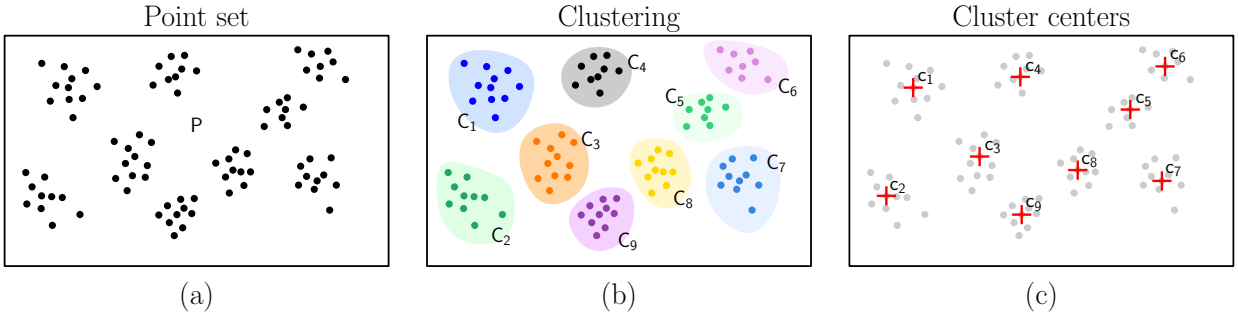


Fig. 26: Center-based geometry clustering.

Depending on the method being employed, it may be required that the center points are drawn from P itself, or they may just be arbitrary points in space. (Points used in an optimization problem that are not drawn from the set are sometimes called *Steiner points*.)

Three Center-Based Clusterings: There are a number of ways to define center-based clusterings. Before getting into our main topic, called the k-center problem, we will contrast three alternatives. In all three cases, we are told the desired number of clusters. (Determining the desired number of clusters is a tricky question, which we will not address here.)

k-Median: Compute the k center points to minimize the *sum of Euclidean distances* to the nearest cluster center.

k-Means: Compute the k center points to minimize the *sum of squared Euclidean distances* to the nearest cluster center.

k-Center: Compute the k center points to minimize the *maximum Euclidean distance* to the nearest cluster center.

Which of these is best? It really depends on your application. In order to better understand the differences between these criteria, it is illustrative to consider the single-cluster version of all three.

1-Median: The 1-median point is the single point that minimizes the sum of distances. In 1-dimensional space, this is realized by the *median*[4] of the point set (see Fig. 26(a)).

[4]This can be proved by a simple variational argument. Suppose the point is not at the median. There are more points on one side, say the right, than on the other. By moving the point some distance Δx to the right, more distances on the right decrease by Δx to compensate for the distances on the left that increase by Δx.

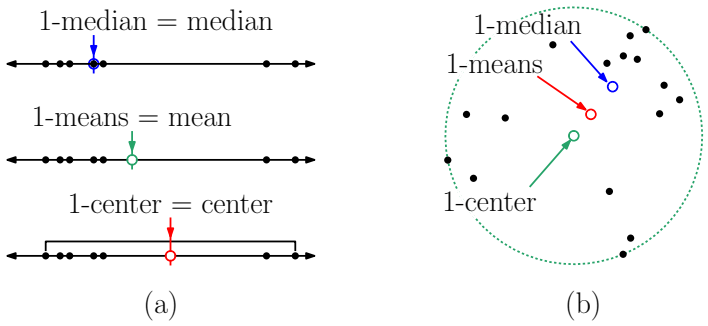


Fig. 27: 1-median, 1-means, and 1-center in 1- and 2-dimensional space.

In higher dimensions, there is no simple way to describe this point. The problem of computing the point in multi-dimensional space that minimizes the sum of Euclidean distances (see Fig. 26(b)). Computing this point is called the *Fermat-Weber problem* (named in honor of the famous 17th-century French mathematician, Pierre de Fermat, and the early 20th-century German economist Alfred Weber). There is no known exact algorithm, and best solutions only can approximate its coordinates.

1-Means: In 1-dimensional space, the 1-means problem is realized by the mean of the set. In higher dimensions, this is the *centroid* of the set. This can be easily computed by taking the mean coordinate along each of the coordinate axes. One of the reasons for the popularity of k-means is that the cluster centers are very easy to compute in $O(n)$ time.

There is a famous heuristic for the k-means problem, called *Lloyd's algorithm*. It starts with an initial set of centers, and then repeatedly moves each center to the centroid of its cluster, and then recomputes the clusters (based on changes in the nearest neighbors). It converges to a local minimum, but this may not be the global minimum.

1-Center: The solution to the 1-center problem is the center of the minimum radius ball that contains the points. In the 1-dimensional case, this is midway between the minimum and maximum. In higher dimensions, this is the center of the smallest enclosing ball for the point set. It is not obvious how to compute this ball. There is an amazingly simple $O(n)$-time randomized algorithm. The algorithm was developed by many researchers over time, but the simplest variant was presented and analyzed by the German computer-scientist Raimund Seidel.)

The k-Center Problem: For the rest of the lecture, we will focus on the k-center problem. Let P be a set of n points in some metric space. We are given a *distance function*, δ, which computes the distance between any two points $p, q \in P$. We assume that δ is a *metric*, meaning that it satisfies the following properties for any $p, q, r \in P$:

Positivity: $\delta(p, q) \geq 0$ and $\delta(p, q) = 0$ if and only if $p = q$.

Symmetry: $\delta(p, q) = \delta(q, p)$

Triangle inequality: $\delta(p, r) \leq \delta(p, q) + \delta(q, r)$

For the rest of the lecture, we can think of δ as the Euclidean distance, but the algorithm that we will present can be applied to any metric on P. Here is a formal problem statement.

k-center problem: Given a set P of n points in a metric space and an integer $k \leq n$, find a set $C \subseteq P$ of k points in order to minimize the maximum distance of any point of P to its closest center in C.

We can state the problem in a more "mathy" form in terms of an objective function

$$\Delta_P(C) \;=\; \max_{p \in P} \; \min_{c \in C} \delta(p, c).$$

This just computes the maximum distance of any point p of P to its closest center in C. Then, the k-center problem is the optimization problem of computing the subset of k-centers from P that minimizes this function, that is,

$$\min_{\substack{C \subseteq P \\ |C|=k}} \; \Delta_P(C).$$

As mentioned above, we can view the k-center problem as finding the smallest radius r such that it is possible cover P using k balls of radius r, each centered at some point of P. The minimum radius r is just the optimum value of $\Delta_P(C)$ a covering problem by balls, and the points of C are the centers of these balls.

Given a point x in space and radius r, define the *ball* $B(x, r)$ to be the (closed) ball of radius r centered at x. Given any solution C to the k-center problem, let $\Delta(C)$ denote the maximum distance from any point of P to its closest center. If we now place balls of radius $\Delta(C)$ about each point in C, it is easy to see that every point of P lies within the union of these balls. By definition of $\Delta(C)$, one of the points of P will lie on the boundary of one of these balls (for otherwise we could make $\Delta(C)$ smaller). The neighborhood of each cluster will lie within its associated ball (see Fig. 28(b)).

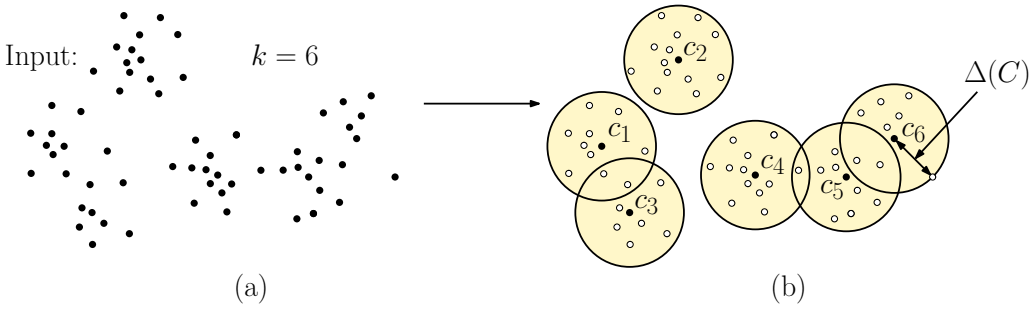


Fig. 28: The k-center problem ($k = 6$) in the Euclidean plane.

Given this perspective, we can see that the k-center problem is equivalent to the following problem:

k-center problem (equivalent form): Given a set P of n points in space and an integer $k \leq n$, find the minimum radius Δ and a set of balls of radius Δ centered at k points of P such that P lies within the union of these balls.

Gonzalez's Algorithm: Like many clustering problems, the k-center problem is known to be NP-hard, and so we will not be able to solve it exactly. (We will show this later this semester for a graph-based variant of the k-center problem.) Instead, we will present simple greedy algorithm, due to Teofilo Gonzalez, an algorithms researcher from UCSB. It does not produce the optimum value of Δ, but the result is at most twice as large as the optimum value of Δ.

Let us start with a couple of useful definitions. Recall that P is our point set, and consider any set $C = \{c_1, \ldots, c_k\}$ of cluster centers. (Since P will be fixed, we'll omit explicit references to it in our notation.) For any $c_i \in C$, recall that $N(c_i)$ are the points of its cluster. Define the *bottleneck distance* of c_i to be the distance to its farthest point in $N(c_i)$, that is,

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

and the overall bottleneck distance is

$$\Delta(C) \;=\; \max_{1 \le i \le k} \Delta(c_i).$$

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

We will build up the greedy centers, denoted $G = \{g_1, g_2, \ldots\}$ as follows. The greedy algorithm begins by selecting any point of P to be the initial center g_1. (There may be better ways to select the point in practice, but it won't affect the worst-case analysis.) We then repeat the following process until we have k centers. For $0 \le i < k$, let $G_i = \{g_1, \ldots, g_i\}$ denote the current set of greedy centers. Recall that $\Delta(G_i)$ is the maximum distance of any point of P from its nearest center. Let p be the point achieving this distance. Intuitively, p is the *most dissatisfied customer*, since he/she has to drive the farthest to get to the nearest Starbucks. The greediest way to satisfy p is to put the next center directly at p. (Thus plopping the next Starbucks right on top of p's house. Are you satisfied now?) In other words, set

$$g_{i+1} \leftarrow p \qquad \text{and} \qquad G_{i+1} \leftarrow G_i \cup \{g_{i+1}\}.$$

The pseudocode is presented in the code block below. The value $d[p]$ denotes the distance from p 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.)

Gonzalez's Algorithm for k-center

```
gonzalez(P, k) {                        // Gonzalez's algorithm for k-center on P
    G = empty-set
    for each (p in P)                   // initialize distances
        d[p] = +infinity

    for (i = 1 to k) {
        p = point of P that maximizes d[p]
        add p to G                      // p is the next cluster center
        for each (q in P) {             // update distances to nearest center
            d[q] = min(d[q], dist(p, q))
        }
    }
    return (G, Delta)                   // final centers and max distance
}
```

It is easy to see that the algorithm's running time is $O(kn)$. One step of the algorithm is illustrated in Fig. 29. Assuming that we have three centers $G = \{g_1, g_2, g_3\}$, let g_4 be the point that is farthest from its nearest center (g_1 in this case). In each step we create a center at g_4, so now $G = \{g_1, \ldots, g_4\}$. In anticipation of the next step, we find the point that maximizes the distance to its closest center (g_5 in this case), and if the algorithm continues, it will be the location of the next center.

Approximation Bound: Now, let us show that this algorithm is at most a factor of two from the optimum. Given a point set P, let $G = \{g_1, \ldots, g_k\}$ denote the set of centers computed by the greedy algorithm, and let $\Delta_P(G)$ denote its bottleneck distance. Let $O = \{o_1, \ldots, o_k\}$ denote the optimum set of centers, that is the set of k centers such that $\Delta_P(O)$ is the smallest possible. Since P is fixed throughout, we'll drop the subscript.

Theorem: For a point set P, let G denote the output of Gonzalez's algorithm and O denote the optimum k-center solution. Then $\Delta(G) \le 2\Delta(O)$.

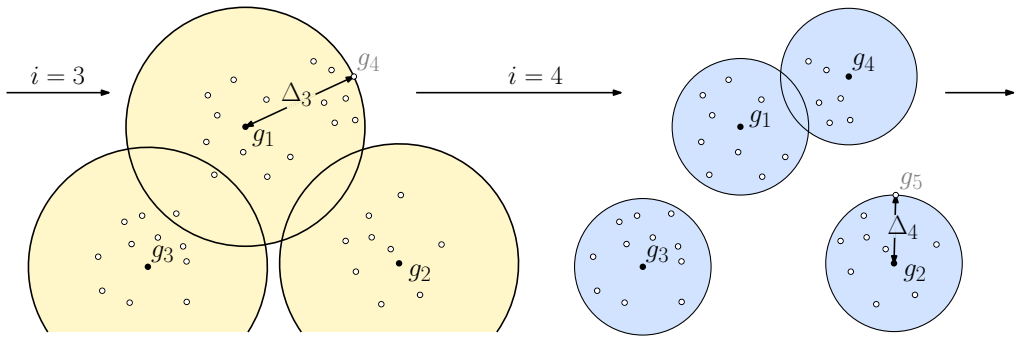


Fig. 29: Greedy approximation to k-center (from stage 3 to 4). (Not necessarily accurate to the algorithm's behavior.)

How can we hope to do this, since (assuming you cannot solve NP-hard problems), you cannot know what $\Delta(O)$ even is! Our approach will be to define estimate, denoted $\Delta_{\min}$ (see Claim 3 below). We will show that this estimate is a lower bound on the optimum, that is, $\Delta_{\min} \leq \Delta(O)$. We will also show that $2\Delta_{\min}$ provides an upper bound on greedy, that is, $\Delta(G) \leq 2\Delta_{\min}$. Combining these yields

$$\Delta(G) \;\leq\; 2\Delta_{\min} \;\leq\; 2\Delta(O),$$

which will imply that greedy is a 2-factor approximation to the optimum. Let's carry out this plan.

The analysis is based on the following three claims, each of which is quite straightforward to prove. Define G_i to be the set of greedy centers after the ith execution of the algorithm, and let $\Delta_i = \Delta(G_i)$ denote its overall bottleneck distance (the farthest any point is from its closest center in G_i). Thus, $\Delta(G) = \Delta_k$.

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 $\Delta(G)$. Also, define $G_{k+1} = \{g_1, \ldots, g_{k+1}\}$.

Claim 1: For $1 \leq i \leq k+1$, $\Delta_{i+1} \leq \Delta_i$. That is, the sequence of bottleneck distances is monotonically nonincreasing. (In Fig. 29 this is represented by the fact that the radii of the covering disks decrease with each stage.)

Proof: Whenever we add a new center, the distance to each point's closest center will either be the same or will decrease. Therefore, the maximum of such a set can never increase.

Claim 2: For $1 \leq i \leq k+1$, every pair of greedy centers in G_i is separated by a distance of at least Δ_{i-1}.

Proof: Consider the ith stage. By the induction hypothesis, the first $i-1$ centers are separated from each other by distance $\Delta_{i-2} \geq \Delta_{i-1}$. The ith center is, by definition, at distance Δ_{i-1} from its closest center, and therefore it is at distance at least Δ_{i-1} from all the other centers.

Since $\Delta(G) = \Delta_k$, we have:

Corollary: Every pair of greedy centers in G_{k+1} is separated by a distance of at least $\Delta(G)$.

Claim 3: Let $\Delta_{\min} = \Delta(G)/2$. Then for any set C of k cluster centers, $\Delta(C) \geq \Delta_{\min}$.

Proof: By definition of $\Delta(C)$, every point of P lies within distance $\Delta(C)$ of some point of C (see Fig. 30(a)). Since $G_{k+1} \subseteq P$, this is true for G_{k+1} as well. Since $|G_{k+1}| = k+1$, and C has only k clusters, by the pigeonhole principle, there exists at least two centers $g, g' \in G_{k+1}$ that are in the same neighborhood of some center $c \in C$ (see Fig. 30(b) and (c)).

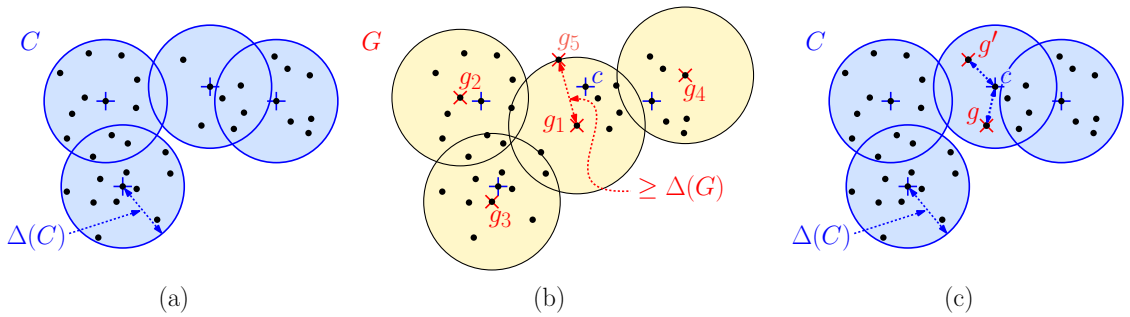


Fig. 30: Proof of Claim 3. (Not necessarily accurate to the algorithm's behavior.)

This implies that both $\delta(g, c)$ and $\delta(g', c)$ are less than or equal to $\Delta(c)$, which is less than or equal to $\Delta(C)$. Since $g, g' \in G_{k+1}$, by the corollary to Claim 2, $\delta(g, g') \geq \Delta(G)$. Combining these observations with basic properties of metric spaces, we have

$$\begin{aligned} \Delta(G) &\leq \delta(g, g') \leq \delta(g, c) + \delta(c, g') \quad \text{(by triangle inequality)} \\ &= \delta(g, c) + \delta(g', c) \quad \text{(by distance symmetry)} \\ &\leq \Delta(c) + \Delta(c) = 2\Delta(C). \end{aligned}$$

Rewriting this, we have $\Delta(C) \geq \Delta(G)/2 = \Delta_{\min}$, as desired.

Since Claim 3 applies to any set of k clusters, it applies to the optimum, O. We conclude that $\Delta(O) \geq \Delta_{\min}$. By definition, $\Delta_{\min} = \Delta(G)/2$, and so $\Delta(G) \leq 2\Delta(O)$, completing the analysis.

You might wonder whether this bound is tight. We will leave it as an exercise to prove that there is a input such that (if you are unlucky about how you choose the first point) the greedy algorithm returns a value that is arbitrarily close to twice that of the optimum.

Example: As an example, let's consider a set of points uniformly distributed along a line segment on the x-axis along the interval $[0, 12]$ and $k = 3$ (see Fig. 31).

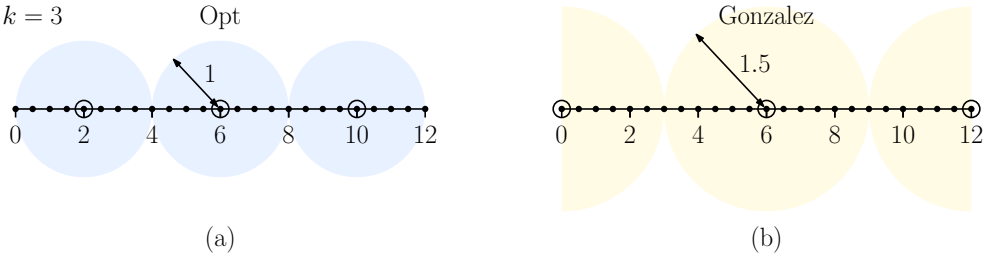


Fig. 31: Example of Gonzalez for $k = 3$. Approximation ratio is

The optimal center placement is at $x = \{2, 6, 10\}$, which yields a radius of 1. In contrast, if we start Gonzalez with the leftmost point, $x = 0$, (since the starting point is arbitrary), it will select the second point at $x = 12$, and the final point at $x = 6$. This yields a covering radius 1.5. Thus, in this case Gonzalez is suboptimal by a factor of $1.5/1 = 1.5$. (Can you come up with an example where the ratio is exactly 2?)

Summary: We introduced a number of common center-based clustering methods (k-means, k-median, and k-center). All of these problems are NP-hard. We presented a simple greedy algorithm, Gonzalez's algorithm, for the k-center problem, and we showed that it produces a result that is at most twice that of the optimal, for any set of points (in any metric space) and any value of k.

Lecture 7: Greedy Approximation: Set Cover

Set Cover: An important class of optimization problems involves covering a certain domain with minimum number of sets. Covering problems arise in many applications of science and engineering:

Surveilance: You want to place cameras in an environment to monitor a given region. Each possible camera position can view a certain region. Cameras are expensive to place and operate. What is the minimum number of camera placements to monitor the entire environment?

Wireless Coverage: You want to place wireless routers around a college campus so that every location is within range of some transmitter. Coverage regions vary due to the presence of obstacles. What is the minimum number of transmitters are needed to cover the entire campus?

Workforce scheduling: You are the manager of a large service center. Workers have various constraints on where/when they can work. Your job is to assign a minimum number of workers to cover all the stations/hours needed.

Many of these problems can be expressed abstractly as the *set cover problem*. We are given a pair $\Sigma = (X, S)$, called a *set system*, where $X = \{x_1, \ldots, x_n\}$ is a finite set of objects, called the *universe*. This is the domain to be covered. $S = \{s_1, \ldots, s_m\}$ is a collection of subsets of X. We assume that every element of X belongs to at least one set of S. Throughout, let $n = |X|$ and and $m = |S|$.

Given such a set system (X, S), the set cover problem is to determine the smallest number of sets of S needed to cover X. For example, in Fig. 32(a), the elements of X are the 12 black circles, and the sets $s_1, \ldots, s_6$ are indicated by the blue regions. In this case there exists a cover of size 3, consisting of s_3, s_4, and s_5 (see Fig. 32(b)). (In this case, the sets of this cover do not overlap. This is called an *exact cover*. In general, the sets of the cover are allowed to overlap.)

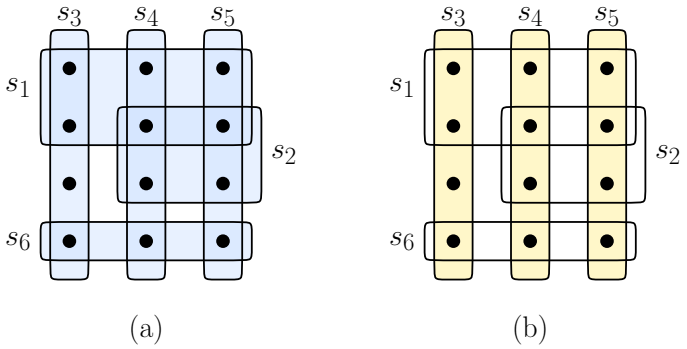


Fig. 32: (a) A set system consisting of 12 elements and 6 sets, and (b) an optimum cover consisting of the three sets $\{s_3, s_4, s_5\}$.

Notice that the output of set cover is not a set, but rather a set of sets. If we think of the sets of S as being indexed by the integers from 1 to m, then we can think of a cover C more conveniently as a subset of $\{1, \ldots, m\}$. This suggests the following definition.

Set Cover Problem: Given a set system $\Sigma = (X, S)$, where $S = \{s_1, \ldots, s_m\}$, compute a set $C \subseteq \{1, \ldots, m\}$ of minimum cardinality such that

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

A more general formulation is a weighted variant, in which each set s_i is associated with a positive weight w_i, which can 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.

Set Cover Approximation: Set Cover is a very useful optimization problem, but it is known to be *NP-hard*. We will present a simple *greedy heuristic* for this problem, and we will show that this heuristic leads to an approximation. As we shall see, the approximation factor is rather weak, but there are reasons to believe that significantly better approximations are not easy to compute. Recall that $n = |X|$. We will show that the size of the greedy heurisitc exceeds the size of an optimum cover by a factor of at most $\ln n$, the natural logarithm of n.

It is worth noting that, in a number of practical instances (including the geometric covering examples mentioned above), the greedy heuristic performs reasonably well. Unfortunately, we will show that it is possible to construct set systems where greedy's performance is $\Omega(\log n)$.

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-set                      // 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
}
```

An example is shown in Fig. 33. Set s_1 has the most elements, six, and is added first (see Fig. 33(a)). Next, s_6 has the most uncovered elements, three (see Fig. 33(b)). Finally, s_2 has the most uncovered elements, two (see Fig. 33(c)). Finally, s_3 covers the only remaining element (see Fig. 33(d)). Thus, it would return a set cover of size 4, whereas the optimal set cover has size 3.

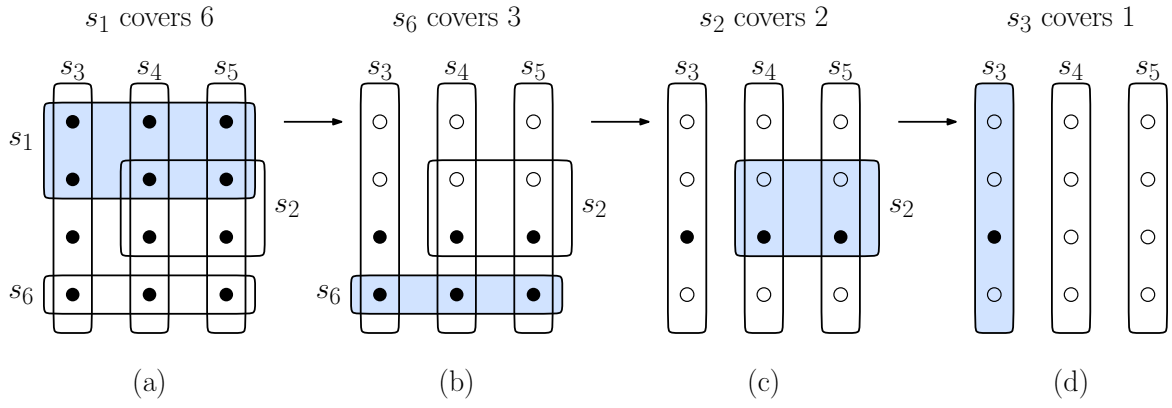


Fig. 33: Example of the greedy set-cover heuristic, returning $\{s_1, s_6, s_2, s_3\}$.

Running time: We will not worry about implementing this algorithm in the most efficient manner. Recall that $n = |X|$, and $m = |S|$ (the number of sets). It is possible implement this algorithm in time $O(nm)$. We will leave this as an exercise, but one way to visualize it is to imagine a bipartite graph, where

the elements X are on one side and the sets S are on the other side. For each pair s and x where set s contains element x, we add the edge (s, x). From this perspective, the greedy algorithm repeatedly removes the highest-degree vertex from the S side, and delete all of its edges.

A bad case for greedy: 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. 34 involving a universe of 32 elements. The optimal set cover consists of the two 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 the set 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.

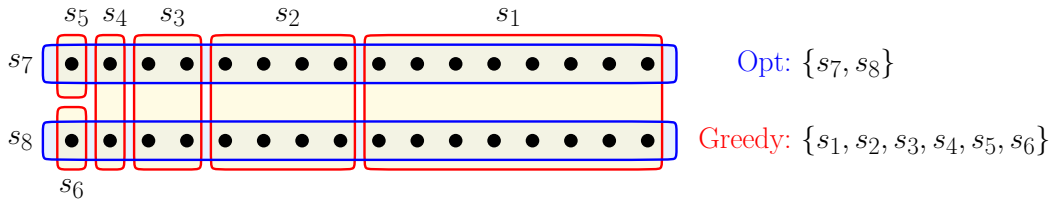


Fig. 34: Repeatedly fooling the greedy heuristic.

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 $\lg n$ sets, where $n = |X|$. (Recall that "lg" denotes logarithm base 2.) Thus, on this example the greedy heuristic achieves an approximation factor of roughly $(\lg n)/2$. 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.

Greedy's approximation factor: Next, we will show that this bad case is close to the worst case. In particular, we'll show that the number of sets generated by the greedy heuristic exceeds the optimum number by a factor of at most $\ln n$.

Theorem: Given any set system $\Sigma = (X, S)$, let G be the output of the greedy heuristic and let O be an optimum cover. Then $|G| \leq 1 + |O| \cdot \ln n$, where $n = |X|$.

By the way, this is just $+1$ off from what we really wanted, namely, $|G| \leq |O| \cdot \ln n$. It is possible to prove this, but the proof is a bit messier. Before giving the proof, we need one useful technical inequality.

Lemma: For all $c > 0$,

$$1 - \frac{1}{c} \leq e^{-1/c}.$$

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 $e^z = 1 + z + \frac{z^2}{2!} + \frac{z^3}{3!} + \ldots \geq 1 + z$.) Setting $z = -1/c$ yields the result.

We now prove the main theorem.

Proof: We will cheat a bit. Let o 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 $g \leq o \cdot \ln n$. (Note that we should really show that $g + 1 \leq o \cdot \ln n$, 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. Let X_i denote the subset of X that remains to be covered after the ith iteration of the algorithm, and let $n_i = |X_i|$. Initially, $X_0 = X$ and $n_0 = n$. At the start of the ith iteration, there are n_{i-1} elements that remain to be covered.

We know that there is a cover of size o for the entire set X, and therefore there is a cover of size o for X_{i-1}. (You might think it should be smaller since we have covered many elements so far. But there is no guarantee that we have used any of the optimum sets in doing so.) Since $n_{i-1} = |X_{i-1}|$, by the pigeonhole principal there exists some set that covers at least n_{i-1}/o elements. Since the greedy algorithm selects the set covering the largest number of remaining elements, it selects a set that covers at least this many elements. Therefore, the number of elements that remain to be covered is at most

$$n_i \;\le\; n_{i-1} - \frac{n_{i-1}}{o} \;=\; n_{i-1}\left(1 - \frac{1}{o}\right).$$

Since this applies to every iteration, we see that with each iteration the number of remaining elements decreases by a factor of at least $(1 - 1/o)$. If we repeat this i times, we have

$$n_i \;\le\; n_0\left(1 - \frac{1}{o}\right)^i \;=\; n\left(1 - \frac{1}{o}\right)^i.$$

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

$$1 \;\le\; n_g \;\le\; n\left(1 - \frac{1}{o}\right)^g.$$

By the technical lemma, with $c = o$, we have

$$1 \;\le\; n\left(e^{-1/o}\right)^g \;=\; ne^{-g/o}$$

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

$$e^{g/o} \;\le\; n \qquad \Rightarrow \qquad \frac{g}{o} \;\le\; \ln n \qquad \Rightarrow \qquad g \;\le\; o \cdot \ln n.$$

Since $g = |G| - 1$, $o = |O|$ and $n = |X|$, we conclude that

$$|G| \;\le\; 1 + |O| \cdot \ln n, \quad \text{where } n = |X|,$$

as desired.

Summary: To summarize. In this short lecture we introduced the set-cover problem. (In a later lecture, we will show that set cover is NP-hard.) We introduced a simple greedy algorithm for set cover, which works by repeatedly taking the set that covers the greatest number of uncovered elements. Finally, we presented a proof that the greedy heuristic is within a factor of $\ln n$ of the size of the optimal cover, where n is the number of elements to be covered.

While greedy often works well in practice, there is no hope in trying to improve this result. It is known that (under fairly well-accepted assumptions about NP-hardness) it is not possible to approximate set cover to any factor smaller than $(1 - o(1)) \cdot \ln n$. (The factor $1 - o(1)$ means that the multiplicative factor is smaller than but arbitrarily close to 1.)

Lecture 8: 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, and this renders straightforward recursive solutions inefficient.

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 sub-optimally 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 $R = \{r_1, \ldots, r_n\}$ of n *requests* to be scheduled on an exclusive resource (e.g., a picnic table at a local park). Each request has a start-finish time interval, $[s_i, f_i]$. The objective is to compute any maximum sized subset of non-overlapping intervals (see Fig. 35(a)).

In *weighted interval scheduling*, we assume that in addition to the start and finish times, each request is associated with a numeric *weight* or *value*, call it v_i, and the objective is to find a set of non-overlapping requests that maximizes the sum of values of the scheduled requests (see Fig. 35(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

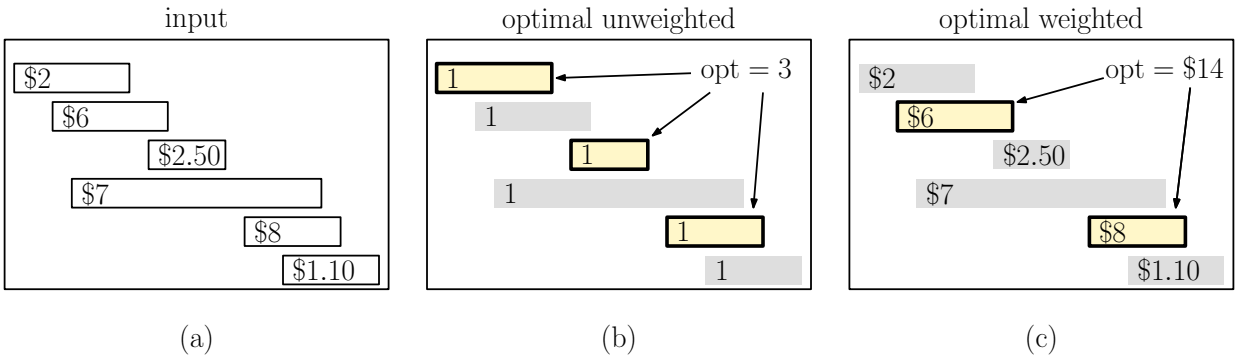


Fig. 35: Weighted and unweighted interval scheduling.

we did in the greedy algorithm, it will be convenient to sort the requests in non-decreasing order of finish time, so that $f_1 \leq \ldots \leq f_n$.

Here is the idea behind DP in a nutshell. Consider the *last* request $[s_n, f_n]$. There are two possibilities. If this request *is not* in the optimum schedule, then we can safely ignore it, and recursively compute the optimum solution of the first $n-1$ requests. Otherwise, this request *is* in the optimal solution. We will schedule this request (receiving the profit of v_n) and then we must eliminate all the requests whose intervals overlap this one. Because requests have been sorted by finish time, this involves finding the largest index p such that $f_p < s_n$. Thus, we solve the problem recursively on the first p requests.

But we don't know the optimum solution, so how can we select among these two options? The answer is that we will compute the cost of both of them recursively, and take the better of the two.

Let's now implement this idea. Recall that the requests are sorted by finish times. For the sake of generality, let's assume that we want to solve the problem on requests 1 through j, where $0 \leq j \leq n$. If $j = 0$, there is nothing to do. Otherwise, given any request j, define prior(j) to be the largest integer such that $f_{\text{prior}(j)} < s_j$, that is, prior$(j)$ is latest request by finish times that does *not* overlap request j. If no such i exists (that is, all the preceding intervals overlap), let prior$(j) = 0$ (see Fig. 36). This means that if request j is put in the schedule, we are free to include request prior(j) and an earlier ones.

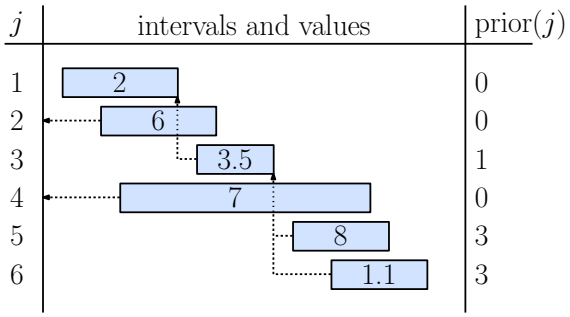


Fig. 36: Weighted interval scheduling input and prior-values.

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 $0 \leq j \leq n$, $W(j)$ denotes the maximum possible value achievable if we consider just requests $\{1, \ldots, j\}$ (assuming that requests are given in order of finish time).

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

Summarizing our earlier observations, in order to compute $W(j)$ for an arbitrary j, 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, $W(j) = W(j-1)$.

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 prior(j). Thus we have $W(j) = v_j + W(\text{prior}(j))$.

Ignoring the basis case ($j = 0$) there are two options. Clearly, we take the better of the two. This yields the following *DP formulation*:

$$W(j) = \begin{cases} 0 & \text{if } j = 0 \text{ (basis)} \\ \max \begin{cases} W(j-1) & \text{(reject } j\text{)} \\ v_j + W(\text{prior}(j)) & \text{(accept } j\text{)} \end{cases} & \text{if } j > 0 \end{cases} \tag{1}$$

The optimal solution for all the requests is $W(n)$. Note that the principle of optimality applies here. When we make the recursive invocations to $W(j-1)$ and $W(\text{prior}(j))$, we should solve these problems optimally. (Their solutions are independent of the choices we have made that brought us here, so there is no harm in doing the best we can in solving them.)

How do we know which of the these two options (accept or reject) to select? We will see in the next section that evaluating both of them in a straightforward manner will lead to a high running time. While it is tempting to try to determine which option is better (e.g., higher value or shorter duration), the basic law of dynamic programming is to *act stupid*—just try both and take the better option.

Recursive Implementation (Slow!): The simplest implementation would be to express the recursive formulation as a recursive function. See the following code block. We first sort the requests by finish time and precompute the values of prior(j), which we store in an array. (We will leave this as an exercise, but it can be done in $O(n \log n)$ time.) We then invoke the recursive function `rec-WIS(n)` to compute the best total cost.

Recursive Weighted Interval Scheduling (Inefficient!)

```
WIS(s[1..n], f[1..n], v[1..n]) {          // recursive WIS (slow)
    Sort requests by finish time
    Compute prior[j] for j = 1, ..., n (Exercise)
    return rec-WIS(n)                     // total value of all requests
}

rec-WIS(j) {                              // total value of 1 .. j
    if (j == 0) return 0                  // basis
    else return max(
        rec-WIS(j-1),                     // reject j
        v[j] + rec-WIS(prior[j]) )        // accept j
}
```

The correctness of this procedure follows from the fact that it is just a straightforward implementation of Eq. (1). The running time is a problem, however. To make this concrete, let us suppose that prior(1) = 0 and prior(j) = $j-2$, for all $j \geq 2$. (Convince yourself that you can construct a set of

intervals to make this happen.) Let $T(j)$ be the number of recursive function calls to `rec-WIS`(0) that result from a single call to `rec-WIS`(j). It is easy to verify that

$$\begin{aligned} T(0) &= 1 \\ T(1) &= T(0) + T(0) \\ T(j) &= T(j-1) + T(j-2), \quad \text{for } j \geq 2. \end{aligned}$$

(see Fig 37). It is not hard to see that this 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, in particular $F(n) \approx \phi^n$, where $\phi \approx 1.618$. Our running time is at least $T(n)$, which is exponential in n.

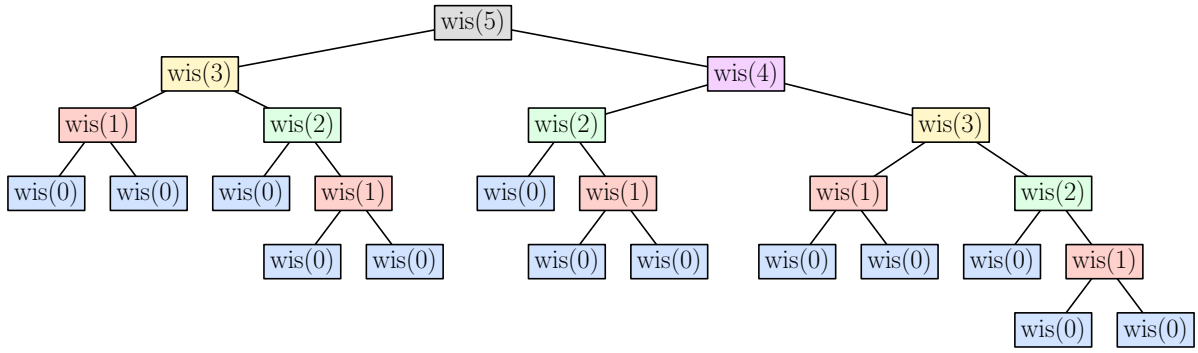


Fig. 37: The exponential nature of recursive-WIS.

Memoized Implementation: The problem with the simple recursive formulation is that it repeatedly calls the same function. To save time, let's just cache (or save) the value, and look it up later if needed. This process is called *memoization*. (Intuitively, we are making a "memo" to ourselves of what the value is for future reference.) The algorithm below saves the values of $W(j)$ in an array $W[j]$. This results in significantly fewer recursive calls (see Fig. 38), with only one recursive call for each j.

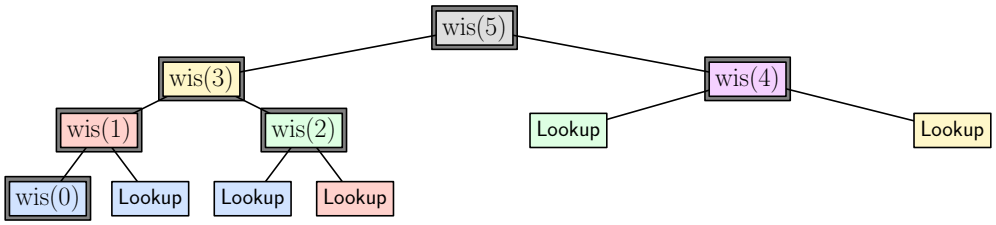


Fig. 38: Recursive calls in memoized WIS. (Highlighted squares are recursive calls, and the others are table lookups.)

Initially, we set $W[j] = -1$ for $0 \leq j \leq n$, so we know that its value has not been initialized. The modified algorithm is presented in the code-block below. 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 flag that indicates whether we accepted or rejected the current request. In particular, we maintain a boolean array accept, and we set accept[j] to true if j was accepted and

```
memo-WIS(j) {                                          // memoized WIS implementation
    if ( W[j] == -1 ) {                                // W[j] undefined?
        if ( j == 0 ) {                                // basis case
            W[j] = 0
        } else {
            rejectVal = memo-WIS(j-1)                  // value if we reject j
            acceptVal = v[j] + memo-WIS(prior[j])      // value if we accept j
            if ( rejectVal > acceptVal ) {             // better to reject
                W[j] = rejectVal
                accept[j] = false                      // remember our choice
            } else {                                   // better to accept
                W[j] = acceptVal
                accept[j] = true                       // remember our choice
            }
        }
    }
    return W[j]                                        // return value
}
```

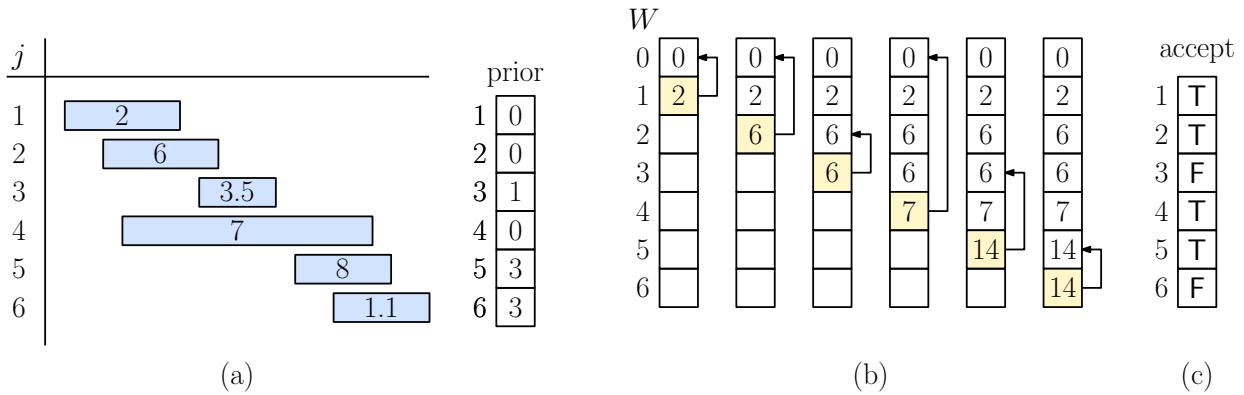


Fig. 39: (a) The input intervals and prior values, (b) the W array, and (c) the final accept values.

otherwise we set it to false. The final answer is `memo-WIS`(n). The resulting algorithm is presented in the following code block and an example is given in Fig. 39.

The correctnes of this algorithm follows from the fact that we are simply implementing the recursive formualation from Eq. (1). Assuming that the intervals have been sorted and the prior values have been computed (each of which take $O(n \log n)$ time), the memoized version runs in $O(n)$ time. To see this, observe that each invocation of memo-WIS either returns in $O(1)$ time (with no recursive calls), or it computes one new entry of W. Since there are n entries in the table, the latter can occur at most n times.

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. As before, we include the computation of the accept flags.

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

Bottom-Up Weighted Interval Scheduling

```
bottom-up-WIS() {                                   // bottom-up WIS implementation
    W[0] = 0                                        // basis
    for (i = 1 to n) {
        rejectVal = W[j-1]                          // value if we reject j
        acceptVal = v[j] + W[prior[j]]              // value if we accept j
        if ( rejectVal > acceptVal ) {              // better to reject
            W[j] = rejectVal
            accept[j] = false                       // remember our choice
        }
        else {                                      // better to accept
            W[j] = acceptVal
            accept[j] = true                        // remember our choice
        }
    }
    return W[n]
}
```

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.

We use the accept[j] values to address this. We start with $W[n]$ and work backwards. We know that value of $W[j]$ arose from two distinct possibilities, accept or reject. If we accepted request j, then we add j to the schedule and continue with prior[j]. Otherwise, we add nothing to the schedule and we continue with $j - 1$. The algorithm for generating the schedule is given in the code block below. Clearly, it runs in $O(n)$ time.

Computing Weighted Interval Scheduling Schedule

```
get-schedule() {                                    // get the WIS schedule
    j = n                                           // start with the last request
    sched = empty
    while (j > 0) {
        if (accept[j]) {                            // accepted request j?
            prepend j to the front of sched
            j = prior[j]
        } else
            j = j-1
    }
    return sched                                    // return the final schedule
}
```

The computation of the final schedule is illustrated in Fig. 40.

- Since accept[6] = F, we *reject* 6 and continue with $6 - 1 = 5$.
- Since accept[5] = T, we *accept* 5 and continue with prior[5] = 3.
- Since accept[3] = F, we *reject* 3 and continue with $3 - 1 = 2$.
- Since accept[2] = T, we *accept* 2 and continue with prior[2] = 0 (and terminate since $j = 0$).

We obtain the final schedule $\langle 2, 5 \rangle$.

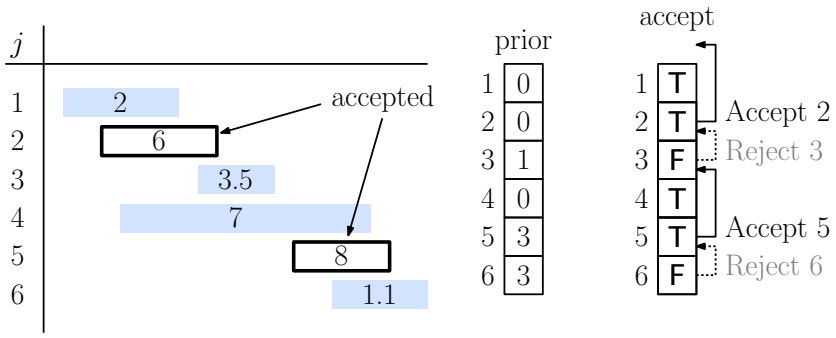


Fig. 40: Using the accept flags to compute the final schedule.

Summary: We introduced a new technique for designing efficient algorithms, called *dynamic programming* (or DP). We showed that the weighted version of the interval-scheduling problem can be solved efficiently using this technique. We showed that the problem has a recursive formulation (Eq. (1)), but that a straightforward implementation yields an exponential-time algorithm. We then provided two methods to obtain an efficient $O(n)$-time algorithm. The first uses recursion combined with memoizing (or caching) computed values. The second computes the table entries in an iterative manner, working in a bottom-up manner. Finally, we showed that by storing some "hints" we could extract the actual schedule, after the DP has finished running.

Lecture 9: DP: Longest Common Subsequence and Edit Distance

Strings: In this lecture we continue our study of dynamic programming algorithms. One important area of algorithm design is the study of algorithms for character strings. Finding patterns or similarities within strings is fundamental to various applications, ranging from document analysis to computational genomics. We study two widely studied measures of string similarity, longest common subsequence and edit distance. Today, we will consider efficient DP solutions to these problems.

Longest Common Subsequence: Consider two character sequences, that is, *strings*,

$$X = \langle x_1, x_2, \ldots, x_m \rangle \qquad \text{and} \qquad Z = \langle z_1, z_2, \ldots, z_k \rangle,$$

where x_i and z_j are elements over some given *alphabet*, Σ. (For example $\Sigma = \{a, b, c, \ldots, z\}$ or $\Sigma = \{A, G, C, T\}$.) Let $|X|$ denote the number of characters in X.

We say that Z is a *subsequence* of X its characters all appear in order in X. More formally, there is a strictly increasing sequence of k indices $\langle i_1, i_2, \ldots, i_k \rangle$ $(1 \le i_1 < i_2 < \ldots < i_k \le n)$ such that $Z = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$ (see Fig. 42).

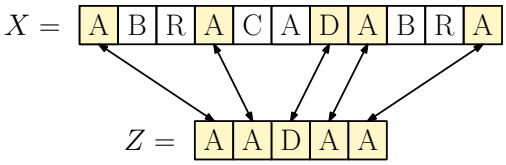


Fig. 41: The string $Z = \langle \text{AADAA} \rangle$ is a subsequence of $X = \langle \text{ABRACADABRA} \rangle$.

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. 42).

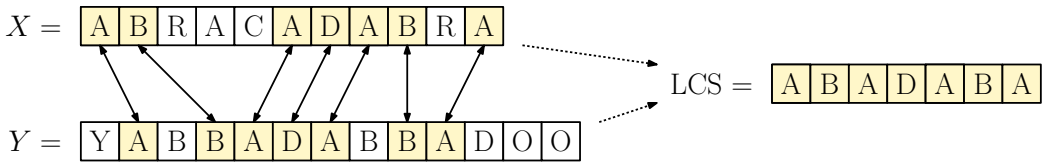


Fig. 42: An example of the LCS of two strings X and Y.

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 $\langle$ABC$\rangle$ and $\langle$BAC$\rangle$ is either $\langle$AC$\rangle$ or $\langle$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 decompose the problem into subproblems, which can be solved recursively. 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. Given $0 \leq i \leq |X|$, the ith *prefix* of X, denoted X_i, is the initial substring length i, that is, $X_i = \langle x_1, \ldots, x_i \rangle$. Define $X_0 = \langle\rangle$ to be the empty sequence.

The idea will be to compute the longest common subsequence for every possible pair of prefixes. For $0 \leq i \leq |X|$ and $\text{lcs}(i, j)$ denote the length of the longest common subsequence of X_i and Y_j. For example, in the above case we have $X_5 = \langle$ABRAC$\rangle$ and $Y_6 = \langle$YABBAD$\rangle$. Their longest common subsequence is $\langle$ABA$\rangle$. Thus, $\text{lcs}(5, 6) = 3$.

Let us start by deriving a recursive formulation for computing $\text{lcs}(i, j)$. Later, we will consider how to implement this recursion efficiently.

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

Last characters match: Suppose $x_i = y_j$. For concreteness. We assert that $\text{LCS}(X_i, Y_j)$ must end in this letter. This is shown in the following lemma.

Lemma: If two strings X and Y share the same last character, then their LCS ends in this character. Furthermore, we may assume that the LCS is constructed using this lasat instance from each string.

Proof: For concreteness, let's suppose that both X and Y end with the letter 'A'. Suppose towards a contradiction that $\text{LCS}(X, Y)$ did not end with 'A'. This means that neither string contributed its last letter 'A' to the LCS. By adding this 'A' we can create a longer common subsequence, a contradiction.

Okay, the LCS ends with 'A', but can we infer that each string contributed this particular instance of 'A'? (For example, X could have contributed an earlier instance 'A' to the LCS.) Since this is the last character of the LCS, we may replace the what instance of 'A' that X contributed with X's last letter. This is still a valid subsequence of X and it has the same length. The same argument applies to Y.

By the above lemma, we may add $x_i = y_j$ to the LCS and remove this character from both strings. To complete the LCS, we recursively compute the LCS of remaining substrings, X_{i-1} and Y_{j-1}. Since the removal of the last character has no impact on this subproblem, we should solve

it optimally.[5] Therefore, the length of the final LCS is $\text{lcs}(X_{i-1}, Y_{j-1}) + 1$ (see Fig. 43). This provides us with the following rule:

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

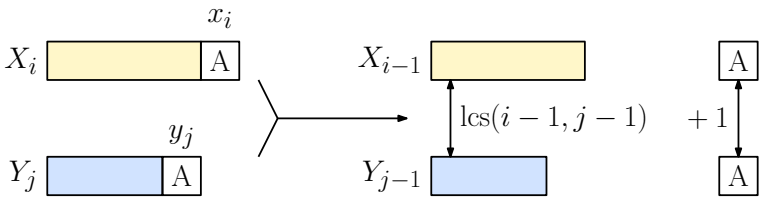


Fig. 43: LCS of two strings, where $x_i = y_j$.

For example, suppose that $X_i = \langle\text{ABCA}\rangle$ and let $Y_j = \langle\text{DACA}\rangle$. We match the final 'A' characters, compute the LCS length of $X_{i-1} = \langle\text{ABC}\rangle$ and $Y_{j-1} = \langle\text{DAC}\rangle$, which is $\langle\text{AC}\rangle$. We then 'A' back, which yields the final LCS of $\langle\text{ACA}\rangle$.

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). Let's consider these two options.

x_i **is not in the LCS:** Since we know that x_i is out, we can remove the last character from X_i, which leaves us with X_{i-1}. We continue to compute the LCS of X_{i-1} and Y_j, which is given by $\text{lcs}(i-1, j)$.

y_j **is not in the LCS:** Since we know that y_j is out, we can remove the last character from Y_j, which leaves us with Y_{j-1}. We continue to compute the LCS of of X_i and Y_{j-1}, which is given by $\text{lcs}(i, j-1)$.

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. This is not a good idea. It will lead to a more complicated, possibly less efficient algorithm. Remember the DP selection principle: *When given a set of feasible options to choose from, try them all and take the best.*

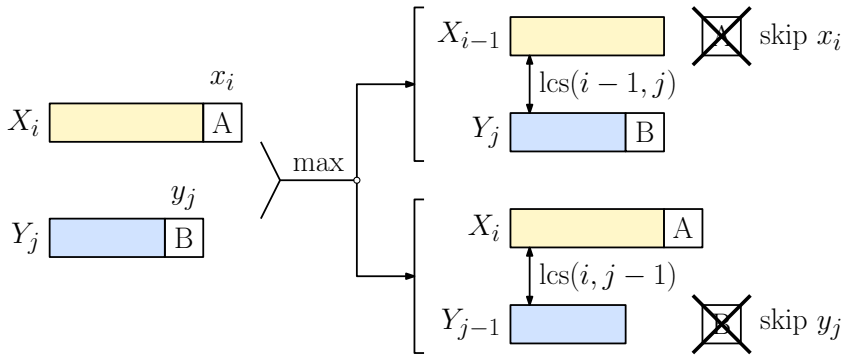


Fig. 44: LCS of two strings, where $x_i \neq y_j$.

[5]Isn't this obvious? Well no. Suppose that we altered the LCS problem to require, say, that it cannot have two consecutive equal characters. This would constrain the subproblem we generate, since it cannot end with this same character. Thus, we may prefer a suboptimal solution to the subproblem, in order to satisfy this global constraint.

We compute both options and take the one that gives us the longer LCS (see Fig. 44). (Hey, did we forget Option 3, where *neither* symbol is in the LCS? Yes, this can happen, but these two rules suffice to handle this. Try it out and you'll see.) Thus, we have the following rule:

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

Combining these observations we have the following recursive *DP formulation*:

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

The final answer is $\text{lcs}(m,n)$.

Memoized implementation: The principal source of the inefficiency in a naive implementation of the recursive rule is that it makes repeated calls to $\text{lcs}(i,j)$ for the same values of i and j. To avoid this, it creates a 2-dimensional array $\text{lcs}[0..m, 0..n]$, where $m = |X|$ and $n = |Y|$. We initialize its elements to -1, which indicates that the entry is currently undefined. The memoized version first checks whether the requested value has already been computed, and if so, it just returns the cached value. Otherwise, it invokes the recursive rule to compute it. Our objective is to compute the LCS of the entire strings of lengths m and n, so the initial call is memo-$\text{lcs}(m,n)$.

Because we will eventually want to construct the final LCS, we will also add some "hooks" to our code to record our decisions. We create a parallel *hook table*, $H[0..n, 0..m]$, which stores three possible values.

$+$: Add $x_i (= y_j)$ to the end of the LCS. (Represented by the symbol '↖'.)

X : Do not include x_i to the LCS. (Represented by the symbol '↑'.)

Y : Do not include y_j to the LCS. (Represented by the symbol '←'.)

The algorithm is presented in the code block below. The final answer is memo-$\text{lcs}(m,n)$. See Fig. 45(a) for an example. (We'll discuss the H-table later.)

Memoized LCS with Hooks

```
memo-lcs(i,j) {                                 // memoized implentation of LCS
    if (lcs[i,j] == -1) {                       // undefined?
        if (i == 0 || j == 0) {                 // basis case
            lcs[i,j] = 0
        } else if (x[i] == y[j]) {              // last characters match
            lcs[i,j] = memo-lcs(i-1, j-1) + 1
            H[i,j] = '+'
        } else {                                // last chars don't match
            skipX = memo-lcs(i-1, j)            // length if we skip X
            skipY = memo-lcs(i, j-1)            // length if we skip Y
            if (skipX >= skipY)                 // better to skip X
                lcs[i,j] = skipX; H[i,j] = 'X'
            else                                // better to skip Y
                lcs[i,j] = skipY; H[i,j] = 'Y'
        }
    }
    return lcs[i,j]                             // return lcs length
}
```

Correctness follows from the correctness of the DP formulation. The running time is $O(mn)$. To see this, observe that there are $(m+1)(n+1) = O(mn)$ entries in the table. The body of each recursive

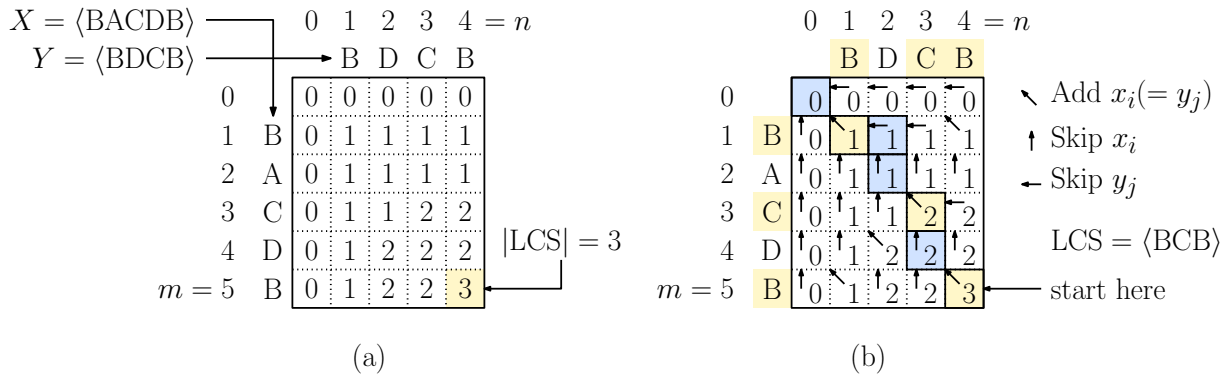


Fig. 45: (a) Contents of the lcs array for the input sequences $X = \langle \text{BACDB} \rangle$ and $Y = \langle \text{BCDB} \rangle$. The numeric table entries are the values of $\text{lcs}[i, j]$. (b) Illustrates the H-table and the extraction of the final sequence.

call runs in $O(1)$ time. Each call either returns immediately or fills in one more entry in the tables. Thus, the total time is proportional to the total number of table entries, which is $O(mn)$.

Extracting the LCS: Next, let us see how to use our hooks to extract the final LCS. We will start at the end with $H[m, n]$ and trace the optimal recursion path back to $H[0, 0]$. If $H[i, j] = +$, this means that $x_i = y_j$, and we are putting this common character into the LCS. We add this character to the LCS, and continue with $H[i - 1, j - 1]$. If $H[i, j] = X$, this means that we are skipping character x_i, and continuing with $H[i - 1, j]$. Finally, if $H[i, j] = Y$, this means that we are skipping character y_j, and continuing with $H[i, j - 1]$. Since each iterations decrements either i or j, the running time is $O(m + n)$. An example of the trace-back is shown in Fig. 45(b).

Extracting the LCS using the Hints

```
get-lcs-sequence() {                              // get the LCS sequence
    LCS = empty
    i = m; j = n                                  // start at lower right
    while( i != 0 or j != 0 )                     // loop until i == j == 0
        switch ( H[i,j] )
            case '+' ->                           // add x[i] (= y[j])
                prepend x[i] to LCS; i--; j--;
            case 'X' ->                           // skip x[i]
                i--
            case 'Y' ->                           // skip y[j]
                j--
    return LCS
}
```

Bottom-up implementation: (Optional) 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 $\text{lcs}[i, j]$, we need to have already computed $\text{lcs}[i - 1, j - 1]$, $\text{lcs}[i - 1, j]$, and $\text{lcs}[i, j - 1]$. Thus, we can compute the entries row-by-row or column-by-column in increasing order. See the code block below and Fig. 45(a). The running time and space used by the algorithm are both clearly $O(mn)$.

Edit Distance: A more widely used measure of string similarity than LCS is the edit-distance. This is widely used in the field of computational genomics, when analyzing the similarity of DNA/RNA sequences.

```
bottom-up-lcs() {                           // bottom-up implementation of 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
}
```

Given two strings $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, the edit distance is the minimum number of primitive operations needed to convert X into Y. Primitive operations include things like inserting a character, deleting a character, changing the value of a character, or swapping two adjacent characters. Generally, we may apply weights to these choices (e.g., favoring insertion over deletion). Let's keep this simple by focusing on just three operations: insert, delete, and change in the unweighted case. (For example, in Fig. 46) we show that the X and be converted Y through 9 edit operations.) The minimum number of insertions, deletions, and changes to convert one string to another is called the *Levenshtein distance* between these strings. It is named for the Soviet mathematician Vladimir Levenshtein, who invented way back in 1965.

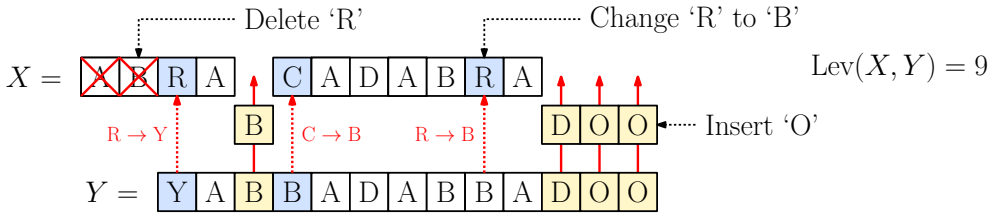


Fig. 46: Levenshtein distance for $X = \langle$ABRACADABRA$\rangle$ and $Y = \langle$YABBADABBADOO$\rangle$.

Let's develop a DP formulation for this problem. We will follow a pattern similar to the LCS problem. For $0 \le i \le m$ and $0 \le j \le n$, let $\text{Lev}(i, j)$ denote the Levenshtein distance between the prefixes $X_i = \langle x_1, \ldots, x_i \rangle$ and $Y_i = \langle y_1, \ldots, y_j \rangle$. Let's explore the various cases.

Basis: If either sequence is empty, then the edit distance is equal to the number of characters in the other string. If X_i is empty, then we need to insert all j characters of Y_j. If Y_j is empty, then we need to delete all i characters of X_i. Thus, we have following rules:

$$\begin{aligned} &\text{if } i = 0 \text{ then } \text{Lev}(i, j) = j \\ &\text{if } j = 0 \text{ then } \text{Lev}(i, j) = i \end{aligned}$$

Last characters match: If $x_i = y_j$, then we should go ahead and match these characters. (It costs us nothing to do so, and if we were to hold out to match one of these with an earlier instance of the same character, this would only limit our future options.) This does not incur any increase in the edit distance, and what remains is to match the remaining prefixes, X_{i-1} and Y_{j-1}. Since the removal of the last character has no impact on this subproblem, we should solve it optimally.

Therefore, the Leveshtein distance is $\text{Lev}(X_{i-1}, Y_{j-1})$ (see Fig. 43). This provides us with the following rule:

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

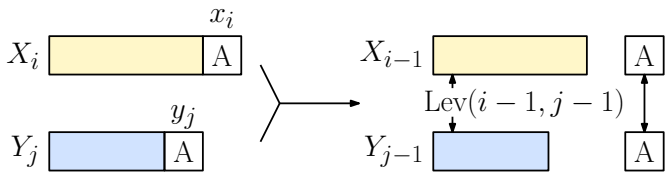


Fig. 47: LCS of two strings, where $x_i = y_j$.

For example, suppose that $X_i = \langle \text{ABCA} \rangle$ and let $Y_j = \langle \text{DACA} \rangle$. We match the final 'A' characters, compute the LCS length of $X_{i-1} = \langle \text{ABC} \rangle$ and $Y_{j-1} = \langle \text{DAC} \rangle$, which is $\langle \text{AC} \rangle$. We then 'A' back, which yields the final LCS of $\langle \text{ACA} \rangle$.

Last characters do not match: If the last character do not match, that is, $x_i \neq y_j$. We know that some edit operation will be needed, but which? There are three options (see Fig. 48).

Insert y_j at the end of X_i: This increases the distance by +1. After doing so, the character y_j has been accounted for. What remains is to compute the distance between X_i with the remainder, Y_{j-1}. In this case, $\text{Lev}(i,j) = 1 + \text{Lev}(i, j-1)$.

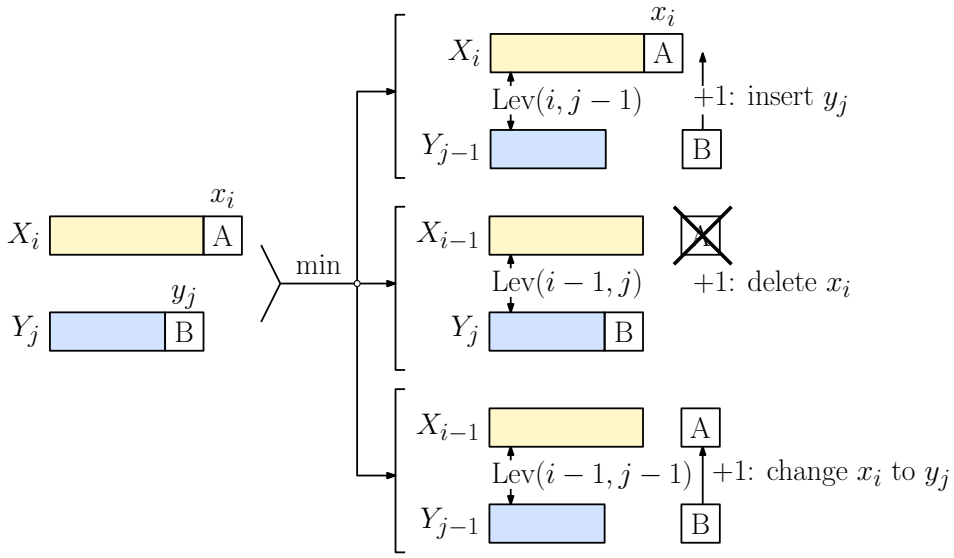


Fig. 48: LCS of two strings, where $x_i = y_j$.

Delete x_i: This increases the distance by +1. After doing so, the character x_i has been accounted for. What remains is to compute the distance between the remainder, X_{i-1}, with Y_j. In this case, $\text{Lev}(i,j) = 1 + \text{Lev}(i-1, j)$.

Change x_i into y_j: This increases the distance by +1. After doing so, both the characters x_i and y_j have been accounted for. What remains is to compute the distance between the remainders, X_{i-1} and Y_{j-1}. In this case, $\text{Lev}(i,j) = 1 + \text{Lev}(i-1, j-1)$.

At this point it may be tempting to try to make a "smart" choice. But, in customary DP fashion, we do not attempt to determine which action is best. We just try them all and take the best, that is, the one that achieves the minimum value. Thus, we have the rule:

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

In summary, we have the following recursive *DP formulation* for the Levenshtein distance:

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

The final answer is $\text{Lev}(m,n)$.

We will leave the implementation (whether memoized or bottom-up) as an exercise. Both are quite similar in structure to the LCS code. The same is true for adding the necessary "hooks" (match, insert, delete, or change). As with LCS, the running time is $O(mn)$. Once the table has been constructed along with the helpers, the edits can be reconstructed in $O(m+n)$ time.

Summary: We have presented DP algorithms for two problems in string similarity, longest common subsequence (LCS) and the edit or Levenshtein distance. Both algorithms run in time that is proportional to the product of the lengths of the two strings. Needless to say, this is unacceptably slow in many applications where string sizes can be large.

Can we do better? There are near linear-time algorithms for LCS (see Wikipedia). There are many tricks and heuristics for speeding up edit distance in practice. Unfortunately, there is pretty strong evidence that in the worst case, you cannot do much better for the Levenshtein distance. It has been proved that the Levenshtein distance for two strings of length n cannot be computed in time $O(n^{2-\varepsilon})$, for any $\varepsilon > 0$, unless the Strong Exponential Time Hypothesis (SETH, for short) is false. It is beyond the scope of this lecture to introduce SETH is, but it is widely held to be true.

Lecture 10: 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. 49). The number of columns of A must equal the number of rows of B. In particular for $1 \leq i \leq p$ and $1 \leq j \leq r$, we have

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

This corresponds to the (hopefully familiar) rule that $C[i,j]$ 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

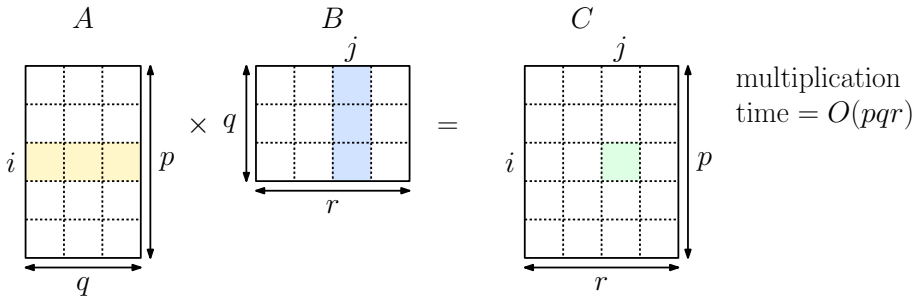


Fig. 49: Matrix Multiplication.

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 way of parenthesizing the matrices 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.

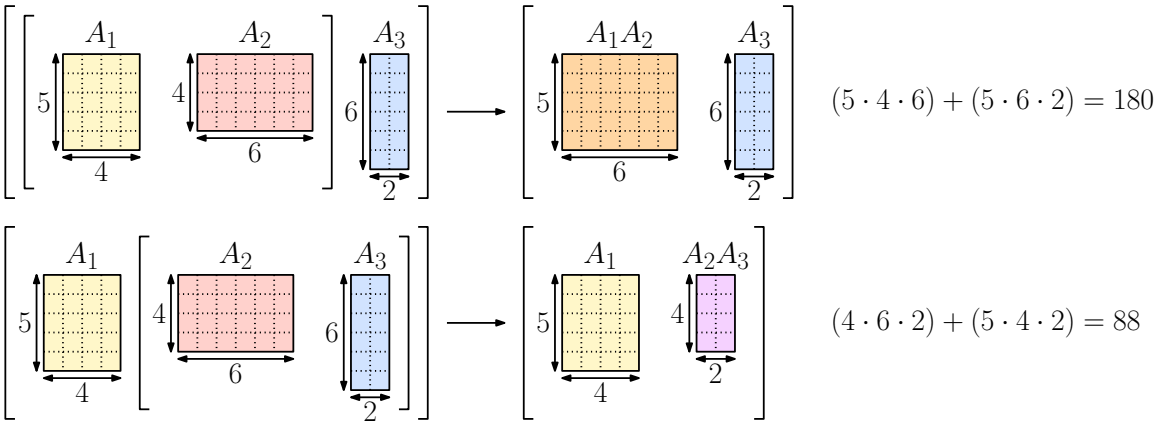


Fig. 50: Order of operations affects total operation count.

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. The output can be thought of as a binary tree whose leaves are the matrices. Indeed, there are many tree-related problems that can be solved using DP, and chain-matrix multiplication is a good archetype for these solutions.

Brute-Force Solution: We could write a procedure which tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is very large. If you have just one or two

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

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

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

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

By applying the definition of $\binom{a}{b}$ and using Stirling's formula,[6] it can be shown that $C(n)$ is $\Omega(4^n/n^{3/2})$. Since 4^n is exponential and $n^{3/2}$ is just polynomial, the exponential will dominate, implying that function grows very fast. Thus, this will not be practical except for very small n. In summary, brute force is not a reasonable option.

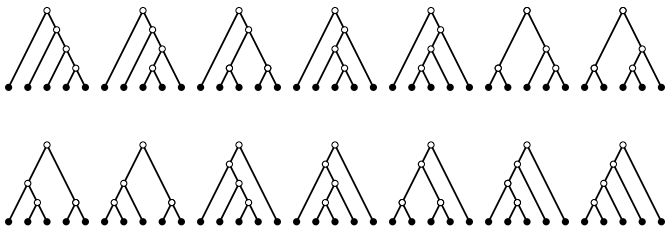


Fig. 51: Different evaluation orders for five matrices.

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. For $1 \leq i \leq j \leq n$, let $A(i,j)$ denote the result of multiplying matrices i through j, that is,

$$A(i,j) \;=\; A_i \cdot A_{i+1} \cdots A_j$$

Our subproblems will then be the minimum number of operations needed to compute $A(i,j)$, which we denote by $M(i,j)$.

Let's explore the properties of this subchain product. 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

[6] Stirling's formula provides an algebraic approximation to the factorial function. It states that, in the limit for large n, $n! \approx \sqrt{2\pi n}(n/e)^n$. From the perspective of asymptotics, this implies that $n!$ grows at least as fast as $\Omega(n^n)$.

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,

$$A(i,j) \;=\; (A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j) \;=\; A(1,k) \cdot A(k+1,n), \qquad \text{for } i \le k \le j-1.$$

Thus the problem of determining the optimal sequence involves the following issues:

- What is the best place to split the chain? (what is k?)
- How much does it cost to compute each of $A(i,k)$ and $A(k+1,j)$?
- How much does it cost for the final product $A(i,k \cdot A(k+1,j)$?

For the first issue, will apply the same ("don't be smart") idea as in other DP problems. We'll try all possible splits, compute their costs, and take the best. For the second issue, the principle of optimality (a requirement of DP solutions) applies here. In order to achieve the globally optimal solution, the subproblems of computing $A(i,k)$ and $A(k+1,j)$ should each be solved optimally. (There is no advantage to be gained by solving one subproblem suboptimally, in order to help do better on the other.)

For the final issue, recall that when you multiply a chain of matrices, the size of the result is the number of rows in the first and the number of columns in the last. Thus, $A(i,k)$ is a $p_{i-1} \times p_k$ matrix, and $A(k+1,j)$ is a $p_k \times p_j$ matrix. It follows that the time to multiply them is $p_{i-1}p_kp_j$. We now have everything we need to give the recursive formulation.

Recursive formulation: For $1 \le i \le j \le n$, recall that $M(i,j)$ denotes the minimum cost (number of operations) needed to compute the product $A(i,j) = A_iA_{i+1}\ldots A_j$. 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, $M(i,i) = 0$.

General case: If $i < j$, then we are asking about the product $A(i,j)$. This can be split into two groups $A(i,k)$ times $A(k+1,j)$, by considering each k, $i \le k < j$ (see Fig. 52) and taking the best.

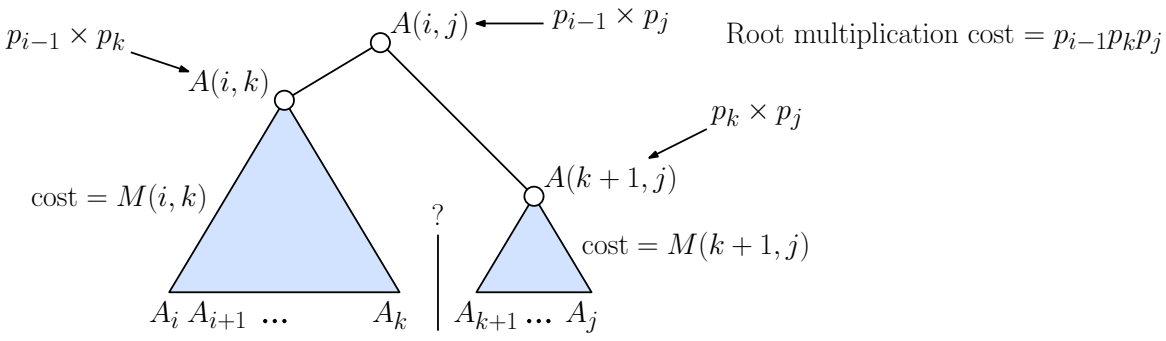


Fig. 52: Dynamic programming recursive formulation.

As observed above, the cost of computing $A(i,k)$ is given recursively by $M(i,k)$, the cost of computing $A(k+1,j)$ is given by $M(k+1,j)$, and the time needed for the final product is $p_{i-1}p_kp_j$. Thus the cost is $M(i,k) + M(k+1,j) + p_{i-1}p_kp_j$. Since we don't know the best choice for k, we try all possibilities.

This yields the following DP formulation:

$$M(i,j) \;=\; \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k \le j-1} \big(M(i,k) + M(k+1,j) + p_{i-1} p_k p_j \big) & \text{if } i < j. \end{cases}$$

The final answer is the product of all the matrices, that is, $M(1,n)$.

Memoized 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 and bottom-up. Let's first present the memoized version. Recall that this involves designing a recursive function which saves results once computed. To do this, we create a table $M[1..n, 1..n]$, where $M[i,j]$ will store the function value $M(i,j)$. Initially, all entries are set to -1, which indicates that they are undefined. Eventually, we are interested in $M[1,n]$ as the final count of the number of operations to multiply all the matrices. In addition, we store a parallel table of "hooks" to allow us to reconstruct the optimal sequence. For each entry $M[i,j]$, it stores the splitting index k that led to the minimum cost. Later we'll see how it is used.

Memoized Chain Matrix Multiplication

```
memo-cmm(i, j) {                                  // memoized chain matrix mult
    if (M[i, j] == -1) {                          // undefined?
        if (i == j) M[i, j] = 0                   // basis case
        else {
            minCost = +INFINITY
            for (k = i to j - 1) {                // get costs for all splits
                cost = memo-cmm(i, k) + memo-cmm(k+1, j) + p[i-1]*p[k]*p[j]
                if (cost < minCost) {             // found a new best?
                    minCost = cost                // ...save it
                    H[i, j] = k                   // ...save the split index
                }
            }
            M[i, j] = minCost                     // save the best cost
        }
    }
    return M[i, j]                                // return table entry
}
```

The final answer is `memo-cmm`$(1,n)$. An example is shown in Fig. 53. We have turned the matrix on its side to better illustrate its relationship to the binary tree.

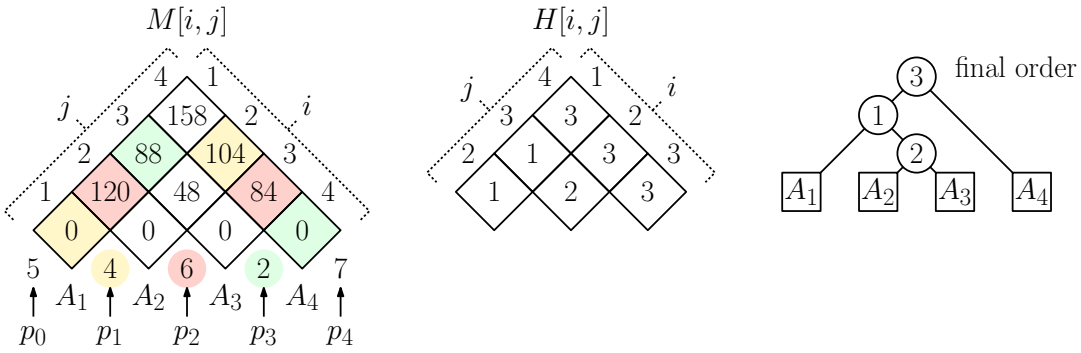


Fig. 53: Chain matrix multiplication for the $A_1 \cdot A_2 \cdots A_4$, where $\langle p_0, \ldots, p_4 \rangle = \langle 5, 4, 6, 2, 7 \rangle$.

For example, when computing $M[1,4]$ in the above example, we take the minimum of the following three options:

$$\begin{aligned}
(k=1)\ M[1,1]+M[2,4]+p_0\cdot p_1\cdot p_4 &= 0+104+140 = 244\\
(k=2)\ M[1,2]+M[3,4]+p_0\cdot p_2\cdot p_4 &= 120+84+210 = 414\\
(k=3)\ M[1,3]+M[4,4]+p_0\cdot p_3\cdot p_4 &= 88+0+70 = 158.
\end{aligned}$$

Clearly, the best choice is 158, so $M[1,4]=158$ and (since this happened when $k=3$) $H[1,4]=3$.

The running time of the procedure is $O(n^3)$. There are $O(n^2)$ table entries to be filled. For each table entry, we need to iterate through the $j-i$ possible splitting points. Since $1\le i\le j\le n$, $j-i\le n-1$, and so it takes $O(n)$ time in the worst case to compute each table entry. (A more careful analysis shows that the total number of operations grows roughly as $n^3/6$.)

Extracting the final Sequence: Extracting the actual multiplication sequence is a fairly easy extension. Recall that when we compute $M[i,j]$, we store the optimal split index k in $H[i,j]$. This tells us that the best way to multiply the subchain $A(i,j)$ is to first multiply the subchain $A(i,k)$ and then multiply the subchain $A(k+1,j)$, and finally multiply these together. Intuitively, $H[i,j]$ tells us what multiplication to perform *last*.

The multiplication algorithm is given in the code block below. The initial call is do-mult$(1,n)$. Since each invocation takes $O(1)$ time and introduces a new split, the total running time is $O(n)$. An example is given in Fig. 53. (It's a good idea to trace through this example to be sure you understand it.)

Extracting Optimum Sequence

```
do-mult(i, j) {                          // multiply the matrices
    if (i == j)                          // basis case
        return A[i]
    else {
        k = H[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
    }
}
```

Bottom-up implementation: The bottom-up process fills the array $M[1..n,1..n]$ by a purely iterative process. This is a bit tricky, however!

You might think that we can just fill the table row-by-row, and column-by-column (as we did with the longest common subsequence problem). However, this simple approach will not work here. 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!

The trick is to compute the matrix *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 implement this, for $1\le i\le j\le n$, let $\ell=j-i+1$ denote the length of the subchain being multiplied. How shall we set up the loops to do this? The case $\ell=1$ (that is, entries of the form $M[i,i]$) 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 $\ell = 2, \ldots, n$. If a subchain of length ℓ starts at position i, then $j = i + \ell - 1$. How high should i go (so we don't index out of bounds)? Since we want $j \leq n$, we have

$$i + \ell - 1 \ \leq \ n, \qquad \text{or equivalently} \qquad i \ \leq \ n - \ell + 1.$$

So our inner loop will be based on i running from 1 up to $n - \ell + 1$. The procedure is presented in the code block below. (Also, see Fig. 53 for an example.) We will skip the H array, but it can easily be added here.

Bottom-Up Chain Matrix Multiplication

```
bottom-up-cmm() {                                  // bottom-up chain matrix mult
    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) {                 // generate all chains of length L
            j = i + L - 1
            minCost = INFINITY
            for (k = i to j - 1)                   // check all splits
                minCost = min(minCost, M[i, k] + M[k+1, j] + p[i-1]*p[k]*p[j])
            M[i, j] = minCost                      // save optimum cost
        }
    }
    return M[1, n]                                 // return the total cost
}
```

Summary: We have presented an $O(n^3)$ time for the problem of determining the best way to multiply n matrices (of various dimensionalities) together. Who cares about this problem? Frankly, I know of no compelling applications. However, there are numerous applications of DP which involve a tree-like partition of the solutions space, and this is an easy example of the general structure.

An interesting application is that of computing the optimum binary search tree for a set keys, where the keys have distinct probabilities of being accessed. This problem is discussed in this Wikipedia article. (The algorithm that we presented corresponds to the naive implementaiton of Knuth's algorithm, which is mentioned in the article.)

Lecture 11: All-Pairs Shortest Paths and the Floyd-Warshall Algorithm

All-Pairs Shortest Paths: Earlier, we saw that Dijkstra's algorithm and the Bellman-Ford algorithm both solved the problem of computing shortest paths in graphs from a single source vertex. Suppose that we want instead to compute shortest paths between *all pairs* of vertices. We could do this applying either Dijkstra and/or Bellman-Ford using every vertex as a source (and this might be the fastest, especially if the graph is sparse). Today, we will consider an alternative approach, which is based on dynamic programming.

Let $G = (V, E)$ be a directed graph with edge weights. For each edge (u, v) E, let $w(u, v)$ denote its weight. As before, the *cost* of a path is the sum of its edge weights, and the *distance* between two vertices is the minimum cost of any path between u and v. We allow negative weight edges, but no negative-cost cycles. (Recall that a negative-cost cycle implies that the shortest path may not be defined, since you make the cost arbitrarily small by repeating the cycle.)

The algorithm that we will present is called the *Floyd-Warshall algorithm*, which was discovered independently at about the same time by Robert Floyd and Stephen Warshall. It runs in $O(n^3)$ time,

where $n = |V|$. It dates back to the early 1960's. The idea is quite natural. It is closely related to Kleene's algorithm (which converts a finite-state automaton to a regular expression). The algorithm was actually discovered even earlier by a French computer scientiest Bernard Roy.

The algorithm can be adapted for use in a number of related applications as well.

Transitive Closure: (This was the application Warshall was interested in). You are given a *binary relation* R on a set X, by which we mean that R is a subset of ordered pairs $(x, y) \subseteq X \times X$. A relation is said to be *transitive* if for any $x, y, z \in X$, if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$. The *transitive closure* of R, denoted R^* is the smallest extension of R that is transitive.

We can think of (X, R) as a directed graph, where X are the vertices and the pairs of R are edges. The transitive closure of R is effectively the same as the *reachability* relation in this graph, that is, $(x, y) \in R^*$ if and only if there exists a path from x to y in R. The Floyd-Warshall algorithm can be modified to compute the transitive closure in time $O(n^3)$, where $n = |X|$.

All-Pairs Max-Capacity Paths: Let $G = (V, E)$ be a directed graph with positive edge capacities $c(u, v)$. Think of each edge as a pipe, and the capacity $c(u, v)$ as the amount of flow that can be pushed through this pipe per unit interval. The *capacity* of a path is the minimum capacity of any edge along the path. (Intuitively, the minimum capacity edge forms a bottleneck, which limits the total amount of flow along the path. By the way, the capacity of the trivial path from u to u is $+\infty$.)

Given any two nodes u and v, we would like to know the maximum capacity path between them. It is easy to modify the Floyd-Warshall algorithm to compute the maximum capacity path between every pair of vertices in $O(n^3)$ time, where $n = |V|$.

Input/Output Representation: We assume that the digraph is represented as an adjacency matrix, rather than the more common adjacency list. (Adjacency lists are generally more efficient for sparse graphs, but storing all the inter-vertex distances will require $\Omega(n^2)$ storage anyway.) 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.

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.

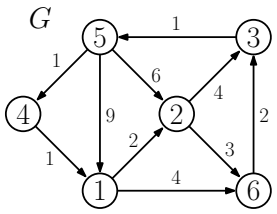
$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i, j) & \text{if } i \neq j \text{ and } (i, j) \in E, \\ +\infty & \text{if } i \neq j \text{ and } (i, j) \notin E. \end{cases}$$

(See Fig. 54(b).) 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.

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 (see Fig. 54(c)). This provides the distance, but how to get the shortest paths? You might think that that this will require $O(n^3)$ storage, since there are $\binom{n}{2} = O(n^2)$ pairs of vertices, and each shortest path might need up to $n - 1$ edges. However, there is a cute trick for reducing the storage to $O(n^2)$. We will create an additional matrix $H = h_{ij}$ of "hooks," which is defined as follows.

$$h_{ij} = \begin{cases} \emptyset & \text{if } i = j \text{ or shortest path is single edge } (i, j) \\ k & \text{where } k \text{ is any vertex along shortest path.} \end{cases}$$

As we shall see below, this allows us to reconstruct a shortest path of any length, by repeatedly inserting a vertex into the path. (This exploits the fact that every subpath of a shortest path is also a shortest path.)



(a)

w

	1	2	3	4	5	6
1	0	2	∞	∞	∞	4
2	∞	0	4	∞	∞	3
3	∞	∞	0	∞	1	∞
4	1	∞	∞	0	∞	∞
5	9	6	∞	1	0	∞
6	∞	∞	2	∞	∞	0

(b)

d

	1	2	3	4	5	6
1	0	2	6	8	7	4
2	7	0	4	6	5	3
3	3	5	0	4	1	7
4	1	3	7	0	8	5
5	1	4	8	1	0	6
6	5	7	2	4	3	0

(c)

Fig. 54: (a) A weighted digraph G, (b) the weight matrix w, and (c) the distance matrix d.

Floyd-Warshall Algorithm: As with any DP algorithm, the key is reducing a large problem to smaller problems. What will these subproblems be? A natural way of doing this is to follow an approach similar to that of Bellman-Ford, constructing shortest paths based on the *number of edges* in the shortest path. For example, we might define $d_{ij}^{(\ell)}$ to be the shortest path from i to j that uses at most ℓ edges. For the basis case, $d_{ij}^{(1)}$ would just be w_{ij}. Then, we could build up larger paths through repeated doubling. The shortest path using at most 2ℓ edges is built by concatenating two paths of length at most ℓ through all possible intermediate vertices k. For example, we might try the following:

$$d_{ij}^{(2\ell)} = \min_{1 \le k \le n} \left(d_{ik}^{(\ell)} + d_{kj}^{(\ell)} \right). \qquad \text{(Possible DP alternative to Floyd-Warshall)}$$

(What would the running time of this approach be if implemented? I'll leave it as an exercise, but it will be *slower* than the Floyd-Warshall algorithm.)

Rather than restricting the *number* of edges on the path, the trick is to restrict the *set of intermediate vertices* that the path is allowed to use. Given a path $p = \langle v_1, v_2, \ldots, v_\ell \rangle$, we refer to the vertices $v_2, \ldots, v_{\ell-1}$ as the *intermediate vertices* of this path. Note that a path consisting of a single edge has no intermediate vertices.

- Given $1 \le i, j \le n$, and $0 \le k \le n$, define $d_{ij}^{(k)}$ to be the cost of the shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, \ldots, k\}$.

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 $\{1, \ldots, k\}$. (It does not need to visit all of these vertices, and it may visit none of them.) 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. 55(a), notice how the value of $d_{5,6}^{(k)}$ changes as k varies.

How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous matrix $d^{(k-1)}$? For the basis case ($k = 0$) the path cannot go through any intermediate vertices, and so $d_{ij}^{(0)} = w(i, j)$ for all i, j. For the induction step ($k \ge 1$), there are two 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 $d_{ij}^{(k-1)}$.

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

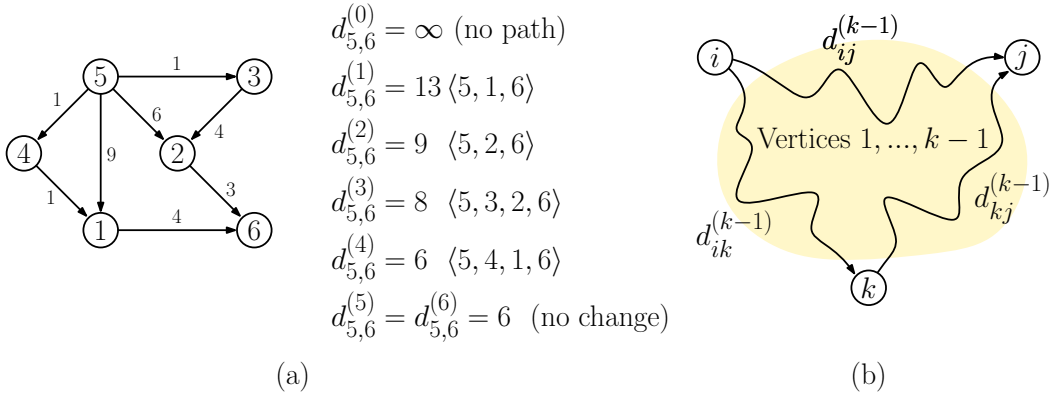


Fig. 55: Limiting intermediate vertices. For example $d^{(3)}_{5,6}$ can go through any combination of the intermediate vertices $\{1,2,3\}$, of which $\langle 5,3,2,6\rangle$ has the lowest cost of 8.

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 both of these paths use intermediate vertices only in $\{1,\ldots,k-1\}$, the length of the path is $d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$.[7]

From the above discussion, we have the following recursive rule (the DP formulation) for computing $d^{(k)}$, which is illustrated in Fig. 55(b).

$$d^{(k)}_{ij} \;=\; \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}\right) & \text{if } k \geq 1. \end{cases}$$

The final answer is $d^{(n)}_{ij}$, as this allows all possible vertices as intermediate vertices.

Bottom-up Implementation: As with other DP problems, a recursive implementation of this rule would be prohibitively slow because the same values may be reevaluated many times. While we could use memoization, we will present a bottom-up implementation.

We will maintain a matrix $d[1..n, 1..n]$. Because the function $d^{(k)}_{ij}$ involves three parameters, a faithful implementation of the above rule would involve a 3-dimensional array, $d[i,j,k]$. We shall see, however, that the algorithm will compute the same final result even if we "cheat" by ignoring the third (k) component.

As usual, we will store a set of "hooks" to save the decisions made. This takes the form of a parallel matrix, $H[1..n, 1..n]$. If we decide that it is best to go through vertex k, we will set $h[i,j] \leftarrow k$ to record this decision. Recall that it stores any vertex along the shortest path from i to j. If the path goes through k, then we can store k as the hook. The complete algorithm is presented in the code fragment below. An example of the algorithm's execution is shown in Fig. 56.

Clearly, the algorithm's running time is $O(n^3)$. The space used by the algorithm is $O(n^2)$.

Overwriting Entries: Recall that a faithful would involve a 3-dimensional array $d[i,j,k]$, and we cheated by omitting the kth component. (Actually, we don't really need to store a separate matrix for every value of k, since at any time we are only using two matrices, one for the current value, k, and one for the previous value, $k-1$.) By omitting this third parameter, we might overwrite some entry $d[i,j]$

[7] Although the figure suggests that $i \neq j \neq k$, you should convince yourself that this holds even if some combination of i, j, and k are equal to each other. For example, if $j = k$, then $d^{(k-1)}_{kj} = d^{(k-1)}_{jj} = w(j,j) = 0$, and so the formula degenerates to $d^{(k-1)}_{ik} + d^{(k-1)}_{kj} = d^{(k-1)}_{ik} = d^{(k-1)}_{ij}$.

```
floyd-warshall(w[1..n, 1..n]) {              // all-pairs shortest distances
    for (i = 1 to n) {                       // initialization (k = 0)
        for (j = 1 to n) {
            d[i, j] = w[i, j]                // initial distance is edge weight
            h[i, j] = null                   // no intermediate vertices
        }
    }
    for (k = 1 to n) {                       // add vertex k as a possible intermediate
        for (i = 1 to n) {                   // ...from i
            for (j = 1 to n) {               // ...to j
                newCost = d[i, k] + d[k, j] // cost if we go through k
                if (newCost < d[i, j]) {     // is it better?
                    d[i, j] = newCost        // update distance
                    h[i, j] = k              // save k as new intermediate
                }
            }
        }
    }
    return d                                 // d[i,j] holds the distance from i to j
}
```

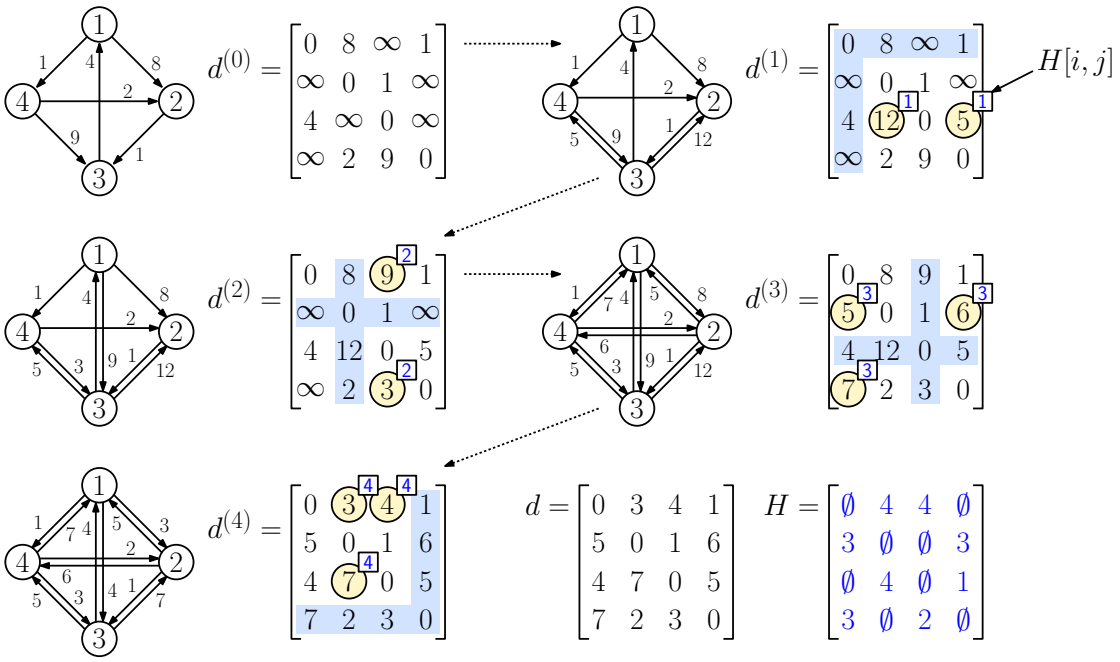


Fig. 56: Floyd-Warshall example. Newly updates entries are circled, and H matrix entries are shown in blue.

and later attempt to access its value. We will show that our laziness does not affect the algorithm's correctness.

Let us consider some iteration k. Overwriting is an issue whenever we alter the value of $d[i, j]$ by setting it to `newCost`, and later we attempt to read this same entry to compute the value of some other entry $d[i', j']$. Which entries are used to compute `newCost`? During iteration k, these are entries of the form $d[i, k]$ or $d[k, j]$. Thus, if entry $d[i, j]$ was overwritten and then reused, it must either be that $d[i, j]$ and $d[i, k]$ are the same entry (implying that $j = k$) or $d[i, j]$ and $d[k, j]$ are the same entry (implying that $i = k$).

In summary, the only instances where overwriting $d[i, j]$ can cause an issue occurs either when $i = k$ or $j = k$. Let's see what happens in case when $i = k$.

$$\texttt{newCost} \;=\; d[i,k] + d[k,j] \;=\; d[i,i] + d[i,j] \;=\; d[i,j] \qquad (\text{since } d[i,i] = 0).$$

By inspecting the pseudocode, we see that the overwriting takes place only if $\texttt{newCost} < d[i, j]$. But since we have shown that $\texttt{newCost} = d[i, j]$, this clearly cannot happen. Thus, $d[i, j]$ is *not overwritten*, and there is no problem. (A similar argument applies for the $j = k$ case.) Thus, we pay no price for our laziness. (If only this were true for life in general!)

Extracting the Shortest Path: Let's next see how to use the hook values $h[i, j]$ to extract the shortest path. Recall that whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set $h[i, j] = k$. If the shortest path does not pass through any intermediate vertex, then $h[i, j] = \texttt{null}$. To find the shortest path from i to j, we consult $h[i, j]$. If it is `null`, then the shortest path is just the direct path along edge (i, j). Otherwise, we recursively compute the shortest path from i to $h[i, j]$ and concatenate this with the shortest path from $h[i, j]$ to j. Since each invocation takes $O(1)$ time, and a shortest path can have at most $n - 1$ edges, the total running time is $O(n)$.

If $d[i, j] = \infty$, there is no path (j is not reachable from i). Otherwise, the function `get-path`(i, j) shown in the following code block returns the sequence of edges along the shortest path. An example is presented in Fig. 57. Note that the running time is proportional to the number of edges on the path.

Printing the Shortest Path

```
get-path(i, j) {                          // return edges on the shortest path
    if (h[i, j] == null)                  // no intermediate?
        return (i, j)                     // shortest path is a single edge
    else {                                // path goes through h[i,j]
        mid = h[i, j]                     // concatenate paths (i --> mid) + (mid --> j)
        return get-path(i, mid) + get-path(mid, j)
    }
}
```

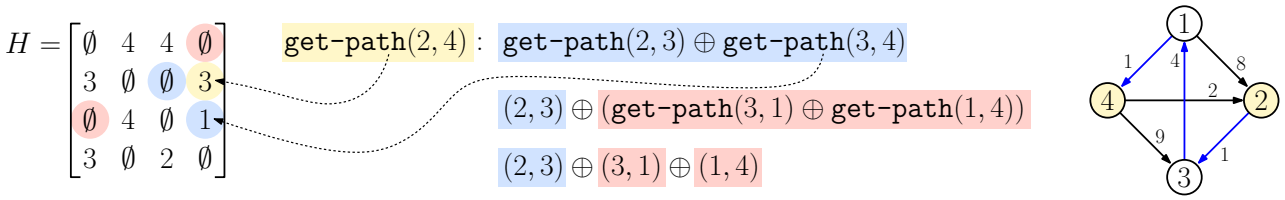


Fig. 57: Extracting the shortest path from 2 to 4.

Summary: We have presented the Floyd-Warshall algorithm, an $O(n^3)$, DP-based algorithm for computing all-pairs shortest paths in a directed graph. The algorithm works even if G has negative cost edges, as

long as there are no negative-cost cycles. The general algorithm structure can be used for a number of other applications, such as computing the transitive closure of a binary relation (which is equivalent to computing the reachability matrix in a digraph). It can be easily modified for computing a number of other all-pairs path-related problems as well.

Lecture 12: Network Flows: Basic Concepts

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

Although the network flow problem is defined in terms of the metaphor of pushing fluids, this problem and its many variations find remarkably diverse applications. These are often studied in the area of operations research. The network flow problem is also of interest because it is a restricted version of a more general optimization problem, called *linear programming*. A good understanding of network flows is helpful in obtaining a deeper understanding of linear programming.

Flow Networks: A *flow network* is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonegative *capacity* $c(u, v) \geq 0$. (In our book, the capacity of edge e is denoted by c_e.) If $(u, v) \notin E$ we model this by setting $c(u, v) = 0$. There are two special vertices: a *source* s, and a *sink* t (see Fig. 58).

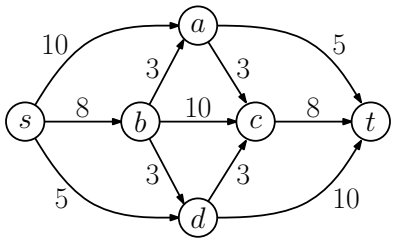


Fig. 58: A flow network.

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

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

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

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

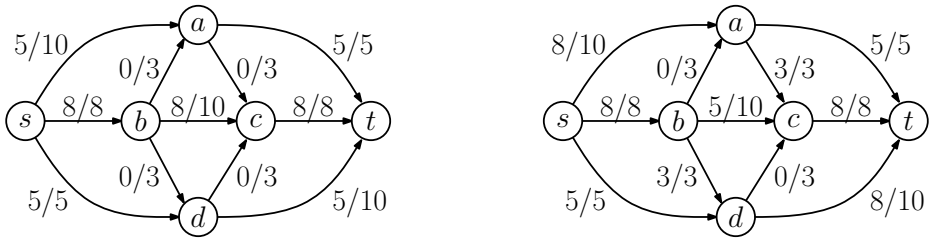
[8]Neither of these is an essential requirement. Given a network that fails to satisfy these assumptions, we can easily generate an equivalent one that satisfies both.

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

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

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. 59, 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. 59: 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. 59(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. 59(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.

This problem has been the subject of a great deal of study. Here is a history of some of some of the highlight results. Recall that $n = |V|$ and $m = |E|$, and let C denote the sum of the edge capacities (assumed to be integers).

- Ford-Fulkerson (1956) - Did not analyze the running time, but roughly $O((n + m)C)$. The same Ford of DP fame and the Bellman-Ford algorithm.
- Dinitz (1970) - $O(n^2m)$. This is often called "Dinic's algorithm", due to a misspelling of the name.
- Edmonds-Karp (1972) - $O(nm^2)$. This incremental algorithm can also be used to compute optimal weighted matchings of all sizes in bipartite graphs. Richard Karp is famous for being one of the founders of NP-completeness (with Stephen Cook).

- Gabow (1985) - $O(nm \log C)$. This paper introduced an interesting optimization technique called *scaling*, where problems are solved at increasingly higher levels of accuracy.
- Goldberg-Tarjan (1986) - $O(nm \log(n^2/m))$ time. This is considered the asymptotically fastest algorithm for Network Flows. This is the same Tarjan known for DFS, Union-Find, and the most efficient version of Dijkstra's algorithm.
- Orlin (2013) - $O(nm)$ time. This algorithm is quite complex, and is based on modifying an earlier algorithm by King, Rao, and Tarjan.

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

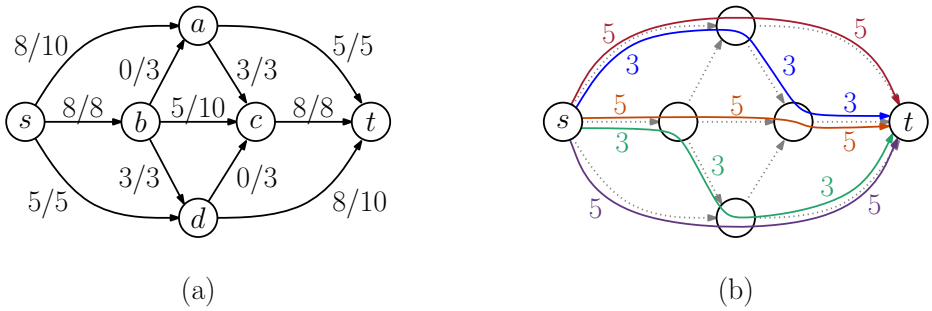


Fig. 60: (a) An edge-based flow and (b) its path-based equivalent.

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. 61). 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).

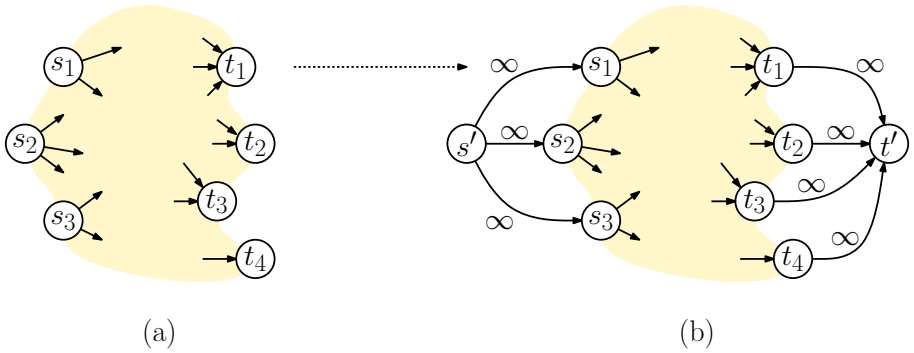


Fig. 61: Reduction from (a) multi-source/multi-sink to (b) single-source/single-sink.

Why Greedy Fails: Before considering algorithms for solving network flows, let's first consider why a simple greedy scheme for computing the maximum flow fails. 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. 62(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. 62(b)) we see that there is no path of positive capacity from s to t. Thus, greedy gets stuck.

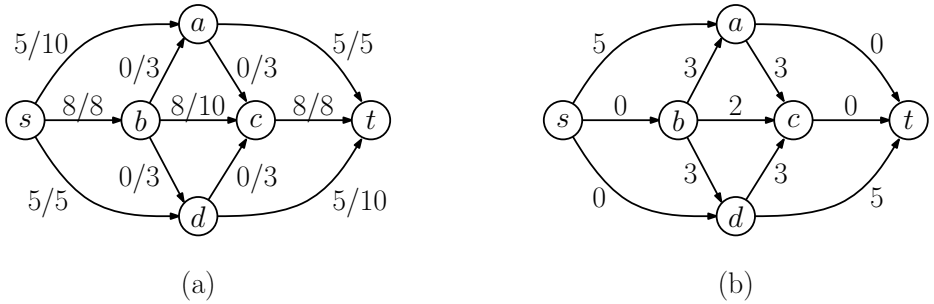


Fig. 62: The greedy flow algorithm can get stuck before finding the maximum flow.

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

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

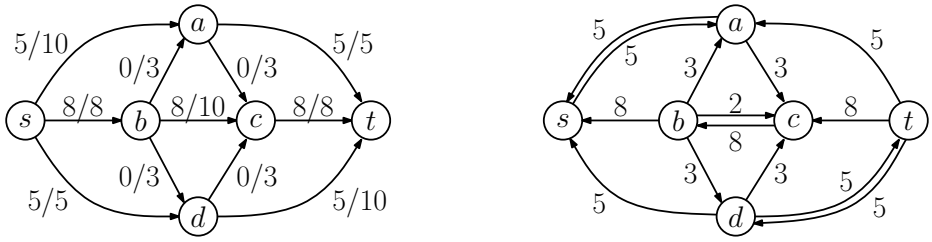
Forward edges: For each edge (u, v) for which $f(u, v) < c(u, v)$, create an edge (u, v) in G_f and assign it the capacity $c_f(u, v) = c(u, v) - f(u, v)$. Intuitively, this edge signifies that we can *increase* the

flow along this edge by up to $c_f(u, v)$ units 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 *decrease* the existing flow by as much as $c_f(u, v)$ units. 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. 63(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. 63: 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 $f''(u, v) = f(u, v) + f'(u, v)$. When a flow is pushed in a direction that is *opposite* to the edge, the addition process effectively *reduces* the flow on the edge. That is, $f''(u, v) = f(u, v) - f'(v, u)$. Clearly, the value of $f + f'$ is equal to $|f| + |f'|$.

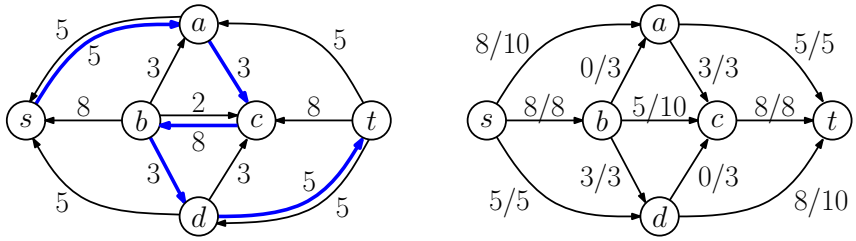
Lemma: Let f be a flow in G and let f' be a flow in G_f. Then $(f + f')$ is a flow in G.

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. 64(a) we show an augmenting path of capacity 3 in the residual network for the flow given earlier in Fig. 63. 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. 64: 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 various strategies in future lectures.) It is summarized in the code fragment below and an example is shown in Fig. 65.

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

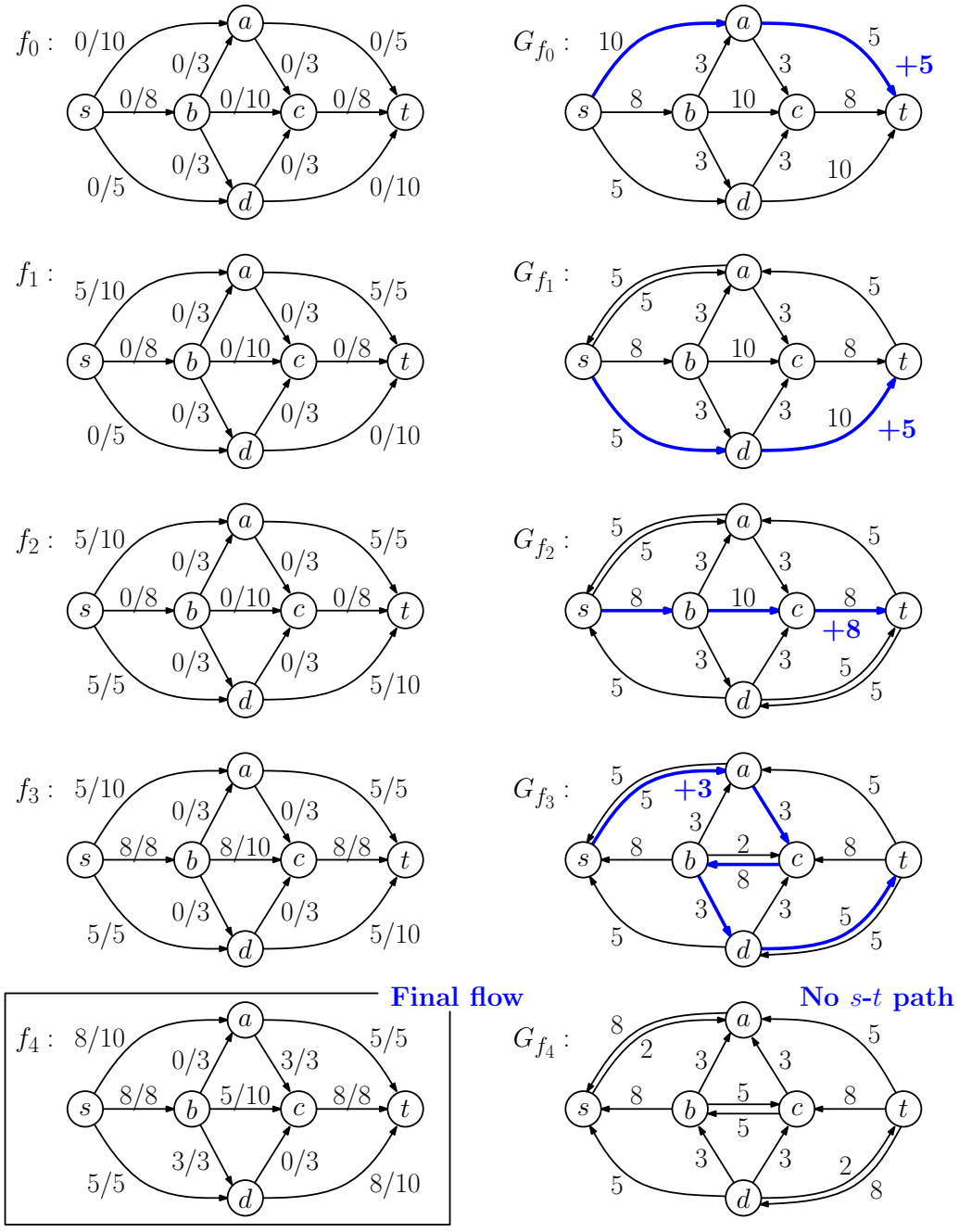


Fig. 65: The Ford-Fulkerson Algorithm. The final total flow is $5 + 5 + 8 + 3 = 21$.

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 and Y (that is, $X \cup Y = V$ and $X \cap Y = \emptyset$) such that $s \in X$ and $t \in Y$. 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,

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

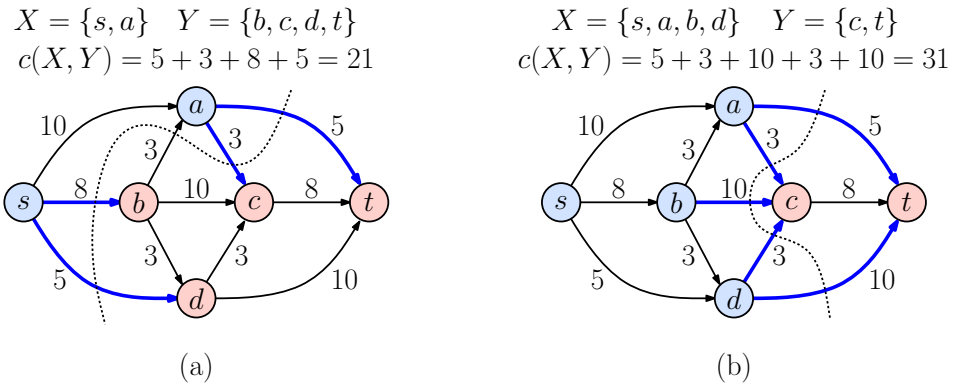


Fig. 66: (a) A cut $X = \{s, a\}$, $Y = \{b, c, d, t\}$) of capacity 5+3+8+5 = 21 and (b) another cut $X = \{s, a, b, d\}$, $Y = \{c, t\}$ of capacity $5 + 3 + 10 + 3 + 10 = 31$.

Given a flow f in G, define the *net flow* across the cut (X, Y) to be the sum of flows from X to Y minus the sum of flows from Y to X, that is,

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

(see Fig. 67). Observe that $f(X, Y) = -f(Y, X)$.

There is a notable asymmetry between net flows and cut capacities. The capacity considers only edges directed from X to Y, whereas the net flow considers both directions. Fig. 67 shows that the net flow across two different cuts is the same, even though these cuts have different capacities. The following lemma shows that this is generally true for all cuts.

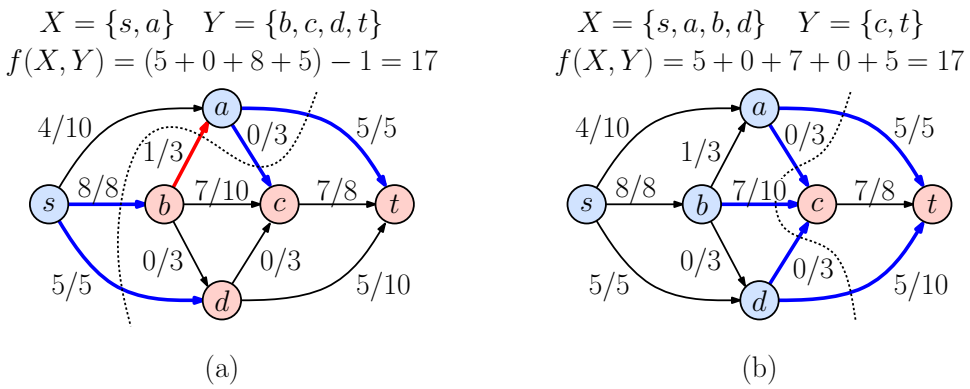


Fig. 67: The flow f across both cuts has the same net flow value, which is equal to $|f| = 17$.

Lemma: Given any network G, any flow f, and any cut (X, Y), the flow value 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 $|f| = f(X, Y)$.

In the net flow, we consider both the flows from X to Y and (negated) from Y to X. In contrast, in the definition of cut capacity, we only consider capacities from X to Y. To understand why this asymmetry exists, suppose that there there are two edges (x, y) and (y, x), where $x \in X$ and $y \in Y$, where the edges have the same capacities and both carry the same flow. The flow from y to x effectively cancels out the flow from x to y (as if you have an "eddy" in the middle of a river, where the flow cycles around). Thus, this cycle does not contribute to the total flow value, and so it is important to consider both in the net flow. On the other hand, the capacity of the edge from y to x does not contribute to nor detract from the maximum amount of flow we can push from the X-side to the Y-side of the cut. Therefore, we do not need to consider it in computing the cut's total capacity.

It is easy to see that 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 network G, any flow f, and any cut (X, Y), the flow value cannot exceed the cut's capacity, that is, $|f| \leq c(X, Y)$.

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) $\Rightarrow$ (ii): (by contradiction) If f is a max flow and there were an augmenting path in G_f, then by pushing flow along this path we would have a larger flow, a contradiction.
- (ii) $\Rightarrow$ (iii): If there are no augmenting paths then s and t are not connected in the residual network (see Fig. 68(a)). Let X be those vertices reachable from s in the residual network, and let Y be the rest. Since there is no augmenting path, $t \in Y$, which implies that (X, Y) forms a cut (see Fig. 68(b)). This also implies that each forward edge $(x, y) \in X \times Y$ is saturated, meaning that $f(x, y) = c(x, y)$ (blue edges in Fig. 68(c)), and each backward edge $(y, x) \in Y \times X$ carries no flow, that is, $f(y, x) = 0$ (red edges in Fig. 68(c)). It follows that the flow across this cut equals the capacity of the cut, that is, $f(X, Y) = c(X, Y)$. By the previous lemma $|f| = f(X, Y)$, and hence we have $|f| = c(X, Y)$.

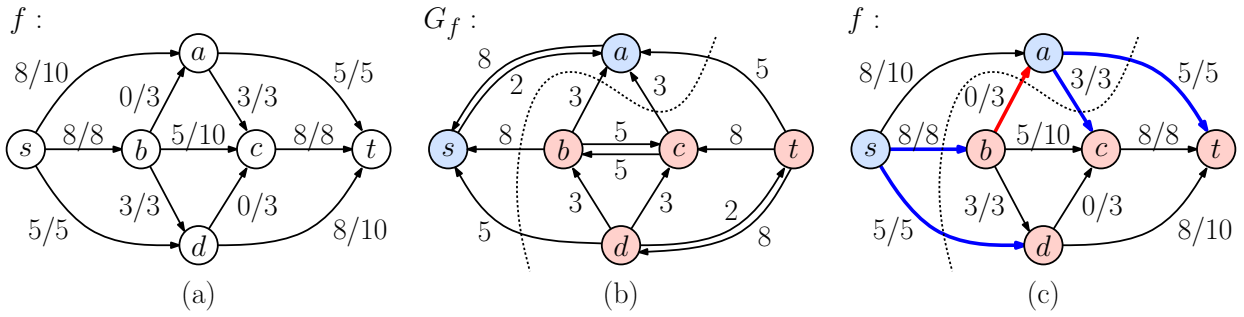


Fig. 68: Proof of the Max-Flow/Min-Cut Theorem.

- (iii) $\Rightarrow$ (i): This follows directly from the previous lemma. No flow value can exceed any cut capacity, and so if *any* flow's value matches *any* cut's capacity, this flow must be maximum.

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 13: Network Flow Algorithms

Algorithmic Aspects of Network Flow: In the previous lecture, we introduced the network-flow problem. We discussed concepts like the residual network, augmenting paths, the Ford-Fulkerson algorithm, cuts, and the Max-Flow/Min-Cut Theorem. We used the Max-Flow/Min-Cut Theorem to prove that the Ford-Fulkerson algorithm is correct. Before reading this lecture, please refer back to that one for these definitions. In this lecture we discuss the running time of the Ford-Fulkerson algorithm, and introduce some more efficient alternatives.

Analysis of Ford-Fulkerson: Before discussing the worst-case running time of the Ford-Fulkerson algorithm, let us first consider whether it is guaranteed to terminate. We assume that all edge capacities are integers.[9] Every augmentation by Ford-Fulkerson increases the flow by a strictly positive integer amount. Since the maximum flow is finite, Ford-Fulkerson must terminate after a finite number of iterations. Thus, we have the following.

[9]This is reasonable, since we can always apply a uniform scale factor to convert fractional values to integers.

Lemma: Given an s-t network with integer capacities, the Ford-Fulkerson algorithm terminates. Furthermore, it produces an integer-valued flow function.

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. 69.

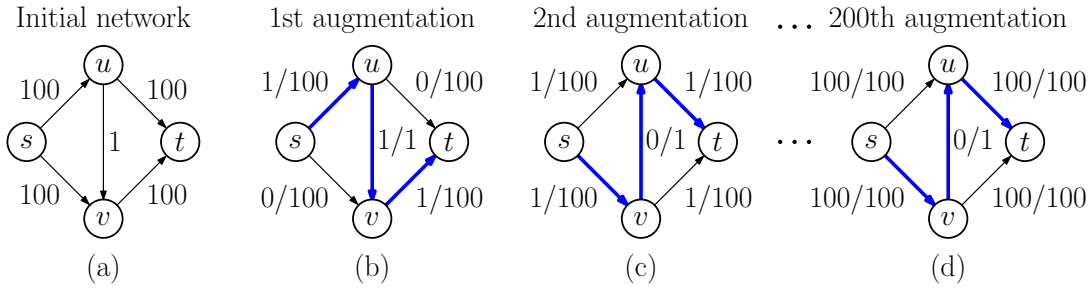


Fig. 69: Bad example for Ford-Fulkerson.

Suppose that we (foolishly) elect to augment using a path from s to t that uses the vertical edge in the middle, alternately increasing its flow to 1 and reducing it to 0. It would take 200 augmentation steps to converge. We could replace 100 with whatever huge value we want and make the running time arbitrarily high.

If we let $|f|$ denote the final maximum flow value, we can see from the example that 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 $\Omega(m|f|)$. In general, if C is any upper bound on the maximum flow, the running time of the Ford-Fulkerson algorithm is $O(mC)$.

Scaling Algorithm: In the above example (Fig. 69), we made the apparently foolish decision to augment on a path of very *low capacity*. Intuitively, it would be smarter to select the *maximum capacity* augmenting path. We can compute the maximum capacity s-t path by a variant of Dijkstra's algorithm in $O(m \log n)$ time. However, we don't need to exact maximum. It suffices to pick any path whose capacity is "close" to maximum. Such a path can be computed in just $O(m)$ time. This is the essence of the *scaling algorithm*. This algorithm (and generally the idea of solving optimization problems through scaling) was introduced by Hal Gabow in the mid 1980's.

The idea of scaling is to eliminate (actually, just disregard) all the edges of small capacity, focusing instead on the high-capacity edges. Once we have pushed as much flow as we can through these "heavy hitters," we can consider edges of smaller and smaller capacities. Eventually, we will get down to the smallest capacity edges.

Let's assume that all of the capacities are integers. (If not, scale them up uniformly so they are.) To get this process started, we'll need to define the highest capacity edges. To do this, let's start by computing a *crude upper bound* on the maximum flow value. We can do this by taking the capacity on any valid cut.[10] For simplicity, lets take the cut that has s on one side and all the other vertices on

[10]Recall that a *cut* is a partition (X, Y) of the vertices such that $s \in X$ and $t \in Y$, and the *capacity* of a cut is the sum of capacities of edges going from X to Y. In the previous lecture we proved that the value of any flow cannot exceed the capacity of any cut.

the other. Let C denote the capacity of this cut, that is,

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

By the Min-Cut/Max-Flow Theorem, we know that any valid flow cannot exceed C.

The algorithm makes use of a *scaling* (or *resolution*) *parameter*, denoted Δ. The value of Δ will be a power of 2. Initially, its value will be close to C, and we will progressively decrease Δ until it equals 1. Intuitively, we will ignore all edges whose capacities are smaller than Δ. This implies that the early phases of the algorithm will focus on pushing flow through the fattest pipes and the final stages will work with the skinniest pipes.

Initially, let Δ be the largest power of 2, such that $\Delta \leq C$. (Formally, $\Delta \leftarrow 2^{\lfloor \lg C \rfloor}$.) Given any flow f, define $G_f(\Delta)$ to be the residual network consisting only of edges of *residual capacity at least* Δ. (There might be none, but this is okay.) An example is shown in Fig. 70.

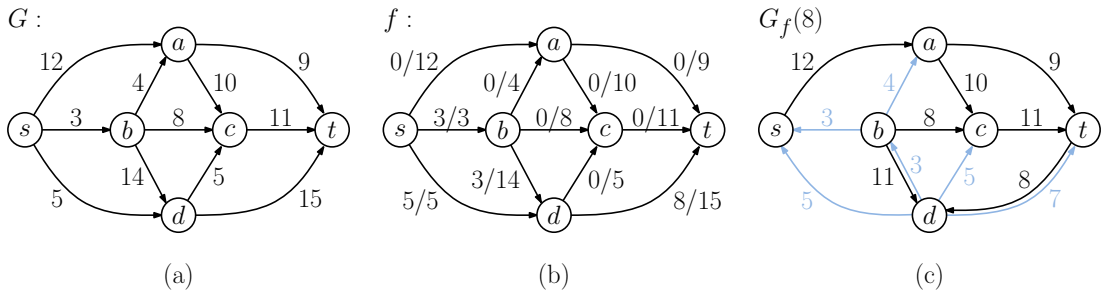


Fig. 70: (a) A network G, (b) a flow f, and (c) the restricted residual $G_f(8)$, which is G_f keeping only edges of weight ≥ 8. The light blue edges are the edges of G_f that have been removed.

Recall that an *augmenting path* in a residual network is an s-t path in the residual. Given such a path, we find the edge of minimum (residual) capacity along the path, and push this much flow along the path. Intuitively, whenever we find an augmenting path in $G_f(\Delta)$, we are guaranteed to push at least Δ units of flow, which means that we are guaranteed to increase our flow by at least Δ. We continue computing such augmenting paths (and updating the residual) until $G_f(\Delta)$ has no augmenting path. We then reduce Δ in half, and repeat the process. When $\Delta = 1$, we are effectively running the standard Ford-Fulkerson algorithm, which implies that we'll obtain the correct final flow. The algorithm is presented in the following code block below. An example is illustrated in Fig. 71.

Analysis of the Scaling Algorithm: We refer to the Kleinberg and Tardos book for a complete analysis of the scaling algorithm. Intuitively, the algorithm is efficient because each augmentation increases the flow by an amount of at least Δ. The minimum cut can have at most m edges. So, after $O(m)$ such augmentations, we will have effectively increased the flow along every edge of the minimum cut so much that its capacity in the residual network falls below Δ. When all the edges of the minimum cut disappear from $G_f(\Delta)$, it is not possible to augment further, and the algorithm goes on to the next smaller value of Δ.

Each augmentation involves computing *any* s-t path, which can be done in $O(n+m) = O(m)$ time by BFS or DFS. It follows that after $O(m^2)$ time, we will exhaust augmentations in $G_f(\Delta)$, and will proceed to the next smaller value of Δ. After $O(\log C)$ halvings of Δ, we will have $\Delta < 1$, and the algorithm will terminate. Thus, the overall running time is $O(m^2 \log C)$. (It can be shown that a more efficient implementation runs in time $O(nm \log C)$.)

Pseudo-Polynomial and Strong Polynomial Time: Earlier, we complained that the Ford-Fulkerson algorithm is not really efficient, since its running time depends on the maximum flow value. The scaling

```
scaling-flow(G) {                                // network flow via scaling
    f = 0                                        // initial flow is zero
    C = sum of capacities out of s               // capacity of any cut
    Del = 2^(floor(log_2(C))                     // largest power of 2 below C
    while (Del >= 1) {                           // repeat until resolution = 1
        R = residual network G_f                 // residual network
        remove from R edges of capacity < Del    // remove small capacity edges
        P = any s-t path in R                    // any augmenting path
        if (P exists) {
            c = min capacity edge in P           // augmentation amount
            f += add c to the flow on every edge of P
        } else {                                 // no more augmentations?
            Del = Del/2                          // reduce Delta by half
        }
    }
    return f                                     // return the final flow
}
```

algorithm also depends on the maximum flow value (albeit logarithmically rather than linearly). So, it what sense can we claim it is "efficient"?

Observe first that if the capacities are all small integers, then we can ignore the $\log C$ component, and so the running time is just $O(m^2)$ which is polynomial in the input size. (Efficient algorithms generally run in polynomial time, as opposed to exponential time.)

The algorithm is really *inefficient* when the capacities are *huge* numbers. Observe that if C is extremely large, then we require many bits to represent these numbers. Indeed, it takes $\Theta(\log C)$ bits to represent a number of magnitude C. Thus, if we think of the input size from the perspective of *total number of bits* needed to represent the input network, it can be shown that the scaling algorithm runs in time that is polynomial in this number of bits. (We will leave the details as an exercise.)

An algorithm whose running time is polynomial in the number of *bits* of input is called a *pseudo-polynomial time algorithm*. In contrast, an algorithm whose running time does *not* depend on the number of bits (assuming that we can process any numeric operation in $O(1)$ time), is said to run in *strongly polynomial time*. The running time should depend only on n and m, not C. We will next show that there exist strongly polynomial time algorithms for network flow.

Edmonds-Karp/Dinic's Algorithm: As mentioned above, neither of the algorithms we have seen runs in strongly polynomial time, that is, polynomial in n and m, irrespective of the magnitudes of the capacity. Dinitz and (independently) Edmonds and Karp developed such an algorithm in the 1970's. (Note that Dinitz's algorithm goes by the name Dinic's algorithm, due to a mistranscription of his name.)

This algorithm uses Ford-Fulkerson as its basis, but with the minor change that the s-t path that is chosen in the residual network has the *smallest number of edges*. In particular, this just means that we run BFS in the residual network from s to t to compute the augmenting path. It can be shown that the total number of augmenting steps using this method is $O(nm)$.[11] Since each augmentation takes $O(m)$ time to run BFS, the overall running time is $O(nm^2)$.

[11] This is not trivial to prove. A proof can be found in the algorithms book by Cormen, Leiserson, Rivest, and Stein. Intuitively, it can be shown that after at most m augmentations, each vertex's distance from s in the residual network increases by at least one edge, never to decrease again. Since a vertex's distance from s cannot exceed n, it follows that there are at most $O(nm)$ augmentations.

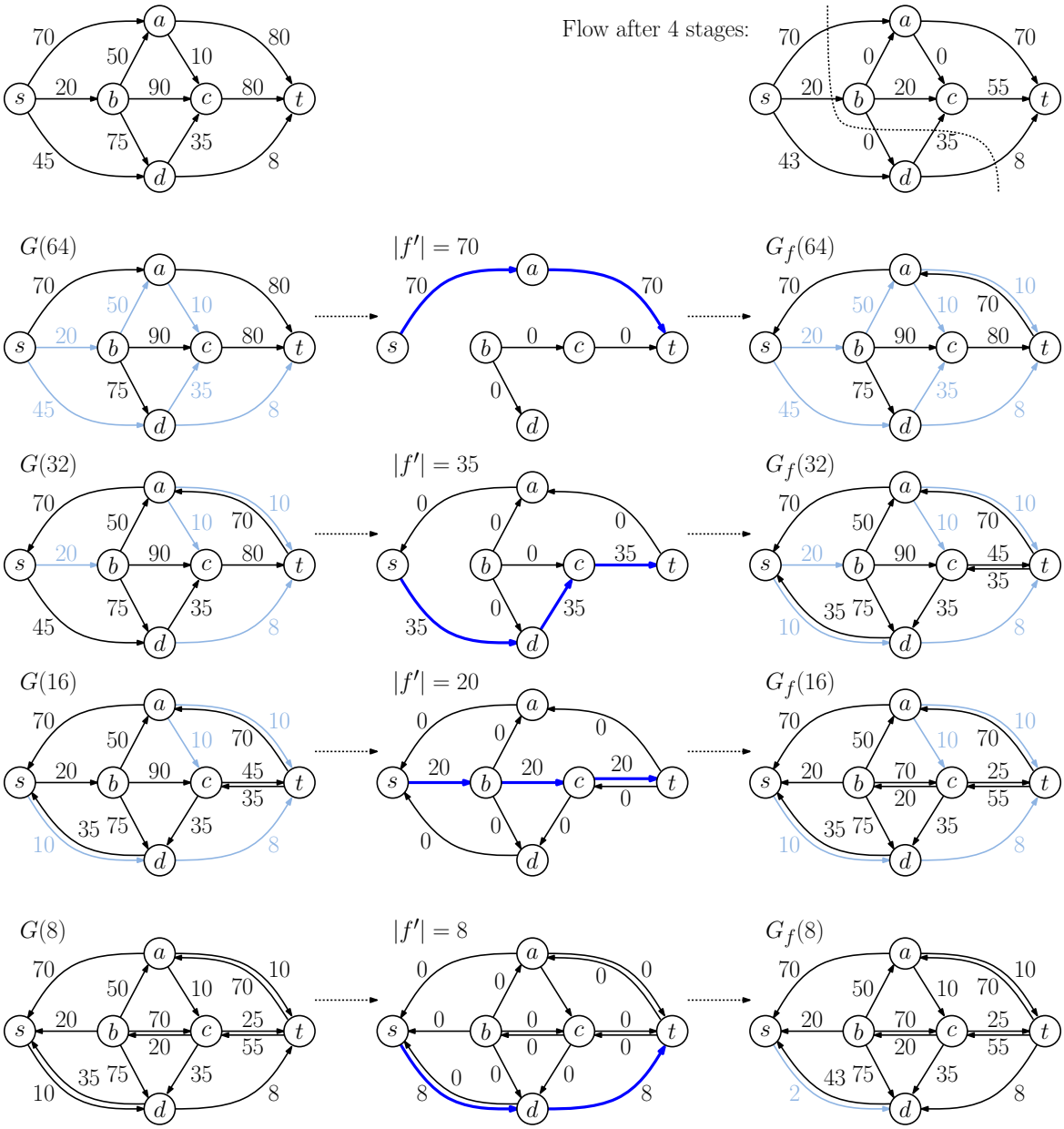


Fig. 71: The first four stages of the scaling algorithm for network flow. Note that at each stage, the value of the augmenting flow f' is at least Δ. (The algorithm will run for $G(4)$, $G(2)$, and $G(1)$, but no changes to the flow will occur.)

Even Faster Algorithms: The max-flow problem is widely studied, and there are many different algorithms. No one knows that what the lowest possible running time is for network flow, but a running time of $O(nm)$ has stood as an important milestone. See Table 1 for a summary of important results. The most recent (and most efficient) algorithms tend to be rather complicated.

Table 1: Running times of various network-flow algorithms, where $n = |V|$, $m = |E|$, C is any upper bound on the maximum flow.

Algorithm	**Year**	**Time**	**Notes**
Ford-Fulkerson (FF)	1956	$O(mC)$	Repeated augmentation
Gabow	1985	$O(nm \log C)$	FF + scaling
Edmonds-Karp	1972	$O(nm^2)$	FF + augment shortest path
Dinic (Dinitz)	1970	$O(n^2m)$	Blocking flows
Dinic + Tarjan	1983	$O(nm \log n)$	Dinic + better data structures
Goldberg and Tarjan	1986	$O(nm \log(n^2/m))$	Preflow push
King, Rao, Tarjan (KRT)	1994	$O(nm \frac{\log n}{\log(m/n \log n)})$	$O(nm)$ for sparse networks
Orlin + KRT	2013	$O(nm)$	

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 Kleinberg and Tardos. There are over 40 problems, most of which involve reductions to network flow.)

Maximum Matching in Bipartite Graphs: There are many applications where it is desirable to compute a pairing between two sets of objects. We present such a problem, called *bipartite matching* in the form of a "dating game," but the algorithm can be applied whenever it is desired to find pairing between objects of different classes subject to some compatibility criterion, for example, pairing medical students seeking residencies with hospitals.

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. The pairing 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 can be paired (see Fig. 72(a)).

Given any undirected 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. Our objective is to compute a matching in G that has the highest number of edges. Such a matching is called a *maximum matching*. Matchings can be computed in any undirected graph, but they are easiest to compute in bipartite graphs. This problem is called the *maximum bipartite matching problem* (see Fig. 72(b)).

We will now present a reduction from maximum bipartite matching to network flow. In particular, we will show that, given any bipartite graph G 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 easily 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. 73(b)).

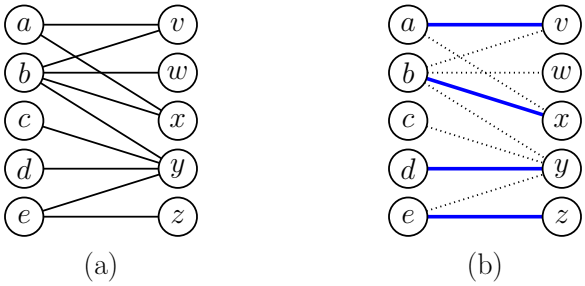


Fig. 72: (a) A bipartite graph G and (b) a maximum matching.

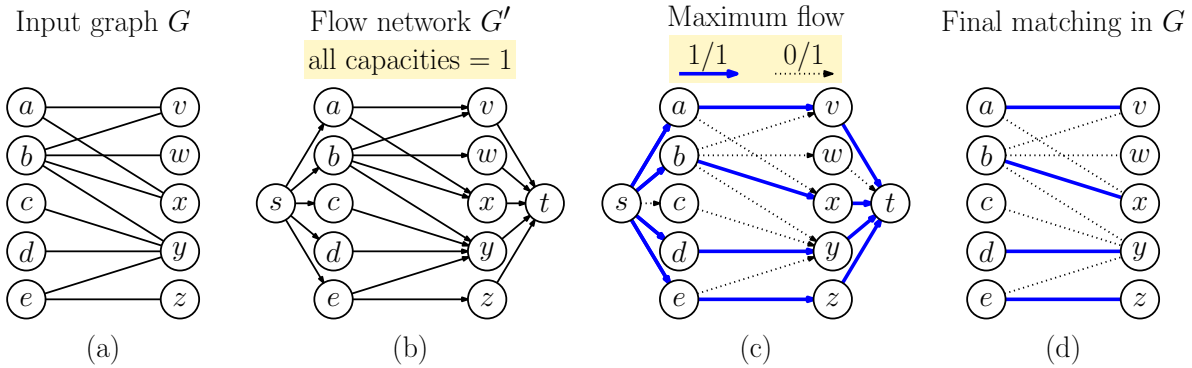


Fig. 73: Reducing bipartite matching to network flow.

Compute the maximum flow in G' (see Fig. 73(c)). The following lemma show that maximizing the flow in G' is equivalent to finding a maximum matching in G.

Lemma: Given a bipartite graph G, G has a matching of size k if and only if G' (constructed above) has a flow of value k.

Proof: ($\Rightarrow$) Let M denote any matching in G. We construct a flow in G' as follows. For each edge $(x, y) \in M$, set the flow along edges (s, x), (x, y), and (y, t) to 1. All the edges remaining edges of G are assigned a flow of 0. We assert that f is a valid flow for G'. By our construction, each vertex x receives one unit of flow coming in from s, sends one unit to y, and y sends one unit to t. Therefore, we have flow conservation. Second, since M is a matching, each vertex of X or Y is incident to a single edge of M, which implies that it carries at most one unit of flow, which implies that the capacity constraints are all satisfied. Therefore, f is a valid flow in G'. By our construction, $|f| = |M|$.

($\Leftarrow$) Suppose that G' has a flow f. We may assume that this is an integer flow. Since all edges have capacity 1, it follows that the flow value on each edge is either 0 or 1.

Let M denote the edges of $X \times Y$ that are carrying unit flow in f. Observe that for every vertex of X, it has exactly one incoming edge (from s) of capacity 1, and hence it can be incident to at most one edge of M. Symmetrically, every vertex of Y has exactly one outgoing edge (to t) of capacity 1, and hence it also can be incident to at most one edge of M. Therefore, M is a matching in the original graph G. (An example is shown in Fig. 73(d)). Since each edge carries one unit of flow, the total value of the flow is the number of edges of M, that is, $|f| = |M|$.

Because the capacities are so low, we do not need to use a fancy implementation. Recall that Ford-Fulkerson runs in time $O(m \cdot C)$, where C is an upper bound on maximum flow. In our case C is at

most n (the size of the largest possible matching). Therefore, the running time of Ford-Fulkerson on this instance is $O(nm)$.

By the way, there are other algorithms designed specifically for maximum bipartite matching. The best is due to Hopcroft and Karp. It runs in $O(\sqrt{n} \cdot m)$ time. There are also efficient algorithms for computing maximum matchings in general undirected graphs. The best known is the *blossom algorithm* of Jack Edmonds, which dates way back to 1961.

Summary: We have shown that the Ford-Fulkerson algorithm can take super-polynomial time to run. As alternatives, we introduced Gabow's scaling algorithm, which runs in time $O(nm \log C)$ and briefly discussed the Edmonds-Karp algorithm, which runs in $O(nm^2)$ time. Finally, we presented a reduction which shows that maximum matchings in bipartite graphs can be reduced to the network flow problem.

Lecture 14: Network Flow: Circulations and Applications

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.[12] 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. 74(a), we show a network in which each node is each labeled with its demand and each edge with its capacity.) 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.

Definition: Given a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ is associated with a positive capacity $c(u, v)$ and each vertex $v \in V$ is associated with a supply/demand value $d(v)$, a *circulation* in G is a function f that assigns a nonnegative real number to each edge that satisfies the following conditions:

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

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

[12] I would not advise applying this in real life. I doubt that the IRS would appreciate it if you paid your \$100 tax bill by demanding that they send you −\$100 dollars.

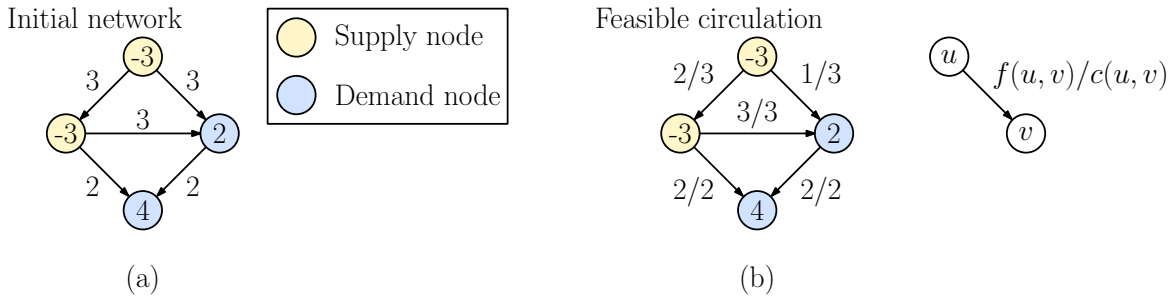


Fig. 74: (a) A circulation network and (b) a feasible circulation.

For example, in Fig. 74(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, $\sum_{v \in S} d(v)$.)

Observe that the max-flow problem is closely related. Suppose that we are given a (standard) s-t network, and we want to know whether it has a flow of value D. This is equivalent to a circulation network where we set $d(s) = -D$, $d(t) = D$ and $d(u) = 0$, for all other vertices.

Reducing Circulation to Max-Flow: Rather than devise an algorithm for the circulation problem, we will show that we can reduce any instance G of the circulation problem into an equivalent network flow problem. The input to our reduction is a network $G = (V, E)$. For each vertex v, let $d(v)$ denote the demand value and for each edge (u, v), let $c(u, v)$ denote its capacity. Intuitively, the idea is push flow into all the supply vertices and extract it from all the demand vertices (see Fig. 75).

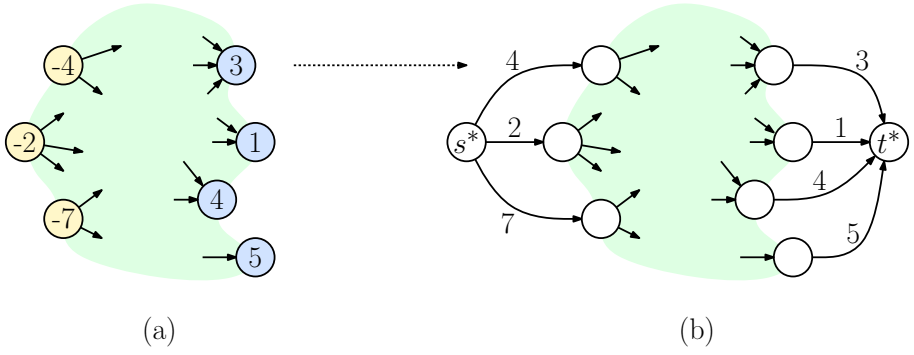


Fig. 75: Reducing the circulation problem (a) to a standard s-t flow network (b).

Here is the formal reduction. Oserve first that we may assume that sum of supplies equals total demand (since if not, we can simply answer "no" immediately.) Otherwise:

- 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-source* 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 $d(v)$

Applying this to the circulation network shown in Fig. 74, yields the s-t network is shown in Fig. 76(b). We then invoke any max-flow algorithm on G'. Recalling that D denotes the total demand, we check whether the value of the maximum flow equals D. If so, we answer "yes," G has a feasible circulation, and otherwise we answer "no," G does not have a feasible circulation. (For example, in Fig. 76(c), there exists a flow of value $D = 6$, implying that the original network has a circulation.)

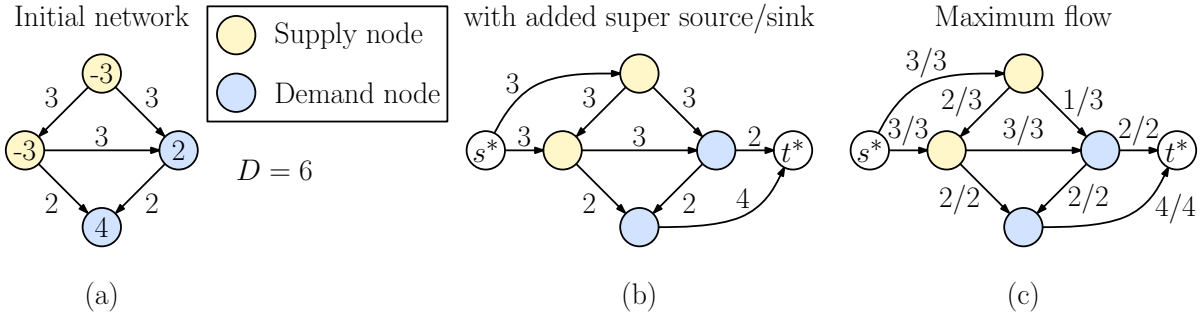


Fig. 76: Reducing the circulation problem to network flow.

We prove correctness below, but intuitively, the newly created edges from s^* will be responsible for providing the necessary supply for the supply vertices of S and newly created edges into t^* are responsible for 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).

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

Proof: ($\Rightarrow$) 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 satisfies flow conservation by the supply/demand constraints.

($\Leftarrow$) 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 the flow conservation of f', all the supply nodes and all the demand nodes have achieve their desired supply/demand constraints. All the other nodes satisfy their supply/demand constraints because by the flow conservation of f' the incoming flow equals the outgoing flow. Therefore, the resulting flow is a circulation for G.

It is not hard to see to that the reduction can be performed in $O(n + m)$ time by a simply analysis of the network's structure. Thus, the overall running time is dominated by the time to compute the network flow (which is $O(nm)$ according to the current state of the art).

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. 77(a), and a valid circulation is shown in Fig. 77(b).

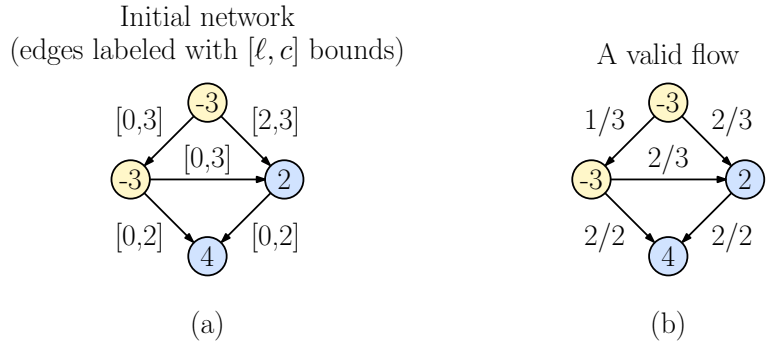


Fig. 77: (a) A network with both upper and lower flow bounds and (b) a valid circulation.

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 (probably invalid) "pseudo-circulation" f_0 that exactly satisfies all the lower flow bounds. In particular, we let $f_0(u, v) = \ell(u, v)$. This circulation may be (in fact will likely be) invalid because f_0 need not satisfy the demand constraints. (Recall these also guarantee flow conservation.) Here's the trick. We will now modify the supply/demand values to compensate for this shortcoming. Why will this work? 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.

Let's carry out this plan. Recall our "pseudo-flow" f_0. For each $v \in V$, let $f_0^{\text{in}}(v)$ and $f_0^{\text{out}}(v)$ denote the flow in and flow out of vertex v. Let $\chi(v)$ denote the *excess flow* coming into v in f_0, that is

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

Note that this may be negative, which means that we have a flow deficit. We will adjust the supply and demand values so that computing any valid circulation f_1 for the adjusted values and combining this with f_0, we will obtain a flow that satisfies all the requirements.

How do we adjust the supply/demand values? For each v, we want to cancel out the excess $\chi(v)$ coming in and generate a net flow of $d(v)$ units coming in. That is, we want f_1 to satisfy

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

To achieve this we can change the demand at vertex v to be $d(v) - \chi(v)$. We'll call this $d'(v)$.

But what about the edge capacities? Recall that we had an upper capacity of $c(u, v)$, but we are essentially "forcing" $\ell(u, v)$ units of flow. So, the remaining capacity is $c(u, v) - \ell(u, v)$. We'll call this $c'(u, v)$.

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 $c(u, v)$), we do the following (see Fig. 78):

1. Create an initial pseudo-circulation f_0 by setting $f(u, v) = \ell(u, v)$.
2. Create a new network $G' = (V', E')$ that has all the same vertices and edges as G (that is, $V' \leftarrow V$ and $E' \leftarrow E$):
   (a) For each $(u, v) \in E'$, set its adjusted capacity to $c'(u, v) \leftarrow c(u, v) - \ell(u, v)$.
   (b) For each $v \in V'$, set its adjusted demand to $d'(v) + \leftarrow d(v) - \chi(v)$, where, $\chi(v) = f_0^{\text{in}}(v) - f_0^{\text{out}}(v)$.
3. Compute a (standard) circulation f_1 in G'. If it does not exist, return "no circulation!" Otherwise, return the final flow $f = f_0 + f_1$ (see Fig. 79(c)).

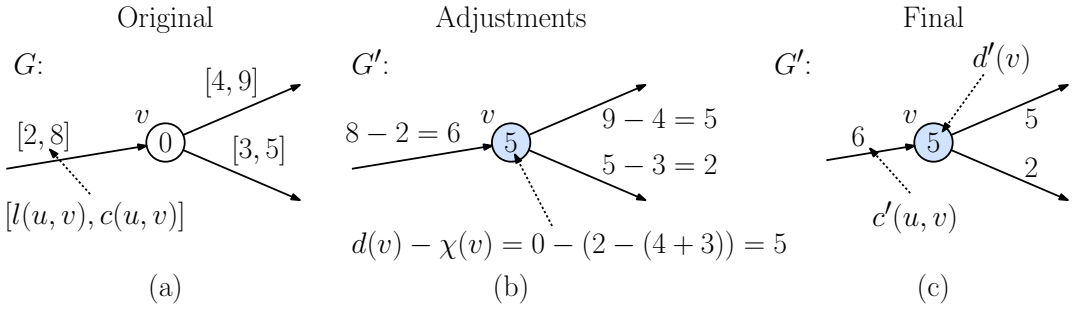


Fig. 78: Eliminating lower bound constraints. The net pseudo-flow into v is $\chi(v) = 3 - (4 + 3) = -4$.

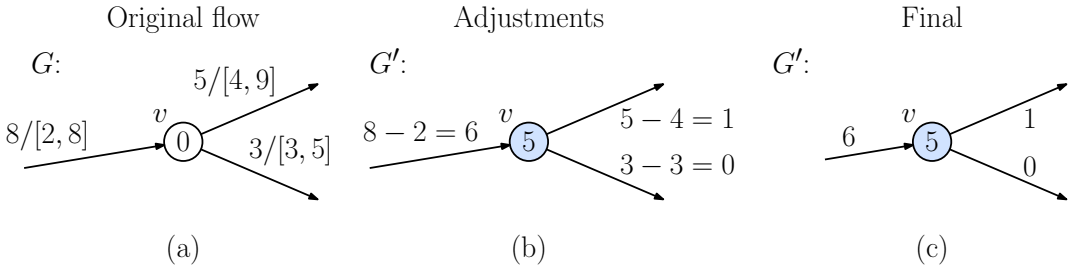


Fig. 79: Adjusting flows between original network and transformed network. Observe that v satisfies the demand requirements, since $6 - (1 + 0) = 5 = d'(v)$.

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 Kleinberg and Tardos 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. 80).

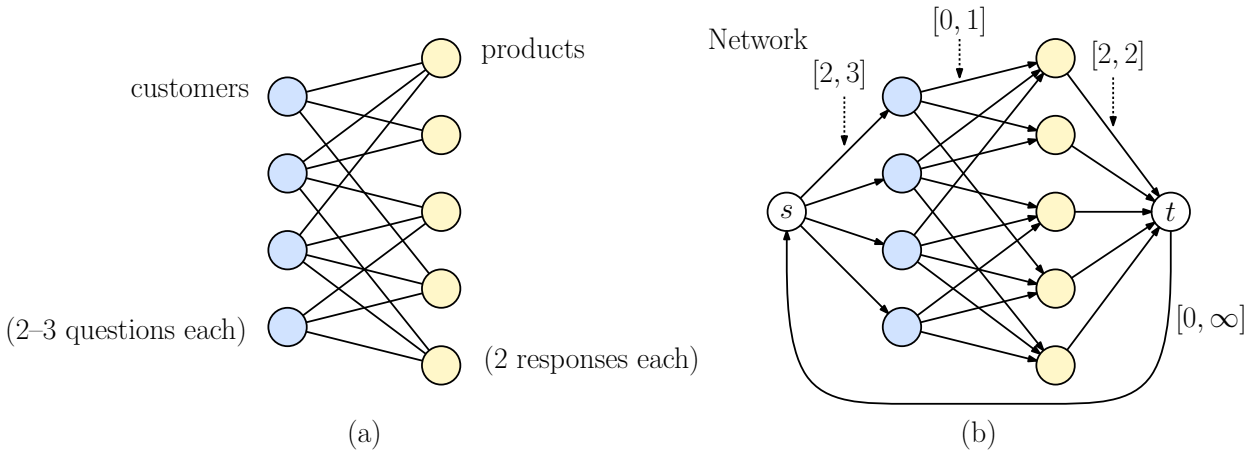


Fig. 80: Reducing the survey design problem to circulation with lower and upper flow bounds. (Assuming $[c_i^-, c_i^+] = [2, 3]$ for all customers and $[p_j^-, p_j^+] = [2, 2]$ for all products.)

- 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.

The correctness of the reduction is established in the following lemma.

Lemma: There exists a valid circulation in G if and only there is a valid survey design.

Proof: ($\Rightarrow$) Suppose that G has a valid (integer-valued) circulation. We create a survey as follows. For each customer-product edge (i, j) that carries one unit of flow, customer i is surveyed about product j. We will show that this is a valid survey. By definition of the edges, customers are surveyed only about products they purchased. From our capacity constraints and the fact that demands are all zero, it follows that the total flow into each customer node is in the interval $[c_i^-, c_i^+]$, implying that this customer is asked about the proper range of products. Similarly, the total flow out of each product node is in the interval $[p_j^-, p_j^+]$, implying that this product is involved in the proper number of surveys. Therefore, this is a valid survey, as desired (see Fig. 81).

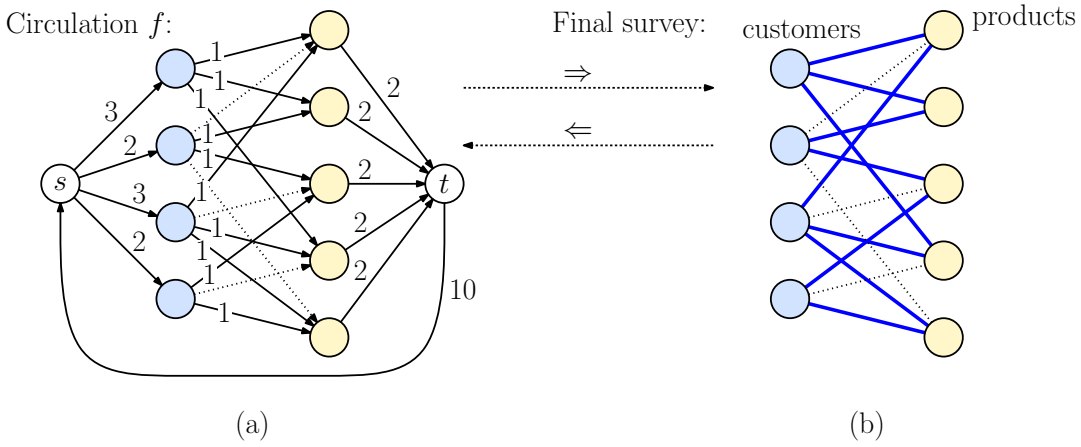


Fig. 81: Correctness of the survey-design reduction to circulation.

($\Leftarrow$) Suppose that there is a valid survey design. We construct a flow in G as follows. For each customer-product pair (i, j) involved in the survey, we create a flow of one unit on edge (i, j), and otherwise we set this flow to 0. Since we only survey pairs where a purchase took place, this edge exists in G. Next, for each customer i, we set the flow along the edge (s, i) to the total number of surveys sent to customer i. For each product j, we set the flow along the edge (j, t) to the total number of surveys involving product j. Finally, to keep everything balanced, we set the flow on edge (t, s) to the total number of surveys.

We will show that this is a valid circulation in G. Because the survey is valid, the number of surveys sent to customer i is in interval $[c_i^-, c_i^+]$, satisfying this edge's capacity constraints. The flow on each (i, j) edge is either 0 or 1, satisfying this edge's capacity constraints. Also, by the validity of the survey, the number of surveys involving product j is in the interval $[p_j^-, p_j^+]$, satisfying this edge's capacity constraints. Finally, observe that the flow into each vertex equals the flow out of this vertex, implying that all the vertex demands (zero) are satisfied. Therefore f is a valid circulation in G, as desired (see Fig. 81).

Summary: We introduced the notion of a circulation in a network. Unlike network flows, each individual vertex has requirements for the net flow through this vertex. These can be used for various applications. We showed that circulations can be solved by any standard network flow algorithm. We also showed that a generalization where edges can have both lower and upper capacities can be reduced to the standard circulation problem (which has only upper capacity bounds). Circulations have many applications, and we one example from survey design.

Lecture 15: NP-Completeness: Basic Definitions

Efficiency and Polynomial Time: Up to this point of the semester we have been building up your "toolkit" for solving algorithmic problems efficiently. Hopefully when presented with a computational problem, you now have a clearer idea the sorts of techniques that could be used to solve the problem efficiently (such as divide-and-conquer, DFS, greedy, dynamic programming, network flow).

What do we mean when we say "efficient"? 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. A general point of agreement is that algorithmic solutions that rely on *brute-force search* run in *exponential time* are not efficient. The alternative is a structured approach. Such solutions run in *polynomial time*, that is, the worst-case running time is $O(n^c)$, where c is a nonnegative constant. (Note that running times like $O(n \log n)$ are polynomial time, since $n \log n = O(n^2)$.) By *exponential time* we mean any function that is at least $\Omega(c^n)$ for a constant $c > 1$. (This includes $n!$, since $n! \approx (n/e)^n > 2^n = \Omega(2^n)$.)

Henceforth, we will use the terms "efficient" and "easy" to mean solvable by an algorithm whose asymptotic worst-case running time is *polynomial in the input size* (even if that polynomial is n^{1000}). Note the emphasis on the term "worst-case". There are problems for which simple heuristics can efficiently solve almost all "typical" input instances but no efficient algorithmic solution is known that works for all inputs. (A good example of this is the Graph Isomorphism problem.)

While the distinction between worst-case polynomial time and worst-case exponential time is quite crude, it has a number of advantages. For example, the composition of any two polynomials is a polynomial. (That is, if $f(n)$ and $g(n)$ are both polynomials, then so is $f(g(n))$. For example, $(n^a)^b = n^{ab}$.) This means that, if a program makes a polynomial number of calls to a function that runs in polynomial time, then the overall running time is a polynomial.

The Emergence of Hard Problems: Near the end of the 1960'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. Often these hard problems involved slight generalizations to problems that are solvable in polynomial time. A list of some of these problems is shown in Table 2.

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 arguably the biggest open problems in computer science:

$$\text{P} = \text{NP?}$$

While we will not be able to provide an answer to this question, we will investigate this concept 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?

Table 2: Computationally hard problems and related (easy) counterparts.

Hard Problems (NP-complete)	**Easy Problems (in P)**
3SAT	2SAT
Traveling Salesman Problem (TSP)	Minimum Spanning Tree (MST)
Longest (Simple) Path	Shortest Path
Hypergraph Matching	Graph Matching
Knapsack	Unary Knapsack
Independent Set in Graphs	Independent Set in Trees
Integer Linear Programming	Linear Programming (weak poly time)
Hamiltonian Cycle	Eulerian Cycle
Balanced Cut	Minimum Cut

Decision Problem: 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 z, does G have a spanning tree whose weight is at most z?" Observe that a solution to the decision problem can often be converted into a solution to the corresponding optimization problem, for example, by combining a decision algorithm with a binary search over possible values z.

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.

Language Recognition: For historical reasons, decision problems are often expressed as *language recognition* problems. A *language* is defined to be a set (finite or infinite) of strings. For example, the set of strings that are palindromes (reading the same backwards and forwards) is an example of a language. We'll see that we can use the concept of languages to describe any computational problem.

To express a computational problem as a language-recognition problem, we first should be able to express its input as a string. Given any mathematical object I, define *serialize*(I) to be a function that maps I to a string. (For example, serializing a graph would involve outputting a string that encodes all its vertices, all its edges, and any additional information such as edge weights.)

Using this, we could define a language MST encoding the Minimum Spanning Tree problem as a language (that is, a collection of strings):

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

Since it will be a hassle to continuously refer to the serialization function, we will just write this more succinctly as

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

Now, consider any string x that is a valid serialization (that is, encoding) of a graph G and integer z. The language recognition question "Is x a member of the language MST?" could amount to carrying out the following procedure:

- Decode x as a graph G and integer z. (If the encoding is invalid, report "no" right away and terminate.)
- Run any MST algorithm (e.g., Kruskal) on G to determine the weight w of MST(G).
- If $w \leq z$, report "yes" and otherwise report "no".

In the terminology of language recognition, reporting "yes" is called *accepting* the input x and reporting "no" is called *rejecting* the input.

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, it follows that 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).

A Harder Example: 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*. (It was named after the great 19th century mathematician and physicist, William Rowan Hamilton, famed for the discovery of quaternions.)

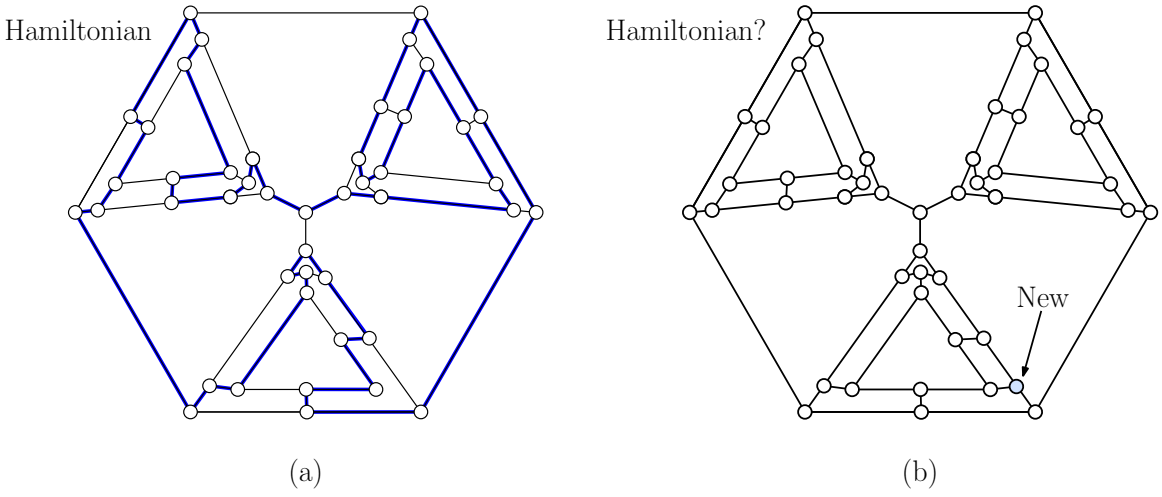


Fig. 82: The Hamiltonian cycle (HC) problem.

In Fig. 82(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. 82(b), which has one additional vertex and one additional edge. Either find a Hamiltonian cycle in this graph or prove than none exists. To make this even harder, imagine a million-vertex graph with many slight variations of this pattern.

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

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

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

To do so, it is important to first introduce the notion of a verification algorithm. Many language recognition problems that may be *hard to solve*, but they have the property that they are *easy to verify* that a string is in the language. Recall the Hamiltonian cycle problem defined above. As we saw, there is no obviously efficient way to find a Hamiltonian cycle in a graph. However, suppose that a graph *does* 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 any 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, a *verification algorithm* is any algorithm that is given an input string x. If $x \in L$, then there exists a string y, called the *certificate*, such that it is possible to prove (with the aid of y) that $x \in L$. If $x \notin L$, then we do not care what the algorithm does. If there exists a verification algorithm that runs in polynomial time, we say that L can be *verified in polynomial time* (or *polynomially verifiable*).

What do we mean by "prove that $x \in L$ with the aid of y"? This is where we will be a bit hand-wavy. In the case of Hamiltonian Cycle, HC is the language of (encoded) graphs that have Hamiltonian cycles. If $x \in$ HC, that is, if this graph has a Hamiltonian cycle, then we can take y to be the sequence of vertices that form the cycle. The "proof" consists of verifying that y indeed defines a cycle in the graph that visits every vertex exactly once. Clearly, if x is represented in any reasonable manner (e.g., an adjacency matrix), we can do this in time that is linear in the number of vertices.

Are All Languages Polynomially Verifable? You might wonder whether given the immense power of being able to pick a certificate out of thin air, isn't every language polynomially verifiable. Let's see some examples of problems that do not seem to be polynomially verifiable.

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

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. For $\overline{\text{HC}}$, it is unclear what certificate could be provided to convince us that the graph has no Hamiltonian cycle.

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.

Reductions: One way to attack the question of whether $\mathrm{P} \neq \mathrm{NP}$ would be show that there exists a language L such that $L \in \mathrm{NP}$ but $L \notin \mathrm{P}$. A good candidate for such a problem would be the "hardest" problem in NP. To define what it means for one problem to be hard relative to another, we introduce the notion of reduction, that is, using an algorithm for one problem to solve another.

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? Effectively, we want to show that

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

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. In other words, we could prove the contrapositive,

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

To make this more concrete, suppose that we had a subroutine[13] that can solve any instance of problem U in polynomial time. Given an input x for the problem H, we could faithfully translate it into an input x' for U. By "faithful" we mean that $x \in H$ if and only if $x' \in U$ (see Fig. 83). Then we run our U subroutine on x' and output whatever it outputs.

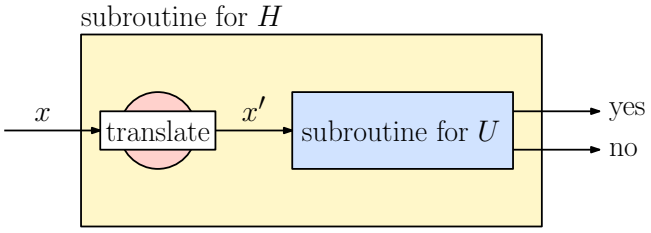


Fig. 83: Reducing H to U.

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 call this a *polynomial reduction*[14] of problem H to problem U, which is denoted $H \leq_P U$.

[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.

[14] Richard Karp used this style of reduction in his influential paper on NP-completeness, and for this reason, it is called a *Karp reduction*. More generally, we might consider calling the subroutine multiple times like a black-box. This is called a *Cook reduction*, after Stephen Cook. While Cook reductions are theoretically more powerful, Karp reductions are simpler and work for virtually all NP-completeness proofs.

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

3-coloring (3Col): Given a graph G, can each of its vertices be labeled with one of three different "colors", such that no two adjacent vertices have the same label (see Fig. 84(a) and (b)).

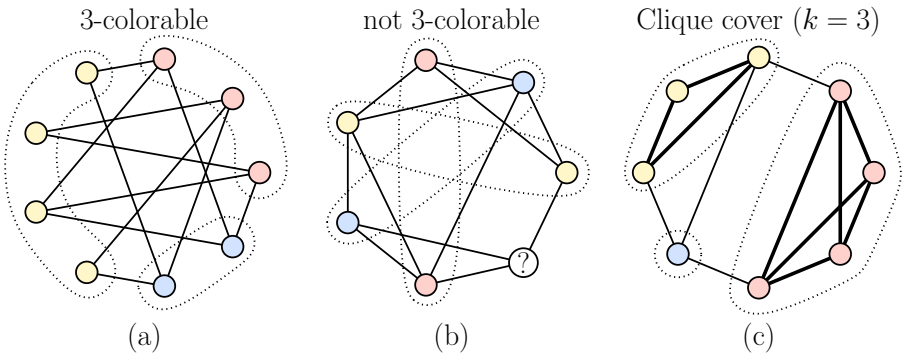


Fig. 84: 3-coloring and Clique Cover.

Coloring arises in various partitioning problems. (E.g., You are arranging your relatives to sit at three big tables during a wedding reception. You have pairs that don't get along, each represented by an edge in your graph. Can you assign them to three tables avoiding warring pairs at the same table?) 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. 84(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 $\text{CCov}(G, k)$. 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. 85).

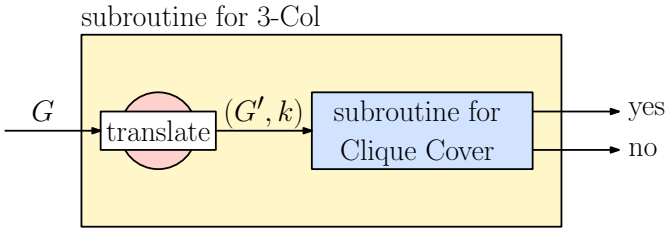


Fig. 85: Reducing 3Col to CCov.

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. 86).

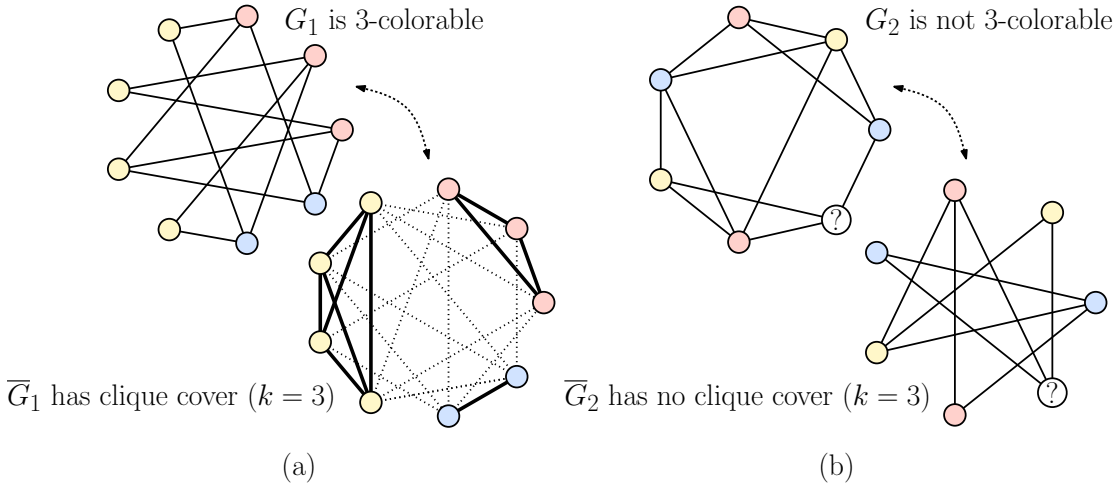


Fig. 86: Clique covers in the complement.

The following formally establishes the correctness of this reduction by showing that we have *faithfully* translated an instance of 3Col to an equivalent instance of CCov.

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.

It is useful to observe that the reduction was from 3Col to CCov, which means that we are at liberty to use any value of k we like, and $k = 3$ was convenient. You might wonder why we didn't need to consider other values of k. The reason is that we didn't need to. Of course, if we were trying to do the reduction in the opposite direction from CCov to 3Col, we would need to worry about this.

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

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

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

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

You shouldn't read too much into the notation or make incorrect inferences. For example, this does not imply that L_1 is necessarily easier in the sense that its running time is smaller. It may very well be that $L_1 \leq_P L_2$ and L_1 takes $O(n^{10})$ time to compute while L_2 takes only $O(n)$ time. It could be that L_1 is solvable in polynomial time, but L_2 takes exponential time to compute. It could be that both take exponential time to compute. What we can infer, however, is that, if L_2 is solvable in polynomial time, then L_1 *cannot* take exponential time.

The way in which inequality is applied 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.

Summarizing the above, and recalling that the composition of poly-time functions is poly-time, we have the following.

Lemma: Given languages L_1, L_2, and L_3,

(i) If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$.
(ii) If $L_1 \leq_P L_2$ and $L_1 \notin P$, then $L_2 \notin P$.
(iii) If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ then $L_1 \leq_P L_3$. (Transitivity of "$\leq_P$")

NP-completeness: We now have the necessary tools to define NP-completeness. This is subset of NP that are "hardest" in the sense that if it is known that if any one is solvable in polynomial time, then they all are. This is made mathematically rigorous 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).

Unfortunately, showing that a problem is NP-hard seems nearly impossible, since it involves a property of *all* languages in NP, an infinite set. In the next lecture, we will present Cook's Theorem, which demonstrates that there does indeed exist an NP-complete problem.

Lecture 16: NP-Completeness: 3SAT and Independent Set

Recap: Recall the following definitions, which were given in the previous lecture.

P: The set of languages (decisions problems) solvable in (worst-case, deterministic) polynomial time.

NP: The set of languages that can be *verified* in polynomial time (with the help of a certificate).

Polynomial reduction: $L_1 \leq_P L_2$ means that there is a polynomial time computable function f such that $x \in L_1$ if and only if $f(x) \in L_2$. A more intuitive way to think about this is that if we had a subroutine to solve L_2 in polynomial time, then we could use it to solve L_1 in polynomial time. Polynomial reductions are *transitive*, that is, $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ implies $L_1 \leq_P L_3$.

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

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

It follows from these definitions that:

- If *any* NP-hard problems is solvable in polynomial time, then *every* NP-complete problem (in fact, every problem in NP) is also solvable in polynomial time.
- If *any* problem in NP 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).

Satisfiability and 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 (T or F) to the variables such that it evaluates to T. (As opposed to the case where every variable assignment results in F.) 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 $x =$ T and $y = z =$ F. 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 truth values to x, y, and z evaluates to F.

The *boolean satisfiability problem* (SAT) is as follows: given a boolean formula F, is it possible to assign truth values (T or F) 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: The certificate consists of an assignment of values true and false to each of the variables. We then plug the values into the formula and evaluate it. If the formula's value is true, we accept the certificate, and otherwise we reject 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 verification program that runs in polynomial time on a given input with a given certificate. Since the program runs in polynomial time, we can express its execution on a specific input in machine-code, which eventually is executed on the machine's logic circuitry, and the function of this circuitry can be faithfully expressed as a boolean formula. (Yes, this formula is *insanely long*, but it is of polynomial length, because the algorithm's running time is polynomial.)

The certificate (which is not given to us) can be encoded as a binary bit string, which we can further decode into a sequence of boolean variables (where T = 1 and F = 0).

This can be done so the formula is satisfiable if and only if there is a certificate that leads to valid verification if and only if the verification succeeds. Therefore, if you *could* determine the satisfiability of this formula in polynomial time, you could determine whether the verification algorithm succeeds.

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[15] boolean formula is satisfiable.

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

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

For example, the graph G shown in Fig. 87 has an independent set of size 6. (I believe this is the largest independent set in this graph.) Therefore $(G, 6) \in$ IS but $(G, 7) \notin$ 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.)

Claim: IS is NP-complete.

Proof: As with all NP-completeness proofs, there are two parts.

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

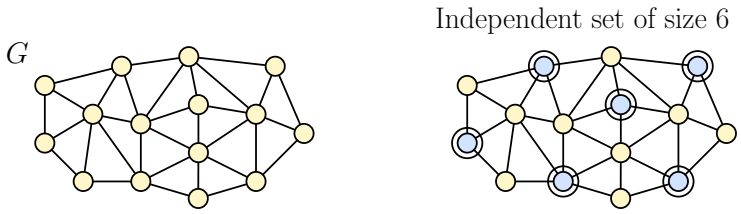


Fig. 87: A graph with an independent set of size $k = 6$.

IS is in NP: Recall that this means that it is possible to present a polynomial time verification procedure. This procedure is given a certificate that allows us to prove that the given graph has an independent set of the desired size. (If the instance does not have an independent set, then we don't care what the certificate contains.)

Given an instance of IS consisting of a graph $G = (V, E)$ and k, the certificate consists of a set of k vertices. We check that we are indeed given k distinct vertices of G, and for each pair of vertices u and v in this set, we check that there is no edge between them in G. If so, we accept the certificate and otherwise we reject it. If G is given by its adjacency matrix, we can do this in time $O(k^2) = O(n^2)$, so the verification runs in polynomial time.

IS is NP hard: It suffices to show that some known NP-complete problem (3SAT) is polynomially reducible to IS, that is, 3SAT $\leq_P$ IS. (Note the direction! We show that the known NP-hard problem is reducible to our new problem.)

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. (This is shown schematically in Fig. 88.) That is, $f(F) = (G, k)$, such that F is satisfiable if and only if G has an independent set of size k.

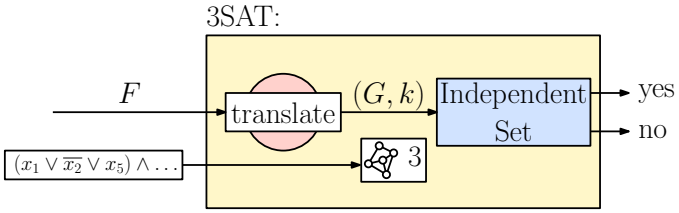


Fig. 88: Reduction of 3-SAT to IS.

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. The rest of this section presents this reduction in detail.

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 true.

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

Idea: Let's create a vertex in G for each literal in each clause. Intuitively, if the literal turns out to be true, we will put the corresponding vertex in our independent set. Note that we don't

know which literals are true or false, so we handle them all the same. (Unfortunately, this idea will not quite work, but we'll fix it in our construction.)

Requirements:

3SAT: By the nature of 3CNF (the conjunction of clauses) each clause must contain at least one literal whose value it true.

IS: V' must contain at least k vertices.

Idea: Let's organize the vertices of the graph into groups of three, called *clusters*, one per clause. We'll connect them together, so that exactly one vertex of each cluster can be in any independent set. We'll set k equal to the number of clauses, which will force us to pick exactly one vertex from each cluster to be in the final independent set. (Again, note that we don't know which these vertices will be, so we treat them all equally.)

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 complementary literals. (We don't know which literals will be true, but at least we konw that we cannot select both x_i and $\overline{x}_i$ to be in any independent set.)

In summary, our strategy will be to create groups of three vertices, one for each literal in each clause, which we call *clause clusters* (see Fig. 89). 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.

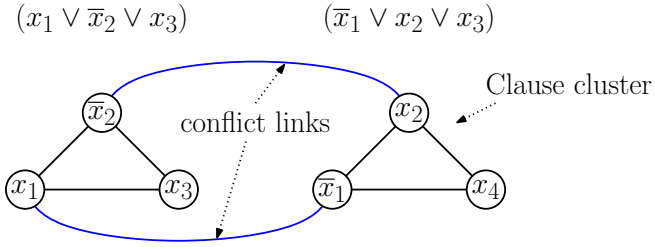


Fig. 89: Clause clusters for the clauses $(x_1 \vee \overline{x}_2 \vee x_3)$ and $(\overline{x}_1 \vee x_2 \vee x_5)$.

To enforce the restriction that only one of x_i and $\overline{x}_i$ can be set to T, we create edges between all vertices associated with x_i to all vertices associated with $\overline{x}_i$. 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.

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. 90. The clauses clusters appear in clockwise order starting from the top.

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

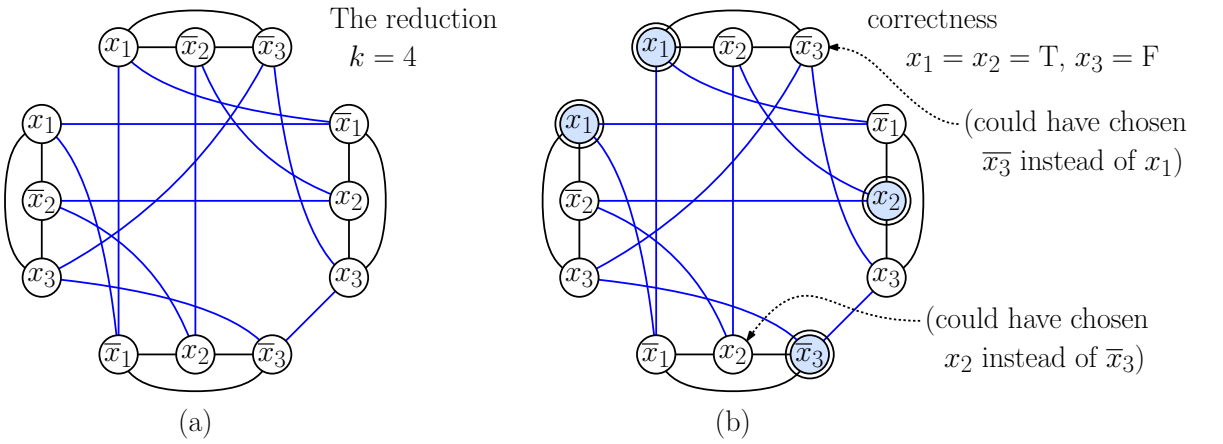


Fig. 90: 3SAT to IS reduction.

In our example, the formula is satisfied by the assignment $x_1 = \mathrm{T}$, $x_2 = \mathrm{T}$, and $x_3 = \mathrm{F}$. Note that the literal x_1 satisfies the first and last clauses, x_2 satisfies the second, and $\overline{x}_3$ satisfies 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 true. 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. 91).

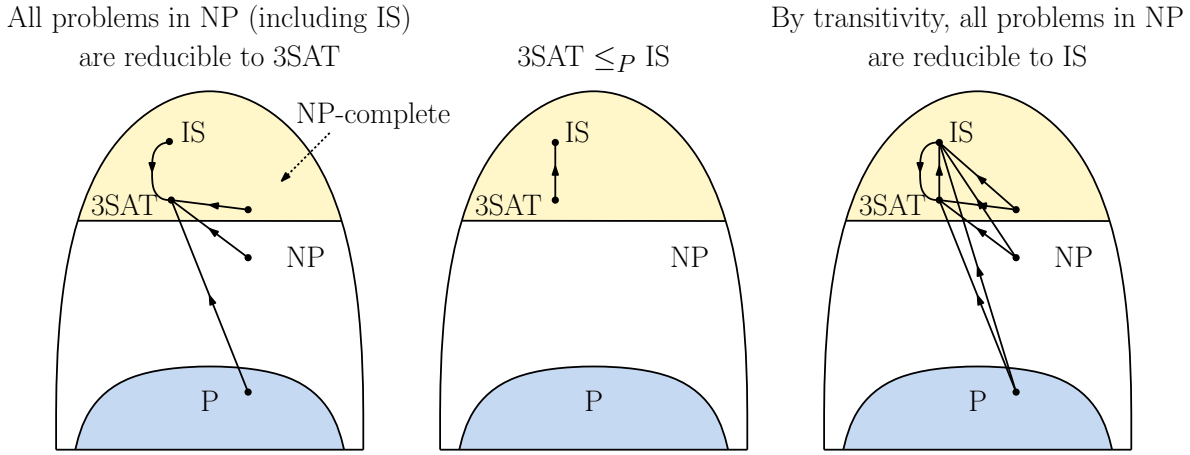


Fig. 91: Our updated picture of NP-completeness.

Lecture 17: NP-Completeness: 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? (See Fig. 92(a).)

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? (See Fig. 92(b).)

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? (See Fig. 92(c).)

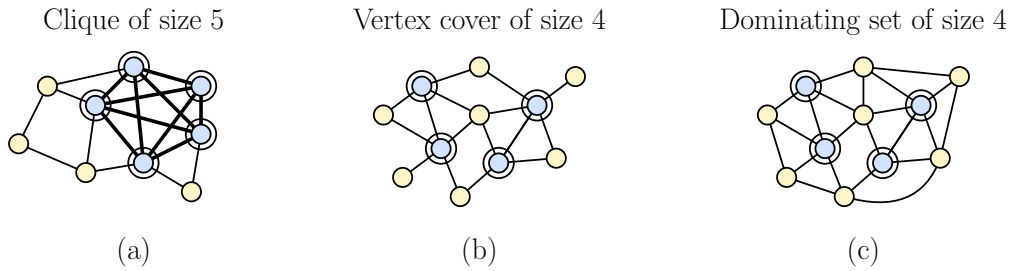


Fig. 92: Clique, Vertex Cover and Dominating Set.

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. 93). 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.

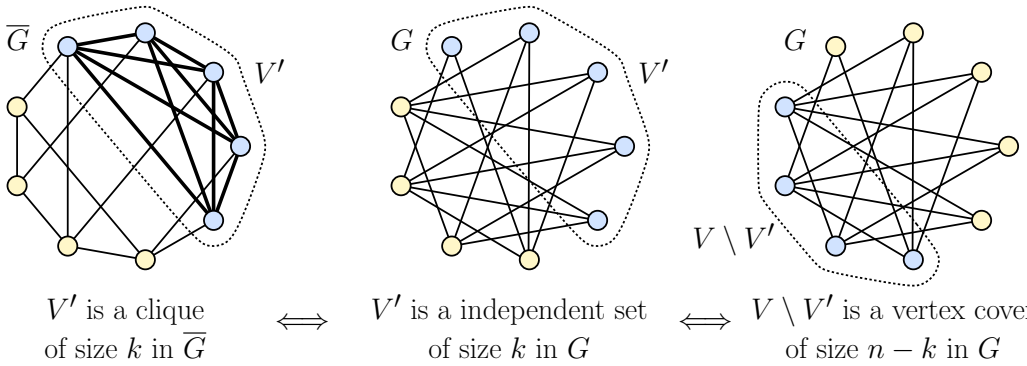


Fig. 93: Clique, Independent set, and Vertex Cover.

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

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

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

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

Proof:

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

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

Theorem: CLIQUE is NP-complete.

Proof:

- CLIQUE $\in$ NP: Consider an instance (G, k) for CLIQUE. The certificate consists of k vertices of G forming the set V'. We can check that all pairs of vertices in V' are adjacent (e.g., by inspection of $O(k^2) = O(n^2)$ entries of the adjacency matrix). If so, the verification succeeds and we accept, and otherwise the verification fails and we reject.
- 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.

Proof:

- VC $\in$ NP: Consider an instance (G, k) for VC. The certificate consists of k vertices of G forming the set V'. In $O(m) = O(n^2)$ time we can check that every edge in G has at least one endpoint in V'. If so, the verification succeeds and we accept, and otherwise the verification fails and we reject.
- 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 above lemma, these instances are equivalent.

We reiterate that in each of the above reductions, the reduction function merely translates similar elements between the two problems. It does not know whether G has an independent set or not. Even if it did, it does not know which vertices are in the independent set.

Dominating Set: In spite of the superficial similarity to Vertex Cover, Dominating Set is a bit trickier to show NP-completeness. As usual the proof has two parts. First, we show that DS $\in$ NP (see below). The trickier part is showing the some known NP-complete problem is reducible to DS. We will show that VC $\leq_P$ DS. That is, we want to show that there is a polynomial time function, which given an instance (G, k) for VC, produces an instance (G', k') for DS, 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 covering condition.

- VC: Every edge is incident to a vertex in V'.
- 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 edge" to "adjacent vertex". 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 inspires the following idea. 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 *mid-edge vertex*, called w_{uv}, and replace the edge (u, v) with the two edges (u, w_{uv}) and (v, w_{uv}) (see Fig. 94). 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.

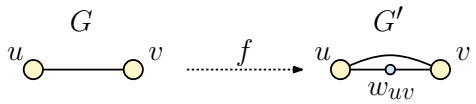


Fig. 94: Gadget for the VC $\leq_P$ DS reduction.

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 $k' = k + n_I$. Okay, we are now ready to state the result and prove it.

Theorem: DS is NP-complete.

Proof:

- DS $\in$ NP: Given an instance (G, k) for DS, we guess the certificate, which consists of the k vertices that will form the dominating set. We then verify that these vertices form a dominating set, by checking that every vertex of G is either in this set or is adjacent to a vertex in this set. If so, we output "yes" and otherwise "no". (Again, if G has a dominating set of size k, one of these guesses will work, and we correctly classify G as having a dominating set of size k. Otherwise, all fail and we classify G as not having such a dominating set.)
- VC $\leq_P$ DS: We want to show that given an instance of the VC problem (G, k), we can produce an equivalent instance of the DS problem in polynomial time. 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. 95. Note that every step can be performed in polynomial time.
  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'.
  ($\Rightarrow$) 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' (see Fig. 96).
  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 mid-edge 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

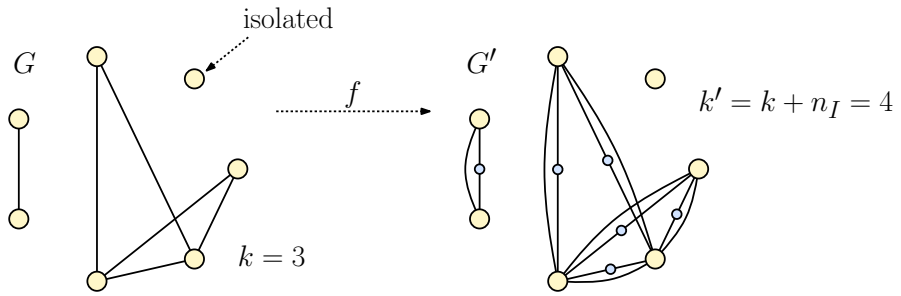


Fig. 95: Dominating set reduction with $k = 3$ and one isolated vertex.

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'' (see Fig. 96).

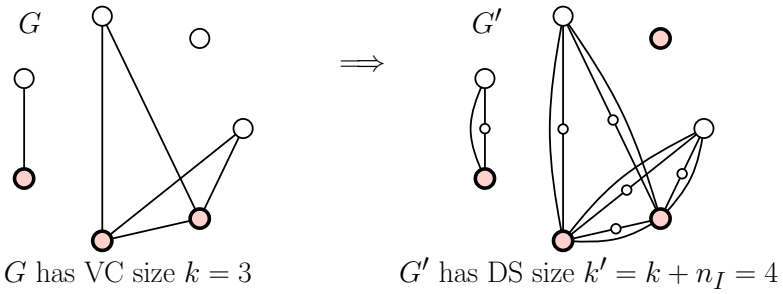


Fig. 96: $\Rightarrow$ part of the correctness of the VC to DS reduction (where $k = 3$ and $I = 1$).

($\Leftarrow$) 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 (see Fig. 97). 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.

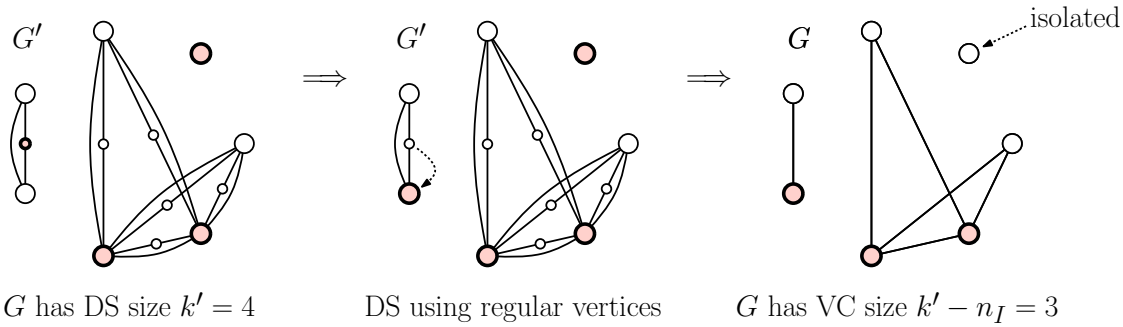


Fig. 97: $\Leftarrow$ part of the correctness of the VC to DS reduction (where $k = 3$ and $I = 1$).

However, we claim that we never need to use any of the newly created mid-edge 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 Fig 97.)
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 mid-edge 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'.

Whew! In conclusion, DS is NP-complete.

Summary: In this lecture we expanded our set of known NP-complete problems to include Clique, Vertex Cover, and Dominating Set.

Lecture 18: Approximations: 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.

Approximation 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, find the vertex cover of *minimum* size, the independent set of *maximum* size, color a graph with the *minimum* number of colors, or find the traveling salesman tour with the *shortest* cost. An approximation algorithm is one that returns a valid (or feasible) answer, but not necessarily one of the optimal size/weight/cost.

How do we measure how good an approximation algorithm is? We define the *approximation ratio* of an approximation algorithm as the worst-case ratio between the answer produced by the approximation algorithm and the optimum solution. For minimization problems (where the approximate solution will usually be larger than the optimum) this is expressed as approx/opt, and for maximization problems (where the approximate solution will be smaller than the optimum) this is expressed as opt/approx. Thus, in either case, the ratio will be at least 1, and the smaller the better.

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. Here are some possibilities (from best to worst):

- Some NP-complete problems can be approximated *arbitrarily well*. In particular, the user provides a parameter $\varepsilon > 0$ and the algorithm achieves an approximation ratio 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*, or PTAS.

- Some NP-complete problems can be approximated and the approximation ratio is a *constant*. (We will see examples later in this lecture.)
- Some NP-complete problems can be approximated, but the approximation ratio is a *function of* n. (For example, the set cover problem, can be approximated to within a factor of $\ln n$ by the greedy heuristic. It is provably hard to do better than $O(\log n)$ unless P = NP.
- Some NP-complete problems are *inapproximable* in the sense no polynomial time algorithm achieves an approximation ratio smaller than ∞ unless P = NP. (For example, there is no bound on the approximability of either independent set or graph coloring, unless P = NP.)

Vertex Cover: We begin by showing that there is an approximation algorithm for vertex cover with an approximation ratio of 2, that is, this algorithm will be guaranteed to find a vertex cover whose size is at most twice that of the optimum. Here is the vertex cover problem.

Vertex Cover (optimization): Given a graph $G = (V, E)$, compute a subset $V' \subseteq V$ of minimum size such that every edge $e \in E$ is incident to some vertex in V'. (See Fig. 98.)

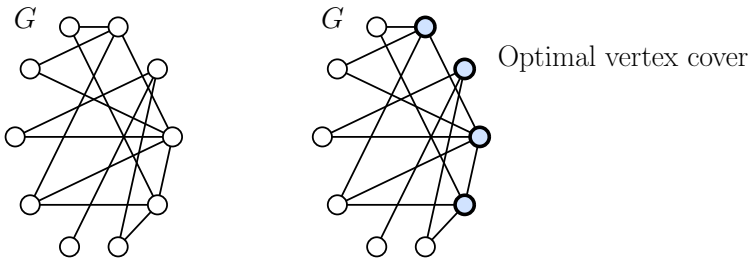


Fig. 98: Vertex cover (optimal solution).

How does one go about finding an approximation algorithm? A reasonably good strategy is the greedy heuristic, which involves repeatedly selecting the vertex of highest degree and add it to V'. Remove this vertex and its incident edges, and recompute degrees. This essentially simulates the greedy set-cover heuristic in the context of selecting vertices to cover edges. It can be shown that this leads to an approximation ratio of $\ln n$, where $n = |V|$.

The greedy heuristic is a good idea (and it probably works well in practice), but here's an approach that achieves an approximation ratio of 2. Recall that a *matching* in a graph $G = (V, E)$ is a subset of edges, $M \subseteq E$, such that each vertex is incident to at most one edge of M. A *maximum matching* has the largest number of edges. A weaker notion is a *maximal matching*, which has the property that it is not possible to add another edge.

While computing the maximum matching in a graph involves a fairly sophisticated algorithm (Edmond's blossom algorithm), there is a very simple algorithm for computing a maximal matching, which is all we need. The relevance of maximal matchings to vertex covers is given in the following claim.

Lemma: Given a graph $G = (V, E)$, let M be a maximal matching and let V^* be the optimum vertex cover in G. Then:

(i) $|V^*| \geq |M|$
(ii) $|V^*| \leq 2|M|$

Proof: To prove (i), observe that for any $(u, v) \in M$ *any* vertex cover needs to include either u or v, since otherwise this edge is not covered. Since this holds for any vertex cover, it holds for V^*.

To prove (ii), observe that since M is maximal, every edge (u, v) of G shares at least one endpoint with an edge of M, for otherwise we could add (u, v) to M, contradicting the fact that M is maximal. Thus, the union of all of M's endpoints, a set of size $2|M|$, is a vertex cover.

This lemma suggests a very simple algorithm. First, let's construct a maximal matching M in G. To do this, we repeatly, take any edge (u, v) of G, add it to M, then remove all the edges incident to either u or v. (If you do not want to actually change the graph, you can instead simply mark the edges, and repeatedly select any unmarked edge.) The result is clearly a matching, since whenever an edge is added, all conflicting edges are removed. It is also clearly maximal, since on termination, all edges have been removed.

To obtain a vertex cover, we exploit (ii) from the above lemma. Putting both endpoints of the edges of any maximal matching in G yields a vertex cover V' of size $2|M|$. By (i), $2|M| \leq 2|V^*|$, implying that V' is at most twice the size of the optimal vertex cover. The algorithm shown in the following code fragment, and it is illustrated in Fig. 99.

Approximation to vertex cover

```
approx-VC(G) {                          // returns an approximation to G's vertex cover
  V' = empty
  while (G has an edge (u,v)) {
    add both u and v to V'                          // add u and v to the cover
    delete from G all edges incident to u and to v  // remove all conflicting edges
  }
  return V' as the approximate vertex cover
}
```

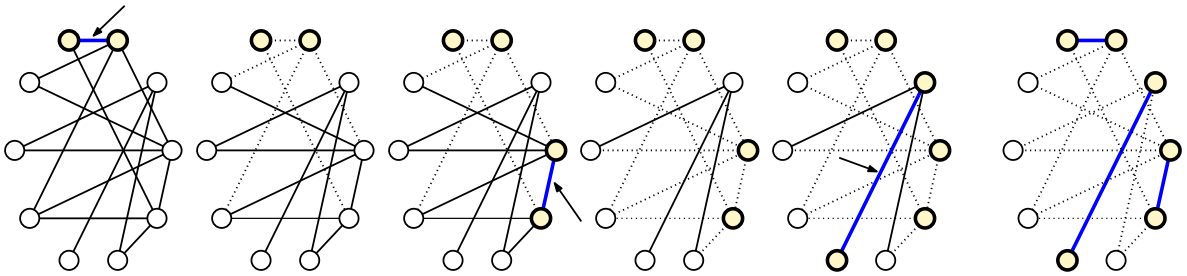


Fig. 99: The matching-based heuristic for vertex cover.

Summarizing the above analysis, we have:

Lemma: Algorithm `approx-VC` has an approximation ratio of 2.

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

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, in an earlier lecture, we proved that if V' is a vertex cover for G, then the complement vertex set, $V \setminus V'$, is an independent set for G. That is, G has a vertex cover of size k if and only if G has an independent set of size $n-k$, where $n = |V|$. Let k^* denote the size of the optimum vertex cover in G. By our heuristic, we can compute a vertex cover of size at most $2k^*$.

What if we try to apply this to independent set? We know that G's optimum independent set has size $n - k^*$. Our heuristic yields an independent set (through complementation) of $n - 2k^*$. Thus, we have achieve the following approximation ratio for independent set:

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

How good is this? It could be terrible! For example, if $n = 1001$ and $k = 500$, then the ratio is $501/(1001-1000) = 501/1 = 501$. Yikes! (And if this is not bad enough, just make both n and k a thousand times larger!)

In summary, just because you have a good approximation algorithm one NP-complete problem, it does not follow that you have a good approximation algorithm for them all.

Metric TSP Approximation: Our next result involves a famous topic called the *Traveling Salesman Problem* (TSP). Before introducing it, let us recall the concept of a metric space (which we saw the earlier lecture on the k-center problem). A (discrete) *metric space* consists of a finite set P points and a *distance function* $\delta(u,v)$ defined on each pair $u, v \in P$. This distance satisfies the following properties:

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

Most natural distance functions are metrics. (E.g., Euclidean distance for vectors, the shortest-path distance in graphs, and edit distance for strings.) Discrete metrics can be represented as an $n \times n$ distance matrix (which is symmetric) or equivalently as a complete undirected graph with nonnegative edge weights, where the weight on edge (u,v) is $\delta(u,v)$. Define the *weight* (or *cost*) of a path to be the sum of edge weights on the path. Given any set of edges T, let $\text{wt}(T)$ denote the sum of edge weights in T.

(Metric) Traveling Salesman Problem: (TSP) Given a set P of n points in a metric space, compute a simple cycle that visits all vertices (a Hamiltonian cycle) of minimum total weight (see Fig. 100).

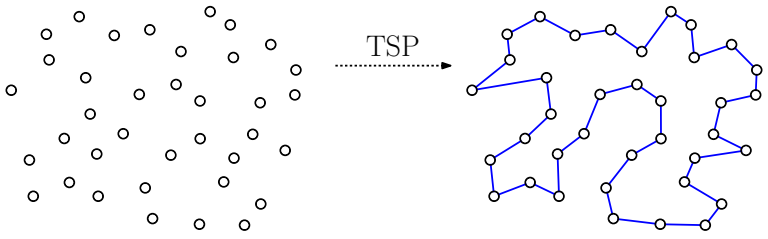


Fig. 100: The metric TSP Problem.

Factor-2 Approximation: We will present a simple MST-based algorithm that achieves a approximation ratio of 2. We will call this algorithm the *twice-around tour algorithm*.

A key insight to the algorithm is the observation if we remove any one edge from the TSP, we have a spanning tree. Of course, it is not necessarily (indeed, very unlikely to be) a minimum spanning tree. Therefore, the cost of the minimum TSP tour for P is at least as large as the cost of the MST, that is:

$$\text{wt}(\text{MST}(P)) \ \leq \ \text{wt}(\text{TSP}(P)).$$

This suggests the following idea for computing an approximation. We know we can compute the MST efficiently (e.g., by Kruskal's algorithm). Suppose we can compute a feasible TSP tour H that is larger than the MST by a constant factor c, then we will have

$$\text{wt}(H) \ \leq \ c \cdot \text{wt}(\text{MST}(P)) \ \leq \ c \cdot \text{wt}(\text{TSP}(P)),$$

implying that we have a factor-c approximation to TSP!

How to convert a spanning tree into a TSP tour? Given any free tree we can generate a tour, called a *twice-around tour*, as follows:

- Starting at any vertex of the tree, traverse the edges by walking around the tree. Thinking of the tree as a wall, place your hand against the wall, and walk along it until returning to the starting point (see Fig. 101).
- This cycle may visit vertices multiple times. Shortcut the cycle by skipping all repeated vertices. Note that, by the triangle inequality, whenever we apply short-cutting, we cannot increase the length of the cycle (see Fig. 101).

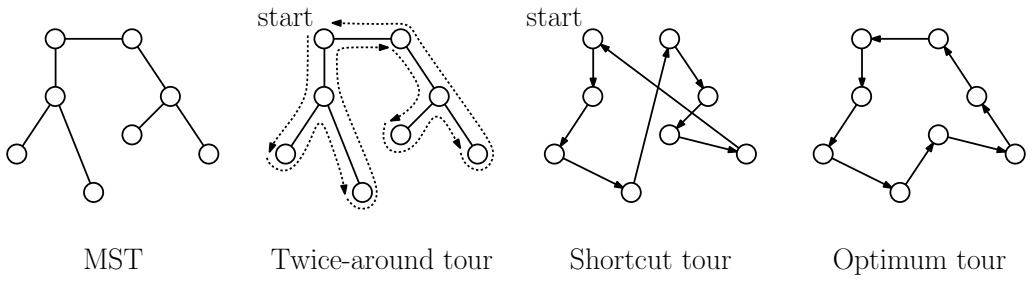


Fig. 101: The twice-around TSP Approximation.

Since the original graph is a complete graph, the MST can be computed in $O(n^2 \log n)$ time, and the twice-around tour can be computed by DFS in $O(n)$ time. Observe that the resulting cycle is a valid TSP tour. How close is it to the optimal TSP tour? (See Fig. 102 for a larger example).

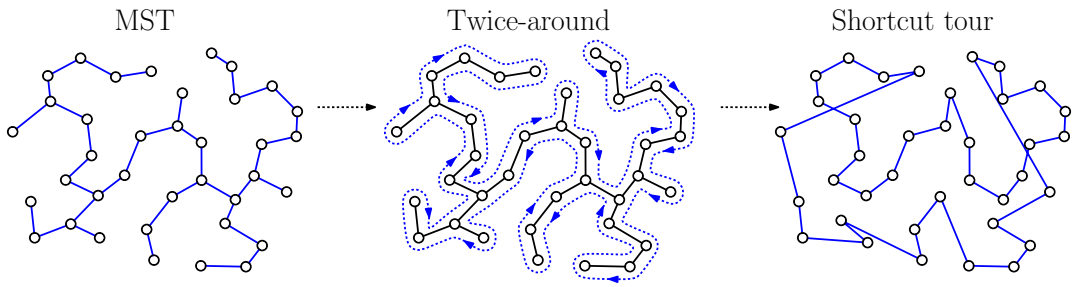


Fig. 102: A larger example of the execution of the twice-around TSP heuristic.

Lemma: The Twice-around TSP heuristic achieves an approximation ratio of 2.

Proof: Let H denote the tour produced by this algorithm, and let H^* be the optimum TSP tour. Let T be the edges of the minimum spanning tree. As observed earlier

$$\text{wt}(T) \;\leq\; \text{wt}(H^*).$$

Since every edge is traversed twice, the pure twice-around tour of T has weight exactly $2 \cdot \text{wt}(T)$. By the triangle inequality, whenever we shortcut an edge of T to form H we do not increase the cost of the tour, and so we have

$$\text{wt}(H) \;\leq\; 2 \cdot \text{wt}(T).$$

Combining these we have

$$\text{wt}(H) \;\leq\; 2 \cdot \text{wt}(T) \;\leq\; 2 \cdot \text{wt}(H^*) \quad\Longrightarrow\quad \frac{\text{wt}(H)}{\text{wt}(H^*)} \;\leq\; 2,$$

as desired.

Christofides Algorithm: We can improve on this 2-factor approximation to obtain a 1.5-factor approximation by a clever observation. The result is called *Christofides algorithm*, and it was discovered by Nicos Christofides in 1976 (and like many algorithms of the day, it was independently discovered in the former Soviet Union by Anatoliy Serdyukov).

The key observation that Christofides made is that a source of wastage in the twice-around tour involves the fact that the lower bound hits every edge twice. What if we only had to hit each edge once? Are there graphs that have a single cycle that hits all the edges exactly once? Such a cycle is called an *Eulerian circuit* (or *Eulerian cycle*). We will use the following fact, without proof.

Lemma: A graph $G = (V, E)$ has an Eulerian circuit if and only if every vertex has even degree (see Fig. 103). Given such a graph, an Eulerian cycle can be computed in time $O(n + m)$, where $n = |V|$ and $m = |E|$.

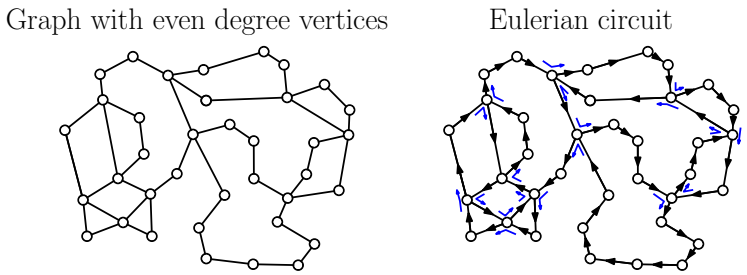


Fig. 103: Eulerian circuit in a graph where every vertex has even degree.

But how can we exploit this? The MST will certainly contain vertices of odd degree. A useful fact (called the *Handshaking Theorem*) states that every graph has an even number of vertices with odd degree. (This is an easy consequence of the observation that the sum of vertex degrees is twice the number of edges, which is always even.) Given the minimum spanning tree of P, let P_0 denote the vertices having even degree in the MST and let P_1 denote the vertices of odd degree. The handshaking theorem states that $|P_1|$ is even. Christofides algorithm works by adding a set of edges to to the vertices of P_1 so that each gets one additional edge. Combining these edges with the MST edges, all the vertices of P now have even degree, and hence by the above lemma, we can efficiently construct an Eulerian tour in this graph. The question is how to do this without increasing the weight very much.

This will be done with the concept of matchings. Recall that a *matching* in a graph is a set of edges M such that each vertex is incident to at most one edge of M. A matching is *perfect* if every vertex is

incident to an edge of M. Consider the complete graph induced by a P of points of even cardinality in a metric space, where the edges are weighted by the distance between vertices. Because this graph has an even number of vertices and is complete, it has a perfect matching. Define the *minimum-weight matching*, denoted MWM(P) to be the perfect matching of minimum total weight on P (see Fig. 104). Such a matching can be computed in $O(|V|^2|E|)$ time by a modification of Edmond's blossom algorithm, which for us is $O(n^4)$, since our graph is complete.

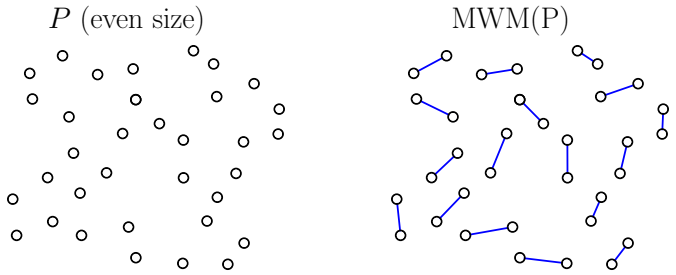


Fig. 104: A point set P of even cardinality and its minimum-weight matching MWM(P).

We can now describe Christofides algorithm. The input is a set of n points P in a metric space.

(a) Compute the edges of the minimum spanning tree of P, that is, $T \leftarrow \text{MST}(P)$ (see Fig. 105(a)).

(b) Partition P into sets P_0 and P_1 of points whose MST vertex has even or odd degree, respectively. (P_1 has an even number of points.)

(c) Compute the minimum-weight perfect matching of the points of P_1, that is, $M \leftarrow \text{MWM}(P_1)$.

(d) Form the graph $G = (P, T \cup M)$ on the point set P whose edge set is the union of T and M. (Note that this can result in some edges being replicated, and hence G is a multi-graph. Nonetheless, all the vertices in G have even degrees.)

(e) Let C' be any Eulerian circuit in G. Apply short-cutting to C' by removing repeated vertices. Let C denote the resulting tour. (By the triangle inequality, $\text{wt}(C) \leq \text{wt}(C')$.)

(f) Return C as the final approximation.

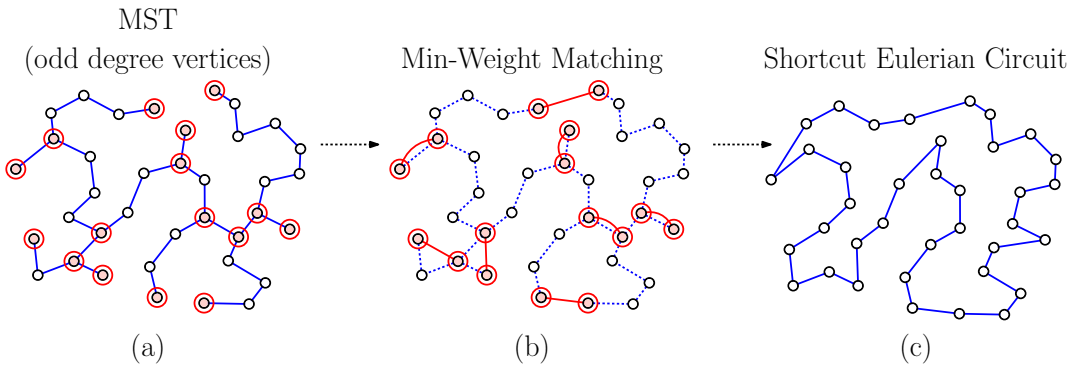


Fig. 105: Christofides algorithm: (a) MST(P) with odd-degree vertices highlighted, (b) MWM(P_1) shown with red edges, (c) the final tour after short-cutting.

A key to the efficiency of Christofides algorithm is that the added edges from the minimum-weight matching are small relative to the weight of the TSP tour. This is shown in the following lemma.

Lemma: Given a point set P, let P_1 be any subset of P of even cardinality. Then

$$\text{wt}(\text{MWM}(P_1)) \;\leq\; \frac{\text{wt}(\text{TSP}(P))}{2}.$$

Proof: First, observe that if we remove any points from P, the weight of the optimum TSP tour can only decrease, and hence $\text{wt}(\text{TSP}(P_1)) \leq \text{wt}(\text{TSP}(P))$. Consider the edges of $\text{TSP}(P_1)$. Since P_1 has even cardinality, the tour has an even number of edges. Let M_0 and M_1 denote result of taking every other edge from $\text{TSP}(P_1)$, with M_0 taking the even numbered edges and M_1 taking the odd numbered edges (see Fig. 106). Thus, we have

$$\text{wt}(M_0) + \text{wt}(M_1) \;=\; \text{wt}(\text{TSP}(P_1)).$$

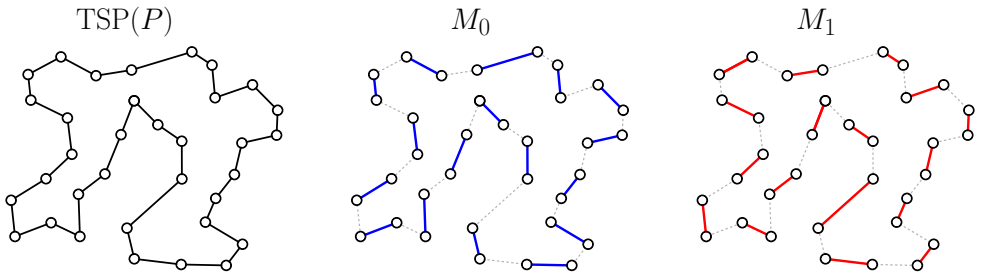


Fig. 106: Given a set P of even cardinality, the edges of $\text{TSP}(P)$ can be partitioned into two matchings.

Observe that both M_0 and M_1 are matchings over P_1. Their weight cannot be smaller than the weight of the minimum weight matching. Using this and the fact that for any numbers a and b, their minimum cannot be larger than their mean ($\min(a,b) \leq (a+b)/2$), we have

$$\text{wt}(\text{MWM}(P_1)) \;\leq\; \min(\text{wt}(M_0), \text{wt}(M_1)) \;\leq\; \frac{\text{wt}(\text{TSP}(P_1))}{2} \;\leq\; \frac{\text{wt}(\text{TSP}(P))}{2},$$

as desired.

We can now prove the approximation bound for Christofides algorithm.

Lemma: Christofides algorithm achieves an approximation ratio of 1.5.

Proof: Let C denote the tour produced by this algorithm and let H^* be the optimum TSP tour. Let T be the edges of the minimum spanning tree, and let M be the edges of the minimum-weight matching on the vertices P_1 of odd degree. As shown in the twice-around heuristic, we have

$$\text{wt}(T) \;\leq\; \text{wt}(H^*).$$

As proved in the previous lemma $\text{wt}(M) \leq \text{wt}(H^*)/2$. Therefore,

$$\text{wt}(T) + \text{wt}(M) \;\leq\; \frac{3}{2} \cdot \text{wt}(H^*).$$

Therefore, the length of the Eulerian circuit C' satisfies this same bound, since it visits each of these edges exactly once. By the triangle inequality, whenever we shortcut the edges of the Eulerian circuit we do not increase the cost of the tour, and so we have

$$\text{wt}(C) \;\leq\; \text{wt}(C') \;\leq\; \frac{3}{2} \cdot \text{wt}(H^*).$$

In summary, letting C denote the result of Christofides algorithm, we have

$$\frac{\text{wt}(C)}{\text{wt}(H^*)} \;\leq\; \frac{3}{2},$$

as desired.

Lecture 19: Approximation: Subset Sum

Subset Sum: The Subset Sum problem (SS) is the following. Given a finite set S of positive integers $S = \{x_1, \ldots, x_n\}$ and a *target value*, t, we want to determine the subset $S' \subseteq S$ that sums to a value that is as close as possible to t, without exceeding it. Consider the following example.

$$S = \{3, 6, 9, 12, 15, 23, 32\} \qquad \text{and} \qquad t = 34.$$

The subset $S' = \{6, 12, 15\}$ sums to $t = 33$, which is optimal, since there is no subset adding up to 34. We can turn this into a decision problem by asking whether there exists a subset that sums exactly to t.

The decision version of the subset sum problem is known to be NP-complete. SS is NP-complete. We will not prove this, but there is a reduction from Vertex Cover. We will present the reduction at the end of the lecture. In this lecture, we will present an approximation algorithm.

Polynomial Approximation Scheme: Next, we will present an approximation algorithm for Subset Sum. In a previous lecture, we introduced the notion of a *polynomial approximation scheme*. This is a function that, for any $\varepsilon > 0$, computes a factor-$(1 + \varepsilon)$ approximation to our problem, such that the running time is a polynomial of the input size n.

For example, the running time of the algorithm might be $O(2^{(1/\varepsilon)}n^2)$. 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 ε component of the running time would be on the order of 2^{100}, which is way too large for practical purposes.) A *fully polynomial approximation scheme* is one in which the running time is polynomial in both n and $1/\varepsilon$. For example, a running time of $O((n/\varepsilon)^2)$ would satisfy this condition. In such cases, reasonably accurate approximations are computationally feasible.

While PTAS's exist for some NP-complete problems, there are a significant number of NP-complete problems, called *APX-complete*, for which is it suspected that none have PTAS solutions. Indeed, if any of these problems had a PTAS solution, then they all would. Examples include optimization versions of many of the NP-complete decision problems we have studied, including

- MAX-3SAT-3: a variant of 3SAT where the objective is to maximize the number of satisfied clauses
- MAX-IS: maximum-sized independent set
- MIN-VC: minimum-sized vertex cover
- MIN-DS: minimum-sized dominating set
- MIN-TSP: minimum-weight TSP tour in metric spaces

For more information, see this Wikipedia article.

Subset Sum Optimization: In the optimization version of the problem, we are given a set of positive integers $S = \{x_1, \ldots, x_n\}$ and a target value t, and we want to determine the subset whose sum is as large as possible but not larger than t. (In knapsack terms, this corresponds to filling the knapsack as full as possible, without overflowing.)

Let z^* denote the optimum sum achievable. (Clearly, $z^* \leq t$.) Any answer we compute cannot exceed z^*, but we don't want it to be too much smaller. Given an approximation parameter ε, where $0 < \varepsilon$, we seek a subset $S' \subseteq S$ whose sum z achieves an approximation ratio of at most $1 + \varepsilon$, that is,

$$\frac{z^*}{1+\varepsilon} \;\leq\; z \;\leq\; z^*.$$

It will be easier to prove a similar bound. Let's assume that $\varepsilon < 1/2$. We will show that our approximation z satisfies $z^*(1-\varepsilon) \leq z$. This is not quite what we want, but if we run our algorithm with the ε parameter set to $\varepsilon/2$, then our approximation error will be

$$1 - \frac{\varepsilon}{2} \;=\; \left(1 - \frac{\varepsilon}{2}\right)\frac{1+\varepsilon}{1+\varepsilon} \;=\; \frac{1+\varepsilon-\varepsilon/2-\varepsilon^2/2}{1+\varepsilon} \;\geq\; \frac{1+\varepsilon-\varepsilon}{1+\varepsilon} \;=\; \frac{1}{1+\varepsilon}.$$

(In the middle inequality, we used the fact that $\varepsilon < 1$ so $\varepsilon^2/2 < \varepsilon/2$.) This change in the value of ε will only affect the constant factors in our running time.

In order to discuss running times, we need to agree on the input size. Each number in the input is of magnitude at most t (since otherwise, we may just ignore it), therefore each needs at most $\lceil \lg t \rceil$ bits to represent. We will take the input size to be $n \lceil \lg t \rceil = O(n \log t)$.

Exact Algorithm: Before presenting the approximation algorithm, let's start with a simple, but very inefficient, exact solutions. The approach will be reminiscent of dynamic programming. We will consider subproblems involving the first i values, for $i = 1, 2, \ldots, n$. Let L_i denote a list of all possible sums of the 2^i subsets of $\{x_1, \ldots, x_i\}$. This includes the empty set whose sum is 0. For example, for the set $\{1, 4, 6\}$, we have

$$L_3 \;=\; \langle 0, 1, 4, 5, 6, 7, 10, 11 \rangle,$$

(which includes the original values plus the sums $1+4$, $1+6$, $4+6$, and $1+4+6$). L_i can have as many as 2^i elements, but may have fewer, since there may be duplicates. There are two things we will want to do for efficiency:

(1) Remove any duplicates from L_i.

(2) Keep only those sums that are less than or equal to t.

Let us assume we have access to a function `merge`(L_1, L_2), which merges two sorted lists and returns the result. Let $L + x$ denote the list resulting by adding the number x to every element of list L. To incorporate x_i, we perform `Merge`$(L, L + x_i)$, which merges L with a shifted copy of L by x_i. We also have a function `trim`(L, t), which eliminates any duplicates that arose in the merging process and also removes any elements larger than t (see Fig. 107). (Of course, both merging and trimming could be done in a single function, but in the next section we will see why it is useful to separate them.) Both functions run in linear time.

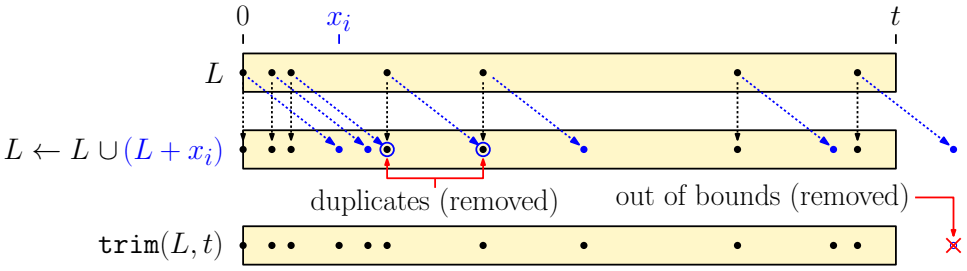


Fig. 107: Merging lists and trimming.

The algorithm operates by initializing L to a sequence containing just the element 0, and then repeatedly includes x_i by merging L with $L + x_i$ and trimming the result. The algorithm is presented in the code block below. The final algorithm is presented in the following code block.

A simulation of the algorithm on the input $S = \{1, 3, 4, 6\}$ and $t = 11$ is illustrated in Fig. 108.) The algorithm runs in $O(2^n)$ (exponential!) time in the worst case, because this is the number of sums that are generated if there are no duplicates, and no items are trimmed.

```
exact-subset-sum(x[1..n], t) {              // exact solution to subset sum
    L = <0>                                 // basis case - no elements in sum
    for (i = 1 to n) {
        L = merge(L, L + x[i])              // add in x[i] and...
        L = trim(L, t)                      // ...remove duplicates and items > t
    }
    return the largest element in L         // return largest sum that is at most t
}
```

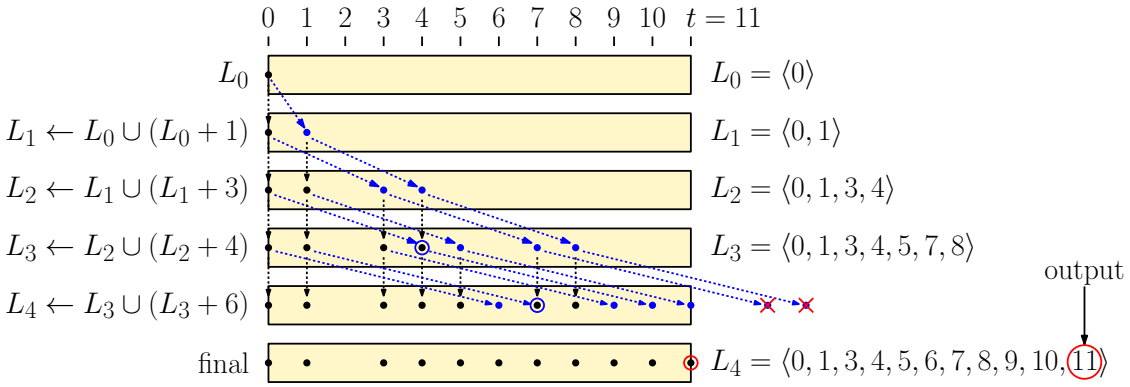


Fig. 108: The execution of `exact-subset-sum` on the input $S = \{1, 3, 4, 6\}$ and $t = 11$.

Approximation Algorithm: The exact algorithm runs in exponential time because it is explicitly generating the sums of all 2^n subsets of the n input values. In order to obtain an efficient approximation algorithm, we will not save all the possible sums. Instead, we will be a employ an operation that "compresses" each of the lists L_i by identifying elements that have very similar values and replacing them by a single value. For example, suppose that some list contains both the values 9851 and 9852. Do we really need to keep both of these values, considering that they are roughly 0.01% different from each other?

We need to be careful, however. Suppose, for example, that we have an entire sorted sequence where each value is nearly equal to its predecessor, e.g., $\langle 9851, 9852, 9853, 9854, \ldots\rangle$. Say we kept 9851, how many subsequent values should we eliminate? Given that we want to achieve a relative error of at most $1 - \varepsilon$, the answer is that as soon as we see a value that exceeds $9851/(1 - \varepsilon)$, we must save it. For example, if $\varepsilon = 0.01$, then $9851/(1 - \varepsilon) \approx 9950$, and as soon as we see a value larger than this, we need to keep it.

However, this alone will lead to errors that are too large. The issue is that the small errors that we commit at the early phases of the algorithm will accumulate as more items are included. Since there are n phases in total, it seems like we should make the allowed error smaller as a function of n. Indeed, we will see that the proper allowable error is not ε but $\delta = \varepsilon/n$. (Our final analysis will bear this out.)

The approximation algorithm works as follows. Let L be a sorted list of integers, and let δ denote the desired error bound. We build the compressed list L' as follows. Start by adding the first element of L to L'. The variable `prev` stores the previous item that was added. We traverse L (in ascending order) and when we find the first next element y that is sufficiently different from `prev`, we add it to L'. Because the relative allowed relative error is δ, the condition that an element y is "sufficiently different" is

$$\frac{y - \texttt{prev}}{y} > \delta \qquad \text{or equivalently} \qquad y > \frac{\texttt{prev}}{1 - \delta}.$$

The final compression algorithm is presented in the following code block.

Compressing a list with precision δ

```
compress(L, delta, t) {    // compress sorted list L with error delta and limit t
    L' = empty                          // start with an empty list
    prev = 0                            // previous item added
    for each (y in L) {
        if (y > t) break                // ignore values larger than t
        if (y > prev/(1-delta)) {       // y is different enough?
            append y to L'              // add y
            prev = y                    // update previous
        }
    }
    return L'                           // return the compressed list
}
```

For example, given $\delta = 0.1$ and $t = 60$, and given the input list

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 51, 53, 54, 56, 80 \rangle,$$

the algorithm first takes 10. It skips 11, since $11 \leq 10/0.9 \approx 11.1$. It takes 12, 15, and 20, but then skips 21 and 22 since both are $\leq 20/0.9 \approx 22.2$. It takes 51, but then skips 53, 54 and 56 since they are all $\leq 51/0.9 \approx 56.7$. Finally, we ignore 80 because it is larger than t. The final compressed list L' is

$$L' = \langle 10, 12, 15, 20, 51 \rangle.$$

There is an alternative way to understand the compression process via bucketing. Suppose that we subdivide the interval from $[1, t]$ into a set of *buckets* of exponentially increasing size. That is, let $\sigma = 1/(1-\delta)$, which is greater than 1. Next, break the real line at the points $\{1, \sigma, \sigma^2, \sigma^3, \ldots\}$, and consider the *bucket* of points lying between any two consecutive break points, $[\sigma^{i-1}, \sigma^i]$. Define an alternative compression procedure $\texttt{compress}^*(L, \delta, t)$, which keeps the smallest element in each bucket, provided that element is at most t (see Fig. 109).

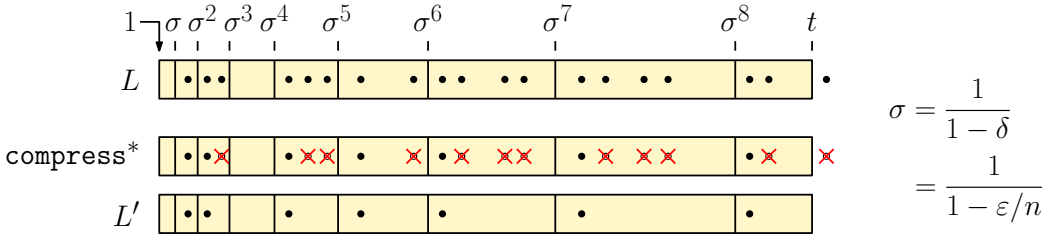


Fig. 109: An alternative view of compression by bucketing.

While these methods do not produce identical results, they are effectively doing the same sort of pruning. For any two such points that lie within the same bucket, $\sigma^{i-1} \leq y \leq y' \leq \sigma^i$, their relative error is

$$\frac{y' - y}{y'} = 1 - \frac{y}{y'} \leq 1 - \frac{\sigma^{i-1}}{\sigma^i} = 1 - \frac{1}{\sigma} = \delta,$$

which means that we can safely keep y and eliminate y'. Thus, we could have equivalently defined the compression process by keeping the smallest value in each bucket and eliminating all the others. (We keep the smallest because any larger value might exceed t, which would be of not use to us.)

Compression is valuable because, rather than storing all 2^n possible sums, we keep a much smaller set. Our next lemma shows that, after compression, the number of items remaining is polynomial in the input size ($n \log t$) and $1/\varepsilon$.

Lemma: After invoking `compress`(L, δ, t), where $\delta = \varepsilon/n$, the number of nonzero items in the resulting list is $O((n \log t)/\varepsilon)$.

Proof: We know that each pair of consecutive elements in the compressed list differ by a ratio of at least σ, where σ is defined to be $1/(1-\delta)$. Let k denote the number of nonzero elements in the compressed list. It follows that the ratio between the maximum and minimum nonzero elements is at least σ^{k-1}. Since the smallest element is at least 1, and the largest is not greater than t, we have

$$\sigma^{k-1} \;\le\; t \;\;\Longrightarrow\;\; k-1 \;\le\; \frac{\ln t}{\ln \sigma} \;=\; \frac{\ln t}{\ln(1/(1-\delta))}.$$

Using the standard facts that $\ln(1/x) = -\ln x$ and $-\ln(1-x) \ge x$, we obtain

$$k \;\le\; 1 + \frac{\ln t}{\ln(1/(1-\delta))} \;=\; 1 + \frac{\ln t}{-\ln(1-\delta)} \;\le\; 1 - \frac{\ln t}{\delta}.$$

By our definition of δ, we have

$$k \;\le\; 1 + \frac{n \ln t}{\varepsilon},$$

which is $O((n \log t)/\varepsilon)$, as desired.

We can now present our approximation algorithm. It is essentially the same as the exact algorithm, except we replace the call to `trim` with our more aggressive `compress` function. It is presented in the following code block.

Approximate Subset Sum

```
approx-subset-sum(x[1..n], t, eps) {      // eps-approximate subset sum
    delta = eps/n                         // approx factor per stage
    L = <0>                               // basis case - no elements in sum
    for (i = 1 to n) {
        L = merge(L, L + x[i])            // add in x[i] and...
        L = compress(L, delta, t)         // ...compress similar values and items > t
    }
    return the largest element in L       // return largest sum that is at most t
}
```

An example of the execution of a single phase of the algorithm is shown in Fig. 110 (where we have take the perspective of compression based on bucketing.)

The running time of the procedure is $O(n|L|)$. By our previous lemma $|L| = O((n \log t)/\varepsilon)$, so the running time is $O((n^2 \log t)/\varepsilon)$. Note that the input size is $O(n \log t)$, so this is (weakly) polynomial as a function of input size.

Approximation Analysis: The final question is why the algorithm achieves an relative error of at most ε over the optimum solution. Let Y^* denote the optimum (largest) subset sum and let Y_ε denote the value returned by our ε-approximation algorithm. We know that $Y_\varepsilon \le Y^*$. We want to show that Y_ε is not too much smaller, that is,

$$Y_\varepsilon \;\ge\; Y^*(1-\varepsilon).$$

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, the total relative error should be (obviously?) $n(\varepsilon/n) = \varepsilon$. The catch

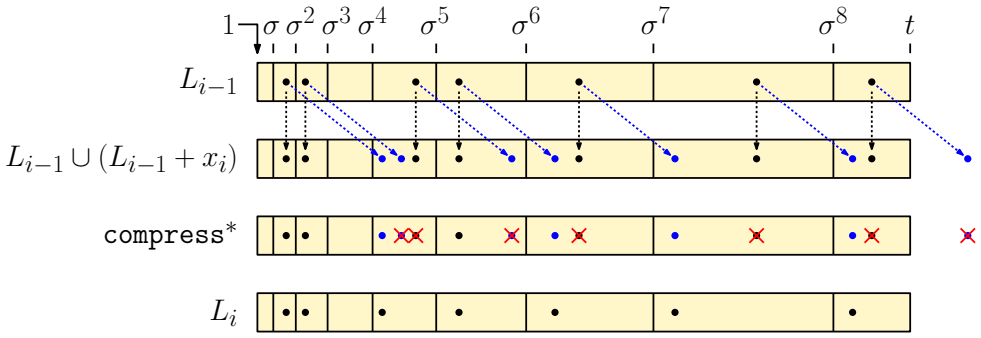


Fig. 110: A single phase of the algorithm (from the bucketing perspective).

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 (exact) algorithm, and let L_i denote the i-th list in the approximate algorithm. We want to show that these two lists are similar. In particular, we will show that for each y in the exact list L_i^*, there is an element z that is close to y. By "close", we mean that z's value is smaller by a factor of just[16] $(1-\varepsilon/n)^i$. We refer to z as y's *representative*.

Lemma: For $0 \le i \le n$, for each $y \in L_i^*$ there exists $z \in L_i$ such that

$$(1-\delta)^i y \;\le\; z \;\le\; y,$$

where $\delta = \varepsilon/n$.

Proof: We will only prove the lower bound ($(1-\delta)^i y \le z$), and we'll leave the upper bound as an exercise. The proof by induction on i. Initially $L_0 = L_0^* = \langle 0 \rangle$. Since both lists are the same, the lemma holds trivially. Let's assume that the induction hypothesis holds for $i-1$, and we will prove it for i. Every element in L_i^* originates in one of two ways, either it is equal to an element $y \in L_{i-1}^*$ or it is $y + x_i$ for some $y \in L_{i-1}^*$. It suffices to show that a representative exists for both y and $y + x_i$.

By our induction hypothesis, there is a representative z to y in L_{i-1}, which means that

$$(1-\delta)^{i-1} y \;\le\; z \tag{2}$$

(see Fig. 111). By adding x_i to each of the above terms and observing that $(1-\delta)^{i-1} < 1$, we obtain a similar bound.

$$(1-\delta)^{i-1}(y + x_i) \;\le\; z + x_i. \tag{3}$$

One or both of z and $z + x_i$ might not appear in L_i because they were removed by compression. Let z' and z'' be elements of L_i that caused them to be removed. (In the bucket perspective, these are the smallest elements of the buckets containing z and $z + x_i$, respectively.) Thus, z' and z'' are elements of L_i, and by the rules of compression

$$(1-\delta)z \;\le z' \qquad \text{and} \qquad (1-\delta)(z + x_i) \;\le z''.$$

Combining with Eq. (2), we have

$$(1-\delta)^i y \;=\; (1-\delta)^{i-1}(1-\delta) y \;\le\; z'.$$

[16] By "just" we mean that this factor is quite close to 1. Recall that we assumed that $\varepsilon < 1$, and $1/n$ is certainly very small for large input sizes. Thus, $1 - \varepsilon/n$ is very close to 1. Raising this to the power i will make is smaller, but we'll show that it won't be much smaller.

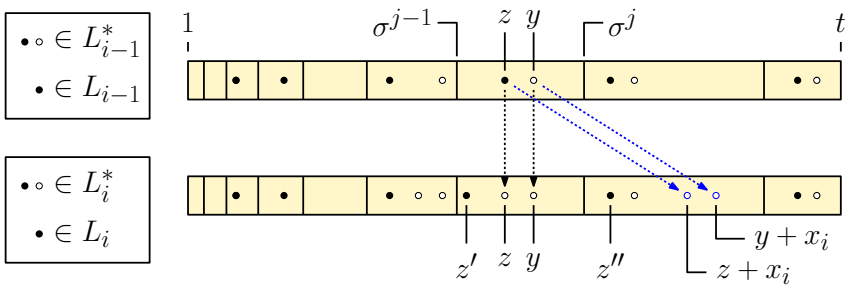


Fig. 111: Subset sum approximation analysis.

Doing the same for z'' yields $(1-\delta)^i(y+x_i) \le z''$. Since this applies to all $y \in L^*_{i-1}$ and z and z'' are both in L_i this completes the proof.

Applying the above lemma with $i = n$, with y playing the role of Y^*, and with Y playing the role of z, it follows that there exists $Y \in L_n$ (the final approximation list) such that

$$\left(1 - \frac{\varepsilon}{n}\right)^n Y^* \;\le\; Y \;\le\; Y^*.$$

To complete the proof, we will apply a standard result from real analysis, which we will not prove.

Lemma: For $n > 0$ and real a,

$$(1+a) \;\le\; \left(1 + \frac{a}{n}\right)^n \;\le\; e^a.$$

By applying this with $a = -\varepsilon$, we have

$$(1-\varepsilon)Y^* \;\le\; Y \;\le\; Y^*.$$

which implies that our final answer Y is a valid answer and satisfies the ε-approximation bound! As observed earlier, the algorithm runs in time $O((n/\varepsilon)\log t)$, which is a polynomial function of both the input size and ε. Therefore, we have obtained the desired PTAS for subset-sum.

Summary: In this lecture, we introduced the subset-sum problem (SS), a simplification of the 0-1 knapsack problem. We showed that the problem is NP-complete, and we also presented a $(1-\varepsilon)$-factor polynomial-time approximation algorithm, that is, a PTAS.

SS is NP-complete: (Optional) 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 $\le_P$ 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 $O(n \cdot b)$ time, which is polynomial in the input size, 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

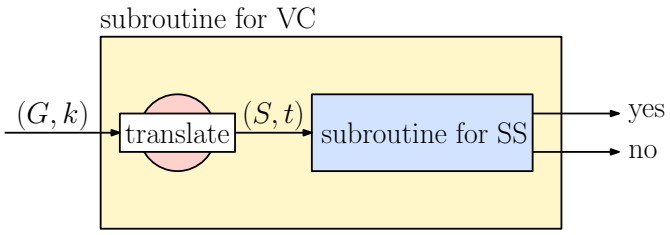


Fig. 112: Reducing VC to SS.

cover of size k if and only if S has a subset summing to t (see Fig. 112). Thus, if subset sum were solvable in polynomial time, so would vertex cover.

How can we encode the notion of selecting a subset of vertices that cover all the edges to that of selecting a subset of numbers that sums to t. In the vertex cover problem we are selecting vertices, and in the subset sum problem we are selecting numbers, so it seems logical that the reduction should map vertices into numbers. The constraint that these vertices should cover all the edges must be mapped to the constraint that the sum of the numbers should equal the target value.

An Initial Approach: Here is an idea, which does not work, but gives a sense of how to proceed. We are given a pair (G, k), and want to know whether G has a vertex cover of size k. Let m denote the number of edges in the graph. First number the edges of the graph from 1 through m. Then represent each vertex v_i as an m-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 (see Fig. 113(a) and (b)).

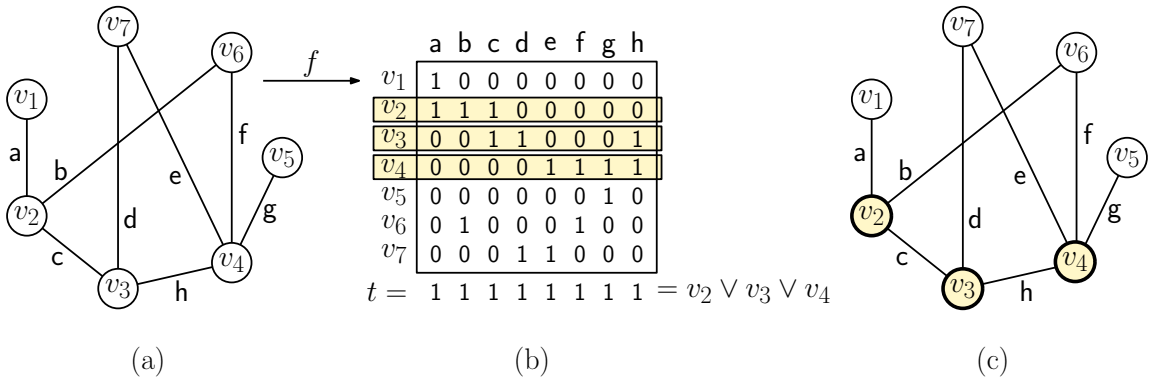


Fig. 113: First attempt at VC to SS reduction.

Suppose that G has a vertex cover V'. If we take the logical-or of the corresponding bit vectors, since every edge is covered, it follows that the result will be the bit vector consisting of m consecutive 1's, where $m = |E|$. If any edge is not covered, that bit position will have a 0.

This is not quite what we want for two reasons. First, the subset sum problem involves addition, not logical-or. Second, we have no way of controlling how many vertices are used in the cover.

Let's address the first problem. There are two ways in which addition differs significantly from logical-or.

- The first issue is that addition can involve "carries". For example, the $1101 \vee 0011 = 1111$, but in binary $1101_2 + 0011_2 = 13 + 3 = 16 = 1000_2$.

  To fix this, we need to spread the bits out further, so the carries from each column cannot spill over into the next. We'll see that we can do this by representing the numbers in base 4. (Each

edge has only two endpoints, so we have at most two 1's in each column.) Of course, the final numbers will be encoded in whatever format our algorithm expects (e.g., binary or decimal).

- The second issue is that an edge may generally be covered either once or twice in the vertex cover. So, a valid vertex cover will be manifest as a sum whose digits are either 1 or 2, e.g. $1211\ldots21112_4$. This is problematic, since we need unique target value t.

  To fix this, we will create a set of m additional *slack values*. For $1 \le i \le m$, the ith slack value, denoted y_i, is a string of all 0's, except for a single 1-digit in the ith position. Since we are working in base 4, this means that $y_i = 4^i$. We will set our final target value to a string of 2's, that is, $t = 2222\ldots222_4$.

  Why does this work? For any edge $e_a = (v_i, v_j)$, if both vertices are in the vertex cover, we know that they will both contribute a 1 in column k, for a total of 2, and we do not need to include the slack value (see Fig. 109(a)). On the other hand, if only one of them is in the cover, then the kth digit will only be 1, and so we will include the slack value y_k to bring the total in column k up to 2, as desired (see Fig. 114(b)).

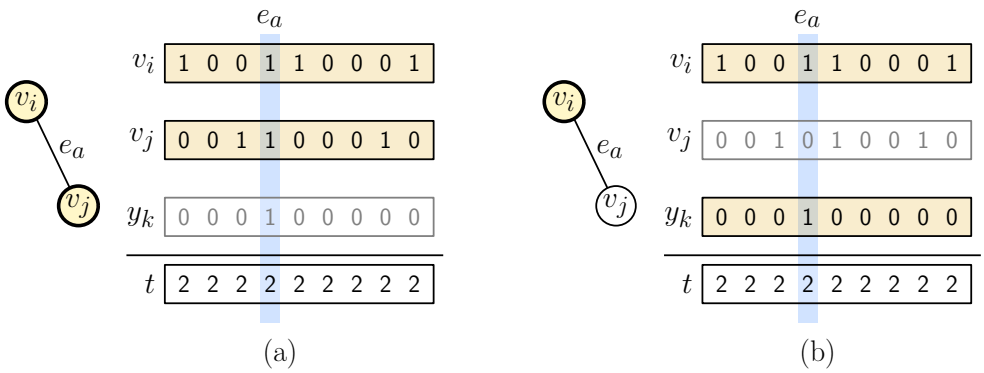


Fig. 114: Using slack values to normalize column sums to 2.

The final issue is how to control the number of vertices in the vertex cover to be k. We will handle this by adding an additional column. For each value arising from a vertex, this column is set to 1. And for each slack variable the value is set to 0. In the target, we will require that the entries in this column sum to the value k (encoded in base-4), the desired size of the vertex cover. Thus, to form the desired sum, we must select exactly k of the vertex values. Note that since we only have a base-4 representation, there might be carries out of this last column (if $k \ge 4$). But since this is the last column, it will not affect any of the other aspects of the construction.

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

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

(2) Create m slack values $y_1, y_2, \ldots, y_m$, where $y_k = 4^k$, that is, a single 1-digit in position k and 0's otherwise.

(3) Let t be the base-4 number whose first digit is k (this may actually span multiple base-4 digits), and whose remaining m digits are all 2.

(4) Convert the x_i's, the y_j's, and t into whatever base numeric base is used for the subset sum problem (e.g. base 10). Output the set $S = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ and t.

Observe that this can be done in polynomial time, in $O(m^2)$, in fact, where $m = |E|$. The construction is illustrated in Fig. 115(a) for the graph shown in Fig. 113(a).

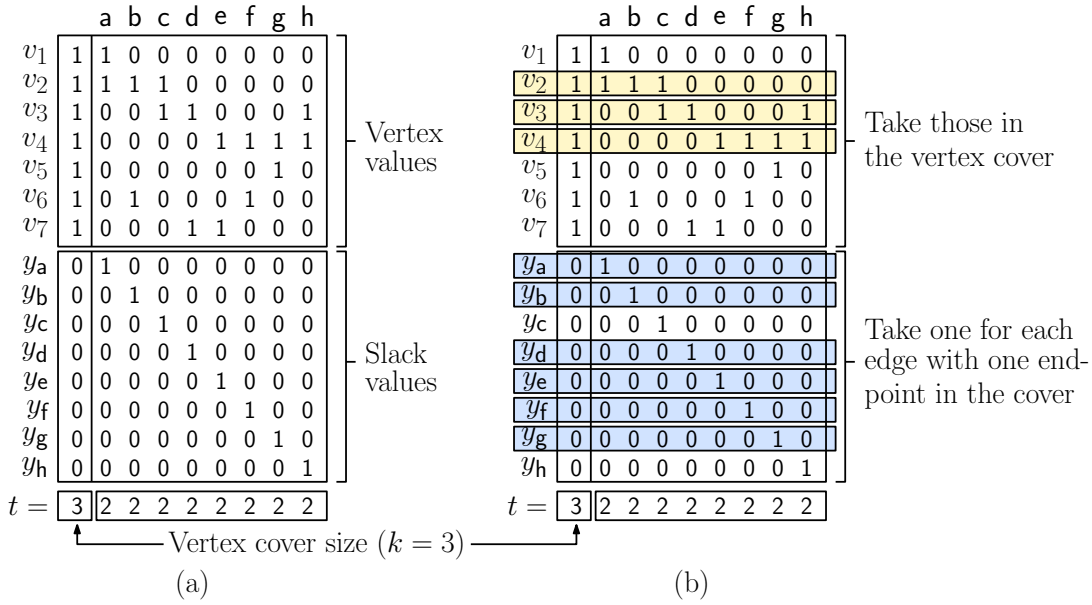


Fig. 115: (a) VC to SS reduction and (b) correctness for the vertex cover $V' = \{v_2, v_3, v_4\}$.

Correctness: Correctness is established in the following lemma.

Lemma: G has a vertex cover of size k if and only if S has a subset that sums to t.

Proof: ($\Rightarrow$) Suppose that G has a vertex cover $V' = \{v_1, \ldots, v_k\}$. We construct a solution to subset sum as follows. First, we take the vertex value x_i for each $v_i \in V'$(see Fig. 115(b)). For each edge e_a that is covered only once in V', we take the corresponding slack variable y_a . It follows from the comments made earlier that the lower-order m 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.

($\Leftarrow$) Conversely, if S has a subset S' that sums to t then we assert that it must select exactly k vertex values x_i, 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 $4^m \geq 4^n$ (where n is the number of vertices in G). In our dynamic programming solution $W = t$, so the DP algorithm would run in $\Omega(n \cdot 4^n)$ time, which is not polynomial time.

Lecture 20: 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)).

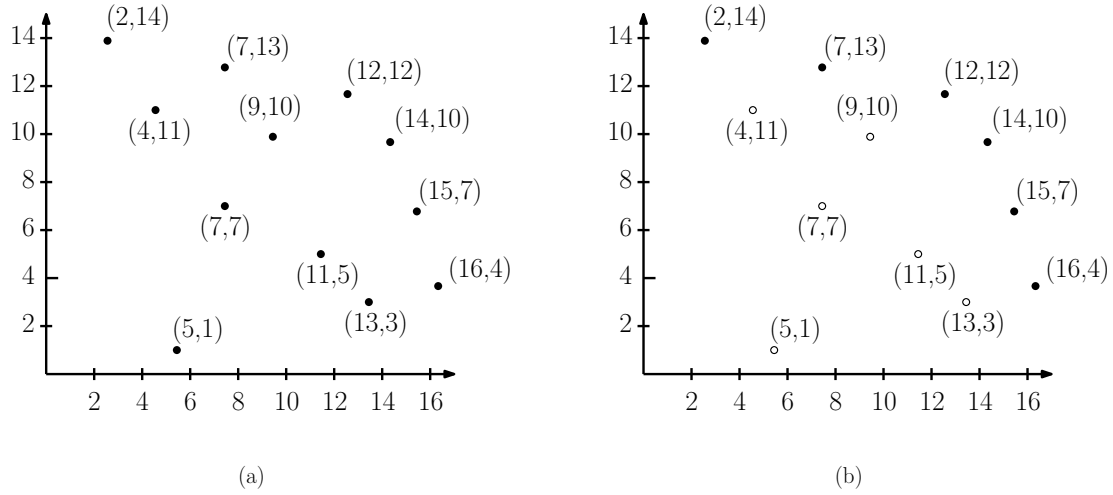


Fig. 116: Max-Dominance.

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.

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:

$$\frac{n^2}{n \lg n} = \frac{n}{\lg n}.$$

```
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
        }
    }
}
```

(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

```
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
}
```

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.

The general process is illustrated in Fig. .

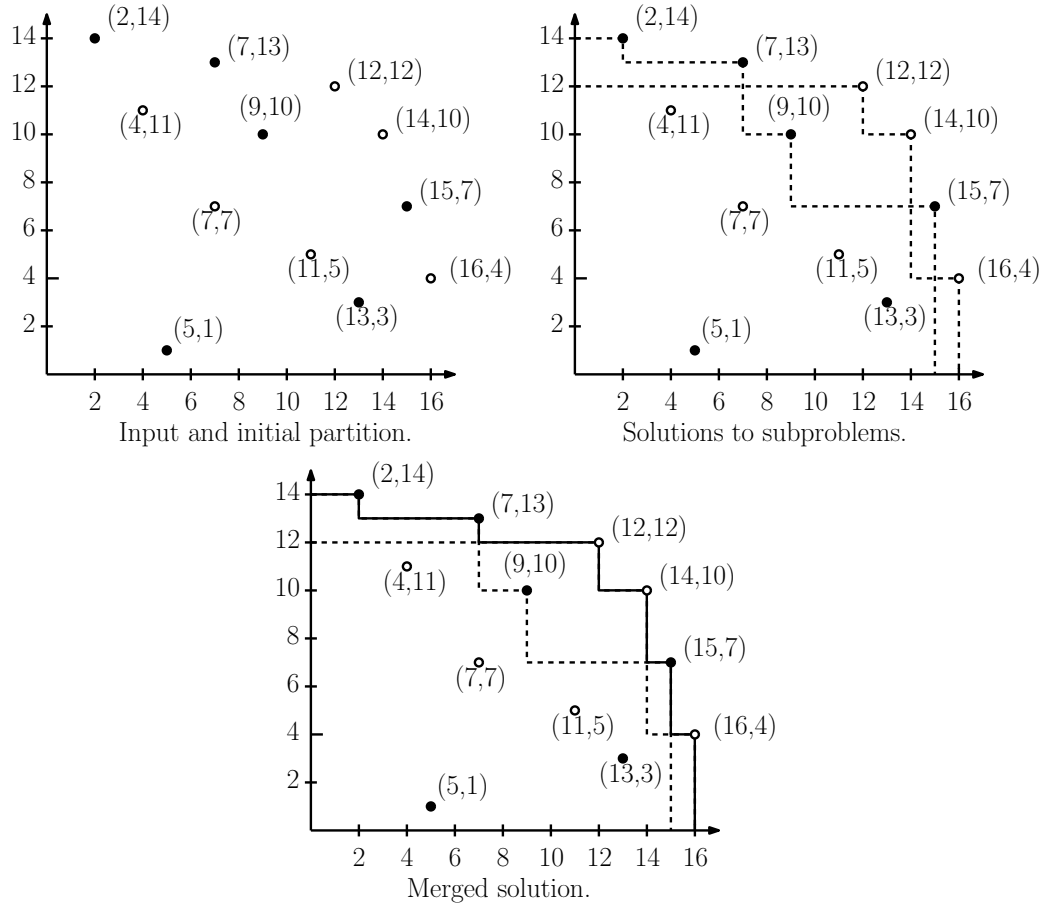


Fig. 117: Divide and conquer approach.

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

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

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

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

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

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

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

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

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

defined for $n \geq 0$.

Case (1): $a > b^k$ then $T(n)$ is $\Theta(n^{\log_b a})$.
Case (2): $a = b^k$ then $T(n)$ is $\Theta(n^k \log n)$.
Case (3): $a < b^k$ then $T(n)$ is $\Theta(n^k)$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

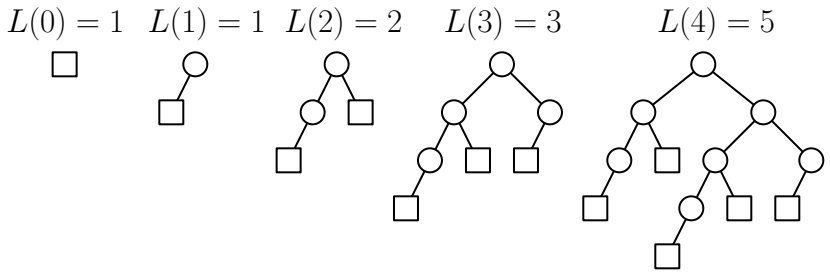


Fig. 118: Minimum-sized AVL trees.

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

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

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

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

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

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

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

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

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

By the quadratic formula we have

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

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

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

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

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

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

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

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

Lecture 21: 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:

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

Now, let's try the following manipulation. Compute $G(z) - zG(z) - z^2 G(z)$, 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:

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

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 22: 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*.[17] 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 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.

[17] 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".

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] ↔ A [G, F]	E [B, A] ↔ C [ ]	E [B, A] ↔ C [ ]
F [B] ↔ C [E]	F [ ] ↔ B [G, E]	F [B, C] ↔ A [G]
G [C] ↔ B [ ]	G [C, B] ↔ A [ ]	G [C] ↔ B [ ]

The answer appears in the footnote below[18] 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.)

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. 119.

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.

[18]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.

```
// 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
        }
    }
}
```

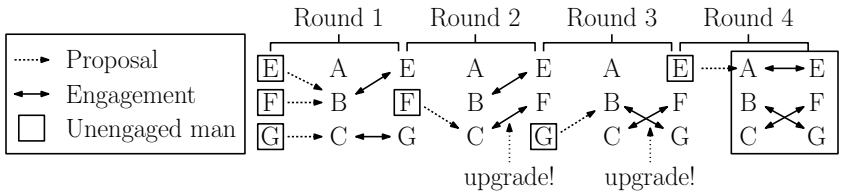


Fig. 119: Example of the round form version of the GS Algorithm on the preference lists given earlier. The final matching is (Eddy ↔ Anny), (Freddy ↔ Betty), (Gerry ↔ Carry).

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. 120),

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

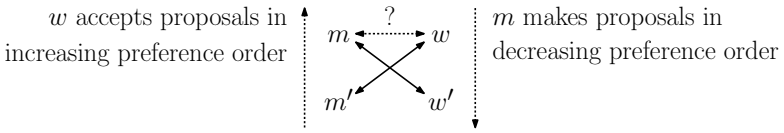


Fig. 120: The proof of Lemma 5.

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 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 23: 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 $\rightarrow$ 001 and b $\rightarrow$ 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. 121.

Character	a	b	c	d
Codeword	0	110	10	111

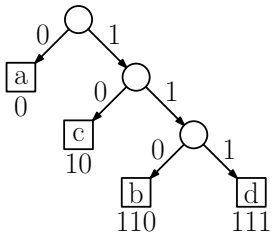


Fig. 121: A tree-representation of a prefix code.

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 single character is given in the following formula:

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

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, $B(T)$.

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 *meta-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 meta-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. 122.

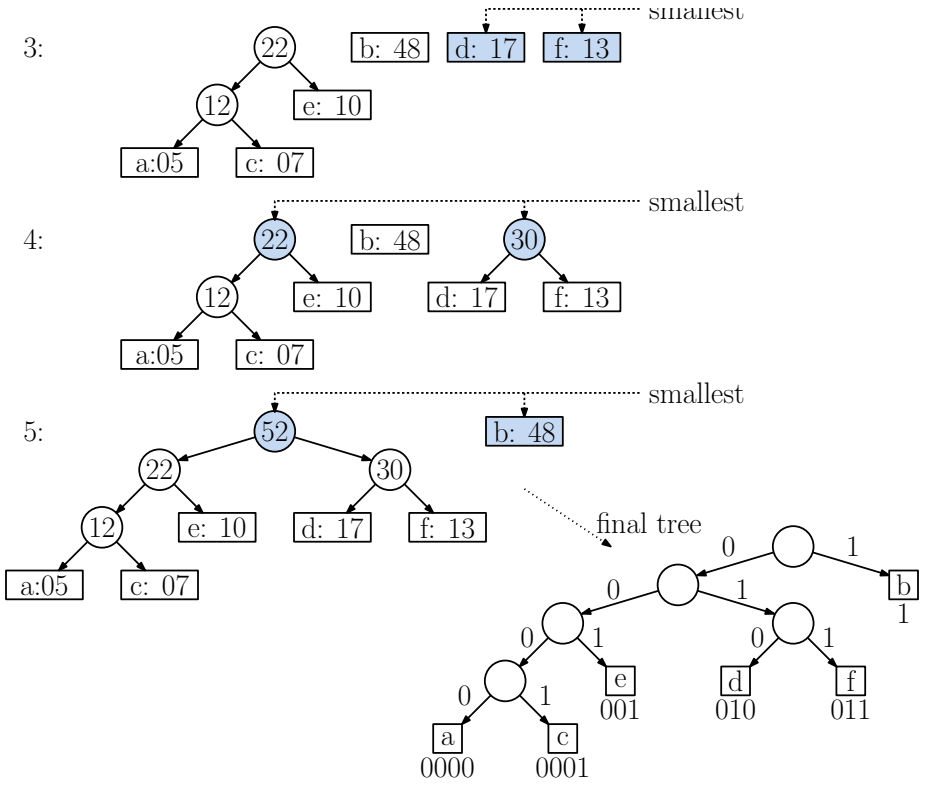


Fig. 122: Huffman's Algorithm.

The pseudocode for Huffman's algorithm is given below. Let C denote the set of characters, and let $n = |C|$. 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.

Correctness: The big question that remains is why is this algorithm correct, that is, does it compute the tree that minimizes the expected encoding length? 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

```
huffman(C, prob) {                          // C = chars, prob = probabilities
    for each (x in C) {
        add x to Q sorted by prob[x]        // add all to priority queue
    }
    for (i = 1 to |C| - 1) {                // repeat until only 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
}
```

constructed by Huffman's algorithm can be converted into one that is equal to Huffman's tree without increasing its cost. This is done by identifying an appropriate place where the two solutions differ, modify the non-greedy solution so that it is a bit closer to the greedy solution, and showing that this modification can be done so that the cost does not increase. By repeating this, we will eventually modify any solution into the greedy solution in a manner that does not increase this cost. This implies that the greedy solution is has the minimum cost.

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 (proved below) 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 meta-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 1: 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. (There may be many such siblings, and if so pick any such pair.) If $\{x, y\} = \{b, c\}$ we are done. Otherwise, from the fact that x and y have the lowest probabilities, we may label the nodes such that $p(b) \leq p(c)$ and $p(x) \leq p(y)$.

Now, since x and y have the two smallest probabilities it follows that $p(x) \leq p(b)$ and $p(y) \leq p(c)$. (In both cases they may be equal.) Because b and c are at the deepest level of the tree we know that $d_T(b) \geq d_T(x)$ and $d_T(c) \geq d_T(y)$. (Again, they may be equal.) Thus, we have $p(b) - p(x) \geq 0$ and $d_T(b) - d_T(x) \geq 0$, and hence their product is nonnegative. Now, suppose that we switch the positions of x and b in the tree, resulting in a new tree T' (see Fig. 123).

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 in both trees. The only exceptions are nodes x and b. By subtracting the old contributions of these nodes and adding in the new contributions we have

$$\begin{aligned} B(T') &= B(T) - (\text{old cost for } b \text{ and } x) + (\text{new cost for } b \text{ and } x) \\ &= B(T) - (p(x)d_T(x) + p(b)d_T(b)) + (p(x)d_T(b) + p(b)d_T(x)). \end{aligned}$$

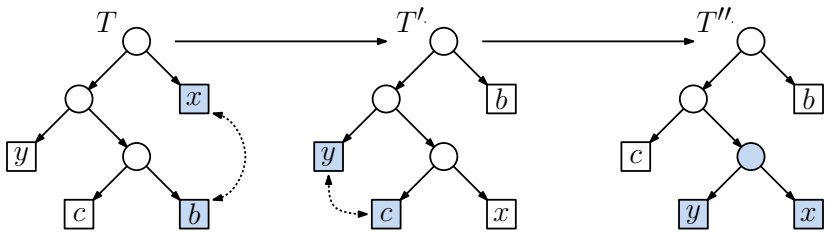


Fig. 123: Showing that the lowest probability nodes are siblings at the tree's lowest level.

With a little algebraic manipulation we obtain

$$\begin{aligned} B(T') &= B(T) + p(x)(d_T(b) - d_T(x)) - p(b)(d_T(b) - d_T(x)) \\ &= B(T) - (p(b) - p(x))(d_T(b) - d_T(x)) \\ &\leq B(T), \end{aligned}$$

where the last step follows because $(p(b) - p(x))(d_T(b) - d_T(x)) \geq 0$. 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.

The above claim applies to just one pair of nodes, those with the lowest probabilities. To show that the *entire* Huffman tree is optimal, we need to extend this argument. We will do this by induction. In order to reduce from n characters to $n-1$, we will do the same reduction that Huffman's algorithm does; namely we will *merge* characters x and y into a new meta-character z, whose probability is the sum of the probabilities of x and y.

Claim 2: Let T_n be any prefix-code tree that satisfies the property of Claim 1 (lowest probability symbols x and y are siblings at the deepest level). Let T_{n-1} be the tree that results by replacing these two nodes and their parent with a single leaf node z of probability $p(z) = p(x) + p(y)$. Then $B(T_n) = B(T_{n-1}) + p(z)$.

Proof: Let d denote the depths of x and y in T_n. Clearly, z is at depth $d-1$ in T_{n-1} (see Fig. 124).

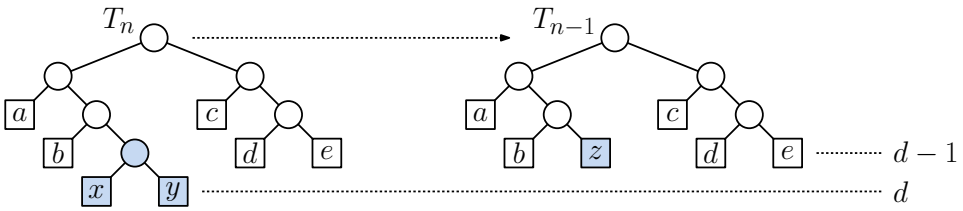


Fig. 124: Proving the correctness of Huffman's algorithm.

Because z replaces x and y the costs of the two trees satisfies

$$\begin{aligned} B(T_n) &= B(T_{n-1}) - (z\text{'s cost in } B(T_{n-1})) + (x \text{ and } y\text{'s costs in } B(T_n)) \\ &= B(T_{n-1}) - p(z)(d-1) + (p(x)\,d + p(y)\,d) \\ &= B(T_{n-1}) - p(z)\,(d-1) + p(z)\,d \\ &= B(T_{n-1}) + p(z). \end{aligned}$$

Note that the cost of trees T_n and T_{n-1} differ only by the fixed term $p(z)$, which does not depend on the tree's structure. Therefore (subject to this replacement), minimizing the cost of T_n is equivalent to minimizing the cost of T_{n-1}. This allows us to prove our main result.

Claim 3: Huffman's algorithm produces an optimal prefix code tree.

Proof: The proof is by induction on n, the number of characters. The basis case ($n = 1$) is trivial, since there is only one tree possible. If $n \geq 2$, then by Claim 1, we know that the two characters x and y of lowest probability are siblings at the deepest level of an optimal tree. Huffman's algorithm replaces these nodes by a character z whose probability is the sum of their probabilities. By induction, Huffman's algorithm computes the optimum tree over the resulting alphabet of $n - 1$ symbols. Call it T_{n-1}. Replacing z with nodes x and y results in a tree T_n whose cost is higher by the fixed amount $p(z) = p(x) + p(y)$. Since T_{n-1} is optimal, and the cost of replacement does not depend on the tree's structure, T_n is also optimal.

Lecture 24: Medians and Selection

Selection: We have discussed recurrences and the divide-and-conquer method of solving problems. Today we will give a rather surprising (and very tricky) algorithm which shows the power of these techniques.

The problem that we will consider is very easy to state, but surprisingly difficult to solve optimally. Suppose that you are given a set of n numbers. Define the *rank* of an element to be one plus the number of elements that are smaller than this element. Since duplicate elements make our life more complex (by creating multiple elements of the same rank), we will make the simplifying assumption that all the elements are distinct for now. It will be easy to get around this assumption later. Thus, the rank of an element is its final position if the set is sorted. The minimum is of rank 1 and the maximum is of rank n.

Of particular interest in statistics is the *median*. If n is odd then the median is defined to be the element of rank $(n+1)/2$. When n is even there are two natural choices, namely the elements of ranks $n/2$ and $(n/2) + 1$. In statistics it is common to return the average of these two elements. We will define the median to be either of these elements.

Medians are useful as measures of the *central tendency* of a set, especially when the distribution of values is highly skewed. For example, the median income in a community is likely to be more meaningful measure of the central tendency than the average is, since if Bill Gates lives in your community then his gigantic income may significantly bias the average, whereas it cannot have a significant influence on the median. They are also useful, since in divide-and-conquer applications, it is often desirable to partition a set about its median value, into two sets of roughly equal size. Today we will focus on the following generalization, called the *selection problem*.

Selection: Given a set A of n distinct numbers and an integer k, $1 \leq k \leq n$, output the element of A of rank k.

The selection problem can easily be solved in $\Theta(n \log n)$ time, simply by sorting the numbers of A, and then returning $A[k]$. The question is whether it is possible to do better. In particular, is it possible to solve this problem in $\Theta(n)$ time? We will see that the answer is yes, and the solution is far from obvious.

The Sieve Technique: The reason for introducing this algorithm is that it illustrates a very important special case of divide-and-conquer, which I call the *sieve technique*. We think of divide-and-conquer as breaking the problem into a small number of smaller subproblems, which are then solved recursively. The sieve technique is a special case, where the number of subproblems is just 1.

The sieve technique works in phases as follows. It applies to problems where we are interested in finding a single item from a larger set of n items. We do not know which item is of interest, however after doing some amount of analysis of the data, taking say $\Theta(n^k)$ time, for some constant k, we find that we do not know what the desired item is, but we can identify a large enough number of elements that *cannot* be the desired value, and can be eliminated from further consideration. In particular "large enough" means that the number of items is at least some fixed constant fraction of n (e.g. $n/2$, $n/3$, $0.0001n$). Then we solve the problem recursively on whatever items remain. Each of the resulting recursive solutions then do the same thing, eliminating a constant fraction of the remaining set.

Applying the Sieve to Selection: To see more concretely how the sieve technique works, let us apply it to the selection problem. Recall that we are given an array $A[1..n]$ and an integer k, and want to find the k-th smallest element of A. Since the algorithm will be applied inductively, we will assume that we are given a subarray $A[p..r]$ as we did in MergeSort, and we want to find the kth smallest item (where $k \leq r - p + 1$). The initial call will be to the entire array $A[1..n]$.

There are two principal algorithms for solving the selection problem, but they differ only in one step, which involves judiciously choosing an item from the array, called the *pivot element*, which we will denote by x. Later we will see how to choose x, but for now just think of it as a random element of A. We then partition A into three parts. $A[q]$ contains the element x, subarray $A[p..q-1]$ will contain all the elements that are less than x, and $A[q+1..r]$, will contain all the element that are greater than x. (Recall that we assumed that all the elements are distinct.) Within each subarray, the items may appear in any order (see Fig. 125).

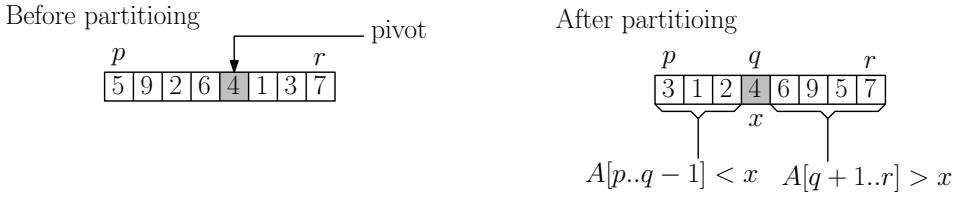


Fig. 125: Partitioning

It is easy to see that the rank of the pivot x is $q-p+1$ in $A[p..r]$. Let xRank $= q-p+1$. If $k =$ xRank, then the pivot is the kth smallest, and we may just return it. If $k <$ xRank, then we know that we need to recursively search in $A[p..q-1]$ and if $k >$ xRank then we need to recursively search $A[q+1..r]$. In this latter case we have eliminated q smaller elements, so we want to find the element of rank $k-q$. Here is the complete pseudocode (see Fig. 126).

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
    }
}
```

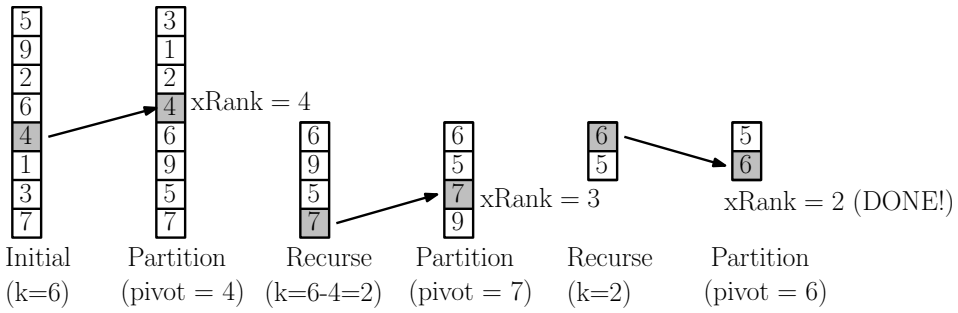


Fig. 126: Selection Algorithm.

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 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

$$T(n) \;=\; n + \frac{n}{2} + \frac{n}{4} + \cdots \;\le\; \sum_{i=0}^{\infty} \frac{n}{2^i} \;=\; n \sum_{i=0}^{\infty} \frac{1}{2^i}.$$

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

$$T(n) \;\le\; 2n \in O(n).$$

(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.

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

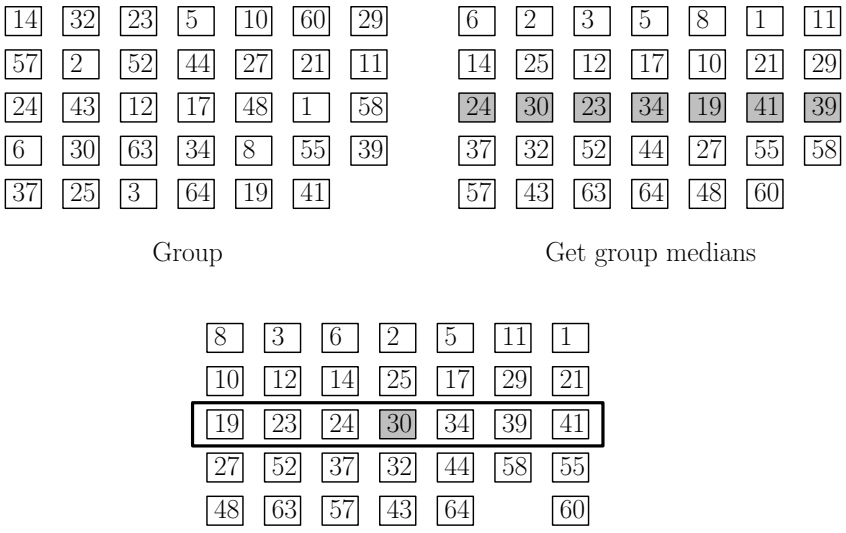


Fig. 127: Choosing the Pivot. (30 is the final pivot.)

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 25: 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.

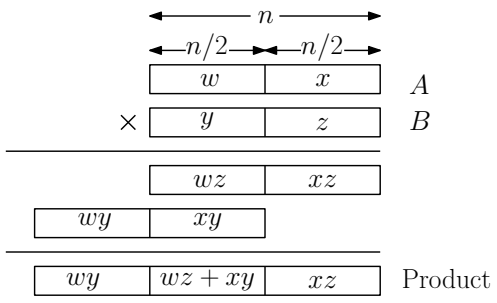


Fig. 128: Long integer multiplication.

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

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

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 26: 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. 129.)

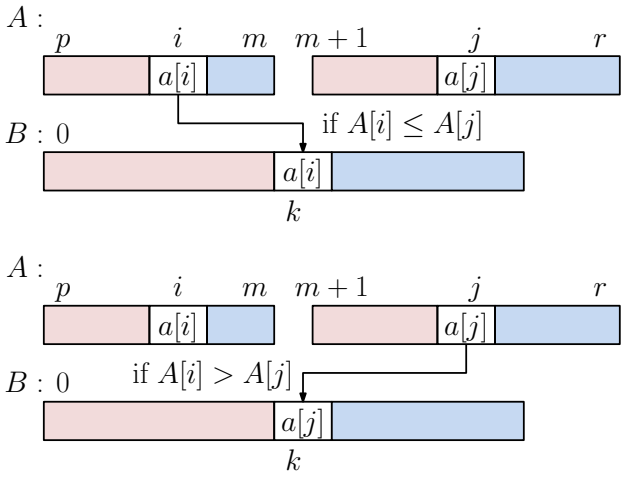


Fig. 129: Merging two sorted lists.

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

MergeSort

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

merge(A, p, m, r) {                         // 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
}
```

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

Fig. 130 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.

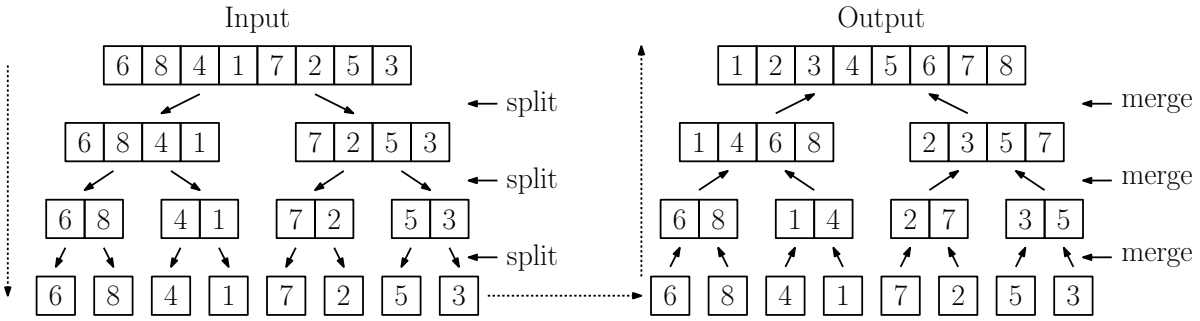


Fig. 130: MergeSort example.

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 $k = \lg n$.[19] 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 one possible measure of similarity between two lists of n numbers. (An example is shown in in Fig. 131.)

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

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

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

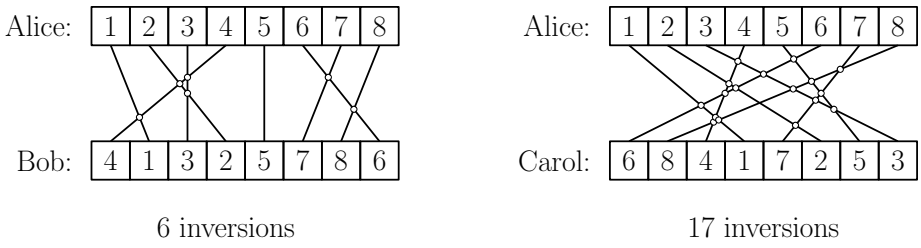


Fig. 131: Movie preferences and inversions.

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. 132).

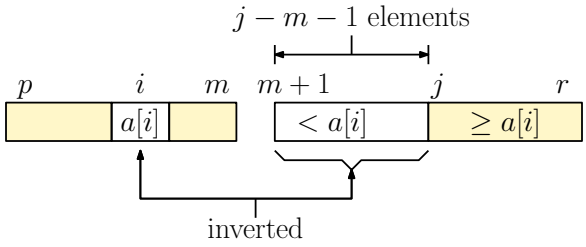


Fig. 132: Counting inversions when $A[i] \leq A[j]$.

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

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

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

Inversion Counting

```
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
}
```

This approach is illustrated in Fig. 133. Observe that inversions are counted in the merging process (shown as small white circles in the figure).

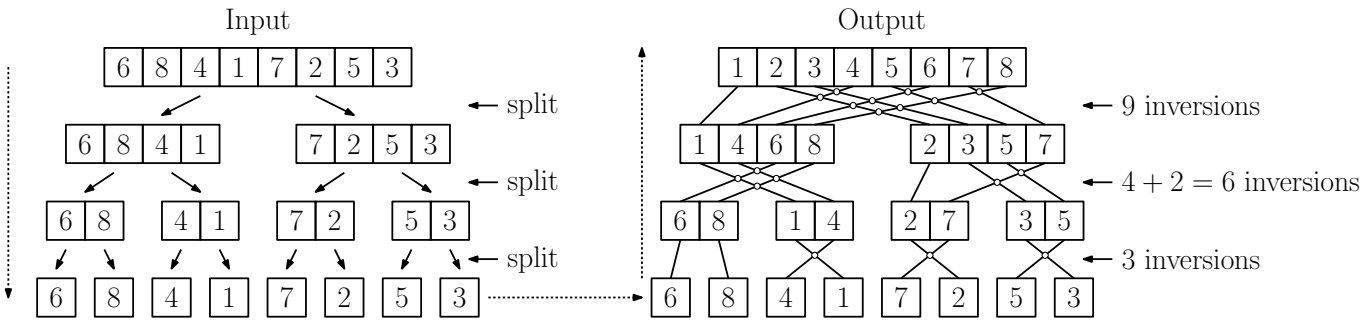


Fig. 133: Inversion counting by divide and conquer.

Lecture 27: 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. 134(a)). This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall that, given two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$, their (Euclidean) distance is

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

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

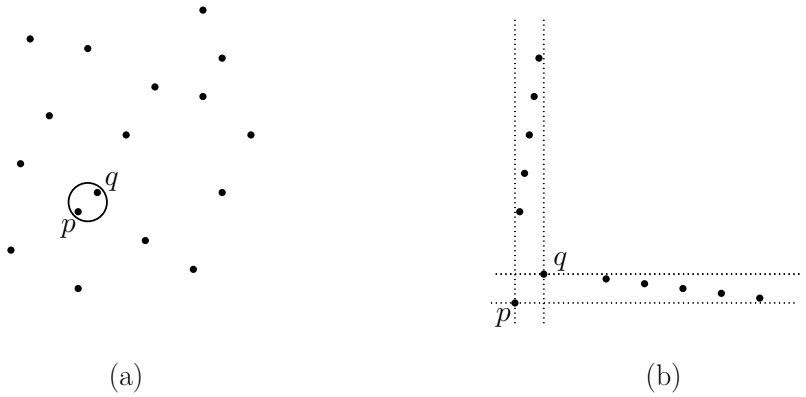


Fig. 134: (a) The closest pair problem and (b) why sorting on x- or y-alone doesn't work.

Before getting into the solution, it is worth pointing out a simple strategy that fails to work. If two points are very close together, then clearly both their x-coordinates and their y-coordinates are close together. So, how about if we *sort* the points based on their x-coordinates and, for each point of the set, we'll consider just *nearby* points in the list. It would seem that (subject to figuring out exactly what "nearby" means) such a strategy might be made to work. The problem is that it could fail miserably. In particular, consider the point set of Fig. 134(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.[20]

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.

[20] 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*.

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

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

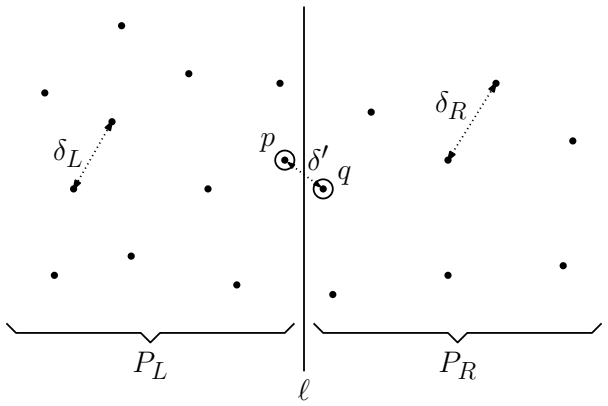


Fig. 135: Divide-and-conquer closest pair algorithm.

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. 135). 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. 135). 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. 136(a)).[21]

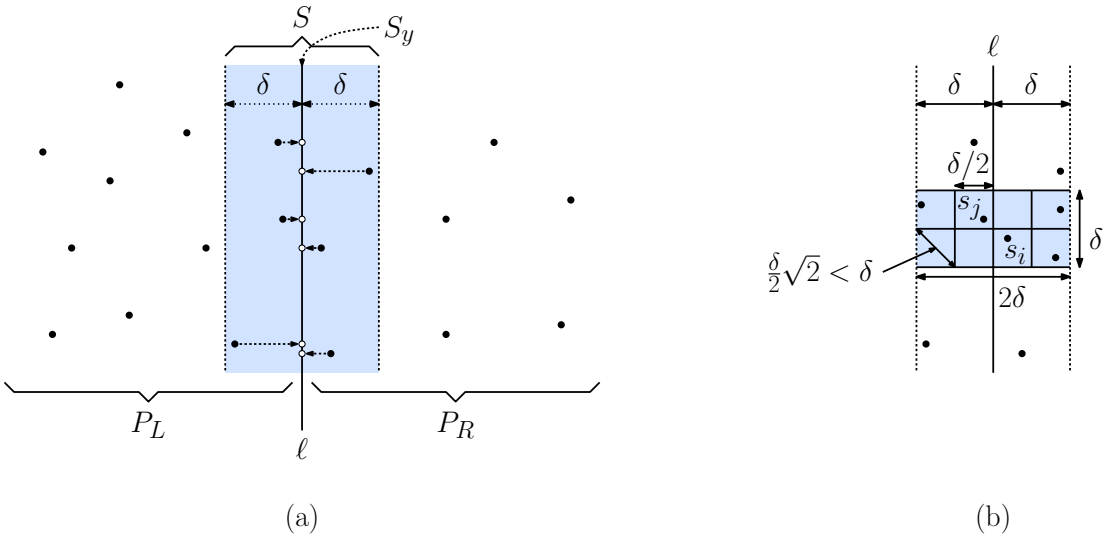


Fig. 136: Closest pair in the strip.

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. 136(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. 136(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 pair seen so far.

[21] 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.

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[22] 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.

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'
}
```

[22] 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.

Lecture 28: 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 numeric *values*, $\langle v_1, \ldots, v_n \rangle$, *weights*, $\langle w_1, \ldots, w_n \rangle$, and a *knapsack capacity* $W > 0$. The *0-1 knapsack problem* is to determine the subset $T \subseteq \{1, \ldots, n\}$ (of items to "take") that maximizes the *total value*,

$$V(T) \;=\; \sum_{i \in T} v_i,$$

subject to the *capacity constraint*:

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

The "0-1" refers to the fact that items may be taken ("1") or not taken ("0"), but you cannot take a fraction of any object. The 0-1 assumption is reasonable if you are stealing say paintings, since who wants half of the Mona Lisa? However, if you were stealing oil, or wheat (or generally, any *fungible commodity*), you could do better by taking fraction of some item, if that is all that fits in your knapsack.

It turns out that this problem is NP-complete, and so we cannot really hope to find an efficient solution. However, if we assume that W is not terribly large, we can devise a solution whose running time grows linearly as a function of W. (For a solution to be considered properly running in polynomial time, the running time should be a polynomial in the input size n alone, and not depend on the magnitude of the inputs, such as W.)

DP Solution: We will present a DP algorithm that solves the 0-1 problem in $O(nW)$ time. Here is our approach. For $0 \leq i \leq n$ and $0 \leq c \leq W$, define $V(i, c)$ to be the maximum total value of any subset of items $\{1, \ldots, i\}$ that can fit into a knapsack of capacity c. Clearly, our ultimate interest is the function value $V(n, W)$.

To compute the entries of the array V we will imply an inductive approach.

Basis: If there are no items to take ($i = 0$) or our knapsack has no capacity ($c = 0$), then clearly we cannot take anything, and hence $V(i, c) = 0$ if either $i = 0$ or $c = 0$.

Otherwise, let us assume that both i and c are strictly positive. We have two options:

Leave object i: If we choose to not take object i, then we obtain no additional value and our knapsack capacity is unchanged. We should do the best we can with the remaining $i - 1$ items. Thus, $V(i, c) = V(i - 1, c)$.

Take object i: First, observe that this is only an option if we have sufficient knapsack capacity, that is, $w_i \leq c$. We gain a value of v_i, but we have used up w_i units of capacity. With the remaining $c - w_i$ units of capacity, we can fill it in the best possible way with the remaining objects $\{1, \ldots, i - 1\}$. Thus, we have $V(i, c) = v_i + V(i - 1, c - w_i)$.

Note that the *principle of optimality* holds here. Whenever we solve a subproblem, we should do so in the optimum manner. Combining these observation, we obtain the following recursive *DP-formulation*

for the 0-1 knapsack problem:

$$V(i,c) \;=\; \begin{cases} 0 & \text{if } i=0 \text{ or } c=0, \\ \max \left\{ \begin{array}{ll} V(i-1,c) & \text{always} \\ v_i + V(i-1,c-w_i) & \text{if } w_i \le c \end{array} \right\} & \text{if } \min(i,j) > 0. \end{cases}$$

Memoized Implementation: This is very easy to implement through memoization. We define an 2-dimensional array $V[0..n, 0..W]$. We initialize all its entries to -1, indicating that the entry is undefined. As usual, we will store a boolean "hook" $H[i,c]$ in a parallel array, which will be set to either `true` (meaning "take") or `false` (meaning "leave"). The algorithm is presented in the code block below. The initial call is `memo-knapsack`(n, W).

0-1 Knapsack Problem

```
memo-knapsack(i, c) {                                  // memoized 0-1 knapsack
    if ( V[i, c] == -1) {                              // entry is undefined?
        if ( i == 0 || c == 0 ) {                      // basis case?
            V[i, c] = 0
        } else {
            leaveValue = memo-Knapsack(i-1, c)         // value if we leave i
            if ( w[i] <= c ) {                         // have capacity to take i?
                takeValue = v[i] + memo-Knapsack(i-1, c - w[i]) // value if we take i
            } else {                                   // no room for i
                takeValue = -1
            }
            if (takeValue > leaveValue) {              // better to take i?
                V[i, c] = takeValue; H[i, c] = true
            } else {                                   // better to leave i?
                V[i, c] = leaveValue; H[i, c] = false
            }
        }
    }
    return V[i, c]                                     // return the final value
}
```

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 $4+3 = 7 \le 10$.

Values of the objects are $\langle 10, 40, 30, 50 \rangle$.
Weights of the objects are $\langle 5, 4, 6, 3 \rangle$.

		Capacity $\rightarrow$	$c=0$	1	2	3	4	5	6	7	8	9	10
Item: i	Value: v_i	Weight: w_i	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

Fig. 137: 0–1 Knapsack for $n = 4$ items and knapsack capacity $W = 10$. Final result is $V[4,10] = 90$ (for taking items 2 and 4).

The algorithm's correctness was established in the derivation of the DP-formulation. The running time is $O(nW)$. The table size is $O((n+1)(W+1)) = O(nW)$, and it takes us $O(1)$ time to compute each table entry.

Of course, we have only computed the maximum value achievable. We should still determine which items should we taken and which should be left to achieve the maximum. We will leave this as an easy exercise. (It may help to recall the weighted interval scheduling problem.)

Summary: We have presented a simple DP algorithm for the 0-1 knapsack problem. In a later lecture, we will show that this problem is NP-hard. Remember that our solution is not considered polynomial time, because the running time depends on W, which is not part of the input size.

You might point out that the knapsack size is part of the problem description, but note that the value of W depends on the *resolution* with which we measure weights. For example, if weights are measured in pounds, then the weights of all items need to be rounded up to the next higher integer pound. This introduces inaccuracies, particularly if we are stealing very light objects, like diamonds. If we were to be more precise, allowing, say, weight measurements in micro-pounds, we would have a much more accurate algorithm. But the value of W (expressed as an integer number of micro-pounds) would increase by a factor of 1,000,000. Hence, the running time and space requirements would increase by this same factor.

Lecture 29: 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.

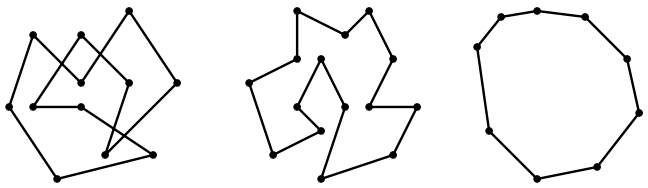


Fig. 138: Polygons.

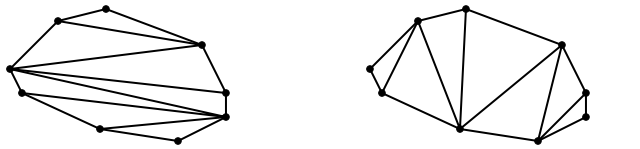
Given a convex polygon, we assume that its vertices are labeled in counterclockwise order $P = \langle v_1, \ldots, v_n \rangle$. We will assume that indexing of vertices is done modulo n, so $v_0 = v_n$. This polygon has n sides, $\overline{v_{i-1}v_i}$.

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. 139.)



A triangulation Lower weight triangulation

Fig. 139: 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 $0 \le i < j \le n$, 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:

$$t[i, j] = \begin{cases} 0 & \text{if } j = i+1 \\ \min_{i<k<j}(t[i, k] + t[k, j] + w(v_i v_k v_j)) & \text{if } j > i+1. \end{cases}$$

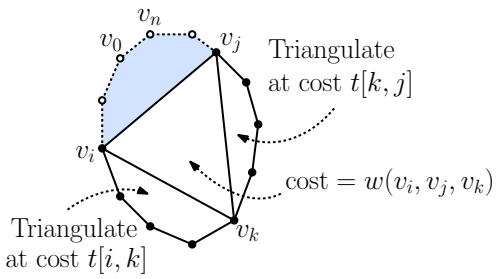


Fig. 140: Triangulations and tree structure.

The final output is the overall minimum weight, which is, $t[0, n]$. This is illustrated in Fig. 140

Note that this has almost exactly the same structure as the recursive definition used in the chain matrix multiplication algorithm (except that some indices are different by 1.) The same $\Theta(n^3)$ algorithm can be applied with only minor changes.

Relationship to Binary Trees: One explanation behind the similarity of triangulations and the chain matrix multiplication algorithm is to observe that both are fundamentally related to binary trees. In the case of the chain matrix multiplication, the associated binary tree is the evaluation tree for the multiplication, where the leaves of the tree correspond to the matrices, and each node of the tree is associated with a product of a sequence of two or more matrices. To see that there is a similar correspondence here, consider an $(n+1)$-sided convex polygon $P = \langle v_0, v_1, \ldots, v_n \rangle$, and fix one side of the polygon (say $\overline{v_0 v_n}$). Now consider a rooted binary tree whose root node is the triangle containing side $\overline{v_0 v_n}$, whose internal nodes are the nodes of the dual tree, and whose leaves correspond to the remaining sides of the tree. Observe that partitioning the polygon into triangles is equivalent to a binary tree with n leaves, and vice versa. This is illustrated in Fig. 141. 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.

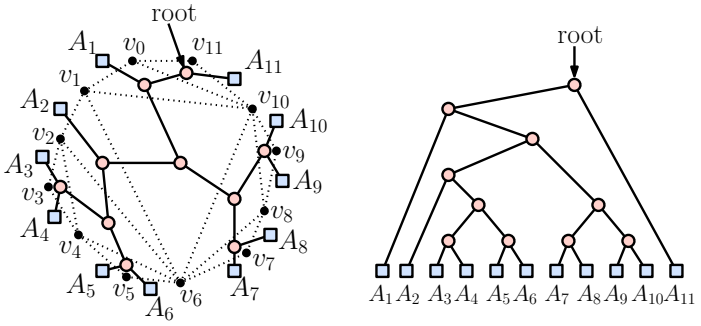


Fig. 141: Triangulations and tree structure.

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 30: Bridges and 2-Edge Connectivity

Higher-Order Graph Connectivity: (The following material applies *only* to undirected graphs!)

Let $G = (V, E)$ be an *connected* undirected graph. We often assume that our graphs are connected, but sometimes it is desirable to have a higher degree of connectivity. For example, if a graph can be disconnected through the removal of a single edge or vertex, the connectivity is rather "fragile." Here are some definitions:

Bridge: Any edge whose removal results in a disconnected graph (see Fig. 142(a)).

2-Edge Connected: A graph is *2-edge connected* if it contains no bridges (see Fig. 142(b)). In general a graph is *k-edge connected* if the removal of any $k-1$ edges results in a connected graph.

Here are also vertex-based equivalents:

Cut Vertex: Any vertex whose removal (together with the removal of any incident edges) results in a disconnected graph (see Fig. 142(a)).

Biconnected: A graph is *biconnected* if it contains no cut vertices (see Fig. 142(c)). In general a graph is *k-connected* if the removal of any $k-1$ vertices results in a connected graph.

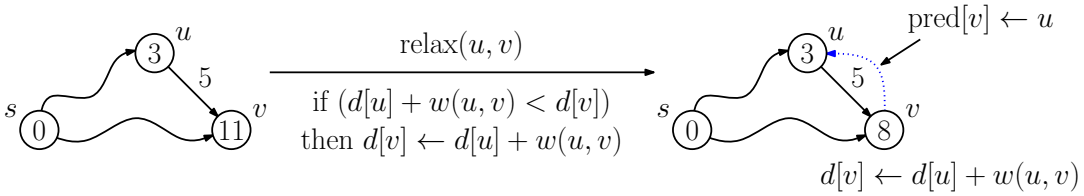


Fig. 142: Higher-order connectivity in graphs

We will present an $O(n+m)$-time algorithm for computing all the bridges of an undirected graph. (The algorithm can be modified to compute the cut vertices as well.)

Although we will not prove it, a useful fact about 2-edge connected graphs is that between every pair of distinct vertices, there are at least two edge-disjoint paths between them. (This is a consequence of a more general result called Menger's Theorem.) If a graph is biconnected, then for every pair of edges there is a simple cycle (that is a cycle that does not repeat any vertices) that contains both edges.

Finding Bridges through DFS: An obvious, but slow, way for computing bridges would be to delete each edge and then apply DFS to determine whether the resulting graph remains connected. However, this would take $O(m(n+m))$ time. We will see that it is possible to identify all the bridges 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.

Suppose that we are currently processing a vertex u in DFSvisit, and we see an edge (u, v) going to a neighbor v of u. If this edge is a back edge (that is, if v is an ancestor of u) then (u, v) cannot be a bridge, because the tree edges between u and v provide a second way to connect these vertices. Therefore, we may limit consideration to when (u, v) is a tree edge, that is, v has not yet been discovered, and so we will invoke DFSvisit(v). While we are doing this, we will keep track of the back edges in the subtree rooted at v. Observe that all these back edges remain entirely within this subtree (see Fig. 143(a)) then (u, v) is a bridge, since its removal completely disconnects this subtree from the rest of the tree. On the other hand, if there is even a single back edge leading out from this subtree, then (u, v) is *not* a

bridge. Such a back edge must go from within the subtree to a proper ancestor of v. By the Parenthesis Lemma, this means that it leads to a vertex whose discovery time is strictly smaller than v's discovery time. (Recall a vertex's ancestors are discovered before it.) In summary, we have established the following claim.

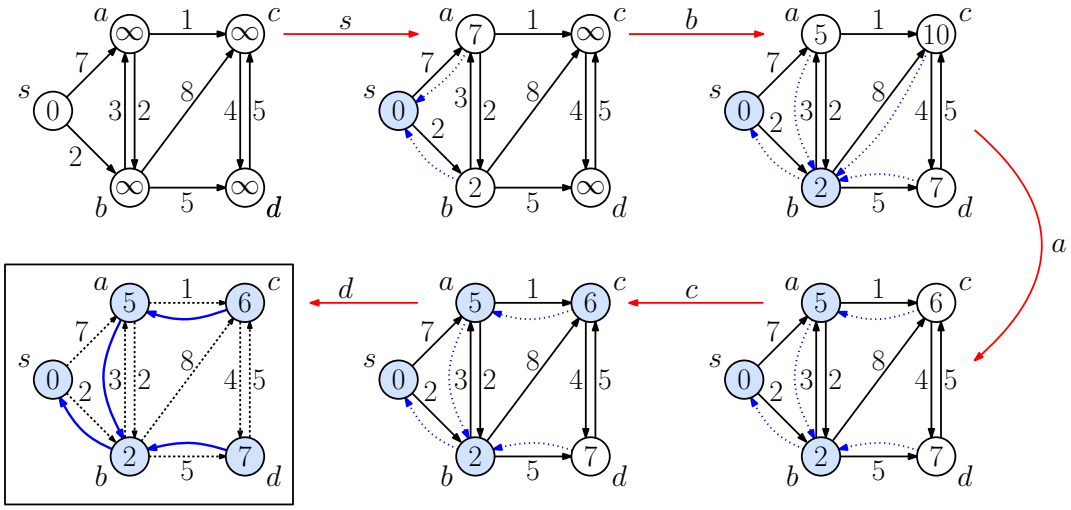


Fig. 143: Conditions for a vertex to be a cut vertex.

Claim: An edge (u, v) is a bridge if and only if it is a tree edge and (assuming that u is the parent of v) there is no back edge within v's subtree that leads to a vertex whose discovery time is strictly smaller than v's discovery time.

Tracking Back Edges: The above claim provides us with a structural characterization of bridges. How can we design an algorithm that tests this condition? To do this, we will introduce an auxiliary quantity, which will be computed as the DFS runs. We define

$$\text{Low}[u] \;=\; \min(d[u], \min\{d[w] : \exists \text{ back edge } (v, w) \text{ where } v \text{ is a descendant of } u\}.$$

Note that we use the term "descendant" in the nonstrict sense, that is, v may be u itself.

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(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 (see Fig. 144(a)).

Tree edge (u, v)**:** Low$[u]$ = min(Low$[u]$, Low$[v]$). Explanation: Since v is in the subtree rooted at u any single back edge leaving the tree rooted at v is a single back edge for the tree rooted at u (see Fig. 144(b)).

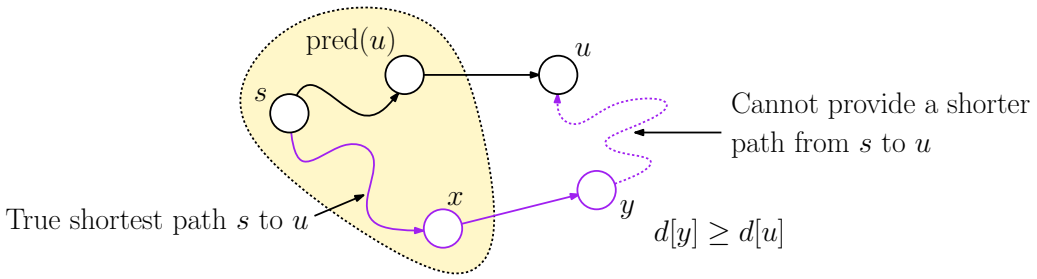


Fig. 144: Cut Vertices and the definition of Low[u].

The code block below shows how to compute Low[u] for all vertices. We do not both computing finish times, since they are not needed for our purposes. Note that there is a subtlety in determining whether an edge is a back edge. Clearly, such an edge must go to a previously discovered vertex (which is why it is in the else-clause), but we also need to check that this vertex is not u's parent. Recall that every edge of an undirected graph is reflected twice in the adjacency list (with v as a neighbor of u and u as a neighbor of v). We need to check that we are not simply seeing the tree edge again, but from the child back to the parent. To do this, we check v is *not* u's parent, that is $v \neq \text{pred}[u]$.

Modification of DFSvisit for Low computation

```
DFSvisit(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
            DFSvisit(v)
            Low[u] = min(Low[u], Low[v])        // update Low[u]
        }
        else if (v != pred[u]) {                // (u,v) is a back edge
            Low[u] = min(Low[u], d[v])          // update Low[u]
        }
    }
}
```

Observe that once Low[u] is computed for all vertices u, we can test whether a given tree edge $(\text{pred}[v], v)$ is a bridge by testing whether there is no back edge in v's subtree going to an earlier discovered vertex, that is, $d[v] = \text{Low}[v]$. (Actually, the test is more naturally stated as $d[v] \geq \text{Low}[v]$, but by definition $\text{Low}[v] \leq d[v]$, so testing for equality is equivalent.) The final code is shown in the code block below.

Wrapup: We have shown how to compute bridges in an undirected graph. There are a number of interesting problems that we still have not discussed. First, we claim that it is possible to adapt this algorithm to compute cut vertices as well. (The computation of Low is the same, but a different condition is applied to determine which vertices are cut vertices.) Second, if a graph fails to be 2-edge connected, it may be desirable to partition the vertices of the graph into 2-edge connected components. For example, in Fig. 145(c) the components consist of $\{d, g, h\}$, $\{b, c, f\}$, $\{a\}$, and $\{e, i, j\}$. This can be done by simple extension as well. (The vertices are stored on a stack, and whenever a bridge is detected, we pop off an appropriate subset of the stack. We will leave the details as an exercise.) A similar approach can be applied to computing the biconnected components of a graph, which is a partition of the edge set of the graph.

```
findAllBridges(G) {
    time = 0
    for each (u in V)                          // initialize
        mark[u] = undiscovered

    for each (u in V)
        if (mark[u] == undiscovered)           // undiscovered vertex?
            DFSVisit(u)                        // ...start a new search here
    for each (v in V) {                        // check for the bridges
        u = pred[v]
        if (u != null and d[v] == Low[v])
            output (u, v) as an bridge
    }
}
```

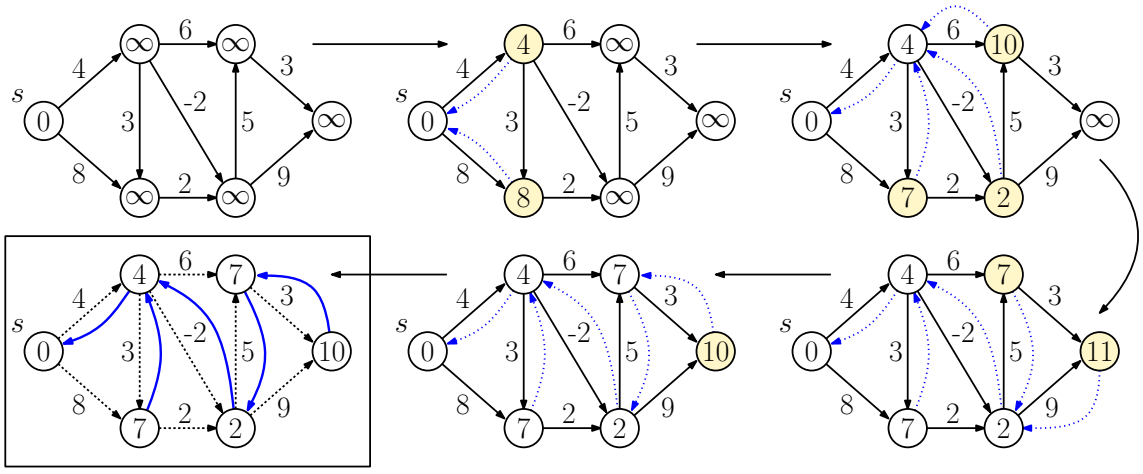


Fig. 145: Computing bridges via DFS.

Lecture 31: NP-Completeness: Hamiltonian Cycle

Hamiltonian Cycle: Today we consider the Hamiltonian cycle problem in directed graphs. Given a digraph $G = (V, E)$, the question is whether there exists a simple cycle that visits all the vertices. The fact that the cycle is simple, implies that every vertex is visited exactly once (except for the first and last vertex.) There are three natural variants: Hamiltonian cycle in undirected graphs, and Hamiltonian paths in both directed and undirected graphs. A Hamiltonian path is a simple path that visits every vertex (exactly once). All four problems are NP-complete. We will present a proof of the Hamiltonian cycle problem for directed graphs and leave the others as exercises. Formally, let us define

$$\text{DHC} \;=\; \{G : G \text{ is a directed graph with a Hamiltonian cycle}\}.$$

An important related problem is the traveling salesman problem (TSP). Given a complete graph (or digraph) with nonnegative edge weights, compute the cycle of minimum weight that visits all the vertices. Since the graph is complete, such a cycle will always exist. We can formulate as a decision problem as follows: given a complete weighted graph G, and integer w, does there exist a Hamiltonian cycle of total weight at most w? TSP is also NP-complete, which we will leave as an exercise.

Another related problem is called the *Eulerian circuit problem*. In this problem, we want to compute a cycle that visits every *edge* exactly once. This is related to a famous math problem, called the Seven Bridges of Königsburg. This problem can be solved in polynomial time by depth-first search. (Again, we'll leave this as an exercise.)

Component Design: Up to now, most of the reductions that we have seen (e.g., for clique, vertex cover, and dominating set) are of a relatively simple variety. They are sometimes called *local replacement reductions*, because they operate by making some local change throughout the graph.

The reduction for Hamiltonian cycle is more complicated, involving a technique called a *component design*. This method involves designing special subgraphs, sometimes called *components* or *gadgets* whose job it is to enforce a particular constraint of the problem. Very complex reductions may involve the creation of many gadgets. This one involves the construction of only one.

DHC Gadget: The main part of the proof is a reduction showing that 3SAT $\leq_P$ DHC. Our reduction will take a 3SAT formula F as input. Recall that such a formula is a conjunction (logical "and") of clauses, each of which is a disjunction (logical "or") of three literals, each of which is either a variable x_j or its complement $\overline{x}_i$.

Our reduction will map each clause to a special subgraph, called a *DHC gadget* (see Fig. 146). This subgraph has three incoming edges and three outgoing edges, one for each of the clause's literals. Let's call the incoming edges i_1, i_2 and i_3, and let's call the outgoing edges o_1, o_2 and o_3.

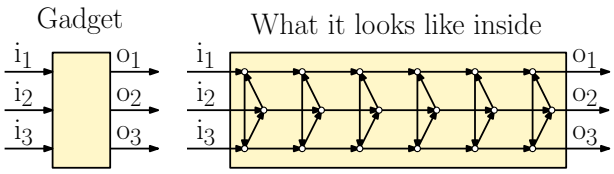


Fig. 146: The DHC gadget.

Our overall plan will be to string these gadgets together in chains, one per literal. We will connect the output from one gadget to the input of the next such that the literals agree (that is, connecting the x_j output of one gadget to the x_j input of the next and connect the $\overline{x}_j$ output of one gadget to the $\overline{x}_j$ input of the next). The strategy is that for each true literal in a clause, the Hamiltonian cycle will travel in along the corresponding input and out along the corresponding output. The DHC gadget is carefully designed to satisfy the following property.

Lemma: (DHC Gadget Properties)

(i) Given any nonempty subset of k input edges, there exists a set of k vertex-disjoint paths that together visit all the vertices of the gadget exactly once and connect each input edge, i_j, with its corresponding output edge, o_j.

(ii) Given any nonempty set of k vertex-disjoint simple paths from entry to exit that hit all the vertices, each path that enters along i_j it must exist on the corresponding output o_j.

Proof: For (i), this is literally a "proof by picture"(see Fig. 147). We show the case of one, two, and three entry edges. (There are a total of $2^3 - 1 = 7$ nonempty subsets of inputs, but due to the symmetry of the gadget, it is easy to see how the omitted cases work.)

The proof of (ii) is a bit harder to see.

- In the case of one entry, in order to hit all the vertices, it is necessary to exit each three-vertex cluster by cycling one turn around. Since there are six clusters of three vertices, on exit the path returns to the same output edge.

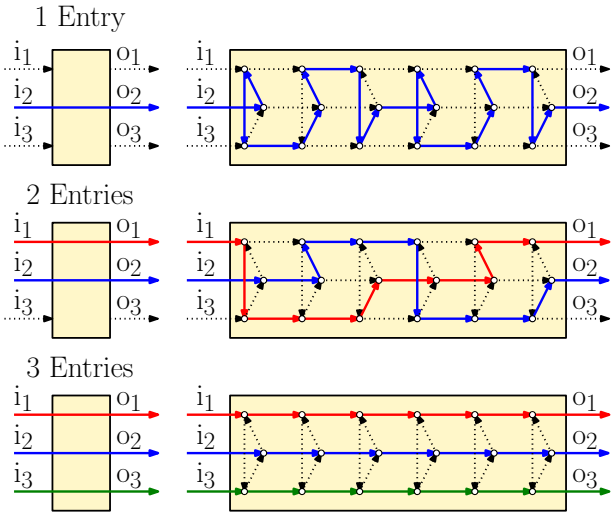


Fig. 147: DHC gadget and examples of path traversals.

- In the case of three entries, it is also easy to see that the only way to hit all the vertices exactly once is for the paths to travel straight through, implying that each input is routed to the corresponding output.
- In the case of two inputs, in order to hit every vertex once, one of the two paths needs to hit two vertices in each of the cluster and the other hits one. If you play around with the gadget, you will see that the choice of which path does which is fixed. This causes the two paths to cycle around. After three cycles they return to the same positions, but their order is switched. After six cycles, they return both to the same position and in the same order.

To see whether you understand the gadget, you might ask the question of why we used exactly 6 three-vertex cycles? Why a multiple of three? Would 3 work? Would 9? Would 12?

The Reduction: Our objective is to show that 3SAT $\leq_P$ DHC. That is, we need to present a polynomial-time function f, which given an boolean formula F for 3SAT, outputs a directed graph G such that F is satisfiable if and only if G has a Hamiltonian cycle. Recall that a satisfying a 3-CNF formula corresponds to assigning truth values to each variable so that every clause has at least one true literal.

Intuitively, we can think of the DHC gadgets as forming a string of holiday light bulbs along a wire. There will be one DHC gadget for each clause in the formula. Each gadget has three entry edges and three exits edges. Think of this like three wires entering and exiting each of these bulbs. Each variable x_j will have two paths which we can choose from. One corresponds to setting $x_j = \mathsf{T}$ and taking the other path corresponds to setting $x_j = \mathsf{F}$ (or equivalently to setting $\overline{x}_j = \mathsf{T}$). We think of the wire that carries the actual truth value for each variable to be the "live wire" that carries electrical current. Ultimately, we want every one of these light bulbs to light up. This will happen if at least one of the wires that is carrying current travels through the light bulb. This is equivalent to saying that at least one of the literals in this clause is true, which is exactly what we need for the formula to be satisfiable. Of course, we don't know whether each variable will be true or false. The idea is that the string of light bulbs will be illuminated if and only if we can assign truth values to all the variables so that the resulting current paths hit every bulb.

For each variable x_j, we will create a special *variable vertex*, named x_j. Each vertex x_j will have two outgoing edges, one labeled T and the other labeled F. Intuitively, any Hamiltonian cycle must visit each variable vertex x_j. If it chooses to take the T edge, we interpret this as setting $x_j = \mathsf{T}$ and taking

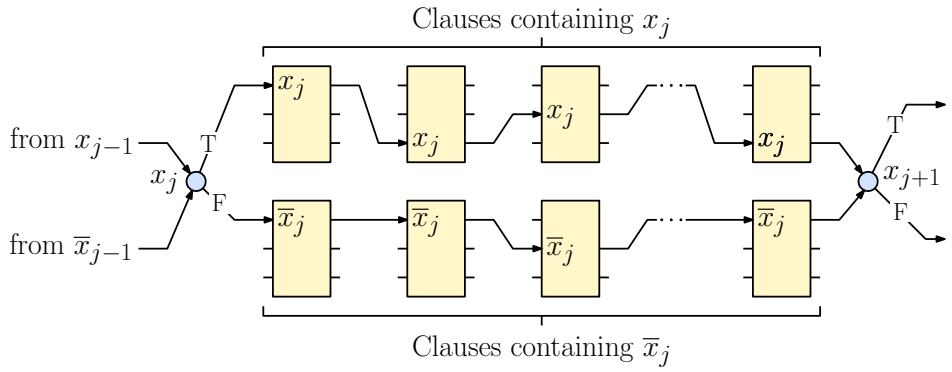


Fig. 148: General structure of reduction from 3SAT to DHC.

the other edge corresponds to setting $x_j = \mathsf{F}$. The path on the T edge will be strung through all the clause gadgets that contain the literal x_j, and the path along the F edge will be strung through all the clause gadgets that contain the literal $\overline{x}_j$ (see Fig. 148).

To connect all of these paths together, we create a start vertex s and add an edge (s, x_1). The two paths leading out from x_1 will converge at x_2, where again two paths will diverge, joining at x_3. We continue in this manner until reaching the final vertex x_n. Its two paths will converge at a final vertex t. To complete the cycle, we generate an edge (t, s). An example of the reduction is shown in Fig. 149.

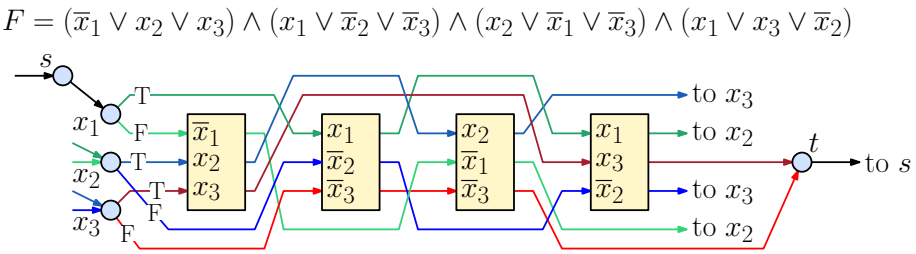


Fig. 149: Example of the 3SAT to DHC reduction. Wires are color-coded (x_1 is green, x_2 is blue, and x_3 is red).

The following lemma establishes the correctness of this reduction.

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

Proof:

($\Rightarrow$) : Suppose that F has a satisfying assignment. We claim that G has a Hamiltonian cycle. This path starts at s and goes immediately to the variable vertex x_1. From now on, whenever it reaches a variable vertex, it follows the outgoing T edge if this variable is assigned true and the F edge otherwise. Depending on which path is taken, it travels through all the gadgets involving this literal. When the final clause containing this literal is visited, it goes to the next variable vertex, x_{j+1}, and the process continues from there.

Because this is a satisfying assignment, every clause will have at least one true literal, and hence each clause gadget will be visited by at least one path. By DHC Gadget Property (i), whenever a path enters a gadget on some entry edge, it exits on the corresponding exit edge (see Fig. 150). (That is, wires never get crossed.)

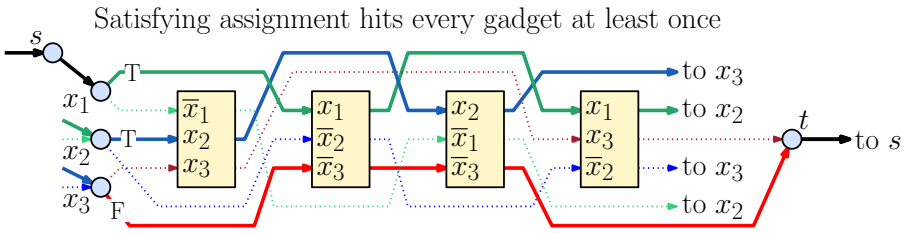


Fig. 150: A satisfying assignment ($x_1 = \mathsf{T}$, $x_2 = \mathsf{T}$, $x_3 = \mathsf{F}$), yields a Hamiltonian cycle.

On finishing with the path for the last variable x_n, we go to the end vertex t, and from there we go back to s, thus completing the Hamiltonian cycle. Since every clause is visited, every vertex is visited exactly once, and therefore, G has a Hamiltonian cycle.

($\Leftarrow$) : Suppose that G has a Hamiltonian cycle. Since the cycle must visit all the vertices, we may assume it starts with s. Since this is a Hamiltonian cycle, every vertex is visited, and hence, every clause gadget is visited by some number of paths (either 1, 2, or 3). From s the path visits x_1. Whenever it visits a variable vertex, the path takes either the outgoing edge labeled T or the edge labeled F. If it takes the former, we assign this variable true, and otherwise we assign it the value false.

We assert that this is a satisfying assignment for the formula. By the DHC Gadget Properties (ii), whenever a Hamiltonian cycle enters a gadget, it must exit along the exit edge corresponding to its entry edge. Therefore, each of the paths from the variable vertices behaves like a wire going through all gadgets in which this literal appears, and then going on to the next variable. When it visits the last variable, its only option is to go to t and then back to s.

Since this is a Hamiltonian cycle, every gadget must be visited by at least one path, and hence every clause must have at least one literal whose assigned value is true. (To illustrate this, consider the non-satisfying assignment in Fig. 151. At least one clause has all literals evaluate to false, and hence the cycle misses the associated gadget. However, this contradicts the hypothesis that G has a Hamiltonian cycle.) Since every clause has at least one true literal, this is a satisfying assignment.

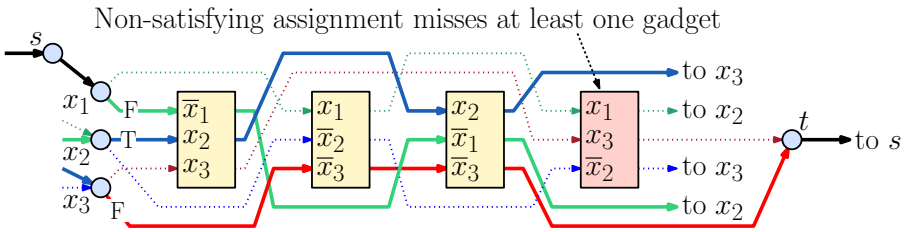


Fig. 151: A non-satisfying assignment ($x_1 = \mathsf{F}$, $x_2 = \mathsf{T}$, $x_3 = \mathsf{F}$), does not yield a Hamiltonian cycle.

Final Conclusion: We can now present the proof that DHC is NP-complete. Recall that we need to show that (i) DHC $\in$ NP and (ii) some known NP-complete problem is reducible to it.

Theorem: Directed Hamiltonian Cycle (DHC) is NP-complete.

Proof:

- (DHC $\in$ NP) The certificate consists of the sequence of vertices in the cycle. In $O(n)$ time, we can check each consecutive pair to see that it is an edge in G. If so, the verification accepts, and if not, the verification rejects.
- (3SAT $\leq_P$ DHC) This follows from correctness of the reduction presented earlier.

Lecture 32: 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. 152. 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

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

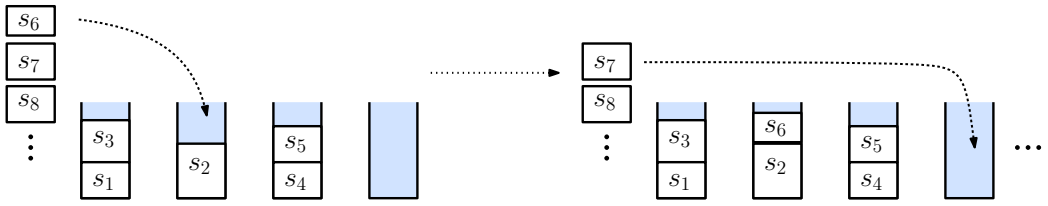


Fig. 152: First-fit Heuristic.

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 $11/9 \approx 1.222$.