Ceterum in problematis natura fundatum est, ut methodi quaecunque *continuo prolixiores evadant, quo maiores sunt numeri, ad quos applicantur.*
[It is in the nature of the problem that any *method will become more prolix as the numbers to which it is applied grow larger.]*

— Carl Friedrich Gauß, *Disquisitiones Arithmeticae* (1801)
English translation by A.A. Clarke (1965)

Illam vero methodum calculi mechanici taedium magis minuere, praxis tentantem docebit. [Truly, that method greatly reduces the tedium of mechanical calculations; practice will teach whoever tries it.]

— Carl Friedrich Gauß, "Theoria interpolationis methodo nova tractata" (c. 1805)

After much deliberation, the distinguished members of the international committee decided unanimously (when the Russian members went out for a caviar break) that since the Chinese emperor invented the method before anybody else had even been born, the method should be named after him. The Chinese emperor's name was Fast, so the method was called the Fast Fourier Transform.

— Thomas S. Huang, "How the fast Fourier transform got its name" (1971)

A

Fast Fourier Transforms

[Read Chapters 0 and 1 first.]
Status: Beta

A.1 Polynomials

Polynomials are functions of one variable built from additions, subtractions, and multiplications (but no divisions). The most common representation for a polynomial $p(x)$ is as a sum of weighted powers of the variable x:

$$p(x) = \sum_{j=0}^{n} a_j x^j.$$

The numbers a_j are called the *coefficients* of the polynomial. The *degree* of the polynomial is the largest power of x whose coefficient is not equal to zero; in the example above, the degree is *at most* n. Any polynomial of degree n can be represented by an array $P[0..n]$ of $n+1$ coefficients, where $P[j]$ is the coefficient of the x^j term, and where $P[n] \neq 0$.

Here are three of the most common operations performed on polynomials:

- **Evaluate:** Give a polynomial p and a number x, compute the number $p(x)$.
- **Add:** Give two polynomials p and q, compute a polynomial $r = p + q$, so that $r(x) = p(x) + q(x)$ for all x. If p and q both have degree n, then their sum $p + q$ also has degree n.
- **Multiply:** Give two polynomials p and q, compute a polynomial $r = p \cdot q$, so that $r(x) = p(x) \cdot q(x)$ for all x. If p and q both have degree n, then their product $p \cdot q$ has degree $2n$.

We learned simple algorithms for all three of these operations in high-school algebra. The addition and multiplication algorithms are straightforward generalizations of the standard algorithms for integer arithmetic.

```
EVALUATE(P[0..n], x):
  X ← 1    ⟨⟨X = x^j⟩⟩
  y ← 0
  for j ← 0 to n
      y ← y + P[j]·X
      X ← X·x
  return y
```

```
ADD(P[0..n], Q[0..n]):
  for j ← 0 to n
      R[j] ← P[j] + Q[j]
  return R[0..n]
```

```
MULTIPLY(P[0..n], Q[0..m]):
  for j ← 0 to n+m
      R[j] ← 0
  for j ← 0 to n
      for k ← 0 to m
          R[j+k] ← R[j+k] + P[j]·Q[k]
  return R[0..n+m]
```

EVALUATE uses $O(n)$ arithmetic operations.[1] This is the best we can hope for, but we can cut the number of multiplications in half using *Horner's rule*:

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + xa_n)).$$

```
HORNER(P[0..n], x):
  y ← P[n]
  for i ← n−1 down to 0
      y ← x·y + P[i]
  return y
```

The addition algorithm also runs in $O(n)$ time, and this is clearly the best we can do.

The multiplication algorithm, however, runs in $O(n^2)$ time. In Chapter 1, we saw a divide-and-conquer algorithm (due to Karatsuba) for multiplying two n-bit integers

[1] All time analysis in this lecture assumes that each arithmetic operation takes $O(1)$ time. This may not be true in practice; in fact, one of the most powerful applications of fast Fourier transforms is fast *integer* multiplication. The fastest algorithm currently known for multiplying two n-bit integers, published by Martin Fürer in 2007, uses $O(n \log n \, 2^{O(\log^* n)})$ bit operations and is based on fast Fourier transforms.

in only $O(n^{\lg 3})$ steps; precisely the same approach can be applied here. Even cleverer divide-and-conquer strategies lead to multiplication algorithms whose running times are arbitrarily close to linear—$O(n^{1+\varepsilon})$ for your favorite value $e > 0$—but except for a few simple cases, these algorithms not worth the trouble in practice, thanks to large hidden constants.

A.2 Alternate Representations

Part of what makes multiplication so much harder than the other two operations is our input representation. Coefficient vectors are the most common representation for polynomials, but there are at least two other useful representations.

Roots

The Fundamental Theorem of Algebra states that every polynomial p of degree n has exactly n *roots* $r_1, r_2, \ldots r_n$ such that $p(r_j) = 0$ for all j. Some of these roots may be irrational; some of these roots may by complex; and some of these roots may be repeated. Despite these complications, this theorem implies a unique representation of any polynomial of the form

$$p(x) = s\prod_{j=1}^{n}(x - r_j)$$

where the r_j's are the roots and s is a scale factor. Once again, to represent a polynomial of degree n, we need a list of $n+1$ numbers: one scale factor and n roots.

Given a polynomial in this root representation, we can clearly evaluate it in $O(n)$ time. Given two polynomials in root representation, we can easily multiply them in $O(n)$ time by multiplying their scale factors and just concatenating the two root sequences.

Unfortunately, if we want to add two polynomials in root representation, we're out of luck. There's essentially *no* correlation between the roots of p, the roots of q, and the roots of $p + q$. We could convert the polynomials to the more familiar coefficient representation first—this takes $O(n^2)$ time using the high-school algorithms—but there's no easy way to convert the answer back. In fact, for most polynomials of degree 5 or more in coefficient form, it's *impossible* to compute roots exactly.[2]

Samples

Our third representation for polynomials comes from a different consequence of the Fundamental Theorem of Algebra. Given a list of $n+1$ pairs $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$, there is *exactly one* polynomial p of degree n such that $p(x_j) = y_j$ for all j. This is just a generalization of the fact that any two points determine a unique line, because a

[2] This is where numerical analysis comes from.

line is the graph of a polynomial of degree 1. We say that the polynomial p ***interpolates*** the points (x_j, y_j). As long as we agree on the sample locations x_j in advance, we once again need exactly $n+1$ numbers to represent a polynomial of degree n.

Adding or multiplying two polynomials in this sample representation is easy, as long as they use the same sample locations x_j. To add the polynomials, just add their sample values. To multiply two polynomials, just multiply their sample values; however, if we're multiplying two polynomials of degree n, we must *start* with $2n+1$ sample values for each polynomial, because that's how many we need to uniquely represent their product. Both algorithms run in $O(n)$ time.

Unfortunately, evaluating a polynomial in this representation is no longer straightforward. The following formula, due to Lagrange, allows us to compute the value of any polynomial of degree n at any point, given a set of $n+1$ samples.

$$p(x) = \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k \neq j}(x_j - x_k)} \prod_{k \neq j} (x - x_k) \right)$$

Hopefully it's clear that formula actually describes a polynomial function of x, since each term in the sum is a scaled product of monomials. It's also not hard to verify that $p(x_j) = y_j$ for every index j; most of the terms of the sum vanish. As I mentioned earlier, the Fundamental Theorem of Algebra implies that p is *the only* polynomial that interpolates the points $\{(x_j, y_j)\}$. Lagrange's formula can be translated mechanically into an $O(n^2)$-time algorithm.

Summary

We find ourselves in the following frustrating situation. We have three representations for polynomials and three basic operations. Each representation allows us to almost trivially perform a different pair of operations in linear time, but the third takes at least quadratic time, if it can be done at all!

representation	evaluate	add	multiply
coefficients	$O(n)$	$O(n)$	$\mathbf{O(n^2)}$
roots + scale	$O(n)$	$\boldsymbol{\infty}$	$O(n)$
samples	$\mathbf{O(n^2)}$	$O(n)$	$O(n)$

A.3 Converting Between Representations

What we need are fast algorithms to convert quickly from one representation to another. Then if we need to perform an operation that's hard for our default representation, we can switch to a different representation that makes the operation easy, perform the desired operation, and then switch back. This strategy immediately rules out the root

representation, since (as I mentioned earlier) finding roots of polynomials is impossible in general, at least if we're interested in exact results.

So how do we convert from coefficients to samples and back? Clearly, once we choose our sample positions x_j, we can compute each sample value $y_j = p(x_j)$ in $O(n)$ time from the coefficients using Horner's rule. So we can convert a polynomial of degree n from coefficients to samples in $O(n^2)$ time. Lagrange's formula can be used to convert the sample representation back to the more familiar coefficient form. If we use the naïve algorithms for adding and multiplying polynomials (in coefficient form), this conversion takes $O(n^3)$ time.

We can improve the cubic running time by observing that *both* conversion problems boil down to computing the product of a matrix and a vector. The explanation will be slightly simpler if we assume the polynomial has degree $n-1$, so that n is the number of coefficients or samples. Fix a sequence $x_0, x_1, \ldots, x_{n-1}$ of sample *positions*, and let V be the $n \times n$ matrix where $v_{ij} = x_i^j$ (indexing rows and columns from 0 to $n-1$):

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}.$$

The matrix V is called a ***Vandermonde matrix***. The vector of coefficients $\vec{a} = (a_0, a_1, \ldots, a_{n-1})$ and the vector of sample *values* $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$ are related by the matrix equation

$$V\vec{a} = \vec{y},$$

or in more detail,

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}.$$

Given this formulation, we can clearly transform any coefficient vector $\vec{a}$ into the corresponding sample vector $\vec{y}$ in $O(n^2)$ time.

Conversely, if we know the sample values $\vec{y}$, we can recover the coefficients by solving a system of n linear equations in n unknowns, which can be done in $O(n^3)$ time using Gaussian elimination.[3] But we can speed this up by implicitly hard-coding the sample

[3] In fact, Lagrange's formula is just a special case of Cramer's rule for solving linear systems.

positions into the algorithm, To convert from samples to coefficients, we can simply multiply the sample vector by the inverse of V, again in $O(n^2)$ time:

$$\vec{a} = V^{-1}\vec{y}.$$

Computing V^{-1} would take $O(n^3)$ time if we had to do it from scratch using Gaussian elimination, but because we fixed the set of sample positions in advance, the matrix V^{-1} can be hard-coded directly into the algorithm.[4]

So we can convert from coefficients to samples and back in $O(n^2)$ time. At first lance, this result seems pointless; we can already add, multiply, or evaluate directly in either representation in $O(n^2)$ time, so why bother? But there's a degree of freedom we haven't exploited yet: ***We get to choose the sample positions!*** Our conversion algorithm is slow only because we're trying to be too general. If we choose a set of sample positions with the right recursive structure, we can perform this conversion more quickly.

A.4 Divide and Conquer

Any polynomial of degree at most $n-1$ can be expressed as a combination of two polynomials of degree at most $(n/2)-1$ as follows:

$$p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2).$$

The coefficients of p_{even} are just the even-degree coefficients of p, and the coefficients of p_{odd} are just the odd-degree coefficients of p. Thus, we can evaluate $p(x)$ by recursively evaluating $p_{\text{even}}(x^2)$ and $p_{\text{odd}}(x^2)$ and performing $O(1)$ additional arithmetic operations.

Now call a set X of n values ***collapsing*** if either of the following conditions holds:

- X has one element.
- The set $X^2 = \left\{x^2 \;\middle|\; x \in X\right\}$ has exactly $n/2$ elements and is (recursively) collapsing.

Clearly the size of any collapsing set is a power of 2. Given a polynomial p of degree $n-1$, and a collapsing set X of size n, we can compute the set $\{p(x) \mid x \in X\}$ of sample values as follows:

1. Recursively compute $\left\{p_{\text{even}}(x^2) \;\middle|\; x \in X\right\} = \left\{p_{\text{even}}(y) \;\middle|\; y \in X^2\right\}$.
2. Recursively compute $\left\{p_{\text{odd}}(x^2) \;\middle|\; x \in X\right\} = \left\{p_{\text{odd}}(y) \;\middle|\; y \in X^2\right\}$.
3. For each $x \in X$, compute $p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2)$.

The running time of this algorithm satisfies the familiar recurrence $T(n) = 2T(n/2) + \Theta(n)$, which as we all know solves to $T(n) = \Theta(n \log n)$.

[4] Actually, it is possible to invert an $n \times n$ matrix in $o(n^3)$ time, using fast matrix multiplication algorithms that closely resemble Karatsuba's sub-quadratic divide-and-conquer algorithm for integer/polynomial multiplication. On the other hand, my numerical-analysis colleagues have reasonable cause to shoot me in the face for daring to suggest, even in passing, that anyone actually invert a matrix at all, ever.

Great! Now all we need is a sequence of arbitrarily large collapsing sets. The simplest method to construct such sets is just to invert the recursive definition: If X is a collapsible set of size n *that does not contain the number* 0, then $\sqrt{X} = \{\pm\sqrt{x} \mid x \in X\}$ is a collapsible set of size $2n$. This observation gives us an infinite sequence of collapsible sets, starting as follows:[5]

$$X_1 := \{1\}$$
$$X_2 := \{1, -1\}$$
$$X_4 := \{1, -1, i, -i\}$$
$$X_8 := \left\{1, -1, i, -i, \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i, -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i\right\}$$

A.5 The Discrete Fourier Transform

For any n, the elements of X_n are called the ***complex nth roots of unity***; these are the roots of the polynomial $x^n - 1 = 0$. These n complex values are spaced exactly evenly around the unit circle in the complex plane. Every nth root of unity is a power of the *primitive* nth root

$$\boxed{\omega_n = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}.}$$

A typical nth root of unity has the form

$$\omega_n^k = e^{(2\pi i/n)k} = \cos\left(\frac{2\pi}{n}k\right) + i\sin\left(\frac{2\pi}{n}k\right).$$

These complex numbers have several useful properties for any integers n and k:

- There are exactly n different nth roots of unity: $\omega_n^k = \omega_n^{k \bmod n}$.
- If n is even, then $\omega_n^{k+n/2} = -\omega_n^k$; in particular, $\omega_n^{n/2} = -\omega_n^0 = -1$.
- $1/\omega_n^k = \omega_n^{-k} = \overline{\omega_n^k} = (\overline{\omega_n})^k$, where the bar represents complex conjugation: $\overline{a + bi} = a - bi$
- $\omega_n = \omega_{kn}^k$. Thus, every nth root of unity is also a (kn)th root of unity.

These properties imply immediately that if n is a power of 2, then the set of all nth roots of unity is collapsible!

If we sample a polynomial of degree $n-1$ at the nth roots of unity, the resulting list of sample values is called the ***discrete Fourier transform*** of the polynomial (or more

[5] In this chapter, lower case italic i always represents the square root of -1. Computer scientists are used to thinking of i as an integer index into a sequence, an array, or a for-loop, but we obviously can't do that here. The engineers' habit of using $j = \sqrt{-1}$ just delays the problem—How do engineers write quaternions?—and typographical hacks like I or $\mathbf{i}$ or $\imath$ or Mathematica's $\mathbb{i}$ are just stupid.

formally, of its coefficient vector). Thus, given an array $P[0..n-1]$ of coefficients, its discrete Fourier transform is the vector $P^*[0..n-1]$ defined as follows:

$$P^*[j] := p(\omega_n^j) = \sum_{k=0}^{n-1} P[k] \cdot \omega_n^{jk}$$

As we already observed, the fact that sets of roots of unity are collapsible implies that we can compute the discrete Fourier transform in $O(n \log n)$ time. The resulting algorithm, called the ***fast Fourier transform***, was popularized by Cooley and Tukey in 1965.[6] The algorithm assumes that n is a power of two; if necessary, we can just pad the coefficient vector with zeros.

```
RADIX2FFT(P[0..n-1]):
  if n = 1
      return P

  for j ← 0 to n/2-1
      U[j] ← P[2j]
      V[j] ← P[2j+1]

  U* ← RADIX2FFT(U[0..n/2-1])
  V* ← RADIX2FFT(V[0..n/2-1])

  ω_n ← cos(2π/n) + i sin(2π/n)
  ω ← 1

  for j ← 0 to n/2-1
      P*[j]       ← U*[j] + ω · V*[j]
      P*[j+n/2] ← U*[j] - ω · V*[j]
      ω ← ω · ω_n

  return P*[0..n-1]
```

Figure A.1. The Cooley-Tukey radix-2 fast Fourier transform algorithm.

Variants of this divide-and-conquer algorithm were previously described by Good in 1958, by Thomas in 1948, by Danielson and Lánczos in 1942, by Stumpf in 1937, by Yates in 1932, and by Runge in 1903; some special cases were published even earlier by Everett in 1860, by Smith in 1846, and by Carlini in 1828. But the algorithm, in its full modern recursive generality, was first described *and used* by Gauss around 1805 for calculating the periodic orbits of asteroids from a finite number of observations. In fact, Gauss's recursive algorithm predates even Fourier's introduction of harmonic analysis by two years. So, of course, the algorithm is universally called the *Cooley-Tukey*

[6] Tukey apparently developed this algorithm to help detect Soviet nuclear tests without actually visiting Soviet nuclear facilities, by interpolating off-shore seismic readings. Without his rediscovery, the nuclear test ban treaty might never have been ratified, and we might all be speaking Russian, or more likely, whatever language radioactive glass speaks.

algorithm. Gauss's work built on earlier research on trigonometric interpolation by Bernoulli, Lagrange, Clairaut, and Euler; in particular, the first explicit description of the discrete "Fourier" transform was published by Clairaut in 1754, more than half a century before Fourier's work. Alas, nobody will understand you if you talk about Gauss's fast Clairaut transform algorithms. Hooray for Stigler's Law![7]

A.6 More General Factoring

The algorithm in Figure A.1 is often called the ***radix-2 fast Fourier transform*** to distinguish it from other FFT algorithms. In fact, both Gauss and (much later) Cooley and Tukey described a more general divide-and-conquer strategy that does not assume n is a power of two, but can be applied to any composite order n. Specifically, if $n = pq$, we can decompose the discrete Fourier transform of order n into simpler discrete Fourier transforms as as follows. For all indices $0 \le a < p$ and $0 \le b < q$, we have

$$\begin{aligned}
x^*_{aq+b} &= \sum_{\ell=0}^{pq-1} x_{jp+k} \, (\omega_{pq}^{jp+k})^{\ell} \\
&= \sum_{k=0}^{p-1} \sum_{j=0}^{q-1} x_{jp+k} \, \omega_{pq}^{(aq+b)(jp+k)} \\
&= \sum_{k=0}^{p-1} \sum_{j=0}^{q-1} x_{jp+k} \, \omega_q^{bj} \, \omega_{pq}^{bk} \, \omega_p^{ak} \\
&= \sum_{k=0}^{p-1} \left(\left(\sum_{j=0}^{q-1} x_{jp+k} \, (\omega_q^b)^j \right) \omega_{pq}^{bk} \right) (\omega_p^a)^k,
\end{aligned}$$

The innermost sum in this expression is one coefficient of a discrete Fourier transform of order q, and the outermost sum is one coefficient of a discrete Fourier transform of order q. The intermediate factors ω_{pq}^{bk} are now formally known as "twiddle factors".

[7] Lest anyone believe that Stigler's Law has treated Gauss unfairly, remember that "Gaussian elimination" was not discovered by Gauss; the algorithm was not even given that name until the mid-20th century! Elimination became the standard method for solving systems of linear equations in Europe in the early 1700s, when it appeared in Isaac Newton's influential textbook *Arithmetica universalis*.[8] Although Newton apparently (and perhaps even correctly) believed he had invented the elimination algorithm, it actually appears in several earlier works, including the eighth chapter of the Chinese manuscript *The Nine Chapters of the Mathematical Art*. The authors and precise age of the *Nine Chapters* are unknown, but commentary written by Liu Hui in 263CE claims that the text was already several centuries old. It was almost certainly *not* invented by a Chinese emperor named Fast.

[8] *Arithmetica universalis* was compiled from Newton's lecture notes and published over Newton's strenuous objections. He refused to have his name associated with the book, and he even considered buying up every copy of the first printing to destroy them. Apparently he didn't want anyone to think it was his latest research. The first edition crediting Newton as the author did not appear until 25 years after his death.

No, seriously, that's actually what they're called. This wall of symbols implies that the discrete Fourier transform of order n can be evaluated as follows:

1. Write the input vector into a $p \times q$ array in row-major order.
2. Apply a discrete Fourier transform of order p to each column of the 2d array.
3. Multiply each entry of the 2d array by the appropriate twiddle factor.
4. Apply a discrete Fourier transform of order q to each row of the 2d array.
5. Extract the output vector from the $p \times q$ array in column-major order.

The algorithm is described in more detail in Figure A.2.

```
FACTORFFT(P[0..pq−1]):
  ⟨⟨Copy/typecast to 2d array in row-major order⟩⟩
  for j ← 0 to p−1
      for k ← 0 to q−1
          A[j,k] ← P[jp+k]

  ⟨⟨Recursively apply order-p FFTs to columns⟩⟩
  for k ← 0 to q−1
      B[·,k] ← FFT(A[·,k])

  ⟨⟨Multiply by twiddle factors⟩⟩
  for j ← 0 to p−1
      for k ← 0 to q−1
          B[·,k] ← B[·,k] · ω_pq^jk

  ⟨⟨Recursively apply order-q FFTs to rows⟩⟩
  for j ← 0 to p−1
      C[j,·] ← FFT(C[j,·])

  ⟨⟨Copy/typecast to 1d array in column-major order⟩⟩
  for j ← 0 to p−1
      for k ← 0 to q−1
          P*[j+kq] ← C[j,k]

  return P*[0..pq−1]
```

Figure A.2. The Gauss-Cooley-Tukey FFT algorithm

We can recover the original radix-2 FFT algorithm in Figure A.1 by setting $p = n/2$ and $q = 2$. The lines $P^*[j] \leftarrow U^*[j] + \omega \cdot V^*[j]$ and $P^*[j+n/2] \leftarrow U^*[j] - \omega \cdot V^*[j]$ are applying an order-2 discrete Fourier transform; in particular, the multipliers ω and $-\omega$ are the "twiddle factors".

Both Gauss[9] and Cooley and Tukey recommended applying this factorization approach recursively. Cooley and Tukey observed further that if all prime factors of n are

[9] Gauss wrote: "*Nulla iam amplius explicatione opus erit, quomodo illa partitio adhuc ulterius extendi et ad eum casum applicari possit, ubi multitudo omnium valorum propositorum numerus e tribus pluribusve factoribus compositus est, e.g. si numerus μ rursus esset compositus, in quo casu manifesto quaevis periodus μ terminorum in plures periodos minores subdividi potest.*" ["There is no need to explain how that division can be extended further and applied to the case where most of the proposed values are composed of three or more factors, for example, if the number μ is again composite, in which case each period of μ terms can

smaller than some constant, so that the subproblems at the leaves of the recursion tree can be solved in $O(1)$ time, the entire algorithm runs in only $O(n \log n)$ time.

Using a completely different approach, Charles Rader and Leo Bluestein described FFT algorithms that run in $O(n \log n)$ time when n is an arbitrary prime number, by reducing to two FFTs whose orders have only small prime factors. Combining these two approaches yields an $O(n \log n)$-time algorithm to compute discrete Fourier transforms of any order n.

A.7 Inverting the FFT

We also need to recover the coefficients of the product from the new sample values. Recall that the transformation from coefficients to sample values is *linear*; the sample vector is the product of a Vandermonde matrix V and the coefficient vector. For the discrete Fourier transform, each entry in V is an nth root of unity; specifically,

$$\boxed{v_{jk} = \omega_n^{jk}}$$

for all integers j and k. More explicitly:

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

To invert the discrete Fourier transform, converting sample values back to coefficients, it suffices to multiply the vector P^* of sample values by the inverse matrix V^{-1}. The following amazing fact implies that this is almost the same as multiplying by V itself:

Lemma A.1. $V^{-1} = \overline{V}/n$

Proof: We just have to show that $M = V\overline{V}$ is the identity matrix scaled by a factor of n. We can compute a single entry in M as follows:

$$m_{jk} = \sum_{l=0}^{n-1} \omega_n^{jl} \cdot \overline{\omega_n}^{lk} = \sum_{l=0}^{n-1} \omega_n^{jl-lk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l$$

obviously be subdivided into several smaller periods."]

If $j = k$, then $\omega_n^{j-k} = \omega_n^0 = 1$, so

$$m_{jk} = \sum_{l=0}^{n-1} 1 = n,$$

and if $j \neq k$, we have a geometric series

$$m_{jk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l = \frac{(\omega_n^{j-k})^n - 1}{\omega_n^{j-k} - 1} = \frac{(\omega_n^n)^{j-k} - 1}{\omega_n^{j-k} - 1} = \frac{1^{j-k} - 1}{\omega_n^{j-k} - 1} = 0. \qquad \square$$

In other words, if $W = V^{-1}$ then $w_{jk} = \overline{v_{jk}}/n = \overline{\omega_n^{jk}}/n = \omega_n^{-jk}/n$. What this observation implies for us computer scientists is that any algorithm for computing the discrete Fourier transform can be trivially adapted or modified to compute the inverse transform as well; see Figure A.3.

```
INVERSEFFT(P*[0..n-1]):
  P[0..n-1] ← FFT(P*)
  for j ← 0 to n-1
      P*[j] ← conj(P*[j])/n
  return P[0..n-1]
```

```
INVERSERADIX2FFT(P*[0..n-1]):
  if n = 1
      return P

  for j ← 0 to n/2-1
      U*[j] ← P*[2j]
      V*[j] ← P*[2j+1]

  U ← INVERSERADIX2FFT(U*[0..n/2-1])
  V ← INVERSERADIX2FFT(V*[0..n/2-1])

  conj(ω_n) ← cos(2π/n) - i sin(2π/n)
  conj(ω) ← 1

  for j ← 0 to n/2-1
      P[j]     ← (U[j] + conj(ω)·V[j])/2
      P[j+n/2] ← (U[j] - conj(ω)·V[j])/2
      conj(ω) ← conj(ω)·conj(ω_n)

  return P[0..n-1]
```

Figure A.3. Generic and radix-2 inverse FFT algorithms.

A.8 Fast Polynomial Multiplication

Finally, given two polynomials p and q, represented by an arrays of length m and n, respectively, we can multiply them in $\Theta((m+n)\log(m+n))$ arithmetic operations as follows. First, pad the coefficient vectors with zeros to length $m+n$ (or to the next larger power of 2 if we plan to use the radix-2 FFT algorithm). Then compute the discrete Fourier transforms of each coefficient vector, and multiply the resulting sample values one by one. Finally, compute the inverse discrete Fourier transform of the resulting sample vector.

```
FFTMultiply(P[0..m−1], Q[0..n−1]):
  for j ← m to m+n−1
      P[j] ← 0
  for j ← n to m+n−1
      Q[j] ← 0

  P* ← FFT(P)
  Q* ← FFT(Q)
  for j ← 0 to 2^ℓ − 1
      R*[j] ← P*[j] · Q*[j]
  return InverseFFT(R*)
```

Figure A.4. Multiplying polynomials in $O((m+n)\log(m+n))$ time.

A.9 Inside the Radix-2 FFT

Fast Fourier transforms are often implemented in hardware as circuits; Cooley and Tukey's radix-2 algorithm unfolds into a particularly nice recursive structure, as shown in Figure A.5 for $n = 16$. On the left, the n top-level inputs and outputs are connected to the inputs and outputs of the recursive calls, represented here as gray boxes. On the left we split the input P into two recursive inputs U and V. On the right, we combine the outputs U^* and V^* to obtain the final output P^*.

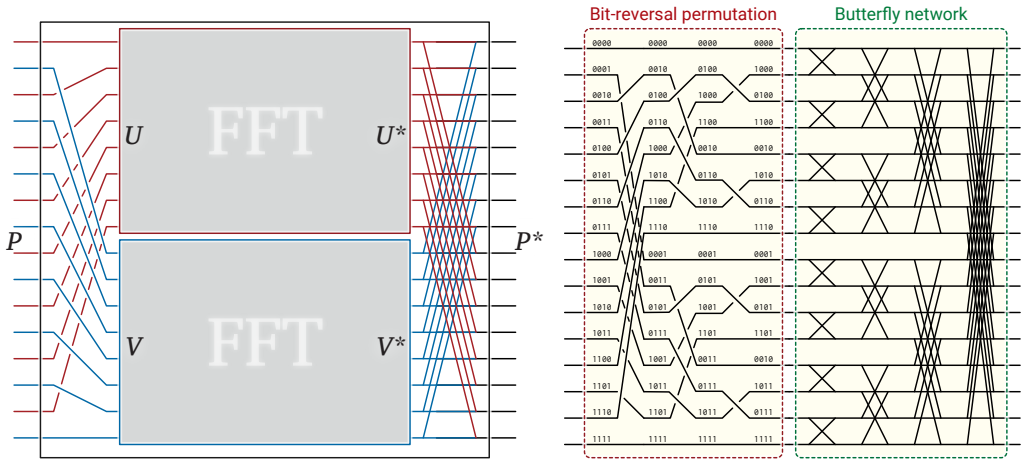


Figure A.5. The recursive structure of the radix-2 FFT algorithm.

If we expand this recursive structure completely, we see that the circuit splits naturally into two parts.

- The left half computes the *bit-reversal permutation* of the input. To find the position of $P[k]$ in this permutation, write k in binary and then read the bits backward. For example, in an 8-element bit-reversal permutation, $P[3] = P[011_2]$ ends up in position $6 = 110_2$.

- The right half of the FFT circuit is called a *butterfly network*. Butterfly networks are often used to route between processors in massively-parallel computers, because they allow any two processors to communicate in only $O(\log n)$ steps.

When n is a power of 2, recursively applying the more general FACTORFFT gives us exactly the same recursive structure, just clustered differently. For many applications of FFTs, including polynomial multiplication, the bit-reversal permutation is unnecessary and can actually be omitted.

Exercises

1. For any two sets X and Y of integers, the Minkowski sum $X + Y$ is the set of all pairwise sums $\{x + y \mid x \in X, y \in Y\}$.

   (a) Describe an analyze and algorithm to compute the number of elements in $X + Y$ in $O(n^2 \log n)$ time. *[Hint: The answer is* ***not*** *always n^2.]*

   (b) Describe and analyze an algorithm to compute the number of elements in $X + Y$ in $O(M \log M)$ time, where M is the largest absolute value of any element of $X \cup Y$. *[Hint: What's this lecture about?]*

2. Suppose we are given a bit string $B[1..n]$. A triple of distinct indices $1 \le i < j < k \le n$ is called a ***well-spaced triple*** in B if $B[i] = B[j] = B[k] = 1$ and $k - j = j - i$.

   (a) Describe a brute-force algorithm to determine whether B has a well-spaced triple in $O(n^2)$ time.

   (b) Describe an algorithm to determine whether B has a well-spaced triple in $O(n \log n)$ time. *[Hint: Hint.]*

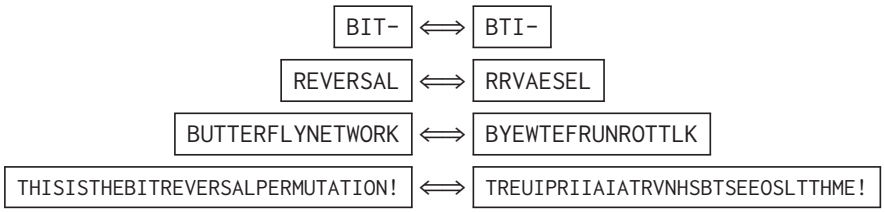
   (c) Describe an algorithm to determine the *number* of well-spaced triples in B in $O(n \log n)$ time.

3. (a) Describe an algorithm that determines whether a given set of n integers contains two elements whose sum is zero, in $O(n \log n)$ time.

   (b) Describe an algorithm that determines whether a given set of n integers contains *three* elements whose sum is zero, in $O(n^2)$ time.

   (c) Now suppose the input set X contains only integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether X contains three elements whose sum is zero, in $O(n \log n)$ time. *[Hint: Hint.]*

4. Describe an algorithm that applies the bit-reversal permutation to an array $A[1..n]$ in $O(n)$ time when n is a power of 2.



5. The FFT algorithm we described in this lecture is limited to polynomials with 2^k coefficients for some integer k. Of course, we can always pad the coefficient vector with zeros to force it into this form, but this padding artificially inflates the input size, leading to a slower algorithm than necessary.

   Describe and analyze a similar DFT algorithm that works for polynomials with 3^k coefficients, by splitting the coefficient vector into three smaller vectors of length 3^{k-1}, recursively computing the DFT of each smaller vector, and correctly combining the results.

6. Fix an integer k. For any two k-bit integers i and j, let $\boldsymbol{i \wedge j}$ denote their bitwise AND, and let $\boldsymbol{\Sigma(i)}$ denote the number of 1s in the binary expansion of i. For example, when $k = 4$, we have $10 \wedge 7 = \texttt{1010} \wedge \texttt{0111} = \texttt{0010} = 2$ and $\Sigma(7) = \Sigma(\texttt{0111}) = 3$.

   The kth ***Sylvester-Hadamard matrix*** $\boldsymbol{H_k}$ is a $2^k \times 2^k$ matrix indexed by k-bit integers, each of whose entries is either $+1$ or -1, defined as follows:

$$H_k[i, j] = (-1)^{\Sigma(i \wedge j)}$$

   For example:

$$H_3 = \begin{bmatrix} +1 & +1 & +1 & +1 & +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 & +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 & +1 & -1 & -1 & +1 \\ +1 & +1 & +1 & +1 & -1 & -1 & -1 & -1 \\ +1 & -1 & +1 & -1 & -1 & +1 & -1 & +1 \\ +1 & +1 & -1 & -1 & -1 & -1 & +1 & +1 \\ +1 & -1 & -1 & +1 & -1 & +1 & +1 & -1 \end{bmatrix}$$

   (a) Prove that the matrix H_k can be decomposed into four copies of H_{k-1} as follows:

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

   (b) Prove that $H_k \cdot H_k = 2^k \cdot I_k$, where I is the $2^k \times 2^k$ identity matrix.

(c) For any vector $\vec{x} \in \mathbb{R}^{2^k}$, the product $H_k\vec{x}$ is called the ***Walsh-Hadamard transform*** of $\vec{x}$. Describe an algorithm to compute the Walsh-Hadamard transform in $O(n \log n)$ time, given the integer k and a vector of $n = 2^k$ integers as input.

♥7. The ***discrete Hartley transform*** of a vector $\vec{x} = (x_0, x_1, \ldots, x_{n-1})$ is another vector $\vec{X} = (X_0, X_1, \ldots, X_{n-1})$ defined as follows:

$$X_j = \sum_{i=0}^{n-1} x_i \left(\cos\left(\frac{2\pi}{n} ij\right) + \sin\left(\frac{2\pi}{n} ij\right) \right)$$

Describe an algorithm to compute the discrete Hartley transform in $O(n \log n)$ time when n is a power of 2.

8. Suppose n is a power of 2. Prove that if we recursively apply the FACTORFFT algorithm, factoring of n into *arbitrary* smaller powers of two, the resulting algorithm still runs in $O(n \log n)$ time.

9. Let $f : \mathbb{Z} \to \mathbb{Z}$ be *any* function such that $2 \le f(n) \le \sqrt{n}$ for all $n \ge 4$. Prove that the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \le 4 \\ f(n) \cdot T\left(\frac{n}{f(n)}\right) + \frac{n}{f(n)} \cdot T(f(n)) + O(n) & \text{otherwise} \end{cases}$$

has the solution $T(n) = O(n \log n)$. For example, setting $f(n) = 2$ for all n gives us the standard mergesort/FFT recurrence.

10. Although the radix-2 FFT algorithm and the more general recursive factoring algorithm both run in $O(n \log n)$ time, the latter algorithm is more efficient in practice (when n is large) due to caching effects.

Consider an idealized two-level memory model, which has an arbitrarily large but slow *main memory* and a significantly faster *cache* that holds the C most recently accessed memory items. The running time of an algorithm in this model is dominated by the number of *cache misses*, when the algorithm needs to access a value in memory that it not stored in the cache.

The number of cache misses seen by the radix-2 FFT algorithm obeys the following recurrence:

$$M(n) \le \begin{cases} 2M(n/2) + O(n) & \text{if } n > C \\ O(n) & \text{if } n \le C \end{cases}$$

If the input array is too large to fit in the cache, then *every* memory access in both the initial and final for-loops will cause a cache miss, and the recursive calls will cause their own cache misses. But if the input array is small enough to fit in cache, the

initial for-loop loads the input into the cache, but there are no more cache misses, even inside the recursive calls.

(a) Solve the previous recurrence for $M(n)$, as a function of both n and C. To simplify the analysis, assume both n and C are powers of 2.

(b) Suppose we always recursively call FACTORFFT with $p = C$ and $q = n/C$. How many cache misses does the resulting algorithm see, as a function of n and C? To simplify the analysis, assume $n = C^k$ for some integer k. (In particular, assume C is a power of 2.)

(c) Unfortunately, it is not always possible for a program (or a compiler) to determine the size of the cache. Suppose we always recursively call FACTORFFT with $p = q = \sqrt{n}$. How many cache misses does the resulting algorithm see, as a function of n and C? To simplify the analysis, assume $n = C^{2^k}$ for some integer k.

Wouldn't the sentence "I want to put a hyphen between the words Fish and And and And and Chips in my Fish-And-Chips sign." have been clearer if quotation marks had been placed before Fish, and between Fish and and, and and and And, and And and and, and and and And, and And and and, and and and and Chips, as well as after Chips?[1]

— Martin Gardner, *Aha! Insight* (1978)

BMO: *I finished with my computations! As I thought, it's a real language they are speaking based on intonations! [speaking to Chips and Ice Cream] Chips? Chips chips chips! Ice cream?*
Chips: *Chips chips chips!*
Ice Cream: *Ice cream ice cream!*

— "Chips and Ice Cream", *Adventure Time* season 6, episode 34 (April 30, 2015)

For every polynomial-time algorithm you have,
there is an exponential algorithm that I would rather run.

— Alan Perlis (first Turing Award winner)

B

Faster Exponential Algorithms

[Read Chapters 2, 3, and 12 first.]
Status: Unfinished.

There are many useful problems—including but not limited to the set of all NP-hard problems—for which the fastest algorithm known runs in exponential time. Moreover, there is *very* strong evidence (but alas, no proof) that it is *impossible* to solve these problems in less than exponential time—it's not that we're all stupid; the problems really are that hard!

But sometimes we have to solve these problems anyway. In these circumstances, it is important to ask: *Which* exponential? An algorithm that runs in $O(2^n)$ time, while still

[1] If you ever decide to read this sentence out loud, be sure to pause briefly between 'Fish and and' and 'and and and And', 'and and and And' and 'and And and and', 'and And and and' and 'and and and And', 'and and and And' and 'and And and and', and 'and And and and' and 'and and and Chips'!

Did you notice the punctuation I carefully inserted between 'Fish and and' and 'and', 'and' and 'and and and And', 'and and and And' and 'and and and And', 'and and and And' and 'and', 'and' and 'and And and and', 'and And and and' and 'and And and and', 'and And and and' and 'and', 'and' and 'and and and And', 'and and and And' and 'and and and And', 'and and and And' and 'and', 'and' and 'and And and and', 'and And and and' and 'and', 'and' and 'and And and and', 'and And and and' and 'and', and 'and' and 'and and and Chips'?

unusable for large instances, is still significantly better than an algorithm that runs in $O(4^n)$ time; an algorithm that runs in $O(1.5^n)$ or $O(1.25^n)$ is better still.

The most straightforward method to find an optimal solution to an NP-hard problem to recursively generate *all* possible solutions via recursive backtracking and check each one: all satisfying assignments, or all vertex colorings, or all subsets, or all permutations, or whatever. However, many (perhaps even most) NP-hard problems have some additional structure that allows us to prune away most of the branches of the recursion tree, thereby drastically reducing the running time.

B.1 3SAT

Let's consider the mother of all NP-hard problems: 3SAT. Given a boolean formula in conjunctive normal form, with at most three literals in each clause, our task is to determine whether any assignment of values of the variables makes the formula true. Yes, this problem is NP-hard, which means that a polynomial algorithm is almost certainly impossible. Too bad; we have to solve the problem anyway.

The trivial solution is to try every possible assignment. We'll evaluate the running time of our 3SAT algorithms in terms of two variables: n is the number of variables in the input formula, and m is the number of clauses. There are 2^n possible assignments, and we can evaluate each assignment in $O(m)$ time, so the overall running time is $O(2^n m)$. Assuming no clause appears in the input formula more than once—a condition that we can easily enforce in polynomial time—then $m = O(n^3)$, so the running time is at most $O(2^n n^3)$.

Polynomial factors like n^3 are essentially noise when the overall running time is exponential, so from now on I'll write $\text{poly}(n)$ to represent any polynomial function of n; in other words, $\text{poly}(n) := n^{O(1)}$. For example, the trivial algorithm for 3SAT runs in time $O(2^n \text{poly}(n))$.

One Clause

Recall that all 3CNF formulas have the following recursive structure

> A 3CNF formula is either nothing
> or (a clause with three literals) $\wedge$ (a 3CNF formula)

Suppose we want to decide whether some 3CNF formula Φ with n variables is satisfiable. Of course this is trivial if Φ is the empty formula, so suppose

$$\Phi = (x \vee y \vee z) \wedge \Phi'$$

for some literals x, y, z and some 3CNF formula Φ'. By distributing the $\wedge$ across the $\vee$s, we can rewrite Φ as follows:

$$\Phi = (x \wedge \Phi') \vee (y \wedge \Phi') \vee (z \wedge \Phi')$$

For any boolean formula Ψ and any literal x, let $\Psi|x$ (pronounced "sigh given eks") denote the simpler boolean formula obtained by setting x to TRUE and simplifying as much as possible. If Ψ has n variables, then $\Psi|x$ has at most $n-1$ variables.

It's not hard to prove by induction (hint, hint) that $x \wedge \Psi = x \wedge (\Psi|x)$, and therefore

$$\Phi = (x \wedge (\Phi'|x)) \vee (y \wedge (\Phi'|y)) \vee (z \wedge (\Phi'|z)).$$

Thus, for any satisfying assignment for Φ, either

- x is true and $\Phi'|x$ is satisfiable, or
- y is true and $\Phi'|y$ is satisfiable, or
- z is true and $\Phi'|z$ is satisfiable.

Each of the smaller formulas $\Phi'|x$, $\Phi'|y$, and $\Phi'|z$ has at most $n-1$ variables. If we recursively check all three possibilities, we get the running time recurrence

$$T(n) \leq 3T(n-1) + \text{poly}(n),$$

whose solution is $O(3^n \text{poly}(n))$. So we've actually done *worse!*

But these three recursive cases are not mutually exclusive! If $\Phi'|x$ is *not* satisfiable, then x *must* be false in any satisfying assignment for Φ. So instead of recursively checking $\Phi'|y$ in the second step, we can check the even simpler formula $\Phi'|\bar{x}y = (\Phi'|\bar{x})|y$. Similarly, if $\Phi'|\bar{x}y$ is not satisfiable, then we know that y must be false in any satisfying assignment, so we can recursively check $\Phi'|\bar{x}\bar{y}z$ in the third step.

```
3SAT(Φ):
  if Φ = ∅
      return TRUE
  (x ∨ y ∨ z) ∧ Φ' ← Φ
  if 3SAT(Φ'|x)
      return TRUE
  if 3SAT(Φ'|x̄y)
      return TRUE
  return 3SAT(Φ'|x̄ȳz)
```

The running time off this algorithm obeys the recurrence

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \text{poly}(n),$$

where $\text{poly}(n)$ denotes the polynomial time required to simplify boolean formulas, handle control flow, move stuff into and out of the recursion stack, and so on.

How do we solve this recurrence? It isn't simple enough to let us guess an accurate solution and check it by induction. Fortunately, there is a standard mechanical solution for all backtracking recurrences of the form

$$T(n) \leq \text{poly}(n) + \sum_{i=1}^{k} T(n - a_i),$$

where $k, a_1, a_2, \ldots, a_k$ are constants. Assume without loss of generality that the constants a_i are indexed in non-increasing order: $a_1 \le a_2 \le \cdots \le a_k$. The fundamental theorem of algebra implies that the ***characteristic polynomial***

$$r^{a_k} - \sum_{i=1}^{k} r^{a_k - a_i}$$

(where r is a formal variable) has exactly k complex roots, some of which may be equal. Let λ be the root with largest magnitude. Then the solution to the recurrence is $\boldsymbol{T(n) = O(\lambda^n \operatorname{poly}(n))}$. This technique is described in more detail in the appendix; in particular, we can actually derived *exact closed form* solutions to backtracking-style recurrences from the complete sequence of characteristic roots and base cases, if the polynomial terms and base cases are specified exactly.

The recurrence $T(n) = T(n-1) + T(n-2) + T(n-3) + \operatorname{poly}(n)$ has characteristic polynomial $r^3 - r^2 - r - 1$, whose roots are approximately

$$1.83928676, \quad -0.419643337 + 0.60629073i, \quad -0.419643337 - 0.60629073i.$$

(Thank you, Wolfram Alpha!) The first root has larger magnitude than the other two, so we get the solution

$$T(n) = O(\lambda^n \operatorname{poly}(n)) = \boldsymbol{O(1.83929^n)}.$$

(Notice that we cleverly eliminated the polynomial noise by ever so slightly increasing the base of the exponent.)

Pure Literals

We can improve this algorithm further by eliminating *pure* literals from the formula before recursing. A literal x is ***pure*** in if it appears in the formula Φ but its negation $\bar{x}$ does not. It's not hard to prove (hint, hint) that if Φ has a satisfying assignment, then it has a satisfying assignment where every pure literal is true. If $\Phi = (x \vee y \vee z) \wedge \Phi'$ has no pure literals, then some clause in Φ contains the literal $\bar{x}$, so we can write

$$\Phi = (x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi'$$

for some literals u and v (each of which might be y, $\bar{y}$, z, or $\bar{z}$). It follows that the first recursive formula $\Phi|x$ contains the clause $(u \vee v)$. We can recursively eliminate the variables u and v just as we eliminated the variables y and x in the second and third cases of our previous algorithm:

$$\Phi|x = (u \vee v) \wedge \Phi'|x = (u \wedge \Phi'|xu) \vee (v \wedge \Phi'|x\bar{u}v).$$

Our new faster algorithm is shown in Figure **??**. The running time $T(n)$ of this algorithm satisfies the recurrence

$$T(n) = \operatorname{poly}(n) + \max \begin{Bmatrix} T(n-1) \\ 2T(n-2) + 2T(n-3) \end{Bmatrix}$$

```
FASTER3SAT(Φ):
  if Φ = ∅
      return TRUE
  if Φ has a pure literal x
      return FASTER3SAT(Φ|x)
  (x ∨ y ∨ z) ∧ (x̄ ∨ u ∨ v) ∧ Φ' ← Φ
  if FASTER3SAT(Φ|xu)
      return TRUE
  if FASTER3SAT(Φ|xūv)
      return TRUE
  if FASTER3SAT(Φ|x̄y)
      return TRUE
  return FASTER3SAT(Φ|x̄ȳz)
```

Figure B.1. A faster backtracking algorithm for 3SAT.

To solve this recurrence, we solve each case separately and take the larger of the two solutions. Here, the comparison is easy; the first case yields a *polynomial* bound, while the second case clearly yields an exponential bound, so for purposes of worst-case analysis, we can simplify the recurrence to $T(n) = O(\text{poly}(n)) + 2T(n-2) + 2T(n-3)$. The solution to this simpler recurrence is

$$T(n) = O(\lambda^n \text{ poly}(n)) = \boldsymbol{O(1.76930^n)},$$

where $\lambda \approx 1.76929235$ is the largest root of the characteristic polynomial $r^3 - 2r - 2$.

Local Search

We can get an even faster algorithm using a different backtracking strategy, called ***local search***. Instead of constructing a satisfying assignment recursively, we search for a path through the set of all assignments that ends at a satisfying assignment.

The ***Hamming distance*** between two different assignments is the number of variables where the two assignments differ.[2] Suppose we could somehow magically find an assignment that is *almost* satisfying, meaning that its Hamming distance from some satisfying assignment is small. Then instead of searching over the entire space of 2^n assignments, we only need to look at the assignments that are close to the assignment we already have. Moreover, we can guide the search by considering only the clauses that are *not* satisfied by our current assignment.

To make this concrete, let A be our current assignment, and suppose there is a satisfying assignment A^* that has Hamming distance r from A. Consider any clause of Φ

[2] Richard Hamming was an early giant of computer and information science, who earned PhD in mathematics at Illinois in 1942, programmed IBM calculating machines for the Manhattan project in 1945, and then went on to create the first error-correcting codes. Arguably the greatest recognition of his achievements is that his name is attached to the number of indices where two bit vectors differ.

that is *not* satisfied by A; at least one the three variables in that clause has a different value in A^*. Thus, if we change that particular variable's value, the resulting assignment has Hamming distance $r-1$ from A^*. Of course, we don't know which variable to change, so we have to try all three of them. And we don't know the target assignment A^*, so we just check at every stage of recursion whether our current assignment is satisfying. The resulting backtracking algorithm, shown in Figure **??**, runs in $O(3^r \text{ poly}(n))$ time.

```
LOCAL3SAT(Φ, A, r):
  if A satisfies Φ
      return TRUE
  if r = 0
      return FALSE
  C ← any clause of Φ that A does not satisfy
  x, y, z ← variables of C

  A[x] ← ¬A[x]
  if LOCAL3SAT(Φ, A, r − 1) = TRUE
      return TRUE

  A[x] ← ¬A[x]
  A[y] ← ¬A[y]
  if LOCAL3SAT(Φ, A_y, r − 1) = TRUE
      return TRUE

  A[y] ← ¬A[y]
  A[z] ← ¬A[z]
  return LOCAL3SAT(Φ, A_z, r − 1)
```

Figure B.2. A local search algorithm for 3SAT.

It must be emphasized that the local search algorithm is *not* explicitly considering every assignment within Hamming distance r of the starting assignment A.

Now we apply the following trivial observation: In any satisfying truth assignment, either at least half the variables are TRUE or at least half the variables are FALSE. Thus, *every* satisfying assignment has Hamming distance at most $n/2$ from either the all-TRUE assignment or the all-FALSE assignment. So using the local search strategy, we can solve 3SAT in $\boldsymbol{O(3^{n/2}\,\text{poly}(n)) = O(1.73206^n)}$ ***time***.

♥Random Starting Assignments

We can improve the local search algorithm further, at the expense of a small probability of a false negative, by repeatedly choosing *random* starting assignments and searching within a small radius of each.

First let's do some preliminary math. Let $\boldsymbol{V(r)}$ denote the number of assignments within Hamming distance r of any fixed assignment. Suppose we run the local search algorithm starting with a random assignment; let $P(r)$ denote the probability that the algorithm finds a satisfying assignment (assuming one exists). A simple counting

argument implies that $\boldsymbol{P(r) \geq V(r)/2^n}$. If we choose $N(r)$ random assignments and run the local search algorithm starting at each one, the probability of *not* finding a satisfying assignment (if one exists) is *at most* $(1-P(r))^{N(r)}$. Thus, if we set $\boldsymbol{N(r) = n2^n/V(r)}$, this failure probability is at most

$$(1-P(r))^{n2^n/V(r)} \leq \left(\left(1-\frac{V(r)}{2^n}\right)^{2^n/V(r)}\right)^n \leq e^{-n}.$$

The overall running time of our algorithm is

$$O(N(r)\,3^r \operatorname{poly}(n)) \;=\; O\!\left(\frac{n}{P(r)} \cdot 3^r \operatorname{poly}(n)\right) \;=\; O\!\left(\frac{2^n 3^r}{V(r)} \operatorname{poly}(n)\right).$$

To complete the analysis, we need to estimate the volume $V(r)$ and choose an appropriate value of r. For any constant $0 < \alpha < 1$, Stirling's approximation $n! \sim \sqrt{2\pi n}\,(n/e)^n$ implies the lower bound

$$V(\alpha n) \;=\; \sum_{i=0}^{r} \binom{n}{i} \;=\; \Omega\big(H(\alpha)^n/\sqrt{n}\big)$$

where $H(\alpha)$ is the *binary entropy* function:

$$H(\alpha) = \left(\frac{1}{\alpha}\right)^{\alpha} \left(\frac{1}{1-\alpha}\right)^{1-\alpha}.$$

Thus, if we set $r = \alpha n$, the running time of our algorithm becomes

$$O\!\left(\frac{2^n 3^{\alpha n}}{H(\alpha)^n} \operatorname{poly}(n)\right) \;=\; O\Big(\big(2(3\alpha)^{\alpha}(1-\alpha)^{1-\alpha}\big)^n \operatorname{poly}(n)\Big)$$

We can find the value of α that minimizes the base of the exponential part of this upper bound using a bit of calculus.

$$\begin{aligned}
\frac{d}{d\alpha} \ln\big((3\alpha)^{\alpha}(1-\alpha)^{1-\alpha}\big) &= \frac{d}{d\alpha}\big(\alpha \ln(3\alpha) + (1-\alpha)\ln(1-\alpha)\big) \\
&= \ln(3\alpha) - \ln(1-\alpha) = 0 \\
&\Longrightarrow\; \alpha = 1/4.
\end{aligned}$$

Finally, if we set $\alpha = 1/4$ (or equivalently, $r = n/4$), the algorithm runs in

$$O\Big(\big(2(3/4)^{1/4}(3/4)^{3/4}\big)^n \operatorname{poly}(n)\Big) \;=\; \boldsymbol{O\big((3/2)^n \operatorname{poly}(n)\big)}\ \textbf{\textit{time}}.$$

Further Extensions

Naturally, these approaches can be extended much further. Since 1998, at least fifteen different 3SAT algorithms have been published, each improving the running time by a small amount; while a few of these improvements use pure backtracking, most of them are based on some variant of local search. For example, in 1999, Uwe Schöning the running time of the randomized local search algorithm to $O((4/3)^n \operatorname{poly}(n))$, still with an exponentially small error probability, by replacing the backtracking search with a random walk of length $3n$. As of 2016, the fastest deterministic algorithm for 3SAT runs in $O(1.3303^n)$ time[3], and the fastest *randomized* algorithm runs in $O(1.30704^n)$ expected time[4], but there is good reason to believe that these are *not* the best possible.

There is a thriving industry of practical SAT solvers, which combine backtracking, local search, random restarts, and other more sophisticated algorithmic techniques with domain-specific heuristics. With these tools in hand, most real-world instances of SAT are actually easy to solve in practice; modern SAT solvers routinely resolve instances with millions of variables and clauses. SAT solvers have become an indispensable tool in several areas of computing, including hardware and software verification, operations research, and computational biology. The practically of SAT does not contradict the fact that the fastest algorithms require exponential time *in the worst case*; it just means that *the worst case* is rare in practice. Unfortunately, a proper discussion of practical techniques for SAT is beyond both the scope of this text and the expertise of its author.

B.2 Maximum Independent Set

Now suppose we are given an undirected graph G and are asked to find the size of the *largest independent set*, that is, the largest subset of the vertices of G with no edges between them. Once again, we have an obvious recursive algorithm: Try every subset of nodes, and return the largest subset with no edges. Expressed recursively, we obtain the following algorithm, where $N(v)$ denotes the *neighborhood* of v: the set containing v and all of its neighbors.

```
MAXIMUMINDSETSIZE(G):
  if G = ∅
      return 0
  else
      v ← any node in G
      withv ← 1 + MAXIMUMINDSETSIZE(G \ N(v))
      withoutv ← MAXIMUMINDSETSIZE(G \ {v})
      return max{withv, withoutv}.
```

[3]Kazuhisa Makino, Suguru Tamaki, Masaki Yamamoto. Derandomizing the HSSW algorithm for 3-SAT. *Algorithmica* 67(2):112–124 (2013)

[4]Timon Hertli. 3-SAT faster and simpler: Unique-SAT Bounds for PPSZ gold in general. *SIAM J. Comput.* 43(2):718–729 (2014).

Our algorithm exploits the fact that if an independent set contains v, then by definition it contains none of v's neighbors. In the worst case, v has no neighbors, so $G \setminus \{v\} = G \setminus N(v)$. Thus, the running time of this algorithm satisfies the recurrence $T(n) = 2T(n-1) + \text{poly}(n) = O(2^n \text{ poly}(n))$. Surprise, surprise.

This algorithm is mirroring a crude recursive upper bound for the number of *maximal* independent sets in a graph; an independent set is maximal if every vertex in G is either already in the set or a neighbor of a vertex in the set. If the graph is non-empty, then every maximal independent set either includes or excludes each vertex. Thus, the number of maximal independent sets satisfies the recurrence $M(n) \leq 2M(n-1)$, with base case $M(1) = 1$. We can easily guess and confirm the solution $M(n) \leq 2^n - 1$. The only subset that we aren't counting with this upper bound is the empty set!

Smarter Case Analysis

We can speed up our algorithm by making several careful modifications to avoid the worst case of the running-time recurrence.

Degree 0. If any vertex v has no neighbors, then $N(v) = \{v\}$, and both recursive calls consider a graph with $n-1$ nodes. But in this case, v is in *every* maximal independent set, so one of the recursive calls is redundant. On the other hand, if v has at least one neighbor, then $G \setminus N(v)$ has at most $n-2$ nodes. So now we have the following recurrence.

$$\begin{aligned} T(n) &\leq \text{poly}(n) + \max\left\{ \begin{array}{c} T(n-1) \\ \boldsymbol{T(n-1) + T(n-2)} \end{array} \right\} \\ &= O(1.61804^n) \end{aligned}$$

As before, the upper bound is derived by solving each case separately using characteristic polynomials and taking the larger of the two solutions. The first case gives us $T(n) = \text{poly}(n)$; the second case yields our old friends the Fibonacci numbers. Here and in later multi-case recurrences, I've typeset the worst case of the recurrence in **bold**.

Degree 1. We can improve this bound even more by examining the new worst case: some vertex v has exactly one neighbor w. In this case, either v or w appears in every maximal independent set. However, given any independent set that includes w, removing w and adding v creates another independent set of the same size. It follows that *some maximum independent set includes v*, so we don't need to search the graph $G \setminus \{v\}$, and the $G \setminus N(v)$ has at most $n-2$ nodes. On the other hand, if the degree of v is at least 2, then $G \setminus N(v)$ has at most $n-3$ nodes.

$$\begin{aligned} T(n) &\leq \text{poly}(n) + \max \left\{ \begin{array}{c} T(n-1) \\ T(n-2) \\ \boldsymbol{T(n-1) + T(n-3)} \end{array} \right\} \\ &= O(1.46558^n) \end{aligned}$$

The base of the exponent is the largest root of the characteristic polynomial $r^3 - r^2 - 1$.

Degree $\geq$ 3. Now the worst-case is a graph where every node has degree at least 2; we split this worst case into two subcases. If G has a node v with degree 3 or more, then $G \setminus N(v)$ has at most $n-4$ nodes. Otherwise (since we have already considered nodes of degree 0 and 1), *every* node in G has degree 2. Let u, v, w be a path of three nodes in G (possibly with u adjacent to w). In any maximal independent set, either v is present and u, w are absent, or u is present and its two neighbors are absent, or w is present and its two neighbors are absent. In all three cases, we recursively count maximal independent sets in a graph with $n-3$ nodes.

$$\begin{aligned} T(n) &\leq \text{poly}(n) + \max \left\{ \begin{array}{c} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \\ \boldsymbol{3T(n-3)} \end{array} \right\} \\ &= O(3^{n/3} \text{poly}(n)) \\ &= O(1.44225^n) \end{aligned}$$

The base of the exponent is $\sqrt[3]{3}$, the largest root of the characteristic polynomial $r^3 - 3$. The third case would give us a bound of $O(1.38028^n)$, where the base is the largest root of the characteristic polynomial $r^4 - r^3 - 1$.

Degree 2. Now the worst case for our algorithm is a graph with an extraordinarily special structure: *Every node has degree 2*. In other words, every component of G is a cycle. It is easy to prove that the largest independent set in a cycle of length k has size $\lfloor k/2 \rfloor$. So we can handle this case directly in polynomial time, with no recursion at all!

$$\begin{aligned} T(n) &\leq \text{poly}(n) + \max \left\{ \begin{array}{c} T(n-1) \\ T(n-2) \\ \boldsymbol{T(n-1) + T(n-4)} \end{array} \right\} \\ &= O(1.38028^n) \end{aligned}$$

Again, the base of the exponential running time is the largest root of the characteristic polynomial $r^4 - r^3 - 1$. The final algorithm after all these improvements is shown in Figure **??**.

```
MaximumIndSetSize(G):
  if G = ∅
      return 0

  else if G has a node v with degree 0 or 1
      return 1 + MaximumIndSetSize(G \ N(v))        ⟨⟨≤ n − 1⟩⟩

  else if G has a node v with degree greater than 2
      withv ← 1 + MaximumIndSetSize(G \ N(v))       ⟨⟨≤ n − 4⟩⟩
      withoutv ← MaximumIndSetSize(G \ {v})         ⟨⟨≤ n − 1⟩⟩
      return max{withv, withoutv}

  else ⟨⟨every node in G has degree 2⟩⟩
      total ← 0
      for each component of G
          k ← number of vertices in the component
          total ← total + ⌊k/2⌋
      return total
```

Figure B.3. A fast backtracking algorithm for MaxIndependentSet

Further Improvements

As with 3Sat, there are faster but increasingly complex algorithms for MaxIndependentSet. For example, in 1977, Bob Tarjan and Anthony Trojanowski published a complex backtracking algorithm that considers several dozen cases, each with its own running time recurrence. Most of these cases are clearly dominated by others; for example, even without solving the recurrences, it's easy to see that the recurrence $T(n) \leq \text{poly}(n) + T(n-4) + T(n-5)$ has a smaller solution than $T(n) \leq \text{poly}(n) + T(n-3) + T(n-5)$. After discarding all such dominated cases, Tarjan and Trojanowski are left with the rather impressive recurrence shown in Figure **??**. The solution to this recurrence is $T(n) = O(1.25603^2) = O(2^{n/3})$, where the base of the exponent is the largest root of the characteristic polynomial $r^{10} - r^8 - r^2 - 2$ of the third case (in bold).

As of 2016, the fastest published algorithm for computing maximum independent sets runs in $O(1.2114^n)$ time[5]. However, in an unpublished technical report, Mike Robson describes a *computer-generated* algorithm that runs in $O(2^{n/4}\,\text{poly}(n)) = O(1.1889^n)$ time; just the description of this algorithm requires more than 15 pages.[6]

B.3 Dynamic Programming

Warmup: Bellman-Held-Karp TSP in $O(2^n\,\text{poly}(n))$ time. (Leave as exercise?) ★★★

★★★

[5]Nicolas Bourgeois, Bruno Escoffier, Vangelis Paschos, and Johan M. M. can Rooij. Fast algorithms for max independent set. *Algorithmica* 62(1):382–415, 2010.

[6]John Michael Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical report 1251-01, LaBRI, 2001. ⟨http://www.labri.fr/perso/robson/mis/techrep.ps⟩.

$$T(n) \le \text{poly}(n) + \max \left\{ \begin{array}{c} T(n-2)+T(n-6) \\ T(n-2)+2T(n-8) \\ \boldsymbol{T(n-2)+T(n-8)+2T(n-10)} \\ T(n-3)+T(n-5) \\ T(n-4)+T(n-6)+T(n-8) \\ T(n-4)+2T(n-7) \\ T(n-4)+T(n-8)+3T(n-11) \\ T(n-4)+2T(n-8)+T(n-10) \\ T(n-4)+T(n-9)+2T(n-12) \\ T(n-4)+4T(n-10) \\ T(n-5)+4T(n-9) \\ T(n-5)+6T(n-11)+4T(n-14)+T(n-17) \end{array} \right\}$$

Figure B.4. The (simplified!) running time recurrence for Tarjan and Trojanowski's independent set algorithm.

3-coloring in $O(2^n \text{ poly}(n))$ time. Let $OPT(X) =$ TRUE if there is a 3-coloring in which every vertex in X is red. Then

$$OPT(X) = \begin{cases} \text{FALSE} & \text{if } X \text{ is not an independent set} \\ \text{TRUE} & \text{if } G[V \setminus X] \text{ is bipartite} \\ \bigvee_{v \in V \setminus X} OPT(X \cup \{v\}) & \text{otherwise} \end{cases}$$

Lawlar improved the running time to $O(3^{n/3} \text{ poly}(n))$ by observing that we only need to care about $OPT(X)$ when X is a *maximal* independent set. There are $O(3^{n/3})$ maximal independent sets, which can be listed in $O(3^{n/3} \text{ poly}(n))$ time.

Alternatively, 3-coloring in $O(2^n \text{ poly}(n))$ time by backtracking through the vertices in some fixed whatever-first order. Each vertex must be colored differently from is parent (and possibly other vertices) and without loss of generality, the first tow nodes are red and green, respectively. So we consider only 2^{n-2} possible colorings, but memoization doesn't help. (Can we do better by looking in max-adjacency order?)

★★★

Lawler's algorithm for chromatic number in $O((1+3^{1/3})^n \text{ poly } n)$ time: Let $OPT(X)$ denote the chromatic number of the subgraph $G[X]$ induced by the vertex subset $X \subseteq V$. Then

$$OPT(X) = \begin{cases} 0 & \text{if } X = \varnothing \\ 1 + \min \left\{ OPT(X \setminus I) \;\middle|\; \text{maximal ind. set } I \text{ in } G[X] \right\} & \text{otherwise} \end{cases}$$

There are $O(3^{k/3})$ maximal independent sets of size k, which can be listed in $O(3^{k/3} \text{ poly}(n))$ time.

★★★

Other techniques to include?

- Horowitz-Sahni partitioning to solve SUBSETSUM in $O(2^{n/2}\,\text{poly}(n))$ time? (See also: 3SUM)
- Decision via counting via Inclusion-Exclusion?

Exercises

1. (a) Prove that any n-vertex graph has at most $3^{n/3}$ maximal independent sets. *[Hint: Modify the MAXIMUMINDSETSIZE algorithm so that it lists all maximal independent sets.]*

   (b) Describe an n-vertex graph with exactly $3^{n/3}$ maximal independent sets, for every integer n that is a multiple of 3.

2. (a) Describe an algorithm for kSAT whose running time satisfies the recurrence

$$T(n) = 2T(n-1) - T(n-k+1) + \text{poly}(n).$$

   (b) Describe an algorithm for kSAT whose running time satisfies the recurrence

$$T(n) = T(n-1) + 2T(n-2) - 2T(n-k+1) + \text{poly}(n).$$

   [Hint: Modify the backtracking algorithms for 3SAT described in the text. No, your algorithm will not make negative recursive calls.]

♥3. Describe an algorithm that solves 3SAT in $O(\phi^n\,\text{poly}(n))$ time, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]*

4. Describe an algorithm to determine whether a graph is 4-colorable in $O(2^n\,\text{poly}(n))$ time. *[Hint: Consider red and green together, but separately from blue and yellow.]*

5. (a) Prove the following upper bound for all constant $0 < \alpha < 1$:

$$\binom{n}{\alpha n} \le \sum_{i=0}^{\alpha n} \binom{n}{i} < \left(\left(\frac{1}{\alpha}\right)^{\alpha} \left(\frac{1}{1-\alpha}\right)^{1-\alpha} \right)^n$$

   [Hint: Expand the expression $(\alpha + (1-\alpha))^n = 1$ using the binomial theorem.]

   (b) Prove the following lower bound for all constant $0 < \alpha < 1$:

$$\binom{n}{\alpha n} \ge \left(\left(\frac{1}{\alpha}\right)^{\alpha} \left(\frac{1}{1-\alpha}\right)^{1-\alpha} \right)^n \cdot \Omega(1/\sqrt{n})$$

   [Hint: Use Stirling's approximation: $\sqrt{2\pi n}\,(n/e)^n \le n! \le 2\sqrt{\pi n}\,(n/e)^n$.]

Need a few more exercises.

★★★

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

C

Dynamic Programming for Formal Languages and Automata

[Read Chapter 3 first.]

Status: Unfinished

This chapter describes dynamic programming algorithms for several problems involving formal languages and finite-state automata. Although I have strived for a self-contained presentation, this material will likely make sense only to people who are already somewhat familiar with formal languages *and* dynamic programming.

C.1 DFA Minimization

Write this. ★★★

C.2 NFA Acceptance

Recall that a ***nondeterministic finite-state automaton***—or ***NFA*** for short—can be described as a directed graph, whose edges are called *states* and whose edges have *labels*

drawn from a finite set Σ called the *alphabet*. Every NFA has a designated *start* state and a subset of *accepting* states. Every walk in this graph has a label, which is a string formed by concatenating the labels of the edges in the walk. A string w is *accepted* by an NFA if and only if there is a walk from the start state to one of the accepting states whose label is w.

More formally (or at least, more symbolically), an NFA consists of a finite set Q of states, a start state $s \in Q$, a set of accepting states $A \subseteq Q$, and a transition function $\delta : Q \times \Sigma \to 2^Q$. We recursively extend the transition function to a function $\delta^* : Q \times \Sigma^* \to 2^Q$ over strings by defining

$$\delta^*(q, w) = \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \bigcup_{r \in \delta(q,a)} \delta^*(r, x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Our NFA ***accepts*** string w if and only if the set $\delta^*(s, w)$ contains at least one accepting state.

We can express this acceptance criterion more directly in terms of the original transition function δ as follows. Let *Accepts?*$(q, w) =$ TRUE if our NFA would accept string w if it started in state q (instead of the usual start state s), and *Accepts?*$(q, w) =$ FALSE otherwise. The function *Accepts?* has the following recursive definition:

$$\textit{Accepts?}(q, w) := \begin{cases} \textsc{True} & \text{if } w = \varepsilon \text{ and } q \in A \\ \textsc{False} & \text{if } w = \varepsilon \text{ and } q \notin A \\ \bigvee_{r \in \delta(q,a)} \textit{Accepts?}(r, x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Then our NFA accepts w if and only if *Accepts?*$(s, w) =$ TRUE.

Backtracking

In the magical world of non-determinism, we can imagine that the NFA always makes the right decision when faced with multiple transitions, or perhaps spawns off an independent parallel thread for each possible choice. Alas, real computers are neither clairvoyant nor (despite the increasing use of multiple cores) infinitely parallel. To simulate the NFA's behavior directly, we must recursively explore the consequences of each choice explicitly.

Before we can turn our recursive definition(s) into an algorithm, we need to nail down the precise input representation. Let's assume, without loss of generality, that the alphabet is $\Sigma = \{1, 2, \ldots, |\Sigma|\}$, the state set is $Q = \{1, 2, \ldots, |Q|\}$, the start state is state 1, and our input consists of three arrays:

- A boolean array $A[1..|Q|]$, where $A[q] =$ TRUE if and only if $q \in A$.

- A boolean array $\delta[1..|Q|, 1..|\Sigma|, 1..|Q|]$, where $\delta[p, a, q] = \textsc{True}$ if and only if $p \in \delta(q, a)$.
- An array $w[1..n]$ of symbols, representing the input string.

Fixing the input string $w[1..n]$ lets us simplify the definition of the acceptance function slightly. For any state q and index i, define $\mathit{Accepts?}(q, i) = \textsc{True}$ if the NFA accepts the suffix $w[i..n]$ starting in state q, and $\mathit{Accepts?}(q, i) = \textsc{False}$ otherwise.

$$\mathit{Accepts?}(q, i) := \begin{cases} \textsc{True} & \text{if } i > n \text{ and } q \in A \\ \textsc{False} & \text{if } i > n \text{ and } q \notin A \\ \displaystyle\bigvee_{r \in \delta(q,a)} \mathit{Accepts?}(r, i+1) & \text{otherwise} \end{cases}$$

In particular, the NFA accepts w if and only if $\mathit{Accepts?}(s, 1) = \textsc{True}$. This is only a minor notational change from the previous definition, but it makes the recursive calls more efficient (passing just an integer instead of a string) and eventually easier to memoize. Our new recursive definition translates directly into the following backtracking algorithm.

```
ACCEPTS?(q, i):
  if i > n
      return A[q]
  for r ← 1 to |Q|
      if δ[q, w[i], r] and ACCEPTS?(r, i + 1)
          return TRUE
  return FALSE
```

Dynamic Programming

The previous algorithm runs in $O(|Q|^n)$ time in the worst case; fortunately, everything is set up to quickly derive a faster dynamic programming solution. The *Accepts?* function can be memoized into a two-dimensional array $\mathit{Accepts?}[1..|Q|, 1..n+1]$. Each entry $\mathit{Accepts?}[q, i]$ depends only on entries of the form $\mathit{Accepts?}[r, i+1]$, so we can fill the memoization table by considering indices i in decreasing order in the outer loop, and states q in arbitrary order in the inner loop. Evaluating each entry $\mathit{Accepts?}[q, i]$ requires $O(|Q|)$ time, using an even deeper loop over all states r, and there are $O(n|Q|)$ such entries. Thus, the entire dynamic programming algorithm requires $\boldsymbol{O(n|Q|^2)}$ ***time***.

```
NFAACCEPTS?(A[1..|Q|], δ[1..|Q|, 1..|Σ|, 1..|Q|], w[1..n]):
  for q ← 1 to |Q|
      Accepts?[q, n+1] ← A[q]
  for i ← n down to 1
      for q ← 1 to |Q|
          Accepts?[q, i] ← FALSE
          for r ← 1 to |Q|
              if δ[q, w[i], r] and Accepts?[r, i+1]
                  Accepts?[q, i] ← TRUE
  return Accepts?[1, 1]
```

C.3 Regular Expression Matching

★★★

Write this. Assume given an expression tree.

C.4 CFG Parsing

Another problem from formal languages that can be solved via dynamic programming is the ***parsing*** problem for context-free languages. Given a string w and a context-free grammar G, does w belong to the language generated by G? Recall that a ***context-free grammar*** over the alphabet Σ consists of a finite set Γ of *non-terminals* (disjoint from Σ) and a finite set of *production rules* of the form $A \to w$, where A is a nonterminal and w is a string over $\Sigma \cup \Gamma$. The ***length*** of a context free grammar is the number of production rules.

Real-world applications of parsing normally require more information than just a single bit. For example, compilers require parsers that output a *parse tree* of the input code; some natural language applications require the *number* of distinct parse trees for a given string; others assign probabilities to the production rules and then ask for the *most likely* parse tree for a given string. However, once we have an algorithm for the decision problem, it it not hard to extend it to answer these more general questions.

For any nonterminal A and any string x, define $\mathit{Gen?}(A, x) = \text{TRUE}$ if x can be derived from A and $\mathit{Gen?}(A, x) = \text{FALSE}$ otherwise. At first glance, it seems that the production rules of the CFL immediately give us a (rather complicated) recursive definition for this function. Unfortunately, there are a few subtle problems.[1]

- Consider the context-free grammar $S \to \varepsilon \mid SS \mid (S)$ that generates all properly balanced strings of parentheses. The most straightforward recursive algorithm for $\mathit{Gen?}(S, w)$ recursively checks whether $x \in L(S)$ and $y \in L(S)$, for *every* possible partition $w = x \bullet y$, *including the trivial partitions* $w = \varepsilon \bullet w$ *and* $w = w \bullet \varepsilon$. Thus, $\mathit{Gen?}(S, w)$ can call itself, leading to an infinite loop.

[1] Similar subtleties arise in induction proofs about context-free grammars.

- Consider another grammar that includes the productions $S \to A$, $A \to B$, and $B \to S$, possibly among others. The "obvious" recursive algorithm for $Gen?(S, w)$ must call $Gen?(A, w)$, which calls $Gen?(B, w)$, which calls $Gen?(S, w)$, and we are again in an infinite loop.

To avoid these issues, we will make the simplifying assumption that our input grammar is in ***Chomsky normal form***, which means that it has the following special structure:

- The starting non-terminal S does not appear on the right side of any production rule.
- The grammar *may* include the production rule $S \to \varepsilon$, where S is the starting non-terminal, but does not contain the rule $A \to \varepsilon$ for any other non-terminal $A \neq S$.
- Otherwise, every production rule has the form $A \to BC$ (two non-terminals) or $A \to a$ (one terminal).

Any context-free grammar can be converted into an equivalent grammar Chomsky normal form; moreover, if the original grammar has length L, an equivalent CFG grammar of length $O(L^2)$ can be computed in $O(L^2)$ time. The conversion algorithm is fairly complex, and we haven't yet seen all the algorithmic tools needed to understand it; for purposes of this chapter, it's enough to know that such an algorithm exists. For example, the language of all properly balanced strings of parentheses is generated by the CNF grammar

$$S \to \varepsilon \mid AA \mid BC \qquad A \to AA \mid BC \qquad B \to LA \qquad C \to RA \qquad L \to (\qquad R \to)$$

⟨⟨This is incorrect.⟩⟩ ⋘⋘⋘

With this simplifying assumption in place, the function *Gen?* has a relatively straightforward recursive definition.

$$Gen?(A, x) = \begin{cases} \textsc{True} & \text{if } |x| \leq 1 \text{ and } A \to x \\ \textsc{False} & \text{if } |x| \leq 1 \text{ and } A \not\to x \\ \displaystyle\bigvee_{A \to BC} \bigvee_{y \bullet z = x} Gen?(B, y) \wedge Gen?(C, z) & \text{otherwise} \end{cases}$$

The first two cases take care of terminal productions $A \to a$ and the ε-production $S \to \varepsilon$ (if the grammar contains it). Here the notation $A \not\to x$ means that $A \to x$ is *not* a production rule in the given grammar. In the third case, for all production rules $A \to BC$, and for all ways of splitting x into a *non-empty* prefix y and a *non-empty* suffix z, we recursively check whether $y \in L(B)$ and $z \in L(C)$. Because we pass strictly smaller strings in the second argument of these recursive calls, every branch of the recursion tree eventually terminates.

This recurrence was transformed into a dynamic programming algorithm by Tadao Kasami in 1965, and again independently by Daniel Younger in 1967, and again independently by John Cocke in 1970, so of course the resulting algorithm is known as "Cocke-Younger-Kasami", or more commonly ***the CYK algorithm***, with the names listed in *reverse* chronological order.

We can derive the CYK algorithm from the previous recurrence as follows. As usual for recurrences involving strings, we modify the function to accept index arguments instead of strings, to ease memoization. Fix the input string w, and then let $Gen?(A, i, j) = \text{TRUE}$ if and only if the substring $w[i..j]$ can be derived from non-terminal A. Now our earlier recurrence can be rewritten as follows:

$$Gen?(A,i,j) = \begin{cases} \text{TRUE} & \text{if } i = j \text{ and } A \to w[i] \\ \text{FALSE} & \text{if } i = j \text{ and } A \not\to w[i] \\ \displaystyle\bigvee_{A \to BC} \bigvee_{k=i}^{j-1} Gen?(B,i,k) \wedge Gen?(C,k+1,j) & \text{otherwise} \end{cases}$$

Then w lies in the language of the grammar if and only if either $Gen?(A, 1, n) = \text{TRUE}$, or $w = \varepsilon$ and the grammar includes the production $S \to \varepsilon$.

We can memoize the function *Gen?* into a three-dimensional boolean array $Gen[1..|\Gamma|, 1..n, 1..n]$, where the first dimension is indexed by the non-terminals Γ in the input grammar. Each entry $Gen[A, i, j]$ in this array depends on entries of the form $Gen[\cdot, i, k]$ for some $k < j$, or $Gen[\cdot, k+1, j]$ for some $k \geq i$. Thus, we can fill the array by increasing j and decreasing i in two outer loops, and considering non-terminals A in arbitrary order in the inner loop. The resulting dynamic programming algorithm runs in ***O(n³L) time***, where L is the length of the input grammar.

```
CYKPARSE(w, G):
  if w = ε
      if G contains the production S → ε
          return TRUE
      else
          return FALSE
  for i ← 1 to n
      for all non-terminals A
          if G contains the production A → w[i]
              Gen[A, i, i] ← TRUE
          else
              Gen[A, i, i] ← FALSE
  for j ← 1 to n
      for i ← n down to j + 1
          for all non-terminals A
              Gen[A, i, j] ← FALSE
          for all production rules A → BC
              for k ← i to j − 1
                  if Gen[B, i, k] and Gen[C, k + 1, j]
                      Gen[A, i, j] ← TRUE
  return Gen[S, 1, n]
```

Exercises

Add some exercises!

It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde, "A Few Maxims for the Instruction Of The Over-Educated" (1894)

Ninety percent of science fiction is crud. But then, ninety percent of everything is crud, and it's the ten percent that isn't crud that is important.

— [Theodore] Sturgeon's Law (1953)

Dis-moi ce que tu manges, je te dirai ce que tu es.

— Jean Anthelme Brillat-Savarin, *Physiologie du Gout* (1825)

D

Advanced Dynamic Programming

[Read Chapter 3 first.]

Status: Unfinished.

Replace Hirschberg with Chowdhury-Ramachandran Four Russians?

Dynamic programming is a powerful technique for efficiently solving recursive problems, but it's hardly the end of the story. In many cases, once we have a basic dynamic programming algorithm in place, we can make further improvements to bring down the running time or the space usage. We saw one example in the Fibonacci number algorithm. Buried inside the naïve iterative Fibonacci algorithm is a recursive problem—computing a power of a matrix—that can be solved more efficiently by dynamic programming techniques—in this case, repeated squaring.

D.1 Saving Space: Divide and Conquer

Just as we did for the Fibonacci recurrence, we can reduce the space complexity of our edit distance algorithm from $O(mn)$ to $O(m+n)$ by only storing the current and previous rows of the memoization table. This "sliding window" technique provides an easy space improvement for most (but *not* all) dynamic programming algorithm.

Unfortunately, this technique seems to be useful only if we are interested in the *cost* of the optimal edit sequence, not if we want the optimal edit sequence itself. By throwing away most of the table, we apparently lose the ability to walk backward through the table to recover the optimal sequence.

Fortunately for memory-misers, in 1975 Dan Hirschberg discovered a simple divide-and-conquer strategy that allows us to compute the optimal edit sequence in $O(mn)$ time, using just $O(m+n)$ space. The trick is to record not just the edit distance for each pair of prefixes, but also a single position in the middle of the optimal editing sequence for that prefix. Specifically, any optimal editing sequence that transforms $A[1..m]$ into $B[1..n]$ can be split into two smaller editing sequences, one transforming $A[1..m/2]$ into $B[1..h]$ for some integer h, the other transforming $A[m/2+1..m]$ into $B[h+1..n]$.

To compute this breakpoint h, we define a second function $Half(i,j)$ such that some optimal edit sequence from $A[1..i]$ into $B[1..j]$ contains an optimal edit sequence from $A[1..m/2]$ to $B[1..Half(i,j)]$. We can define this function recursively as follows:

$$Half(i,j) = \begin{cases} \infty & \text{if } i < m/2 \\ j & \text{if } i = m/2 \\ Half(i-1,j) & \text{if } i > m/2 \text{ and } Edit(i,j) = Edit(i-1,j)+1 \\ Half(i,j-1) & \text{if } i > m/2 \text{ and } Edit(i,j) = Edit(i,j-1)+1 \\ Half(i-1,j-1) & \text{otherwise} \end{cases}$$

(Because there there may be more than one optimal edit sequence, this is not the only correct definition.) A simple inductive argument implies that $Half(m,n)$ is indeed the correct value of h. We can easily modify our earlier algorithm so that it computes $Half(m,n)$ at the same time as the edit distance $Edit(m,n)$, all in $O(mn)$ time, using only $O(m)$ space.

Edit		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Half		A	L	G	O	R	I	T	H	M
	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
A	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
L	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
T	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
R	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
U	0	1	2	3	4	5	6	7	8	9
I	0	1	2	3	4	5	**5**	5	5	5
S	0	1	2	3	4	5	5	5	5	5
T	0	1	2	3	4	5	5	**5**	5	5
I	0	1	2	3	4	5	**5**	5	5	5
C	0	1	2	3	4	5	5	5	5	5

Finally, to compute the optimal editing sequence that transforms A into B, we recursively compute the optimal sequences transforming $A[1..m/2]$ into $B[1..Half(m,n)]$ and transforming $A[m/2+1..m]$ into $B[Half(m,n)+1..n]$. The recursion bottoms out

when one string has only constant length, in which case we can determine the optimal editing sequence in linear time using our old dynamic programming algorithm. The running time of the resulting algorithm satisfies the following recurrence:

$$T(m,n) = \begin{cases} O(n) & \text{if } m \le 1 \\ O(m) & \text{if } n \le 1 \\ O(mn) + T(m/2,h) + T(m/2,n-h) & \text{otherwise} \end{cases}$$

It's easy to prove inductively that $T(m,n) = O(mn)$, no matter what the value of h is. Specifically, the entire algorithm's running time is at most twice the time for the initial dynamic programming phase.

$$\begin{aligned} T(m,n) &\le \alpha mn + T(m/2,h) + T(m/2,n-h) \\ &\le \alpha mn + 2\alpha mh/2 + 2\alpha m(n-h)/2 \qquad \text{[inductive hypothesis]} \\ &= 2\alpha mn \end{aligned}$$

A similar inductive argument implies that the algorithm uses only $O(n+m)$ space.

Compare with Chowdhury and Ramachandran [SODA 2006]: Compute values on boundary via divide-and-conquer (instead of by sweeping). Reconstruct sequence recursively as well. Subproblem: given a submatrix with values on top/left boundary, and target indices on bottom/right boundary, report LCS leading to target and top/left index.

Also, this isn't actually how Hirschberg described "Hirschberg's algorithm".

★★★

Hirschberg's divide-and-conquer trick can be applied to almost any dynamic programming problem to obtain an algorithm to construct an optimal *structure* (in this case, the cheapest edit sequence) within the same space and time bounds as computing the *cost* of that optimal structure (in this case, edit distance). For this reason, we will almost always ask you for algorithms to compute the cost of some optimal structure, not the optimal structure itself.

D.2 Saving Time: Sparseness

In many applications of dynamic programming, we are faced with instances where almost every recursive subproblem will be resolved exactly the same way. We call such instances *sparse*. For example, we might want to compute the edit distance between two strings that have few characters in common, which means there are few "free" substitutions anywhere in the table. Most of the table has exactly the same structure. If we can reconstruct the entire table from just a few key entries, then why compute the entire table?

To better illustrate how to exploit sparseness, let's consider a simplification of the edit distance problem, in which substitutions are not allowed (or equivalently, where a

substitution counts as two operations instead of one). Now our goal is to maximize the number of "free" substitutions, or equivalently, to find the *longest common subsequence* of the two input strings.

Fix the two input strings $A[1..n]$ and $B[1..m]$. For any indices i and j, let $LCS(i,j)$ denote the length of the longest common subsequence of the prefixes $A[1..i]$ and $B[1..j]$. This function can be defined recursively as follows:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } A[i] = B[j] \\ \max\{LCS(i, j-1),\ LCS(i-1, j)\} & \text{otherwise} \end{cases}$$

This recursive definition directly translates into an $O(mn)$-time dynamic programming algorithm.

LCS	«	A	L	G	O	R	I	T	H	M	S	»
«	**0**	0	0	0	0	0	0	0	0	0	0	0
A	0	**1**	1	1	1	1	1	1	1	1	1	1
L	0	1	**2**	2	2	2	2	2	2	2	2	2
T	0	1	2	2	2	2	2	**3**	3	3	3	3
R	0	1	2	2	2	**3**	3	3	3	3	3	3
U	0	1	2	2	3	3	3	3	3	3	3	3
I	0	1	2	2	3	3	**4**	4	4	4	4	4
S	0	1	2	2	3	3	4	4	4	4	**5**	5
T	0	1	2	2	3	3	4	**5**	5	5	5	5
I	0	1	2	2	3	3	**4**	5	5	5	5	5
C	0	1	2	2	3	3	4	5	5	5	5	5
»	0	1	2	2	3	3	4	5	5	5	5	**6**

Figure D.1. The *LCS* memoization table for the strings ALGORITHMS and ALTRUISTIC; the brackets « and » are sentinel characters. Match points are indicated in red.

Call an index pair (i,j) a ***match point*** if $A[i] = B[j]$. In some sense, match points are the only "interesting" locations in the memoization table. Given a list of the match points, we can reconstruct the entire table using the following recurrence:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = j = 0 \\ \max\left\{LCS(i',j') \;\middle|\; A[i'] = B[j'] \text{ and } i' < i \text{ and } j' < j\right\} + 1 & \text{if } A[i] = B[j] \\ \max\left\{LCS(i',j') \;\middle|\; A[i'] = B[j'] \text{ and } i' \le i \text{ and } j' \le j\right\} & \text{otherwise} \end{cases}$$

(Notice that the inequalities are strict in the second case, but not in the third.) To simplify boundary issues, we add unique sentinel characters $A[0] = B[0]$ and $A[m+1] = B[n+1]$ to both strings (brackets « and » in the example above). These sentinels ensure that the sets on the right side of the recurrence equation are non-empty, and that we *only* have to consider match points to compute $LCS(m,n) = LCS(m+1, n+1) - 1$.

If there are K match points, we can actually compute them in $O(m \log m + n \log n + K)$ time. Sort the characters in each input string, remembering the original index of each character, and then essentially merge the two sorted arrays, as follows:

```
FindMatches(A[1..m], B[1..n]):
  for i ← 1 to m:  I[i] ← i
  for j ← 1 to n:  J[j] ← j

  sort A and permute I to match
  sort B and permute J to match

  i ← 1;  j ← 1
  while i < m and j < n
      if A[i] < B[j]
          i ← i + 1
      else if A[i] > B[j]
          j ← j + 1
      else                          ⟨⟨Found a match!⟩⟩
          ii ← i
          while A[ii] = A[i]
              jj ← j
              while B[jj] = B[j]
                  report (I[ii], J[jj])
                  jj ← jj + 1
              ii ← i + 1
          i ← ii;  j ← jj
```

To efficiently evaluate our modified recurrence, we once again turn to dynamic programming. We consider the match points in lexicographic order—the order they would be encountered in a standard row-major traversal of the $m \times n$ table—so that when we need to evaluate $LCS(i, j)$, all match points (i', j') with $i' < i$ and $j' < j$ have already been evaluated.

```
SparseLCS(A[1..m], B[1..n]):
  Match[1..K] ← FindMatches(A, B)
  Match[K + 1] ← (m + 1, n + 1)        ⟨⟨Add end sentinel⟩⟩
  Sort M lexicographically
  for k ← 1 to K
      (i, j) ← Match[k]
      LCS[k] ← 1                       ⟨⟨From start sentinel⟩⟩
      for ℓ ← 1 to k − 1
          (i', j') ← Match[ℓ]
          if i' < i and j' < j
              LCS[k] ← min{LCS[k], 1 + LCS[ℓ]}
  return LCS[K + 1] − 1
```

The overall running time of this algorithm is $O(m \log m + n \log n + K^2)$. So as long as $K = o(\sqrt{mn})$, this algorithm is actually faster than naïve dynamic programming. With

some additional work, the running time can be further improved to $O(m \log m + n \log n + K \log K)$.

D.3 Saving Time: Monotonicity

Recall the optimal binary search tree problem from the previous lecture. Given an array $F[1..n]$ of access frequencies for n items, the problem it to compute the binary search tree that minimizes the cost of all accesses. A relatively straightforward dynamic programming algorithm solves this problem in $O(n^3)$ time.

As for longest common subsequence problem, the algorithm can be improved by exploiting some structure in the memoization table. In this case, however, the relevant structure isn't in the table of costs, but rather in the table used to reconstruct the actual optimal tree. Let $OptRoot[i, j]$ denote the index of the root of the optimal search tree for the frequencies $F[i..j]$; this is always an integer between i and j. Donald Knuth proved the following nice monotonicity property for optimal subtrees: If we move either end of the subarray, the optimal root moves in the same direction or not at all. More formally:

$$\boxed{OptRoot[i, j-1] \le OptRoot[i, j] \le OptRoot[i+1, j] \text{ for all } i \text{ and } j.}$$

In other words, every row and column in the array $OptRoot[1..n, 1..n]$ is sorted. This (nontrivial!) observation suggests the following more efficient algorithm:

```
FasterOptimalSearchTree(f[1..n]):
  InitF(f[1..n])
  for i ← 1 downto n
      OptCost[i, i-1] ← 0
      OptRoot[i, i-1] ← i
  for d ← 0 to n
      for i ← 1 to n
          ComputeCostAndRoot(i, i+d)
  return OptCost[1, n]

ComputeCostAndRoot(i, j):
  OptCost[i, j] ← ∞

  for r ← OptRoot[i, j-1] to OptRoot[i+1, j]
      tmp ← OptCost[i, r-1] + OptCost[r+1, j]
      if OptCost[i, j] > tmp
          OptCost[i, j] ← tmp
          OptRoot[i, j] ← r

  OptCost[i, j] ← OptCost[i, j] + F[i, j]
```

It's not hard to see that the loop index r increases monotonically from 1 to n during each iteration of the *outermost* for loop of FasterOptimalSearchTree. Consequently, the total cost of all calls to ComputeCostAndRoot is only $O(n^2)$.

If we formulate the problem slightly differently, this algorithm can be improved even further. Suppose we require the optimum *external* binary tree, where the keys $A[1..n]$ are all stored at the leaves, and intermediate pivot values are stored at the internal nodes. An algorithm discovered by Ching Hu and Alan Tucker[1] computes the optimal binary search tree in this setting in only $O(n \log n)$ time!

D.4 Saving Time: More Monotoniticy

Knuth's algorithm can be significantly generalized by considering a more subtle form of monotonicity in the *cost* array. A common (but often implicit) operation in many dynamic programming algorithms is finding the minimum element in every row of a two-dimensional array. For example, consider a single iteration of the outer loop in FasterOptimalSearchTree, for some fixed value of d. Define an array M by setting

$$M[i,r] = \begin{cases} OptCost[i,r-1] + OptCost[r+1,i+d] & \text{if } i \le r \le i+d \\ \infty & \text{otherwise} \end{cases}$$

Each call to ComputeCostAndRoot$(i, i+d)$ computes the smallest element in the ith row of this array M.

Let $M[1..m, 1..n]$ be an arbitrary two-dimensional array. We say that M is ***monotone*** if the leftmost smallest element in any row is either directly above or to the left of the leftmost smallest element in any later row. To state this condition more formally, let $LM(i)$ denote the index of the smallest item in the ith row $M[i, \cdot]$; if there is more than one smallest item, choose the one furthest to the left. Then M is monotone if and only if $LM(i) \le LM(i+1)$ for every index i. For example, the following 5×5 array is monotone (as shown by the hilighted row minima).

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 27 \\ 74 & 14 & 14 & 29 & 60 \\ 21 & 8 & 25 & 10 & 71 \\ 68 & 45 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

Given a monotone $m \times n$ array M, we can compute the array $LM[i]$ containing the index of the leftmost minimum element in every row as follows. We begin by recursively

[1] T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J. Applied Math.* 21:514–532, 1971. For a slightly simpler algorithm with the same running time, see A. M. Garsia and M. L. Wachs, A new algorithms for minimal binary search trees, *SIAM J. Comput.* 6:622–642, 1977. The original correctness proofs for both algorithms are rather intricate; for simpler proofs, see Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter, Correctness of constructing optimal alphabetic trees revisited, *Theoretical Computer Science*, 180:309-324, 1997.

computing the leftmost minimum elements in all odd-indexed rows of M. Then for each even index $2i$, the monotonicity of M implies the bounds

$$LM[2i-1] \le LM[2i] \le LM[2i+1],$$

so we can compute $LM[2i]$ by brute force by searching only within that range of indices. This search requires exactly $LM[2i+1]-LM[2i-1]$ comparisons, because finding the minimum of k numbers requires $k-1$ comparisons. In particular, if $LM[2i-1] = LM[2i+1]$, then we don't need to perform any comparisons on row $2i$. Summing over all even indices, we find that the total number of comparisons is

$$\sum_{i=1}^{m/2} LM[2i+1]-LM[2i-1] \;=\; LM[m+1]-LM[1] \;\le\; n.$$

We also need to spend constant time on each row, as overhead for the main loop, so the total time for our algorithm is $O(n+m)$ plus the time for the recursive call. Thus, the running time satisfies the recurrence

$$T(m,n) = T(m/2,n) + O(n+m),$$

which implies that our algorithm runs in $\boldsymbol{O(m+n\log m)}$ ***time***.

Alternatively, we could use the following divide-and-conquer procedure, similar in spirit to Hirschberg's divide-and-conquer algorithm. Compute the middle leftmost-minimum index $h = LM[m/2]$ by brute force in $O(n)$ time, and then recursively find the leftmost minimum entries in the following submatrices:

$$M\,[1\,..\,m/2-1,\ 1\,..\,h] \qquad M\,[m/2+1\,..\,m,\ h\,..\,n]$$

The worst-case running time $T(m,n)$ for this algorithm obeys the following recurrence (after removing some irrelevant ±1s from the recursive arguments, as usual):

$$T(m,n) = \begin{cases} 0 & \text{if } m < 1 \\ O(n) + \max_k \big(T(m/2,k) + T(m/2,n-k)\big) & \text{otherwise} \end{cases}$$

The recursion tree for this recurrence is a balanced binary tree of depth $\log_2 m$, and therefore with $O(m)$ nodes. The total number of *comparisons* performed at each level of the recursion tree is $O(n)$, but we also need to spend at least constant time at each node in the recursion tree. Thus, this divide-and-conquer formulation also runs in $\boldsymbol{O(m+n\log m)}$ ***time***.

In fact, these two algorithms are morally identical. Both algorithms examine the same subset of array entries and perform the same pairwise comparisons, although in different orders. Specifically, the divide-and-conquer algorithm performs the usual depth-first traversal of its recursion tree. The even-odd algorithm actually performs a *breadth*-first traversal of the exactly the same recursion tree, handling each level of the recursion tree in a single for-loop. The breadth-first formulation of the algorithm will prove more useful in the long run.

D.5 Saving More Time: Total Monotonicity

A more general technique for exploiting structure in dynamic programming arrays was discovered by Alok Aggarwal, Maria Klawe, Shlomo Moran, Peter Shor, and Robert Wilber in 1987; their algorithm is now universally known as ***SMAWK*** (pronounced "smoke") after their suitably-permuted initials.

SMAWK requires a stricter form of monotonicity in the input array. We say that M is ***totally monotone*** if the subarray defined by any subset of (not necessarily consecutive) rows and columns is monotone. For example, the following 5×5 array is monotone (as shown by the highlighted row minima), but not totally monotone (as shown by the gray 2×2 subarray).

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 27 \\ 74 & 14 & 14 & 29 & 60 \\ 21 & 8 & 25 & 10 & 71 \\ 68 & 45 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

On the other hand, the following array is totally monotone:

$$\begin{bmatrix} 12 & 21 & 38 & 76 & 89 \\ 47 & 14 & 14 & 29 & 60 \\ 21 & 8 & 20 & 10 & 71 \\ 68 & 16 & 29 & 15 & 76 \\ 97 & 8 & 12 & 2 & 6 \end{bmatrix}$$

Given a totally monotone $m \times n$ array as input, SMAWK finds the leftmost smallest element of every row in only $\boldsymbol{O(n+m)}$ ***time***.

The Monge property

Before we go into the details of the SMAWK algorithm, it's useful to consider the most common special case of totally monotone matrices. A ***Monge array***[2] is a two-dimensional array M where

$$M[i,j] + M[i',j'] \le M[i,j'] + M[i',j]$$

for all row indices $i < i'$ and all column indices $j < j'$. This inequality is sometimes called the *Monge property* or the *quadrangle inequality* or *submodularity*.

Lemma: *Every Monge array is totally monotone.*

[2]Monge arrays are named after Gaspard Monge, a French geometer who was one of the first to consider problems related to flows and cuts, in his 1781 *Mémoire sur la Théorie des Déblais et des Remblais*, which (in context) should be translated as "Treatise on the Theory of Holes and Dirt".

Proof: Let M be a two-dimensional array that is *not* totally monotone. Then there must be row indices $i < i'$ and column indices $j < j'$ such that $M[i, j] > M[i, j']$ and $M[i', j] \leq M[i', j']$. These two inequalities imply that $M[i, j] + M[i', j'] > M[i, j'] + M[i', j]$, from which it follows immediately that M is *not* Monge. □

Monge arrays have several useful properties that make identifying them easier. For example, an easy inductive argument implies that we do not need to check every quadruple of indices; in fact, we can determine whether an $m \times n$ array is Monge in just $O(mn)$ time.

Lemma: *An array M is Monge if and only if $M[i, j] + M[i+1, j+1] \leq M[i, j+1] + M[i+1, j]$ for all indices i and j.*

Monge arrays arise naturally in geometric settings; the following canonical example of a Monge array was suggested by Monge himself in 1781. Fix two parallel lines ℓ and ℓ' in the plane. Let $p_1, p_2, \ldots, p_m$ be points on ℓ, and let $q_1, q_2, \ldots, q_n$ be points on ℓ', with each set indexed in order along their respective lines. Let $M[1..m, 1..n]$ be the array of Euclidean distances between these points:

$$M[i, j] := |p_i q_j| = \sqrt{(p_i.x - q_i.x)^2 + (p_i.y - q_i.y)^2}$$

An easy argument with the triangle inequality implies that this array is Monge. Fix arbitrary indices $i < i'$ and $j < j'$, and let x denote the intersection point of segments $p_i q_{j'}$ and $p_{i'} q_j$.

$$\begin{aligned}
M[i, j] + M[i', j'] &= |p_i q_j| + |p_{i'} q_{j'}| \\
&\leq |p_i x| + |x q_j| + |p_{i'} x| + |x q_{j'}| && \text{[triangle inequality]} \\
&= \big(|p_i x| + |x q_{j'}|\big) + \big(|p_{i'} x| + |x q_j|\big) \\
&= |p_i q_{j'}| + |p_{i'} q_j| \\
&= M[i, j'] + M[i', j]
\end{aligned}$$

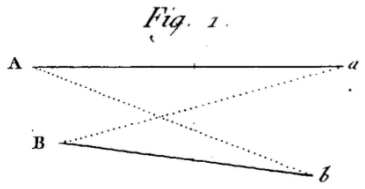


Figure D.2. The cheapest way to move two specks of dirt (on the left) into two tiny holes (on the right). From Monge's 1781 treatise on the theory of holes and dirt.

There are also several easy ways to construct and combine Monge arrays, either from scratch or by manipulating other Monge arrays.

Lemma: *The following arrays are Monge:*

(a) Any array with constant rows.
(b) Any array with constant columns.
(c) Any array that is all 0*s except for an upper-right rectangular block of* 1*s.*
(d) Any array that is all 0*s except for an lower-left rectangular block of* 1*s.*
(e) Any positive multiple of any Monge array.
(f) The sum of any two Monge arrays.
(g) The transpose of any Monge array.

Each of these properties follows from straightforward definition chasing. For example, to prove part (f), let X and Y be arbitrary Monge arrays, and let $Z = X + Y$. For all row indices $i < i'$ and column indices $j < j'$, we immediately have

$$\begin{aligned} Z[i,j] + Z[i',j'] &= X[i,j] + X[i',j'] + Y[i,j] + Y[i',j'] \\ &\le X[i',j] + X[i,j'] + Y[i',j] + Y[i,j'] \\ &= Z[i',j] + Z[i,j']. \end{aligned}$$

The other properties have similarly elementary proofs.

In fact, the properties listed in the previous lemma precisely characterize Monge arrays: ***Every*** Monge array (including the geometric example above) is a positive linear combination of row-constant, column-constant, and upper-right block arrays. (This characterization was proved independently by Rudolf and Woeginger in 1995, Bein and Pathak in 1990, Burdyok and Trofimov in 1976, and possibly others.)

D.6 The SMAWK algorithm

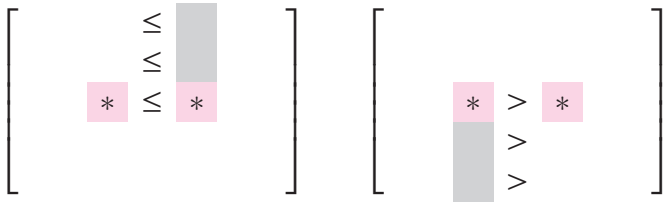
The SMAWK algorithm alternates between two different subroutines, one for "tall" arrays with more rows than columns, the other for "wide" arrays with more columns than rows. The "tall" subroutine is just the divide-and-conquer algorithm described in the previous section—recursively compute the leftmost row minima in every other row, and then fill in the remaining row minima in $O(n + m)$ time. The secret to SMAWK's success is its handling of "wide" arrays.

For any row index i and any two column indices $p < q$, the definition of total monotonizity implies the following observations:

- If $M[i,p] \le M[i,q]$, then for all $h \le i$, we have $M[h,p] \le M[h,q]$ and thus $LM[h] \ne q$.
- If $M[i,p] > M[i,q]$, then for all $j \ge i$, we have $M[j,p] > M[j,q]$ and thus $LM[j] \ne p$.

Call an array entry $M[i,j]$ ***dead*** if we have enough information to conclude that $LM[i] \ne j$. Then after we compare any two entries in the same row, either the left entry

and everything below it is dead, or the right entry and everything above it is dead.



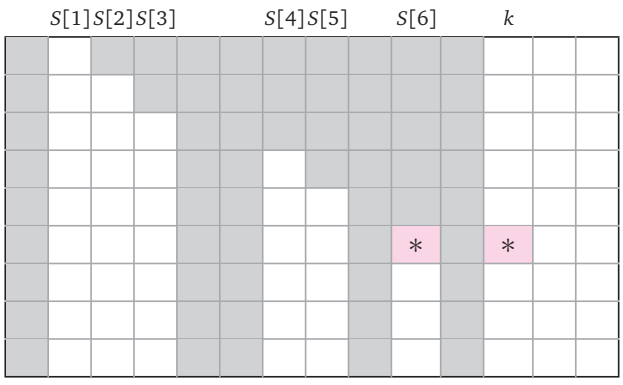
The following algorithm maintains a stack of column indices in an array $S[1..m]$ (yes, really m, the number of *rows*) and an index t indicating the number of indices on the stack.

```
REDUCE(M[1..m, 1..n]):
  t ← 1
  S[t] ← 1
  for k ← 1 to n
      while t > 0 and M[t, S[t]] ≥ M[t, k]
          t ← t − 1        ⟨⟨pop⟩⟩
      if t < m
          t ← t + 1
          S[t] ← k         ⟨⟨push k⟩⟩
  return S[1..t]
```

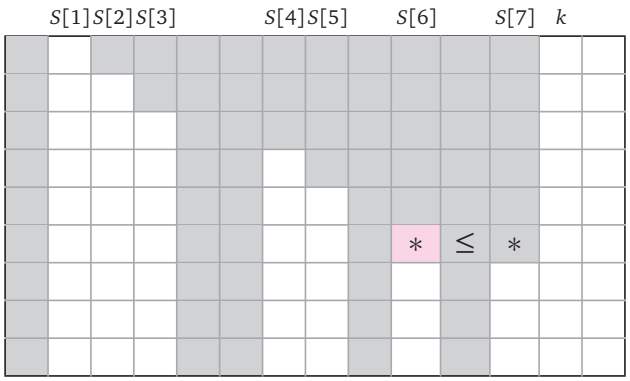
This algorithm maintains three important invariants:

- $S[1..t]$ is sorted in increasing order.
- For all $1 \le j \le t$, the top $j-1$ entires of column $S[j]$ are dead.
- If $j < k$ and j is not on the stack, then *every* entry in column j is dead.

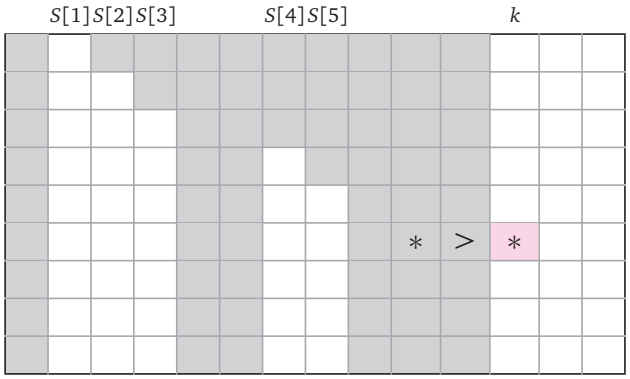
The first invariant follows immediately from the fact that indices are pushed onto the stack in increasing order. The second and third are more subtle, but both follow inductively from the total monotonicity of the array. The following figure shows a typical state of the algorithm when it is about to compare the entries $M[t, S[t]]$ and $M[t, k]$ (indicated by stars). Dead entries are indicated in gray.



If $M[t, S[t]] < M[t, k]$, then $M[t, k]$ and everything above it is dead, so we can safely push column k onto the stack (unless we already have $t = n$, in which case every entry in column k is dead) and then increment k.



On the other hand, if $M[t, S[t]] > M[t, k]$, then we know that $M[t, S[t]]$ and everything below it is dead. But by the inductive hypothesis, every entry above $M[t, S[t]]$ is already dead. Thus, the entire column $S[t]$ is dead, so we can safely pop it off the stack.



In both cases, all invariants are maintained.

Immediately after every comparison $M[t, S[t]] \geq M[t, k]$? in the REDUCE algorithm, we either increment the column index k or declare a column to be dead; each of these events can happen at most once per column. It follows that REDUCE performs at most $2n$ comparisons and thus runs in ***O(n) time*** overall.

Moreover, when REDUCE ends, every column whose index is not on the stack is completely dead. Thus, to compute the leftmost minimum element in every row of M, it suffices to examine only the $t < m$ columns with indices in the output array $S[1..t]$.

Finally, the main SMAWK algorithm first REDUCEs the input array (if it has fewer rows than columns), then recursively computes the leftmost row minima in every other, and finally fills in the rest of the row minima in $O(m + n)$ additional time. The argument

of the recursive call is an array with exactly $m/2$ rows and at most $\min\{n, m\}$ columns, so the running time obeys the following recurrence:

$$T(m,n) \leq \begin{cases} O(m) + T(m/2, n) & \text{if } m \geq n, \\ O(n) + T(m/2, m) & \text{if } m < n. \end{cases}$$

The algorithm needs $O(n)$ time to REDUCE the input array to at most m rows, after which every recursive call reduces the (worst-case) number of rows and columns by a factor of two. We conclude that SMAWK runs in ***$O(m+n)$ time***.

Later variants of SMAWK can compute row minima in totally monotone arrays in $O(m+n)$ time even when the rows are revealed one by one, and we require the smallest element in each row before we are given the next. These later variants can be used to speed up many dynamic programming algorithms by a factor of n when the final memoization table is totally monotone.

D.7 Using SMAWK

Finally, let's consider an (admittedly artificial) example of a dynamic programming algorithm that can be improved by SMAWK. Recall the Shimbel-Bellman-Ford algorithm for computing shortest paths:

```
BELLMANFORD(V, E, w):
  dist(s) ← 0
  for every vertex v ≠ s
      dist(v) ← ∞
  repeat V − 1 times
      for every edge u→v
          if dist(v) > dist(u) + w(u→v)
              dist(v) ← dist(u) + w(u→v)
```

We can rewrite this algorithm slightly; instead of relaxing *every* edge, we first identify the *tensest* edge leading into each vertex, and then relax only those edges. (This modification is actually closer to the natural dynamic-programming algorithm to compute shortest paths.)

```
MODBELLMANFORD(V, E, w):
  dist(s) ← 0
  for every vertex v ≠ s
      dist(v) ← ∞
  repeat V − 1 times
      for every vertex v
          mind(v) ← min_u (dist(u) + w(u→v))
      for every vertex v
          dist(v) ← min{dist(v), mind(v)}
```

The two lines in red can be interpreted as finding the minimum element in every row of a two-dimensional array M indexed by vertices, where

$$M[v,u] := dist(u) + w(u \to v)$$

Now consider the following input graph. Fix n arbitrary points $p_1, p_2, \ldots, p_n$ on some line ℓ and n more arbitrary points $q_1, q_2, \ldots, q_n$ on another line parallel to ℓ, with each set indexed in order along their respective lines. Let G be the complete bipartite graph whose vertices are the points p_i or q_j, whose edges are the segments $p_i q_j$, and where the weight of each edge is its natural Euclidean length. The standard version of Bellman-Ford requires $O(VE) = O(n^3)$ time to compute shortest paths in this graph.

The $2n \times 2n$ array M is not itself Monge, but we can easily decompose it into $n \times n$ Monge arrays as follows. Index the rows and columns of M by the vertices $p_1, p_2, \ldots, p_n, q_1, q_2, \ldots, q_n$ in that order. Then M decomposes naturally into four $n \times n$ blocks

$$M = \begin{bmatrix} \infty & U \\ V & \infty \end{bmatrix},$$

where every entry in the upper left and lower right blocks is ∞, and the other two blocks satisfy

$$U[i,j] = dist(q_j) + |p_i q_j| \qquad \text{and} \qquad V[i,j] = dist(p_j) + |p_j q_i|$$

for all indices i and j. Let P, Q, and D be the $n \times n$ arrays where

$$P[i,j] = dist(p_j) \qquad Q[i,j] = dist(q_j) \qquad D[i,j] = |p_i q_j|$$

for all indices i and j. Arrays P and Q are Monge, because all entries in the same column are equal, and the matrix D is Monge by the triangle inequality. It follows that the blocks $U = Q + D$ and $V = P + D^T$ are both Monge. Thus, by calling SMAWK twice, once on U and once on V, we can find all the row minima in M in $O(n)$ time. The resulting modification of Bellman-Ford runs in ***$O(n^2)$ time***.[3]

An important feature of this algorithm is that ***it never explicitly constructs the array M***. Instead, whenever the SMAWK algorithm needs a particular entry $M[u,v]$, we compute it on the fly in $O(1)$ time.

D.8 Saving Time: Four Russians

Some day. Maybe.

★★★

[3]Yes, we could also achieve this running time using Disjktra's algorithm with Fibonacci heaps.

Exercises

1. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K^2)$ time, where K is the number of match points. *[Hint: Use our earlier FINDMATCHES algorithm as a subroutine.]*

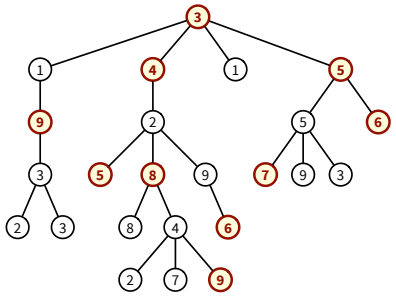
2. (a) Describe an algorithm to compute the *longest increasing subsequence* of a string $X[1..n]$ in $O(n \log n)$ time.

   (b) Using your solution to part (a) as a subroutine, describe an algorithm to compute the longest common subsequence of two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points.

3. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points. *[Hint: Combine your answers for problems 1 and 2(b).]*

4. Let T be an arbitrary rooted tree, where each vertex is labeled with a positive integer. A subset S of the nodes of T is *heap-ordered* if it satisfies two properties:

   - S contains a node that is an ancestor of every other node in S.
   - For any node v in S, the label of v is larger than the labels of any ancestor of v in S.



A heap-ordered subset of nodes in a tree.

   (a) Describe an algorithm to find the largest heap-ordered subset S of nodes in T that has the heap property in $O(n^2)$ time.

   (b) Modify your algorithm from part (a) so that it runs in $O(n \log n)$ time when T is a linked list. *[Hint: This special case is equivalent to a problem you've seen before.]*

   ♥(c) Describe an algorithm to find the largest subset S of nodes in T that has the heap property, in $O(n \log n)$ time. *[Hint: Find an algorithm to merge two sorted lists of lengths k and ℓ in $O(\log \binom{k+\ell}{k})$ time.]*

5. Suppose you are given a sorted array $X[1..n]$ of distinct numbers and a positive integer k. A set of k intervals ***covers*** X if every element of X lies inside one of the k intervals. Your aim is to find k intervals $[a_1, z_1], [a_2, z_2], \ldots, [a_k, z_k]$ that cover X where the function $\sum_{i=1}^{k}(z_i - a_i)^2$ is as small as possible. Intuitively, you are trying to cover the points with k intervals whose lengths are as close to equal as possible.

   (a) Describe an algorithm that finds k intervals with minimum total squared length that cover X. The running time of your algorithm should be a simple function of n and k.

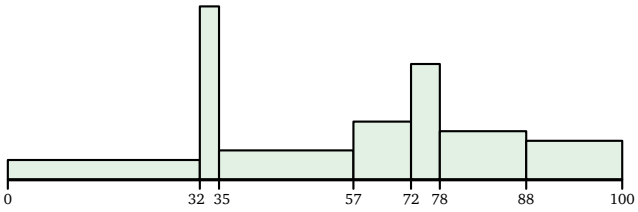
   (b) Consider the two-dimensional matrix $M[1..n, 1..n]$ defined as follows:

$$M[i,j] = \begin{cases} (X[j] - X[i])^2 & \text{if } i \le j \\ \infty & \text{otherwise} \end{cases}$$

   Prove that M satisfies the ***Monge property***: $M[i,j] + M[i',j'] \le M[i,j'] + M[i',j]$ for all indices $i < i'$ and $j < j'$.

   (c) Describe an algorithm that finds k intervals with minimum total squared length that cover X ***in $O(nk)$ time***. *[Hint: Solve part (a) first, then use part (b).]*

6. Suppose we want to summarize a large set S of values—for example, grade-point averages for every student who has ever attended UIUC—using a variable-width histogram. To construct a histogram, we choose a sorted sequence of ***breakpoints*** $b_0 < b_1 < \cdots < b_k$, such that every element of S lies between b_0 and b_k. Then for each interval $[b_{i-1}, b_i]$, the histogram includes a rectangle whose height is the number of elements of S that lie inside that interval.



A variable-width histogram with seven bins.

   Unlike a standard histogram, which requires the intervals to have equal width, we are free to choose the breakpoints arbitrarily. For statistical purposes, it is useful for the *areas* of the rectangles to be as close to equal as possible. To that end, define the ***cost*** of a histogram to be the sum of the *squares* of the rectangle areas; we seek a histogram with minimum cost.

   More formally, suppose we fix a sequence of breakpoints $b_0 < b_1 < \cdots < b_k$. For each index i, let n_i denote the number of input values in the ith interval:

$$n_i := \#\left\{x \in S \;\middle|\; b_{i-1} \le x < b_i\right\}.$$

Then the ***cost*** of the resulting histogram is $\sum_{i=1}^{k} (n_i(b_i - b_{i-1}))^2$. We want to compute a histogram with minimum cost for the given set S, where every breakpoint b_i is equal to some value in S.[4] In particular, b_0 must be the minimum value in S, and b_k must be the maximum value in S. Informally, you are trying to compute a histogram with k rectangles whose areas are as close to equal as possible.

Describe and analyze an algorithm to compute a variable-width histogram with minimum cost for a given set of data values. Your input is a sorted array $S[1..n]$ of distinct real numbers and an integer k. Your algorithm should return a sorted array $B[0..k]$ of breakpoints that minimizes the cost of the resulting histogram, where every breakpoint $B[i]$ is equal to some input value $S[j]$, and in particular $B[0] = S[1]$ and $B[k] = S[n]$.

[4]Thanks to the non-linear cost function, removing this assumption makes the problem *considerably* more difficult!

The problem is that we attempt to solve the simplest questions cleverly,
thereby rendering them unusually complex.
One should seek the simple solution.

— Anton Pavlovich Chekhov (c. 1890)

I love deadlines. I like the whooshing sound they make as they fly by.

— Douglas Adams, *The Salmon of Doubt* (2002)

E

Matroids

[Read Chapter 4 and 7 first.]

Status: Needs significant revision or deletion

Merge into greedy notes as hardsection?

E.1 Definitions

Many problems that can be correctly solved by greedy algorithms can be described in terms of an abstract combinatorial object called a *matroid*. Matroids were first described in 1935 by the mathematician Hassler Whitney as a combinatorial generalization of linear independence of vectors—"matroid" means "something sort of like a matrix".

A matroid $(S, \mathcal{I})$ consists of a finite ***ground set*** S and a collection $\mathcal{I}$ of subsets of X that satisfies three axioms:

- **Non-emptiness:** The empty set is in $\mathcal{I}$. (Thus, $\mathcal{I}$ is not itself empty.)
- **Heredity:** If $\mathcal{I}$ contains a subset $X \subseteq S$, then $\mathcal{I}$ contains every subset of X.
- **Exchange:** If X and Y are two sets in $\mathcal{I}$ where $|X| > |Y|$, then $Y \cup \{x\} \in \mathcal{I}$ for some element $x \in X \setminus Y$.

The subsets in $\mathcal{I}$ are typically called ***independent sets***; for example, we would say that any subset of an independent set is independent. An independent set is called a ***basis*** of the matroid if it is not a proper subset of another independent set. The exchange property implies that every basis of a matroid has the same cardinality. The ***rank*** of a subset X of the ground set is the size of the largest independent subset of X. A subset of S that is not in $\mathcal{I}$ is called ***dependent*** (surprise, surprise). Finally, a dependent set is called a ***circuit*** if every proper subset is independent.

Most of this terminology is justified by Whitney's original example:

- ***Linear matroid:*** Let A be any $n \times m$ matrix. A subset $I \subseteq \{1, 2, \ldots, n\}$ is independent if and only if the corresponding subset of columns of A is linearly independent.

The heredity property follows directly from the definition of linear independence; the exchange property is implied by an easy dimensionality argument. A basis in any linear matroid is also a basis (in the linear-algebra sense) of the vector space spanned by the columns of A. Similarly, the rank of a set of indices is precisely the rank (in the linear-algebra sense) of the corresponding set of column vectors.

Here are several other examples of matroids; some of these we will see again later. I will leave the proofs that these are actually matroids as exercises for the reader.

- ***Uniform matroid $U_{k,n}$:*** A subset $X \subseteq \{1, 2, \ldots, n\}$ is independent if and only if $|X| \leq k$. Any subset of $\{1, 2, \ldots, n\}$ of size k is a basis; any subset of size $k+1$ is a circuit.
- ***Graphic/cycle matroid $\mathcal{M}(G)$:*** Let $G = (V, E)$ be an arbitrary undirected graph. A subset of E is independent if it defines an *acyclic* subgraph of G. A basis in the graphic matroid is a spanning tree of G; a circuit in this matroid is a cycle in G.
- ***Cographic/cocycle matroid $\mathcal{M}^*(G)$:*** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq E$ is independent if the complementary subgraph $(V, E \setminus I)$ of G is *connected*. A basis in this matroid is the complement of a spanning tree; a circuit in this matroid is a *cocycle*—a minimal set of edges that disconnects the graph.
- ***Matching matroid:*** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq V$ is independent if there is a matching in G that covers I.
- ***Disjoint path matroid:*** Let $G = (V, E)$ be an arbitrary *directed* graph, and let s be a fixed vertex of G. A subset $I \subseteq V$ is independent if and only if there are edge-disjoint paths from s to each vertex in I.

Now suppose each element of the ground set of a matroid $\mathcal{M}$ is given an arbitrary non-negative weight. The ***matroid optimization problem*** is to compute a basis with maximum total weight. For example, if $\mathcal{M}$ is the cycle matroid for a graph G, the matroid optimization problem asks us to find the maximum spanning tree of G. Similarly, if $\mathcal{M}$ is the cocycle matroid for G, the matroid optimization problem seeks (the complement of) the *minimum* spanning tree.

The following natural greedy strategy computes a basis for any weighted matroid $(S, \mathcal{I})$, where the ground set S is represented by an array $S[1..n]$, and the weights of ground elements are represented by another array $w[1..n]$.

```
GreedyBasis(S[1..n], I, w[1..n]):
  sort S in decreasing order of weight w
  G ← ∅
  for i ← 1 to n
      if G ∪ {S[i]} ∈ I
          add S[i] to G
  return G
```

Suppose we can test in $F(n)$ time whether a given subset $X \subset S$ is independent. Then this algorithm runs in $O(n \log n + n \cdot F(n))$ time.

Theorem E.1. *For any matroid $\mathcal{M} = (S, \mathcal{I})$ and any weight function w, GreedyBasis$(S, \mathcal{I}, w)$ returns a maximum-weight basis of $\mathcal{M}$.*

Proof: We use a standard exchange argument. Let $G = \{g_1, g_2, \ldots, g_k\}$ be the independent set returned by GreedyBasis$(S, \mathcal{I}, w)$. If any other element of S could be added to G to obtain a larger independent set, the greedy algorithm would have added it. Thus, G is a basis.

For purposes of deriving a contradiction, suppose there is an independent set $H = \{h_1, h_2, \ldots, h_\ell\}$ such that

$$\sum_{i=1}^{k} w(g_i) < \sum_{j=1}^{\ell} w(h_i).$$

Without loss of generality, assume that H is a basis. The exchange property now implies that $k = \ell$.

Now suppose the elements of G and H are indexed in order of decreasing weight. Let i be the smallest index such that $w(g_i) < w(h_i)$, and consider the independent sets

$$G_{i-1} = \{g_1, g_2, \ldots, g_{i-1}\} \quad \text{and} \quad H_i = \{h_1, h_2, \ldots, h_{i-1}, h_i\}.$$

By the exchange property, there is some element $h_j \in H_i$ such that $G_{i-1} \cup \{h_j\}$ is an independent set. We have $w(h_j) \geq w(h_i) > w(g_i)$. Thus, the greedy algorithm considers *and rejects* the heavier element h_j before it considers the lighter element g_i. But this is impossible—the greedy algorithm accepts elements in decreasing order of weight. □

We now immediately have a correct greedy optimization algorithm for *any* matroid. Returning to our examples:

- Linear matroid: Given a matrix A, compute a subset of vectors of maximum total weight that span the column space of A.

- Uniform matroid: Given a set of weighted objects, compute its k largest elements.
- Cycle matroid: Given a graph with weighted edges, compute its maximum spanning tree. In this setting, the greedy algorithm is better known as *Kruskal's algorithm*.
- Cocycle matroid: Given a graph with weighted edges, compute its minimum spanning tree.
- Matching matroid: Given a graph, determine whether it has a perfect matching.
- Disjoint path matroid: Given a directed graph with a special vertex s, find the largest set of edge-disjoint paths from s to other vertices.

The exchange condition for matroids turns out to be crucial for the success of this algorithm. A *subset system* is a finite collection $\mathcal{S}$ of finite sets that satisfies the heredity condition—If $X \in \mathcal{S}$ and $Y \subseteq X$, then $Y \in \mathcal{S}$—but not necessarily the exchange condition.

Theorem E.2. *For any subset system $\mathcal{S}$ that is **not** a matroid, there is a weight function w such that* GREEDYBASIS$(\mathcal{S}, w)$ *does **not** return a maximum-weight set in $\mathcal{S}$.*

Proof: Let X and Y be two sets in $\mathcal{S}$ that violate the exchange property—$|X| > |Y|$, but for any element $x \in X \setminus Y$, the set $Y \cup \{x\}$ is not in $\mathcal{S}$. Let $m = |Y|$. We define a weight function as follows:

- Every element of Y has weight $m + 2$.
- Every element of $X \setminus Y$ has weight $m + 1$.
- Every other element of the ground set has weight zero.

With these weights, the greedy algorithm will consider and accept every element of Y, then consider and reject every element of X, and finally consider all the other elements. The algorithm returns a set with total weight $m(m+2) = m^2 + 2m$. But the total weight of X is at least $(m+1)^2 = m^2 + 2m + 1$. Thus, the output of the greedy algorithm is not the maximum-weight set in $\mathcal{S}$. □

Recall the Applied Chaos scheduling problem considered in Chapter 4. There is a natural subset system associated with this problem: A set of classes is independent if and only if no two classes overlap. (This is just the graph-theory notion of "independent set"!) This subset system is *not* a matroid, because there can be maximal independent sets of different sizes, which violates the exchange property. If we consider a *weighted* version of the class scheduling problem, where each class is worth a different number of hours, Theorem **??** implies that the greedy algorithm will *not* always find the optimal schedule. (In fact, there's an easy counterexample with only two classes!) However, Theorem **??** does *not* contradict the correctness of the greedy algorithm for the original *unweighted* problem; that problem uses a particularly lucky choice of weights (all equal).

E.2 Scheduling with Deadlines

Suppose you have n tasks to complete in n days; each task requires your attention for a full day. Each task comes with a *deadline*, the last day by which the job should be completed, and a *penalty* that you must pay if you do not complete each task by its assigned deadline. What order should you perform your tasks in to minimize the total penalty you must pay?

More formally, you are given an array $D[1..n]$ of deadlines an array $P[1..n]$ of penalties. Each deadline $D[i]$ is an integer between 1 and n, and each penalty $P[i]$ is a non-negative real number. A *schedule* is a permutation of the integers $\{1, 2, \ldots, n\}$. The scheduling problem asks you to find a schedule π that minimizes the following cost:

$$cost(\pi) := \sum_{i=1}^{n} P[i] \cdot [\pi(i) > D[i]].$$

This doesn't look anything like a matroid optimization problem. For one thing, matroid optimization problems ask us to find an optimal *set*; this problem asks us to find an optimal *permutation*. Surprisingly, however, this scheduling problem is actually a matroid optimization in disguise! For any schedule π, call tasks i such that $\pi(i) > D[i]$ *late*, and all other tasks *on time*. The following trivial observation reveals the underlying matroid structure.

> The cost of a schedule is determined by the subset of tasks that are on time.

Call a subset X of the tasks ***realistic*** if there is a schedule π in which every task in X is on time. We can precisely characterize the realistic subsets as follows. Let $X(t)$ denote the subset of tasks in X whose deadline is on or before t:

$$X(t) := \{i \in X \mid D[i] \le t\}.$$

In particular, $X(0) = \emptyset$ and $X(n) = X$.

Lemma E.3. *Let $X \subseteq \{1, 2, \ldots, n\}$ be an arbitrary subset of the n tasks. X is realistic if and only if $|X(t)| \le t$ for every integer t.*

Proof: Let π be a schedule in which every task in X is on time. Let i_t be the tth task in X to be completed. On the one hand, we have $\pi(i_t) \ge t$, since otherwise, we could not have completed $t-1$ other jobs in X before i_t. On the other hand, $\pi(i_t) \le D[i]$, because i_t is on time. We conclude that $D[i_t] \ge t$, which immediately implies that $|X(t)| \le t$.

Now suppose $|X(t)| \le t$ for every integer t. If we perform the tasks in X in increasing order of deadline, then we complete all tasks in X with deadlines t or less by day t. In particular, for any $i \in X$, we perform task i on or before its deadline $D[i]$. Thus, X is realistic. □

We can now define a ***canonical schedule*** for any set X as follows: execute the tasks in X in increasing deadline order, and then execute the remaining tasks in any order. The previous proof implies that a set X is realistic if and only if every task in X is on time in the canonical schedule for X. Thus, our scheduling problem can be rephrased as follows:

> Find a realistic subset X such that $\sum_{i \in X} P[i]$ is maximized.

So we're looking for optimal subsets after all!

Lemma E.4. *The collection of realistic sets of jobs forms a matroid.*

Proof: The empty set is vacuously realistic, and any subset of a realistic set is clearly realistic. Thus, to prove the lemma, it suffices to show that the exchange property holds. Let X and Y be realistic sets of jobs with $|X| > |Y|$.

Let t^* be the largest integer such that $|X(t^*)| \le |Y(t^*)|$. This integer must exist, because $|X(0)| = 0 \le 0 = |Y(0)|$ and $|X(n)| = |X| > |Y| = |Y(n)|$. By definition of t^*, there are more tasks with deadline $t^* + 1$ in X than in Y. Thus, we can choose a task j in $X \setminus Y$ with deadline $t^* + 1$; let $Z = Y \cup \{j\}$.

Let t be an arbitrary integer. If $t \le t^*$, then $|Z(t)| = |Y(t)| \le t$, because Y is realistic. On the other hand, if $t > t^*$, then $|Z(t)| = |Y(t)| + 1 \le |X(t)| < t$ by definition of t^* and because X is realistic. The previous lemma now implies that Z is realistic. This completes the proof of the exchange property. □

This lemma implies that our scheduling problem is actually a matroid optimization problem, so the greedy algorithm finds the optimal schedule.

```
GREEDYSCHEDULE(D[1..n], P[1..n]):
  Sort P in increasing order, and permute D to match
  j ← 0
  for i ← 1 to n
      X[j + 1] ← i
      if X[1..j + 1] is realistic
          j ← j + 1
  return the canonical schedule for X[1..j]
```

To turn this outline into a real algorithm, we need a procedure to test whether a given subset of jobs is realistic. Lemma 9 immediately suggests the following strategy to answer this question in $O(n)$ time.

```
REALISTIC?(X[1..m], D[1..n]):
  ⟨⟨X is sorted by increasing deadline: i ≤ j ⟹ D[X[i]] ≤ D[X[j]]⟩⟩
  N ← 0
  j ← 0
  for t ← 1 to n
      if D[X[j]] = t
          N ← N + 1
          j ← j + 1
      ⟨⟨Now N = |X(t)|⟩⟩
      if N > t
          return FALSE
  return TRUE
```

If we use this subroutine, GREEDYSCHEDULE runs in $O(n^2)$ time. By using some appropriate data structures, the running time can be reduced to $O(n \log n)$; details are left as an exercise for the reader.

Exercises

1. Prove that for any graph G, the "graphic matroid" $\mathcal{M}(G)$ is in fact a matroid. (This problem is really asking you to prove that Kruskal's algorithm is correct!)

2. Prove that for any graph G, the "cographic matroid" $\mathcal{M}^*(G)$ is in fact a matroid.

♦3. Prove that for any graph G, the "matching matroid" of G is in fact a matroid. *[Hint: What is the symmetric difference of two matchings?]*

♦4. Prove that for any directed graph G and any vertex s of G, the resulting "disjoint path matroid" of G is in fact a matroid. *[Hint: This question is **much** easier if you're already familiar with maximum flows.]*

5. Let G be an undirected graph. A set of cycles $\{c_1, c_2, \ldots, c_k\}$ in G is called *redundant* if every edge in G appears in an even number of c_i's. A set of cycles is *independent* if it contains no redundant subset. A maximal independent set of cycles is called a *cycle basis* for G.

   (a) Let C be any cycle basis for G. Prove that for any cycle γ in G, there is a subset $A \subseteq C$ such that $A \cap \{\gamma\}$ is redundant. In other words, γ is the 'exclusive or' of the cycles in A.

   (b) Prove that the set of independent cycle sets form a matroid.

   ♥(c) Now suppose each edge of G has a weight. Define the weight of a cycle to be the total weight of its edges, and the weight of a *set* of cycles to be the total weight of all cycles in the set. (Thus, each edge is counted once for every cycle in

which it appears.) Describe and analyze an efficient algorithm to compute the minimum-weight cycle basis in G.

6. Describe a modification of GREEDYSCHEDULE that runs in $O(n \log n)$ time. *[Hint: Store X in an appropriate data structure that supports the operations "Is $X \cup \{i\}$ realistic?" and "Add i to X" in $O(\log n)$ time each.]*

Man muss immer generalisieren. [One must always generalize.]

— attributed to Carl Gustav Jacob Jacobi (c. 1830) by
Philip J. Davis and Reuben Hersh, *The Mathematical Experience* (1981)

Life is like riding a bicycle. To keep your balance you must keep moving.

— Albert Einstein, in a letter to his son Eduard (February 5, 1930)

Scarcely pausing for breath, Vroomfondel shouted, "We don't demand solid facts! What we demand is a total absence of solid facts. I demand that I may or may not be Vroomfondel!"

— Douglas Adams, *The Hitchhiker's Guide to the Galaxy* (1979)

F

Balances and Pseudoflows

[Read Chapters 10 and 11 first.]

Status: Beta. Needs more figures.

F.1 Unbalanced Flows

In this chapter, we consider a generalization of flows that allows "stuff" to be injected or extracted from the flow network at the vertices. For the moment, consider a flow network $G = (V, E)$ without any specific source and target vertices. Let $b : V \to \mathbb{R}$ be a ***balance*** function describing how much flow should be injected (if the value is positive) or extracted (if the value is negative) at each vertex. We interpret positive balances as ***demand*** or ***deficit*** and negative balances as (negated) ***supply*** or ***excess***.

We now redefine the word ***flow*** to mean a function $f : E \to \mathbb{R}$ that satisfies the modified balance condition

$$\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) = b(v)$$

at *every* node v. A flow f is ***feasible*** if it satisfies the usual capacity constraints $0 \le f(e) \le c(e)$ at every edge e. Our problem now is to compute, given a flow network

with edge capacities and vertex balances, either a *feasible* flow in or a proof that no such flow exists.

Note two significant differences from the standard maximum flow problem. First, there are no special source and target vertices that are exempt from the balance constraint. Second, we are not trying to optimize the value of the flow; in fact, the "value" of a flow is no longer even *defined*, because the network has no source and target vertices.

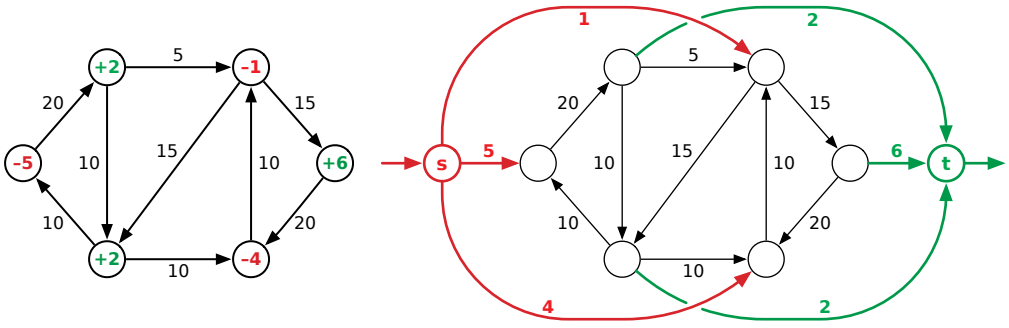
One easy necessary condition is that all the vertex balances must sum to zero; intuitively, every edge $u \to v$ adds the same amount to v's balance that it subtracts from u's balance. More formally, for every feasible flow f, we have

$$\begin{aligned}\sum_v b(v) &= \sum_v \left(\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) \right) \\ &= \sum_{u \to v \in E} f(u \to v) - \sum_{v \to w \in E} f(v \to w) = 0.\end{aligned}$$

F.2 Reduction to Maximum Flow

We can reduce the problem of finding a feasible flow in a network with non-zero balance constraints to the standard maximum-flow problem, by adding new vertices and edges to the input graph as follows.

Starting with with original graph G, we construct a new graph $G' = (V', E')$ by adding a new source vertex s with edges to every supply vertex, a new target vertex t with edges from every demand vertex. Specifically, for each vertex v in G, if $b(v) > 0$, we add an edge $v \to t$ with capacity $b(v)$, and if $b(v) < 0$, we add a new edge $s \to v$ with capacity $-b(v)$. Let $c' : E' \to \mathbb{R}$ be the resulting capacity function; by construction, $c'|_E = c$.



A flow network G with non-zero balance constraints, and the transformed network G'.

Now call an (s, t)-flow in G' ***saturating*** if every edge leaving s (or equivalently, every edge entering t) is saturated. Every saturating flow is a maximum flow; conversely, either all maximum flows in G' are saturating, or G' has no saturating flow.

Lemma F.1. *G has a feasible flow if and only if G' has a saturating (s, t)-flow.*

Proof: Let $f : E \to \mathbb{R}$ be any feasible flow in G, and consider the function $f' : E' \to \mathbb{R}$ defined as follows:

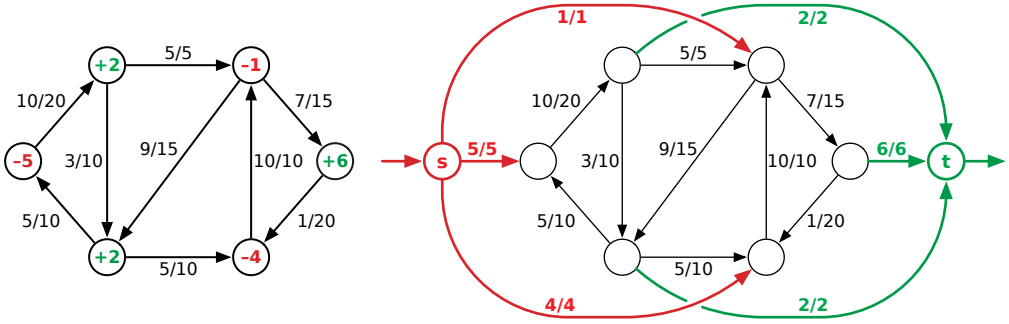
$$f'(e') = \begin{cases} f(e') & \text{if } e' \in E \\ c'(e') & \text{otherwise} \end{cases}$$

Every edge incident to s or t is saturated, and every edge in E satisfies the capacity constraint $0 \le f'(e) = f(e) \le c(e) = c'(e)$. For each vertex v except s and t, we immediately have

$$\begin{aligned} \sum_{u \in V'} f'(u \to v) - \sum_{w \in V'} f'(v \to w) &= \left(\sum_{u \in V} f(u \to v) - \sum_{w \in V} f(v \to w) \right) - b(v) \\ &= b(v) - b(v) \\ &= 0. \end{aligned}$$

We conclude that f' is a feasible (s,t)-flow in G'. Every edge out of s or into t is saturated by definition, so f' is a saturating flow in G'.

Similarly tedious algebra implies that for any saturating (s,t)-flow $f' : E' \to \mathbb{R}$ through G', the restriction $f = f'|_E$ is a feasible flow in G. □



A feasible flow in G and the corresponding saturating flow in G'.

We emphasize that there are flow networks with no feasible flow, even when the sum of the balances is zero. Suppose we partition the vertices of G into two arbitrary subsets S and T. As usual, let $\|S, T\|$ be the total capacity of the cut (S, T):

$$\|S, T\| = \sum_{u \in S} \sum_{v \in T} c(u \to v).$$

Let $b(S)$ and $b(T)$ denote the sum of the balances of vertices in S and T, respectively:

$$b(S) := \sum_{u \in S} b(u) \qquad \text{and} \qquad b(T) := \sum_{v \in T} b(v).$$

We call the cut (S, T) ***infeasible*** if $\|S, T\| < b(T)$; that is, if T has more demand than can be moved across the cut. The following theorem is a straightforward consequence of the maxflow/mincut theorem (hint, hint):

Theorem F.2. *Every flow network has either a feasible flow or an infeasible cut.*

F.3 Pseudoflows

Instead of reducing our new unbalanced flow problem to the classical maximum flow problem, it is instructive to project the behavior of Ford-Fulkerson on the transformed flow network G' back to the original flow network G. The resulting algorithm no longer maintains and incrementally improves a feasible *flow* in G, but a more general function called a ***pseudoflow***. Formally, a pseudoflow in G is *any* function $\psi : E \to \mathbb{R}$. We say that a pseudoflow ψ is ***feasible*** if $0 \le \psi(e) \le c(e)$ for every edge $e \in E$. A ***flow*** is any pseudoflow that also satisfies the balance constraints at every vertex.

For any pseudoflow ψ in G, we define the residual capacity of any edge $u \to v$ as usual:

$$c_\psi(u \to v) := \begin{cases} c(u \to v) - \psi(u \to v) & \text{if } u \to v \in E \\ \psi(v \to u) & \text{if } v \to u \in E \end{cases}$$

We also define the ***residual balance*** of any node to be its original balance minus its net incoming pseudoflow:

$$b_\psi(v) := b(v) - \sum_u \psi(u \to v) + \sum_w \psi(v \to w).$$

A pseudoflow ψ is a flow if and only if $b_\psi(v) = 0$ for every vertex v. Finally, we define the ***residual network*** G_ψ to be the graph of all edges with positive residual capacity, where the balance of each node v is $b_\psi(v)$. As usual, if ψ is zero everywhere, then G_ψ is the original network G, with its original capacities and balances.

Now we redefine an ***augmenting path*** to be any path in the residual graph G_ψ from any vertex with negative residual balance (supply or excess) to any vertex with positive residual balance (demand or deficit). Pushing flow along an augmenting path decreases the total residual supply

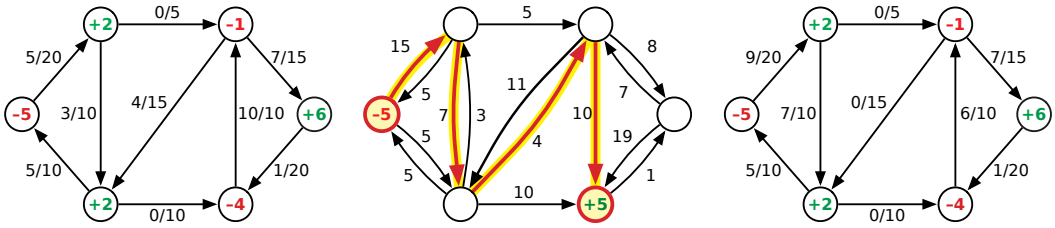
$$B_\psi := \sum_v \left| b_\psi(v) \right|$$

and therefore moves ψ closer to being a feasible flow. The largest amount of flow that we can push along an augmenting path from u to v, before it ceases to be an augmenting path, is the minimum of three quantities:

- The residual supply $-b_\psi(u)$ at the beginning of the path,
- The residual demand $b_\psi(v)$ at the end of the path, and
- The minimum residual capacity among the edges in the path.

On the other hand, if G_ψ contains a vertex v with non-zero residual balance, but does not contain an augmenting path starting or ending at v, then G has no feasible flow. Specifically, if $b_\psi(v) > 0$, then the set S of vertices reachable from v and its complement $T = V \setminus S$ define an infeasible cut; symmetrically, if $b_\psi(v) < 0$, then the set T of vertices that can reach v and its complement $S = V \setminus T$ define an infeasible cut.

Putting all these pieces together, we obtain a simple algorithm for computing either a feasible flow or an infeasible cut. Initialize $\psi(e) = 0$ at every edge e. Then as long



From left to right: A pseudoflow ψ in a flow network G; the residual graph G_ψ with one augmenting path highlighted; and the updated pseudoflow after pushing 4 units along the augmenting path.

as the residual graph G_ψ has a vertex with non-zero balance, update ψ by pushing as much flow as possible along an arbitrary augmenting path. When all residual balances are zero, the current pseudoflow ψ is actual a feasible flow.

```
FeasibleFlow(V, E, c, b):
  for every edge e ∈ E
      ψ(e) ← 0
  B ← Σ_v |b(v)|/2
  while B > 0
      construct G_ψ
      ⟨⟨Find augmenting path π⟩⟩
      s ← any vertex with b_ψ(s) < 0
      if s cannot reach a vertex t in G_ψ with b_ψ(t) > 0
          return Infeasible
      t ← any vertex reachable from s with b_ψ(t) > 0
      π ← any path in G_ψ from s to t
      ⟨⟨Push as much flow as possible along π⟩⟩
      R ← min{−b_ψ(s), b_ψ(t), min_{e∈π} c_ψ(e)}
      B ← B − R
      for every directed edge e ∈ π
          if e ∈ E
              ψ(e) ← ψ(e) + R
          else ⟨⟨rev(e) ∈ E⟩⟩
              ψ(e) ← ψ(e) − R
  return ψ
```

Naturally this algorithm comes with both the same power and the same limitations as Ford-Fulkerson. We can find a single augmenting path in $O(V+E)$ time via whatever-first search. If all capacities and balances are integers, the basic algorithm halts after at most B iterations, where $B = \sum_v |b(v)|/2$, but if we allow irrational capacities **⟨⟨or irrational balances?⟩⟩**, the algorithm could run forever without converging to a flow. Choosing the augmenting path with maximum residual capacity or with the fewest edges leads to

faster algorithms; in particular, if we always choose the shortest augmenting path, the algorithm runs in $O(VE^2)$ time.

F.4 Variations on a Theme

There are several variations on the standard maximum-flow problem, with additional or modified constraints, that can be solved quickly using the pseudoflow formulation. These variations are all solved using a two-stage process:

- First find a *feasible* flow f in the original input graph G.
- Then find a *maximum* flow f' in the residual graph G_f.
- Finally, return the flow $f + f'$.

In each variation, the residual graph G_f we use in the second stage will be a textbook-standard flow network. Notice that Ford-Fulkerson itself can be seen as an example of this two-stage algorithm, where the flow f found in the first stage is zero everywhere, or more subtly, where f is obtained by interrupting Ford-Fulkerson after *any* augmentation step.

Maximum Flows with Non-Zero Balances

Suppose we are given a flow network $G = (V, E)$ with edge capacities $c : E \to \mathbb{R}^+$, non-trivial vertex balances $b : V \to \mathbb{R}$, and two special vertices s and t, and we are asked to compute the maximum (s, t)-flow in this network. In this context, an (s, t)-flow is a function $f : E \to \mathbb{R}^+$ that satisfies the modified balance conditions

$$\sum_w f(v \to w) - \sum_u f(u \to v) = b(v)$$

at every vertex v ***except s and t***. As usual, our goal is to find an (s, t)-flow that maximizes the net flow out of s:

$$|f| = \sum_w f(s \to w) - \sum_u f(u \to s)$$

The algorithms in the previous sections *almost* solve the first stage directly, except for two issues: (1) the terminal vertices s and t are not subject to balance constraints, and (2) the sum of the vertex balances need not be zero. In fact, we can handle both of these issues at once by modifying the graph as follows. First, to avoid any ambiguity, we (re)define $b(s) = b(t) = 0$. Then we add one new vertex z with balance $b(z) = -\sum_v b(v)$, and two new infinite-capacity edges $t \to z$ and $z \to s$. Call the resulting modified flow network G'. Straightforward definition-chasing implies that any feasible flow in G' restricts to a feasible (s, t)-flow in G, and conversely, any feasible (s, t)-flow in G can be extended to a feasible flow in G'.

★★★

Figure!

Thus, we can find a feasible (s,t)-flow f in G in $O(VE^2)$ time by repeatedly pushing flow along shortest augmenting paths, or in $O(VE)$ time using Orlin's maximum-flow algorithm. In the resulting residual graph G_f, every vertex (except at s and t) has residual balance zero, so we can find a maximum flow in G_f using any standard algorithm.

Lower Bounds on Edges

In another standard variant, the input includes a ***lower bound $\ell(e)$*** on the flow value of each edge e, in addition to the capacity $c(e)$. In this context, a flow f is feasible if and only if $\ell(e) \le f(e) \le c(e)$ for every edge e. In the standard flow problem, we have $\ell(e) = 0$ for every edge e.

Although it is natural to require the lower bounds $\ell(e)$ to be non-negative, we can in fact allow negative lower bounds, and therefore negative flow values $f(e)$, if we interpret negative flow along an edge $u \to v$ as positive flow along its reversal $v \to u$. More formally, we define a pseudoflow as a function $\psi : E \to \mathbb{R}$ such that

$$\psi(v \to u) = -\psi(u \to v)$$

for every edge $u \to v$; more simply, a pseudoflow is an *antisymmetric* function over the edges. The antisymmetry is reflected in the upper and lower bounds on flow:

$$\ell(v \to u) = -c(u \to v) \qquad c(v \to u) = -\ell(u \to v)$$

Then for any pseudoflow ψ, each edge $u \to v$ has both a residual capacity and a residual lower bound, defined as follows to maintain antisymmetry:

$$\ell_\psi(u \to v) := \begin{cases} \ell(u \to v) - \psi(u \to v) & \text{if } u \to v \in E \\ \psi(u \to v) - c(v \to u) & \text{if } v \to u \in E \end{cases}$$

$$c_\psi(u \to v) := \begin{cases} c(u \to v) - \psi(u \to v) & \text{if } u \to v \in E \\ \psi(u \to v) - \ell(v \to u) & \text{if } v \to u \in E \end{cases}$$

Now the residual network G_ψ consists of all edges $u \to v$ with non-negative residual capacity $c_\psi(u \to v) \ge 0$. We can easily verify (hint, hint) that this antisymmetric formulation of (pseudo)flows and residual graphs is completely consistent with our usual formulation of flows as *non-negative* functions.

Given a flow network with both lower bounds and capacities on the edges, we can compute a maximum (s,t)-flow as follows. If all lower bounds are zero or negative, we can apply any standard maxflow algorithm, after replacing each edge $u \to v$ with a negative lower bound with a pair of opposing edges $u \to v$ and $v \to u$, each with positive capacity.[1] Otherwise, we first define an initial feasible pseudoflow ψ that meets every positive lower bound:

$$\psi(u \to v) = \max\{\ell(u \to v), 0\}$$

[1] Wait, didn't we forbid opposing edges two chapters ago? Okay, fine: Replace each edge $u \to v$ with a pair of opposing *paths* $u \to x \to v$ and $v \to y \to u$, where x and y are new vertices.

Vertices in the resulting residual graph G_ψ may have non-zero residual balance, but every edge has a lower bound that is either zero or negative. Thus, we can compute a maximum (s,t)-flow in G_ψ using the previous two-stage approach, in $O(VE)$ time.

★★★

Figure!

♥F.5 Push-Relabel

★★★

This section needs figures and an editing sweep. Cite Alexander Karzanov (preflows)?

The pseudoflow formulation is the foundation of another family of efficient maximum-flow algorithms that is *not* based on path-augmentation, called ***push-relabel*** or ***preflow-push*** algorithms, discovered by Andrew Goldberg and then refined in collaboration with Robert Tarjan before formal publication in 1986, while Goldberg was still a PhD student.

Every push-relabel algorithm maintains a special type of pseudoflow, called a ***preflow***, in which every node (except s) has non-negative residual balance:

$$b_\psi(v) := \sum_u \psi(u{\to}v) - \sum_w \psi(v{\to}w) \geq 0.$$

We call a vertex ***active*** if it is not the target vertex t and its residual balance is positive; we immediately observe that ψ is actually a *flow* if and only if no vertex is active. The algorithm also maintains a non-negative integer ***height*** $\boldsymbol{ht(v)}$ for each vertex v. We call any edge $u{\to}v$ with $ht(u) > ht(v)$ a ***downward*** edge.

The push-relabel algorithm begins by setting $ht(s) = |V|$ and $ht(v) = 0$ for every node $v \neq s$, and choosing an initial pseudoflow ψ that saturates all edges leaving s and leaves all other edges empty:

$$\psi(u{\to}v) = \begin{cases} c(u{\to}v) & \text{if } u = s \\ 0 & \text{otherwise} \end{cases}$$

Then for as long as there are active nodes, the algorithm repeatedly performs one of the following operations at an arbitrary active node u:

- ***Push:*** For some downward residual edge $u{\to}v$ out of u, increase $\psi(u{\to}v)$ by the minimum of the excess at u and the residual capacity of $u{\to}v$.
- ***Relabel:*** If u has no downward outgoing residual edges, increase the height of u by 1.

It is not at all obvious (at least, it wasn't obvious to me at first) that the push-relabel algorithm correctly computes a maximum flow, or even that it halts in finite time. To prove that the algorithm is correct, we need a series of relatively simple observations.

First, we say that the height function $ht : V \to \mathbb{N}$ and the pseudoflow $\psi : E \to \mathbb{R}$ are ***compatible*** if $ht(u) \leq ht(v) + 1$ for every edge $u{\to}v$ in the residual graph G_ψ.

Lemma F.3. *After each step of the push-relabel algorithm, the height function ht and the pseudoflow ψ are compatible.*

Proof: We prove the lemma by induction. Initially, every residual edge either enters the source vertex s or has both endpoints with height zero, so the initial heights and pseudoflow are compatible. For each later step of the algorithm, there are two cases to consider.

- Just before we push flow along the residual edge $u \to v$, then $u \to v$ must be a downward edge, so $ht(v) < ht(u)$. If the edge $u \to v$ was empty before the push, then this step adds the reversed edge $v \to u$ to the residual graph, and $ht(v) < ht(u) \le ht(u) + 1$.
- On the other hand, just before we relabel an active vertex v, we must have $ht(v) \le ht(w)$ for every outgoing residual edge $v \to w$ and (by the induction hypothesis) $ht(u) \le ht(v) + 1$ for every incoming residual edge $u \to v$. Thus, after relabeling, we have $ht(v) \le ht(w) + 1$ for every outgoing residual edge $v \to w$ and $ht(u) \le ht(v) \le ht(v) + 1$ for every incoming residual edge $u \to v$.

In both cases, compatibility of the new height function and the new pseudoflow follows from the inductive hypothesis. □

Lemma F.4. *After each step of the push-relabel algorithm, there is no residual path from s to t.*

Proof: Suppose to the contrary that there is a simple residual path $v_0 \to v_1 \to \cdots \to v_k$, where $v_0 = s$ and $v_k = t$. This path does not repeat vertices, so $k < |V|$. Because the height of s and t never change, we have $ht(v_0) = |V|$ and $ht(v_k) = 0$. Finally, compatibility implies $ht(v_i) \ge ht(v_{i-1}) - 1$, and thus by induction $ht(v_i) \ge ht(v_0) - i$, for each index i. In particular, we have $ht(v_k) \ge |V| - k > 0$, giving us a contradiction. □

Lemma F.5. ***If** the push-relabel algorithm terminates, it returns a maximum (s, t)-flow.*

Proof: The algorithm terminates only when there are no active vertices, which means that every vertex except s and t has zero residual balance. A pseudoflow with non-zero residual balance is *definition* of a flow! Any flow whose residual graph has no paths from s to t is a maximum (s, t)-flow. □

For a full proof of correctness, we still need to prove that the algorithm terminates, but the easiest way to prove termination is by proving an upper bound on the running time. In the analysis, we distinguish between two types of push operations. A push along edge $u \to v$ is ***saturating*** if we have $\psi(u \to v) = c(u \to v)$ after the push, and ***non-saturating*** otherwise.

Lemma F.6. *The push-relabel algorithm performs at most $O(V^2)$ relabel operations.*

Proof: At every stage of the algorithm, each active node v has a residual path back to s (because we can follow some unit of flow from s to v). Compatibility now implies that the height of each active node is less than $2|V|$. But we only change the height of a node when it is active, and then only upward, so the height of *every* node is less than $2|V|$. We conclude that each node is relabeled at most $2|V|$ times. □

Lemma F.7. *The push-relabel algorithm performs at most $O(VE)$ saturating pushes.*

Proof: Consider an arbitrary edge $u \to v$. We only push along $u \to v$ when $ht(u) > ht(v)$. If this push is saturating, it removes $u \to v$ from the residual graph. This edge reappears in the residual graph only when we push along the reversed edge $v \to u$, which is only possible when $ht(u) < ht(v)$. Thus, between any saturating push through $u \to v$ and the reappearance of $u \to v$ in the residual graph, vertex v must be relabeled at least twice. Similarly, vertex u must be relabeled at least twice before the next push along $u \to v$. By the previous lemma, u and v are each relabeled at most $2|V|$ times. We conclude that there are at most $|V|$ saturating pushes along $u \to v$. □

Lemma F.8. *The push-relabel algorithm performs at most $O(V^2E)$ non-saturating pushes.*

Proof: Define the ***potential* Φ** of the current residual graph to be the sum of the heights of all active vertices. This potential is always non-negative, because heights are non-negative. Moreover, we have $\Phi = 0$ when the algorithm starts (because every active node has height zero) and $\Phi = 0$ again when the algorithm terminates (because there are no active vertices).

Every relabel operation increases Φ by 1. Every saturating push along $u \to v$ makes v active (if it wasn't already), and therefore increases Φ by at most $ht(v) \leq 2|V|$. Thus, the total potential increase from all relabels and saturating pushes is at most $O(V^2E)$.

On the other hand, every non-saturating push along $u \to v$ makes the vertex u inactive and makes v active (if it wasn't already) and therefore *decreases* the potential by at least $ht(u) - ht(v) \geq 1$.

Because the potential starts and ends at zero, the total potential *decrease* from all non-saturating pushes must equal the total potential *increase* from the other operations. The lemma follows immediately. □

With appropriate elementary data structures, we can perform each push in $O(1)$ time, and each relabel in time proportional to the degree of the node. It follows that the algorithm runs in ***$O(V^2E)$ time***; in particular, the algorithm always terminates with the correct output!

Like the Ford-Fulkerson algorithm, the push-relabel approach can be made more efficient by carefully choosing *which* push or relabel operation to perform at each step. Two natural choices lead to faster algorithms:

- **FIFO:** The active vertices are kept in a standard queue. At each step, we remove the active vertex from the front of the queue, and then either push from or relabel that vertex until it becomes inactive. Any newly active vertices are inserted at the back of the queue. This rule reduces the number of non-saturating pushes to $O(V^3)$, and so the resulting algorithm runs in $\boldsymbol{O(V^3)}$ ***time***.
- **Highest label:** At each step, we either push from or relabel the vertex with maximum height (breaking ties arbitrarily). This rule reduces the number of non-saturating pushes to $O(V^2\sqrt{E})$, and so the resulting algorithm runs in $\boldsymbol{O(V^2\sqrt{E})}$ ***time***.

With more advanced data structures that support pushing flow along more than one edge at a time, the running time of the push-relabel algorithm can be improved to $\boldsymbol{O(VE\log(V^2/E))}$. (This was one of the theoretically-fastest algorithms known before Orlin's algorithm.) In practice, however, this optimization is usually slower than the more basic algorithm that handles one edge at a time.

Exercises

1. Recall from Chapter 11 that a ***path cover*** of a directed acyclic graph is a collection of directed paths, such that every vertex in G appears in at least one path. We previously saw how to compute *disjoint* path covers (where each vertex lies on *exactly* one path) by reduction to maximum bipartite matching. Your task in this problem is to compute path covers *without* the disjointness constraint.
   (a) Suppose you are given a dag G with a unique source s and a unique sink t. Describe an algorithm to find the smallest path cover of G in which every path starts at s and ends at t.
   (b) Describe an algorithm to find the smallest path cover of an arbitrary dag G, with no additional restrictions on the paths.

2. (a) Prove that any flow f in a network $G = (V, E)$ with non-zero balance constraints (and no source or target) can be expressed as a weighted sum of directed paths and directed cycles, such that
      - each path leads from an excess node to a deficit node;
      - a directed edge $u \to v$ appears in at least one path or cycle if and only if $f(u \to v) > 0$; and
      - the total number of paths and cycles is at most E.
   (b) Describe an algorithm to construct such a decomposition in $O(VE)$ time.

3. Let $G = (V, E)$ be an arbitrary flow network with source s and sink t. Recall that a ***preflow*** is a pseudoflow $\psi : E \to \mathbb{R}$ where every vertex except s has non-negative residual balance; that is, for each vertex v, we have
$$\sum_u \psi(u \to v) > \sum_w \psi(v \to w).$$

A preflow ψ is *feasible* if $0 \le \psi(u{\to}v) \le c(u{\to}v)$ for every edge $u{\to}v$. A ***maximum*** preflow if is a feasible preflow such that the net flow into t

$$\sum_u \psi(u{\to}t) - \sum_w \psi(t{\to}w)$$

is as large as possible. Describe an algorithm to transform an arbitrary maximum pseudoflow into a maximum flow in $O(VE)$ time. *[Hint: Flow decomposition!]*

4. (a) An ***edge cover*** of an undirected graph $G = (V, E)$ is a subset of edges $C \subseteq E$ such that every vertex is the endpoint of *at least* one edge in C. (Thus, edge covers are the "opposite" of matchings.) Describe and analyze an efficient algorithm to compute the *smallest* edge cover of a given bipartite graph.

   (b) Describe and analyze an algorithm for the following more general problem. The input consists of a bipartite graph $G = (V, E)$ and two functions $\ell, u : V \to \mathbb{N}$. Your algorithm should either output a subset of edges $C \subseteq E$ such that each vertex $v \in V$ is incident to at least $\ell(v)$ edges and at most $u(v)$ edges in C, or correctly report that no such subset exists.

5. (a) Suppose we are given a directed graph G, two vertices s and t, and two functions $\ell, c : V \to \mathbb{R}$ over the *vertices*. An (s,t)-flow $f : E \to \mathbb{R}$ in this network is feasible if and only if the total flow into each vertex v (except s and t) lies between $\ell(v)$ and $c(v)$:

$$\ell(v) \le \sum_{u \to v} f(u{\to}v) \le c(v).$$

   Describe an efficient algorithm to compute a maximum (s,t)-flow in this network.

   (b) Suppose we are given a directed graph G, two vertices s and t, and two functions $b^-, b^+ : V \to \mathbb{R}$ over the vertices. An (s,t)-flow $f : E \to \mathbb{R}$ in this network is feasible if and only if the total ***net*** flow into each vertex v (except s and t) lies between $b^-(v)$ and $b^+(v)$:

$$b^-(v) \le \sum_{u \to v} f(u{\to}v) - \sum_{v \to w} f(v{\to}w) \le b^+(v).$$

   Describe an efficient algorithm to compute a maximum (s,t)-flow in this network.

   (c) Describe an efficient algorithm to compute a maximum (s,t)-flow in a network with *all* of the features we've seen so far:
   - upper and lower bounds on the flow through each edge,
   - upper and lower bounds on the flow into each vertex, and
   - upper and lower bounds on the flow balance at each vertex.

La distance n'y fait rien; il n'y a que le premier pas qui coûte.
[The distance is nothing; it is only the first step that costs.]

— Marie Anne de Vichy-Chamrond, marquise du Deffand,
letter to Jean le Rond d'Alembert, July 7, 1763

Tam quaestiones altioris indaginis poscuntur.
[Then these questions require further investigation.]

— Carl Gustav Jacob Jacobi, unpublished notes (c. 1836)

Cecil Graham: *What is a cynic?*
Lord Darlington: *A man who knows the price of everything, and the value of nothing.*
Cecil Graham: *And a sentimentalist, my dear Darlington, is a man who sees an absurd value in everything and doesn't know the market price of any single thing.*

— Oscar Wilde, *Lady Windermere's Fan, A Play About a Good Woman* (1892)

G

Minimum-Cost Flows

[Read Chapters 8, 10, 11, and F first.]
Status: Text beta. Needs figures.

In this final chapter on flows, we consider a significant generalization of the maximum-flow problem that (as far as we know) cannot be solved by modifying the graph and applying a standard flow algorithm. The input to our problem consists of a flow network without special source and target vertices, where each edge e has a ***cost $\$(e)$***, in addition to the usual edge capacities and vertex balances. Edge costs can be positive, negative, or zero. The total cost of any flow f is then defined as

$$\$(f) = \sum_{e \in E} \$(e) \cdot f(e).$$

The ***minimum-cost flow*** problem asks for a feasible flow with minimum cost, instead of a feasible flow with maximum value.

G.1 Minimum-Cost Circulations

The simplest special case of the minimum-cost flow problem requires the input flow network to satisfy two constraints:

- All vertex balances are zero. Thus, every flow f must satisfy the conservation constraint $\sum_u f(u{\to}v) = \sum_w f(v{\to}w)$ at every vertex v.
- Edges have non-negative capacities and zero lower bounds. Thus, every *feasible* flow f satisfies the capacity constraint $0 \le f(u{\to}v) \le c(u{\to}v)$ at every edge $u{\to}v$.

Because the vertex balances are all zero, any feasible flow in such a network is actually a *circulation*—a sum of directed cycles. This special case is normally called the ***minimum-cost circulation*** problem.

The standard maximum-flow problem can be reduced to the minimum-cost circulation problem as follows. Suppose we are given a flow network $G = (V, E)$ with source and target vertices s and t and non-negative edge capacities $c(e)$. Let $G' = (V, E')$ be the network constructed from G by assigning cost 0 to every edge, and then adding a single edge $t{\to}s$ with infinite capacity and cost -1. Every feasible (s,t)-flow f in G can be extended to a feasible circulation in G' by defining

$$f(t{\to}s) = |f| = \sum_w f(s{\to}w) - \sum_u f(u{\to}s);$$

moreover, the cost of the resulting circulation is precisely $-|f|$. Conversely, for any feasible circulation $f : E' \to \mathbb{R}$, the restriction $f|_E$ is a feasible (s,t)-flow in G with value $-f(t{\to}s)$. Thus, any maximum-value (s,t)-flow in G corresponds to a minimum-cost circulation in G' and vice versa.

★★★ Figure!

Cycle Canceling

We can solve any instance of the minimum-cost circulation problem using a natural generalization of the Ford-Fulkerson augmenting path algorithm called ***cycle canceling***. This algorithm is normally attributed to Morton Klein in 1967, but the key insights date back at least to A. N. Tolstoy's studies of railway transportation networks in the late 1920s.

Consider an arbitrary circulation f in G. As usual, the ***residual network*** G_f consists of all edges $u{\to}v$ with non-zero residual capacity $c_f(u{\to}v)$. For each residual edge $u{\to}v$ in G_f that is not an edge in the original graph G, we define its cost as follows:

$$\$(u{\to}v) := -\$(v{\to}u).$$

★★★ Figure showing flow f and residual graph G_f

Now let γ be a directed cycle in the residual graph G_f. We define the residual capacity $c_f(\gamma)$ of γ to be the minimum residual capacity of the edges in γ, and the cost $\$(\gamma)$ of γ to be the *sum* of the costs of the edges in γ. More concisely:

$$c_f(\gamma) := \min_{e\in\gamma} c_f(e) \qquad \$(\gamma) := \sum_{e\in\gamma} \$(e)$$

Exactly as in Ford-Fulkerson, we can ***augment*** the circulation f, by pushing R units of flow around γ, to obtain a new circulation f':

$$f'(u \to v) = \begin{cases} f(u \to v) + c_f(\gamma) & \text{if } u \to v \in \gamma \\ f(u \to v) - c_f(\gamma) & \text{if } v \to u \in \gamma \\ f(u \to v) & \text{otherwise} \end{cases}$$

Straightforward calculation now implies that

$$\$(f') = \$(f) + c_f(\gamma) \cdot \$(\gamma).$$

In particular, if $\$(\gamma) < 0$, the new circulation f' has lower cost than the original circulation f. We immediately conclude that ***a feasible circulation f is a minimum-cost circulation if and only if G_f contains no negative cycles.***

Figure showing negative cycle and updated flow f'

★★★

Finally, Klein's cycle canceling algorithm initializes f to the all-zero circulation, and then repeatedly augments f along an arbitrary negative residual cycle, until there are no more negative residual cycles. For the special instances generated by our reduction from maximum flow, every negative-cost residual cycle consists of a path from s to t and the sole negative edge $t \to s$, so cycle canceling is equivalent to Ford and Fulkerson's augmenting path algorithm.

Analysis

In each iteration of the algorithm, we can find a negative-cost cycle in $O(VE)$ time using a straightforward modification of the Shimbel-Bellman-Ford shortest path algorithm (hint, hint). If both the capacity and the cost of each edge is an integer, then each cycle-augmentation decreases the cost of the circulation by a positive integer, and therefore by at least 1. It follows that the cycle canceling algorithm runs in $\boldsymbol{O(VE \cdot |\$(f^*)|)}$ ***time***, where f^* is the final circulation. The crude bound $\$(f^*) \geq -ECK$, where $C = \max_e c(e)$ and $K = \max_e |\$(e)|$, also implies that the algorithm runs in at most $\boldsymbol{O(VE^2CK)}$ ***time***. As one might expect from Ford-Fulkerson, these time bounds are exponential in the complexity of the input even when capacities are integers, both time bounds are tight in the worst case, and the algorithm may not terminate or even approach a minimum-cost circulation in the limit if any capacities are irrational.[1]

Like Ford-Fulkerson, more careful choices of *which* cycle to cancel lead to more efficient algorithms. Unfortunately, some natural choices are NP-hard to compute,

[1] If the input network has integer capacities, the algorithm must eventually terminate even when costs are irrational, because there are only a finite number of feasible integer circulations. However, the number of iterations could still be exponential, even if we always cancel the cycle with most negative total cost!

including the cycle with the most negative total cost and the negative cycle with the fewest edges. In the late 1980s, Andrew Goldberg and Bob Tarjan developed a minimum-cost flow algorithm that repeatedly cancels the so-called ***minimum-mean cycle***, which is the cycle whose *average cost per edge* is smallest. By combining an algorithm of Karp to compute minimum-mean cycles in $O(VE)$ time, efficient dynamic tree data structures, and other sophisticated techniques that are (unfortunately) beyond the scope of this class, their algorithm achieves a running time of $\boldsymbol{O(VE^2 \log V)}$.

Reduction from General Minimum-Cost Flow

In the standard general formulation of the minimum-cost flow problem, we are given a directed flow network $G = (V, E)$ with the following additional data:

- a capacity function $c : E \to \mathbb{R}$ such that $c(e) \geq 0$ for all $e \in E$;
- a lower bound function $\ell : E \to \mathbb{R}$ such that $0 \leq \ell(e) \leq c(e)$ for all $e \in E$;
- a balance function $b : V \to \mathbb{R}$ such that $\sum_v b(v) = 0$; and
- a cost function $\$: E \to \mathbb{R}$.

As usual, a feasible flow in G is a function $f : E \to \mathbb{R}$ that satisfies the capacity constraints $\ell(e) \leq f(e) \leq c(e)$ at every edge e and the balance constraints $\sum_u f(u \to v) - \sum_w f(v \to w) = b(v)$ at every vertex v. The minimum-cost flow problem asks for a feasible flow f with minimum total cost $\$(f) = \sum_e \$(e) \cdot f(e)$.

We can solve the general minimum-cost flow problem in two stages. First, find *any* feasible flow f in G, by reducing to the maximum-flow problem as described in the previous chapter. If there is no feasible flow in G, we can immediately report failure. Otherwise, in the second stage, we compute a minimum-cost circulation f' in the residual graph G_f, using (for example) cycle canceling. Tedious definition-chasing implies that the function $f + f'$ is a minimum-cost flow in the original network G. In total, this algorithm runs in $\boldsymbol{O(VE^2 \log V)}$ ***time***.

G.2 Successive Shortest Paths

The cycle canceling algorithm implies a natural characterization of minimum-cost flows. Consider a general flow network $G = (V, E)$ with capacities, lower bounds, costs, and balances. A pseudoflow $\psi : E \to \mathbb{R}$ is a minimum-cost flow if and only if it satisfies three conditions:

- **Feasible:** $\ell(e) \leq \psi(e) \leq c(e)$ for every edge e.
- **Balanced:** $\sum_u \psi(u \to v) - \sum_w \psi(v \to w) = b(v)$ for every vertex v.
- **Locally optimal:** The residual graph G_ψ contains no negative-cost cycles.

The cycle-canceling algorithm incrementally improves a feasible and balanced pseudoflow (that is, a circulation) until it is also locally optimal. A complementary strategy called ***successive shortest paths*** incrementally improves a feasible and *locally optimal*

pseudoflow until it is also *balanced*. The successive shortest paths algorithm was initially proposed by Ford and Fulkerson in the mid-1950s, and then later rediscovered by William Jewell in 1958, Masao Iri in 1960, and Robert Busacker and Paul Gowen in 1960.

The successive shortest paths algorithm requires two simplifying assumptions about the initial flow network G:

- **All lower bounds are zero.** Otherwise, think of the lower-bound function ℓ as a pseudoflow (sending $\ell(e)$ units of flow across each edge e), and consider the residual graph G_ℓ.
- **All costs are non-negative.** Otherwise, consider the residual graph of the following pseudoflow:

$$\overline{c}(u{\to}v) = \begin{cases} c(u{\to}v) & \text{if } \$(u{\to}v) \le 0 \\ 0 & \text{otherwise} \end{cases}$$

Enforcing these assumptions may introduce non-zero balances at the vertices, even if all balances in the initial flow network are zero. These two assumptions imply that the all-zero pseudoflow is both feasible and locally optimal, but not necessarily balanced.

The successive shortest path algorithm begins by initializing ψ to the all-zero pseudoflow, and then repeatedly augments ψ by pushing flow along a *shortest* path in the residual graph G_ψ from any vertex with negative residual balance to any vertex with positive residual balance, where the length of a path in G_ψ is the sum of the residual costs of its edges. The algorithm is described in more detail below:

```
SuccessiveShortestPaths(V, E, c, b, $):
  for every edge e ∈ E
      ψ(e) ← 0
  B ← Σ_v |b(v)|/2
  while B > 0
      construct G_ψ
      s ← any vertex with b_ψ(s) < 0
      t ← any vertex with b_ψ(t) > 0
      σ ← shortest path in G_ψ from s to t
      Augment(s, t, σ)
  return ψ
```

```
Augment(s, t, σ):
  R ← min{−b_ψ(s), b_ψ(t), min_{e∈σ} c_ψ(e)}
  B ← B − R
  for every directed edge e ∈ σ
      if e ∈ E
          ψ(e) ← ψ(e) + R
      else ⟨⟨rev(e) ∈ E⟩⟩
          ψ(e) ← ψ(e) − R
```

(To simplify the pseudocode, I have assumed implicitly that the residual graph G_ψ is always strongly connected. If necessary, this assumption can be enforced by adding a new vertex with zero balance and with infinite-cost edges to and from every other vertex in G. If SuccessiveShortestPaths ever tries to push flow along one of these new edges, we can immediately report that the original flow problem is infeasible.)

SuccessiveShortestPaths is *obviously* an instantiation of the FeasibleFlow algorithm from the previous chapter; the only difference is that we require each augmenting path σ to be a shortest path, instead of an arbitrary path. So *obviously* the

algorithm either returns a feasible flow (that is, a feasible and balanced pseudoflow) or reports correctly that no such flow exists.

Unfortunately, like almost(?) every other sentence that uses the words "obviously", the previous argument misses a crucial subtlety: Shortest paths are only *well-defined* in graphs without negative cycles! To argue that SUCCESSIVESHORTESTPATHS is even a well-defined algorithm, much less a *correct* algorithm, we need to prove the following lemma:

Lemma G.1. *After every iteration of the main loop of SUCCESSIVESHORTESTPATHS, the feasible pseudoflow ψ is locally optimal; that is, the residual graph G_ψ contains no negative cycles.*

Proof: We prove the lemma by induction. The base case is immediate; the initial all-zero pseudoflow is locally optimal because the graph G has no negative edges.

Suppose that ψ is locally optimal at the beginning of an iteration of the main loop. By definition, there are no negative cycles in the residual graph G_ψ, so shortest paths are well-defined. Let s be an arbitrary supply vertex and let t be an arbitrary demand vertex in G_ψ. Let σ be a shortest path in G_ψ, and let ψ' be the resulting pseudoflow after augmenting along σ. We need to prove that ψ' is locally optimal.

For the sake of argument, suppose the residual graph $G_{\psi'}$ contains a negative cycle γ. Let $f = \sigma + \gamma$ denote the pseudoflow in G_ψ obtained by sending one unit of flow along σ and then one unit of flow along γ. Although γ may contain edges that are not in G_ψ, all such edges must be reversals of edges in σ; thus, f is positive only on edges in G_ψ. Routine calculations imply that f is an integral (s,t)-flow in G_ψ with value 1.[2] Thus, by the Flow Decomposition Lemma, f can be decomposed into a single path σ' from s to t and possibly one or more cycles $\gamma_1, \ldots, \gamma_k$. Our assumption that ψ is locally optimal implies that each cycle γ_i has non-negative cost, and therefore $\$(\sigma') \le \$(f) < \$(\sigma)$. But this is impossible, because σ is a *shortest* path from s to t. □

★★★

Figure with a complicated instance of σ and γ.

For the special instances of minimum-cost flow generated by our reduction from maximum flow, the only supply node is the source vertex s, the only deficit node is the target vertex t, and every edge in the network has cost 0 except the edge $s \to t$, which has cost 1. Thus, for these instances, the successive shortest paths is equivalent to Ford and Fulkerson's augmenting path algorithm (followed by a single augmentation of the edge $s \to t$).

[2] Here I'm ignoring the residual balances in G_ψ. Note that f may not be a *feasible* flow in G_ψ, but that doesn't matter.

Analysis

In each iteration of the algorithm, we can find a shortest in $O(VE)$ time using a the Shimbel-Bellman-Ford shortest path algorithm. (Actually, we can use Dijkstra's algorithm to compute the *first* shortest path, but augmenting along a path of positive-cost edges introduces residual edges with negative costs, so we can't use Dijkstra's algorithm in later iterations.)

Assuming the capacity of each edge is an integer, then each augmentation decreases the total absolute balance B by a positive integer, and therefore by at least 1. It follows that the cycle canceling algorithm halts after at most B iterations and thus runs in $\boldsymbol{O(VEB)}$ ***time***. As one might expect from Ford-Fulkerson, this time bound is exponential in the complexity of the input when capacities are integers, the time bound are tight in the worst case, and the algorithm may not terminate or even approach a minimum-cost circulation in the limit if any capacities are irrational.

G.3 Node Potentials and Reduced Costs

The main bottleneck in Ford and Fulkerson's successive shortest paths algorithm is computing a shortest paths in every iteration. Because some edges in the residual graph may have negative costs, we are forced to use Bellman-Ford instead of Dijkstra's algorithm. Or so it seems.

The following clever trick to speed up successive shortest paths was proposed by Jack Edmonds and Richard Karp in 1969 (published in 1972) and independently by Nobuaki Tomizawa in 1970 (published in 1971). The same trick was later applied by Donald Johnson in the late 1970s to speed up all-pairs shortest-path algorithms; Johnson attributes the trick to Edmonds and Karp.

Suppose each vertex v in the flow network has been assigned an arbitrary real ***potential*** $\boldsymbol{\pi(v)}$. We define the ***reduced cost*** of each edge with respect to potential function π as follows:

$$\boldsymbol{\$^{\pi}(u \to v)} := \pi(u) + \$(u \to v) - \pi(v).$$

Then for any path α from u to v, we immediately have

$$\$^{\pi}(\alpha) = \sum_{e \in \alpha} \$^{\pi}(e) = \pi(u) + \sum_{e \in \alpha} \$(e) - \pi(v) = \pi(u) + \$(\alpha) - \pi(v).$$

The potentials of all the intermediate vertices on α cancel out; intuitively, for each intermediate vertex x, we get an "entrance gift" of $\pi(x)$ when the path enters x, which we immediately pay back as an "exit tax" when the path leaves x. This simple observation has two important consequences:

- Shortest paths with respect to the reduced cost function $\$^{\pi}$ are identical to shortest paths with respect to the original cost function $\$$.

- For any cycle γ, we immediately have $\$^\pi(\gamma) = \$(\gamma)$; thus, a flow network G has no cycles with negative cost if and only if G has no cycles with negative *reduced* cost.

Edmonds, Karp, and Tomizawa observed that if we choose the right potential function, we can ensure that all reduced edge costs are non-negative, allowing us to compute shortest paths using Dijkstra's algorithm instead of Bellman-Ford.

Lemma G.2. *A flow network G has no negative-cost cycles if and only if, for some potential function $\pi : V \to \mathbb{R}$, every edge in G has non-negative reduced cost.*

Proof: One direction is easy. Suppose G contains a cycle γ with negative cost. Then for every potential function π, we have $\$^\pi(\gamma) = \$(\gamma) < 0$, which implies that at least one edge in γ has negative reduced cost.

Conversely, suppose G has no negative-cost cycles. Then shortest path distances with respect to the cost function $\$$ are well-defined. Fix an arbitrary vertex s, and for every vertex v, let $\boldsymbol{dist(v)}$ denote the shortest-path distance from s to v. (If necessary, add zero-capacity, high-cost edges from s to every other vertex, so that these distances are all finite.) Then for every edge $u \to v$, we immediately have

$$dist(v) \le dist(u) + \$(u \to v),$$

since otherwise there would be a path from s through u to v with cost less than $dist(v)$; this inequality can be rewritten as

$$dist(u) + \$(u \to v) - dist(v) \ge 0.$$

Thus, if use these shortest-path distances as potentials, every edge in G has non-negative reduced cost. □

Speeding up Successive Shortest Paths

Now we seem to have a chicken-and-egg problem: We need a good potential function to compute shortest paths quickly, but we need shortest paths to compute a good potential function!

We can break this apparent cycle as follows. Because the original flow network G has only non-negative edge costs, we can start with $\pi(v) = 0$ at every vertex v. Then after each iteration of the main loop, we update the vertex potentials to ensure that reduced costs in the new residual graph are still non-negative. Specifically, ***we use the shortest-path distances from each iteration as the vertex potentials in the next iteration***, as described in the pseudocode below.

```
FastSuccessiveShortestPaths(V, E, c, b, $):
  for each edge e ∈ E
      ψ(e) ← 0
  for each vertex v ∈ V
      π(e) ← 0
  B ← Σ_v |b(v)|/2
  while B > 0
      construct G_ψ
      s ← any vertex with b_ψ(s) < 0
      t ← any vertex with b_ψ(t) > 0
      dist ← Disjkstra(G_ψ, s, $^π)
      σ ← shortest path in G_ψ from s to t    ⟨⟨computed in the previous line⟩⟩
      Augment(s, t, σ)
      for each vertex v ∈ V
          π(v) ← dist(v)
  return ψ
```

Lemma G.3. *After every iteration of the main loop of* FastSuccessiveShortestPaths*, every edge $u \to v$ in the residual graph G_ψ has non-negative reduced cost: $\$^\pi(u \to v) \geq 0$.*

Proof: We prove the lemma by induction. The base case is immediate, because the original graph G has no negative-cost edges.

Suppose at the beginning of some iteration that for every edge $u \to v$ in G_ψ, we have $\$^\pi(u \to v) \geq 0$. Let s be an arbitrary supply vertex, and for any vertex v, let $dist(v)$ denote the shortest-path distance from s to v with respect to the reduced cost function $\$^\pi$. Let σ be a shortest path in G_ψ from s to some demand vertex t, and let ψ' be the resulting pseudoflow after augmenting along σ.

Now let $u \to v$ be an arbitrary edge in the new residual graph $G_{\psi'}$. We need to prove that $\$^{dist}(u \to v) \geq 0$. There are two cases to consider:

- Suppose $u \to v$ is an edge in the old residual graph G_ψ. Then following the proof of Lemma 2, we immediately have $\$^{dist}(u \to v) = dist(u) + \$^\pi(u \to v) - dist(v) \geq 0$.
- Suppose $u \to v$ is not an edge in the old residual graph G_ψ. Then $u \to v$ must be the reversal of an edge in σ. It follows that $dist(u) = dist(v) + \$^\pi(v \to u)$, and therefore

$$\begin{aligned}
\$^{dist}(u \to v) &:= dist(u) + \$^\pi(u \to v) - dist(v) \\
&= dist(u) + \pi(u) + \$(u \to v) - \pi(v) - dist(v) \\
&= dist(u) + \pi(u) - \$(v \to u) - \pi(v) - dist(v) \\
&= dist(u) - \$^\pi(v \to u) - dist(v) \\
&= 0.
\end{aligned}$$

In both cases, we conclude that $\$^{dist}(u \to v) \geq 0$, as required. □

This lemma implies that we can compute shortest paths at every iteration of FAST-SUCCESSIVESHORTESTPATHS using Disjktra's algorithm in $O(E \log V)$ time. It follows that the overall algorithm runs in $\boldsymbol{O(BE \log V)}$ ***time***.

The fastest minimum-cost flow algorithm currently known (at least among algorithms whose running times depend only on V and E), due to James Orlin in the early 1990s, reduces the problem to $O(E \log V)$ iterations of Dijkstra's shortest-path algorithm and therefore runs in $\boldsymbol{O(E^2 \log^2 V)}$ ***time***.

G.4 Transshipment and Transportation

The ***transshipment*** and ***transportation*** problems are two interesting special cases of minimum-cost flows, where the flow values are not directly limited by capacity constraints on the edges, but rather indirectly limited by the balance conditions at the vertices.

A transshipment network is a directed graph $G = (V, E)$ with vertex balances $b : V \to \mathbb{R}$ and edge costs $\$: E \to \mathbb{R}$, but *no* capacity function; each edge is allowed to carry *any* non-negative amount of flow. Thus, a feasible flow in such a network is a function $f : E \to \mathbb{R}$ such that $f(e) \geq 0$ for every edge e and $\sum_u f(u \to v) - \sum_w f(v \to w) = b(v)$ for every vertex v. As usual, the transshipment problem asks for a feasible flow with minimum total cost $\$(f) = \sum_e \$(e) \cdot f(e)$.

The transshipment problem was first cast in this discrete form by Alex Orden in 1956. However, transshipment has been a common practice in long-distance commerce for as long as there has been long-distance commerce.

Traditionally, the vertices in a transshipment network are divided into three categories: *supply* or *source* vertices have negative balance; *demand* or *target* vertices have positive balance, and ***transshipment*** vertices have zero balance. The *transportation* problem is a special case of the transshipment problem in which the input satisfies two additional conditions:

- There are no transshipment nodes; every vertex has a non-zero balance constraint.
- Every edge $u \to v$ leaves a supply node u and enters a demand node v. Thus, the flow network is a directed bipartite graph, where the vertex partition separates supply vertices from demand vertices.

Some formulations of the transportation problem require G to be a *complete* bipartite graph, but if necessary we can introduce any missing edges with infinite cost.

The transportation problem was first cast in this modern form by Frank Hitchcock in 1942 and independently by Tjalling Koopmans in 1947; however, earlier versions of the problem were studied by Leonid Kantorivich in the late 1930s, A. N. Tolstoy in the 1920s and 1930s, and even earlier (as a continuous mass-transport problem) by Gaspard Monge in the late 1700s.

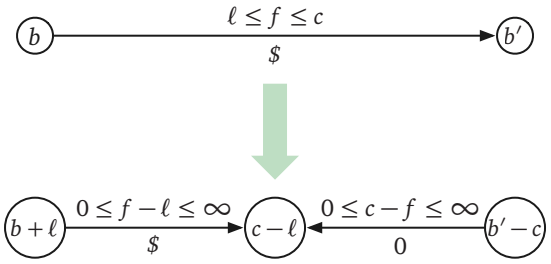
Reducing Minimum-Cost Flow to Transportation

Like the minimum-cost circulation problem, is is possible to reduce arbitrary instances of minimum-cost flow to the transportation problem; the following reduction was described by Delbert Fulkerson in 1960, generalizing Orden's 1956 reduction from the transshipment problem. Let $G = (V, E)$ be an arbitrary flow network, where every edge e has an associated lower bound $\ell(e)$, capacity $c(e)$, and cost $\$(e)$, and every vertex v has an associated balance $b(v)$.

We can remove the capacity constraints from any single edge $u \to v$ as follows. First, replace the edge with a pair of edges $u \to z_{uv}$ and $v \to z_{uv}$, where z_{uv} is a new vertex. Next, set the various constraints for the new vertex and edges as follows:

- $\ell(u \to z_{uv}) = \ell(v \to z_{uv}) = 0$,
- $c(u \to z_{uv}) = c(v \to z_{uv}) = \infty$,
- $\$(u \to z_{uv}) = \$(u \to v)$ and $\$(v \to z_{uv}) = 0$, and
- $b(z_{uv}) = c(u \to v) - \ell(u \to v)$;

Finally, increase the balance $b(u)$ by $\ell(u \to v)$ and decrease the balance $b(v)$ by $c(u \to v)$. Call the resulting flow network G'.



Removing capacity constraints from a single edge.

Given any feasible flow f in the original network G, we can construct a corresponding flow f' in G' by setting

$$f'(e) = \begin{cases} f(u \to v) - \ell(u \to v) & \text{if } e = u \to z_{uv}, \\ c(u \to v) - f(u \to v) & \text{if } e = v \to z_{uv}, \\ f(e) & \text{otherwise.} \end{cases}$$

Routine calculations imply that f' is a feasible flow in G' with cost $\$(f) - \ell(u \to v) \cdot \$(u \to v)$. Conversely, given a feasible flow f' in G', we can construct a corresponding flow f in G by setting

$$f(e) = \begin{cases} f'(u \to z_{uv}) + \ell(u \to v) & \text{if } e = u \to v, \\ f'(e) & \text{otherwise.} \end{cases}$$

Again, routine calculations imply that f is a feasible flow in G with cost $\$(f) + \ell(u \to v) \cdot \$(u \to v)$. We conclude, for both flow transformations, that f is a minimum-cost flow in G if and only if f' is a minimum cost flow in G'.

Applying this transformation to every edge in G, we obtain a bipartite transportation network H with $V + E$ vertices and $2E$ uncapacitated edges in $O(E)$ time. Given a minimum-cost flow in H, we can easily recover a minimum-cost flow in the original network G in $O(E)$ time.

Special case: Assignment

We conclude by describing two further interesting special cases of the transshipment/transportation problem.

The ***assignment*** problem is a special case of the Hitchcock-Koopman's transportation problem in which every supply vertex has balance -1 and every demand vertex has balance $+1$. In other words, the assignment problem asks for a ***minimum-weight perfect matching*** in a bipartite graph with weighted edges (where "weight" is what we previously called "cost"). The assignment problem can also be reformulated as a ***maximum***-weight perfect matching problem by negating all edge weights.

The assignment problem was first posed *and solved* by the German mathematician Carl Jacobi in the early 1800s, motivated by the problem of solving a system of ordinary differential equations. Jacobi's algorithm was not published until after his death.[3] Jacobi's algorithm was rediscovered by Harold Kuhn in the mid 1950s. Kuhn called his algorithm the "Hungarian method", because its relied on combinatorial results on perfect matchings published by Hungarian mathematicians Dénes Kőnig and Jenő Egerváry in 1931. (Alas, Jacobi was not Hungarian.) In 1957, topologist James Munkres observed that the Jacobi-Egerváry-Kuhn algorithm runs in $O(V^4)$ time; with more care, the running time can be improved to $O(V^3)$.

Any instance of the transportation problem *with integer balances* can be transformed into an equivalent instance of the assignment problem, by replacing each vertex v with $|b(v)|$ equivalent copies.

Special Case: Single-Supply Transshipment

Perhaps the most important special case of the *transhipment* problem requires that the network has exactly one supply node s. This special case is better known as the ***single-source shortest path*** problem. Specifically, let T be a shortest-path tree of G rooted at the supply vertex s, and let $dist(v)$ denote the shortest-path distance from s to v. We define a canonical flow $f_T : E \to \mathbb{R}$ for this spanning tree as follows:

$$f_T(u \to v) = \begin{cases} \sum_{w \downarrow v} b(w) & \text{if } u \to v \in T \\ 0 & \text{otherwise} \end{cases}$$

[3]Carl Gustav Jacob Jacobi. De investigando ordine systematis aequationum differentialum vulgarium cujuscunque. *J. Reine Angew. Math.* 64(4):297–320, 1865. Posthumously published by Carl Borchardt. English translation by François Ollivier: Looking for the order of a system of arbitrary ordinary differential equations. *Applicable Algebra in Engineering, Communication and Computing*, 20(1):7–32, 2009.

(Here $w \downarrow v$ indicates that w is a descendant of v in T.) Equivalently, f_t) is the sum of $V-1$ path flows, each sending $b(v)$ units of flow along the unique path from s to some other vertex v; thus,

$$\$(f_T) = \sum_v dist(v) \cdot b(v).$$

We can prove this canonical flow is optimal in (at least) two different ways. First, f_T is precisely the minimum-cost flow constructed by the successive shortest-path algorithm (at least if the shortest path tree T is unique). Second, following the proof of Lemma 1, if the residual graph G_{f_T} contained any negative cycles, we could find a shorter path from s to some other vertex v.

Ford's generic relaxation algorithm to compute shortest path trees is morally equivalent to cycle canceling. Ford's algorithm maintains a tentative shortest path tree T. Any tense edge $u \to v$ indicates a negative cycle in the residual graph $G_T = G_{f_T}$, consisting of $u \to v$ and the unique path from v to u in T. Relaxing $u \to v$ replaces the unique edge $x \to v$ in T with $u \to v$.

In fact, for *every* transhipment network G without negative cycles, there is an minimum-cost flow is non-zero only on the edges of some spanning tree of G.

Exercises

Need more!

★★★

1. Describe and analyze an algorithm for the following problem, first posed and solved by the German mathematician Carl Jacobi in the early 1800s.

   > *Disponantur nn quantitates $h_k^{(i)}$ quaecunque in schema Quadrati, ita ut k habeantur n series horizontales et n series verticales, quarum quaeque est n terminorum. Ex illis quantitatibus eligantur n transversales, i.e. in seriebus horizontalibus simul atque verticalibus diversis positae, quod fieri potest* $1.2 \ldots n$ *modis; ex omnibus illis modis quaerendum est is, qui summam n numerorum electorum suppeditet maximam.*

   For the tiny minority of students who are not fluent in mid-19th century academic Latin, here is a modern English formulation of Jacobi's problem. Suppose we are given an $n \times n$ matrix M. Describe and analyze an algorithm that computes a permutation σ that maximizes the sum $\sum_{i=1}^{n} M_{i,\sigma(i)}$, or equivalently, permutes the columns of M so that the sum of the elements along the diagonal is as large as possible.

   Please write a complete, self-contained solution in English, not in mid-19th century academic Latin. Or Hungarian.

2. An ***(s, t)-series-parallel*** graph is an directed acyclic graph with two designated vertices s (the *source*) and t (the *target* or *sink*) and with one of the following structures:

- **Base case:** A single directed edge from s to t.
- **Series:** The union of an (s,u)-series-parallel graph and a (u,t)-series-parallel graph that share a common vertex u but no other vertices or edges.
- **Parallel:** The union of two smaller (s,t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

Recall that any series-parallel graph can be represented by a binary *decomposition tree*, whose interior nodes correspond to series compositions and parallel compositions, and whose leaves correspond to individual edges. The decomposition tree can be constructed in $O(V+E)$ time.

(a) Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s,t)-series-parallel graph whose edges have *unit* capacity and arbitrary costs.

♥(b) Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s,t)-series-parallel graph whose edges have *arbitrary* capacities and costs.

3. Every year, Professor Dumbledore assigns the instructors at Hogwarts to various faculty committees. There are n faculty members and c committees. Each committee member has submitted a list of their *prices* for serving on each committee; each price could be positive, negative, zero, or even infinite. For example, Professor Snape might declare that he would serve on the Student Recruiting Committee for 1000 Galleons, that he would *pay* 10000 Galleons to serve on the Defense Against the Dark Arts Course Revision Committee, and that he would not serve on the Muggle Relations committee for any price.

   Conversely, Dumbledore knows how many instructors are needed for each committee, as well as a list of instructors who would be suitable members for each committee. (For example: "Dark Arts Revision: 5 members, anyone but Snape.") If Dumbledore assigns an instructor to a committee, he must pay that instructor's price from the Hogwarts treasury.

   Dumbledore needs to assign instructors to committees so that (1) each committee is full, (3) no instructor is assigned to more than three committees, (2) only suitable and willing instructors are assigned to each committee, and (4) the total cost of the assignment is as small as possible. Describe and analyze an efficient algorithm that either solves Dumbledore's problem, or correctly reports that there is no valid assignment whose total cost is finite.

4. Vince wants to borrow a certain amount of money from his friends as cheaply as possible, possibly after first arranging a sequence of intermediate loans. Each of Vince's friends have a different amount of money that they can lend (possibly zero). For any two people x and y, there is a maximum amount of money (possibly zero

or infinite) that x is willing to lend to y and a certain profit (possibly zero or even negative) that x expects from any loan to y.

For example, suppose Vince wants to borrow $\$100$ from his friends Ben and Naomi, who have the following constraints:

- Ben has $\$500$ available to lend.
- Ben is willing to lend up to $\$150$ to Vince at a profit of 20¢ per dollar.
- Ben is willing to lend up to $\$50$ to Naomi, at a loss of 10¢ per dollar.
- Naomi has $\$50$ available to lend.
- Naomi is willing to lend any amount of money to Vince, at a profit of 10¢ per dollar.
- Naomi is not willing to lend money to Ben.

If Vince borrows $\$100$ directly from Ben, he needs $\$120$ to pay off the loan. If Vince borrows $\$50$ from Ben and $\$50$ from Naomi, he needs $\$115$ to pay off the loan: $\$60$ for Ben and $\$55$ for Naomi. But if Vince asks Naomi to borrow $\$50$ from Ben and then borrows the entire $\$100$ from Naomi, then he needs only $\$110$ to pay off Naomi, who can then pay off Ben with just $\$45$. With the same constraints, the maximum amount of money that Vince can borrow is $\$250$.

Describe and analyze an algorithm that finds a sequence of loans that minimizes the amount Vince needs to pay everyone off, or correctly reports that Vince cannot borrow his desired amount. The input has the following components:

- An array $Money[1..n]$, where $Money[i]$ is the amount of money that friend i has.
- An array $MaxLoan[1..n, 0..n]$, where $MaxLoan[i, j]$ is the amount of money that friend i is willing to lend to friend j. "Friend 0" is Vince.
- An array $Profit[1..n, 0..n]$, where $Profit[i, j]$ is the profit per dollar that friend i expects from any load to friend j. Again, "friend 0" is Vince.
- The total amount T that Vince wants to borrow.

5. Suppose you are given a directed flow network G, in which every edge has an *integer* capacity and an *integer* cost, and each vertex has an *integer* balance, along with an *integer* minimum-cost flow f^* in G. Describe algorithm for the following problems:

   (a) IncCost($u \to v$): Increase the cost of $u \to v$ by 1 and update the minimum-cost flow.

   (b) DecCost($u \to v$): Decrease the cost of $u \to v$ by 1 and update the minimum-cost flow.

   Both algorithms should modify f^* so that it is still a minimum-cost flow, more quickly than recomputing a minimum-cost flow from scratch.

The greatest flood has the soonest ebb; the sorest tempest the most sudden calm; the hottest love the coldest end; and from the deepest desire oftentimes ensues the deadliest hate.

— Socrates

Th' extremes of glory and of shame, Like east and west, become the same.

— Samuel Butler, *Hudibras* Part II, Canto I (c. 1670)

Everything is dual; everything has poles; everything has its pair of opposites; like and unlike are the same; opposites are identical in nature, but different in degree; extremes meet; all truths are but half-truths; all paradoxes may be reconciled.

— *The Kybalion: A Study of The Hermetic Philosophy of Ancient Egypt and Greece* (1908)

Oh, that!

— John von Neumann to George Dantzig (May 1947)

H

Linear Programming

[Read Chapters F and G first.]

Status: Text beta. Needs figures.

H.1 Introduction

The maximum flow and minimum cut problems are examples of a general class of problems called ***linear programming***. Many other optimization problems fall into this class, including minimum spanning trees and shortest paths, as well as several common problems in scheduling, logistics, and economics. Linear programming was used implicitly by Fourier and Jacobi in the early 1800s, but it was first formalized and applied to problems in economics in the 1930s by Leonid Kantorovich. Kantorivich's work was hidden behind the Iron Curtain (where it was largely ignored) and therefore unknown in the West. Linear programming was rediscovered and applied to shipping problems in the late 1930s by Tjalling Koopmans. The first complete algorithm to solve linear programming problems, called the ***simplex method***, was published by George Dantzig in 1947. Koopmans first proposed the name "linear programming" in a discussion with Dantzig in 1948. Kantorovich and Koopmans shared the 1975 Nobel Prize in Economics

"for their contributions to the theory of optimum allocation of resources". Dantzig did not; his work was apparently too pure. Koopmans wrote to Kantorovich suggesting that they refuse the prize in protest of Dantzig's exclusion, but Kantorovich saw the prize as a vindication of his use of mathematics in economics, which his Soviet colleagues had written off as "a means for apologists of capitalism".

A linear programming problem—or more simply, a ***linear program***—asks for a vector $x \in \mathbb{R}^d$ that maximizes or minimizes a given linear function, among all vectors x that satisfy a given set of linear inequalities. The general form of a linear programming problem is the following:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\
\text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1..p \\
& \sum_{j=1}^{d} a_{ij} x_j = b_i \quad \text{for each } i = p+1..p+q \\
& \sum_{j=1}^{d} a_{ij} x_j \ge b_i \quad \text{for each } i = p+q+1..n
\end{aligned}$$

Here, the input consists of a *constraint matrix* $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, an *offset vector* $b \in \mathbb{R}^n$, and an *objective vector* $c \in \mathbb{R}^d$. Each coordinate of the vector x is called a *variable*. Each of the linear inequalities is called a *constraint*. The function $x \mapsto c \cdot x$ is called the *objective function*.

A linear program is said to be in ***canonical form***[1] if it has the following structure:

$$\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\
\text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1..n \\
& x_j \ge 0 \quad \text{for each } j = 1..d
\end{aligned}$$

A canonical linear program is completely determined by its constraint matrix A, its offset vector b, and its objective vector c. In addition to the n matrix constraints $Ax \ge b$, every canonical linear program includes d *sign constraints* of the form $x_j \ge 0$.

[1] Confusingly, some authors call this *standard form*. Linear programs in this form are also sometimes called *packing* LPs.

We can express this canonical form more compactly as follows. For two vectors $x = (x_1, x_2, \ldots, x_d)$ and $y = (y_1, y_2, \ldots, y_d)$, the expression $x \geq y$ means that $x_i \geq y_i$ for every index i.

$$\begin{array}{rl} \max & c \cdot x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{array}$$

Any linear programming problem can be converted into canonical form as follows:

- For each variable x_j, add two new variables x_i^+ and x_i^-, an equality constraint $x_j = x_j^+ - x_j^-$, and inequalities $x_j^+ \geq 0$ and $x_j^- \geq 0$.
- Replace any equality constraint $\sum_j a_{ij} x_j = b_i$ with two inequality constraints $\sum_j a_{ij} x_j \geq b_i$ and $\sum_j a_{ij} x_j \leq b_i$.
- Finally, replace any upper bound constraint $\sum_j a_{ij} x_j \geq b_i$ with the equivalent lower bound $\sum_j -a_{ij} x_j \leq -b_i$.

This conversion potentially doubles the number of variables and the number of constraints; fortunately, it is rarely necessary in practice.

Another useful format for linear programs is ***slack form***[2], in which every inequality is of the form $x_j \geq 0$:

$$\begin{array}{rl} \max & c \cdot x \\ \text{s.t.} & Ax = b \\ & x \geq 0 \end{array}$$

It's fairly easy to convert any linear programming problem into slack form; I'll leave the details as an easy exercise (hint, hint). Slack form is especially useful in executing the simplex algorithm, which we'll see in the next chapter.

H.2 The Geometry of Linear Programming

A point $x \in \mathbb{R}^d$ is ***feasible*** with respect to some linear programming problem if it satisfies all the linear constraints. The set of all feasible points is called the *feasible region* for that linear program. The feasible region has a particularly nice geometric structure that lends some useful intuition to the linear programming algorithms we'll see later.

Any linear equation in d variables defines a *hyperplane* in $\mathbb{R}^d$; think of a line when $d = 2$, or a plane when $d = 3$. This hyperplane divides $\mathbb{R}^d$ into two *halfspaces*; each halfspace is the set of points that satisfy some linear inequality. Thus, the set of feasible points is the intersection of several hyperplanes (one for each equality constraint) and halfspaces (one for each inequality constraint). The intersection of a finite number of hyperplanes and halfspaces is called a ***polyhedron***. It's not hard to verify that any

[2] Confusingly, some authors call this *standard form*.

halfspace, and therefore any polyhedron, is *convex*—if a polyhedron contains two points x and y, then it contains the entire line segment $\overline{xy}$.

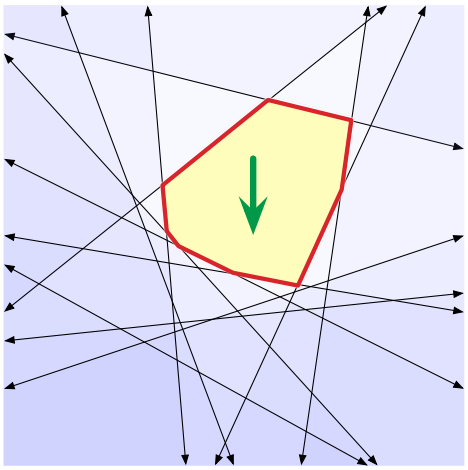


Figure H.1. A two-dimensional polyhedron defined by 11 linear inequalities.

By rotating $\mathbb{R}^d$ (or choosing an appropriate coordinate frame) so that the objective function points downward, we can express *any* linear programming problem in the following geometric form:

> Find the lowest point in a given polyhedron.

With this geometry in hand, we can easily picture two pathological cases where a given linear programming problem has no solution. The first possibility is that there are no feasible points; in this case the problem is called ***infeasible***.

$$\begin{array}{ll} \text{maximize} & x - y \\ \text{subject to} & 2x + y \le 1 \\ & x + y \ge 2 \\ & x, y \ge 0 \end{array}$$

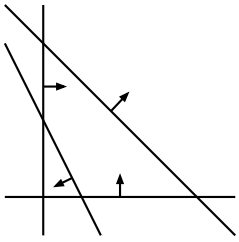


Figure H.2. An infeasible linear programming problem; arrows indicate the constraints.

The second possibility is that there are feasible points at which the objective function is arbitrarily large; in this case, we call the problem ***unbounded***. The same polyhedron could be unbounded for some objective functions but not others, or it could be unbounded for every objective function.

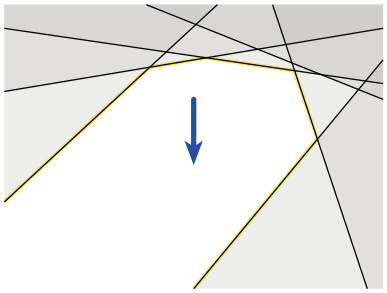


Figure H.3. A two-dimensional polyhedron (white) that is unbounded downward but bounded upward.

H.3 Examples

One Shortest Path

We can compute the length of the shortest path from s to t in a weighted directed graph by solving the following simple linear program.

$$\begin{aligned} &\text{maximize} && dist(t) \\ &\text{subject to} && dist(s) = 0 \\ &&& dist(v) - dist(u) \le \ell(u \to v) \quad \text{for every edge } u \to v \end{aligned}$$

Here the input consists of a graph $G = (V, E)$ and a vector $\ell \in \mathbb{R}^E$ (or equivalently, a function $\ell : E \to \mathbb{R}$) representing the lengths of edges in G, and our goal is to compute a vector $dist \in \mathbb{R}^V$ (or equivalently, a function $dist : V \to \mathbb{R}$) such that $dist(t)$ is the shortest path distance from s to t. The constraints in the linear program encode Ford's generic requirement that every edge in the graph must be relaxed. These relaxation constraints imply that in any feasible solution, $dist(v)$ is *at most* the shortest path distance from s to v, for *every* vertex v. Thus, somewhat counterintuitively, we are correctly *maximizing* $dist(t)$ to compute the *shortest*-path distance from s to t!

- This linear program is feasible if and only if the graph has no negative cycles. On the one hand, if the graph has no negative cycles, then the true shortest-path distances give a feasible solution. On the other hand, if the graph has a negative cycle, then for any distance values $dist(v)$, at least one edge in that cycle will be tense.
- This linear program is bounded if and only if the graph contains a path from s to t. On the one hand, if there is a path from s to t, the length of that path is an upper bound on the optimal objective value. On the other hand, if there is no path from s to t, we can obtain a feasible solution by setting $dist(v) = 0$ for all vertices v that are reachable from s, and $dist(v) = \infty$ for all vertices v that are not reachable from s.
- The optimal solution to this linear program is not necessarily unique. For example, if the graph contains a vertex v that cannot reach t, we can transform any optimal solution into another optimal solution by arbitrarily decreasing $dist(v)$. More

generally, this LP has a unique optimal solution if and only if (1) every vertex lies on a path from s to t and (2) every path from s to t has the same length.

One Shortest Path, Take Two

Alternatively, and perhaps more intuitively, we can formulate the shortest-path problem as a minimization problem. However, instead of using variables that represent distances, this linear program uses variables $x(u \to v)$ that *intuitively* indicate which edges $u \to v$ lie on the shortest path from s to t.

$$\begin{aligned}
\text{minimize} \quad & \sum_{u \to v} \ell(u \to v) \cdot x(u \to v) \\
\text{subject to} \quad & \sum_{u} x(u \to t) - \sum_{w} x(t \to w) = 1 \\
& \sum_{u} x(u \to v) - \sum_{w} x(v \to w) = 0 \quad \text{for every vertex } v \neq s, t \\
& x(u \to v) \geq 0 \quad \text{for every edge } u \to v
\end{aligned}$$

Any path from s to t—in particular, the *shortest* from s to t path—yields a feasible point x for this linear program, where $x(u \to v) = 1$ for each edge $u \to v$ in the path and $x(u \to v) = 0$ for every other edge. More generally, any *walk* from s to t implies a feasible solution, where $x(u \to v)$ is the number of times $u \to v$ appears in the walk.

Sharp-eyed readers might recognize this linear program as a *minimum-cost flow* problem, where each edge $u \to v$ has cost $\ell(u \to v)$ and infinite capacity, the source vertex s has balance -1 (one unit of supply), the target vertex t has balance 1 (one unit of demand), and all other vertices have balance 0. (The missing balance constraint for s is implied by the other $V - 1$ balance constraints.) More concisely, the feasible points for this linear program are flows from s to t with value 1, and the objective function $\ell \cdot x$ is the *cost* of this flow.

Because all balances and (nonexistent) capacities in this network are integers, we know that there is an integral minimum-cost flow from s to t. (The existence of an integral optimal solution a feature of this particular linear program! It is possible for a linear program with only integral input data A, b, and c to have only non-integral optimal solutions.) A simple flow decomposition argument now implies that the minimum-cost flow from s to t is actually a simple path from s to t.

- This linear program is feasible if and only if the underlying graph contains a path from s to t. If G contains a path from s to t, then sending one unit of flow along that path gives us a feasible solution. On the other hand, any (s, t)-flow is a weighted sum of (s, t)-paths and possibly cycles.
- This linear program is bounded if and only if the underlying graph has no negative cycles. If the G contains a negative cycle, we can arbitrarily decrease the cost of any solution by pushing more flow around that cycle. On the other hand, suppose G has

no negative cycles, and let x be any feasible solution. If x contains any cycles, we can decrease the cost of x by removing them. Then the maxflow-mincut theorem implies that $x(u \to v) \le 1$ for every edge $u \to v$. We conclude that the cost of any feasible solution is at least the sum of all negative edge lengths in G.

- The optimal solution to this linear program is not necessarily unique, because shortest paths are not necessarily unique. Moreover, if there is more than one optimal solution, then there are infinitely many optimal solutions. Specifically, if x and x' are optimal solutions, then for any constant $0 < \alpha < 1$, the convex combination $\alpha x + (1-\alpha)x'$ is also an optimal solution.

Single-Source Shortest Paths

In the optimal solution for the first shortest-path linear program, the objective function $dist(t)$ is the actual shortest-path distance from s to t, but for any vertex v that is not on the shortest path from s to t, the corresponding value $dist(v)$ may underestimate the true distance from s to v. We can obtain the true distances from s to *every* other vertex by modifying the objective function as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_v dist(v) \\ \text{subject to} \quad & dist(s) = 0 \\ & dist(v) - dist(u) \le \ell(u \to v) \quad \text{for every edge } u \to v \end{aligned}$$

Again, we are counterintuitively faced with a *maximization* problem to find *shortest* paths.

We can similarly modify the second shortest-path linear program to compute a spanning tree of shortest paths.

$$\begin{aligned} \text{minimize} \quad & \sum_{u \to v} \ell(u \to v) \cdot x(u \to v) \\ \text{subject to} \quad & \sum_u x(u \to v) - \sum_w x(v \to w) = 1 \quad \text{for every vertex } v \ne s \\ & x(u \to v) \ge 0 \quad \text{for every edge } u \to v \end{aligned}$$

Let T be any shortest-path tree rooted at s, and for any vertex v, let D_v denote set of all descendants of v in T. (In particular, $D_s = V$, and $D_v = |v|$ if v is a leaf of T.) Then we can obtain an optimal solution to this second LP by setting

$$x(u \to v) = \begin{cases} |D_v| & \text{if } u \to v \in T \\ 0 & \text{otherwise} \end{cases}$$

Again, sharp-eyed readers may recognize this linear program as an uncapacitated minimum-cost flow problem.

Maximum Flows and Minimum Cuts

Recall that the input to the maximum (s,t)-flow problem consists of a directed graph $G=(V,E)$, two special vertices s and t, and a function assigning a non-negative capacity $c(e)$ to each edge e. Our task is to compute a flow function $f \in E^{\mathbb{R}}$ subject to the usual capacity and balance constraints. The maximum-flow problem can be encoded as a linear program as follows:

$$\begin{aligned}
\text{maximize} \quad & \sum_w f(s{\to}w) - \sum_u f(u{\to}s) \\
\text{subject to} \quad & \sum_w f(v{\to}w) - \sum_u f(u{\to}v) = 0 && \text{for every vertex } v \neq s,t \\
& f(u{\to}v) \le c(u{\to}v) && \text{for every edge } u{\to}v \\
& f(u{\to}v) \ge 0 && \text{for every edge } u{\to}v
\end{aligned}$$

Similarly, the minimum cut problem can be formulated using "indicator" variables similarly to the shortest-path problem. We define a variable $S(v)$ for each vertex v, indicating whether $v \in S$ or $v \in T$, and a variable $x(u{\to}v)$ for each edge $u{\to}v$, indicating whether $u \in S$ and $v \in T$, where (S,T) is some (s,t)-cut.

$$\begin{aligned}
\text{minimize} \quad & \sum_{u\to v} c(u{\to}v) \cdot x(u{\to}v) \\
\text{subject to} \quad & x(u{\to}v) + S(v) - S(u) \ge 0 && \text{for every edge } u{\to}v \\
& x(u{\to}v) \ge 0 && \text{for every edge } u{\to}v \\
& S(s) = 1 \\
& S(t) = 0
\end{aligned}$$

Like the minimization linear programs for shortest paths, there can be optimal solutions that assign fractional values to the variables. Nevertheless, the minimum value for the objective function is the cost of the minimum cut, and there is an optimal solution for which every variable is either 0 or 1, representing an actual minimum cut.

H.4 Linear Programming Duality

Each of the preceding pairs of linear programs is an example of a general relationship called *duality*. For any linear programming problem, there is a corresponding *dual* linear program that can be obtained by a mechanical translation, essentially by swapping the constraints and the variables. The translation is simplest when the original LP is in canonical form:

Primal (Π)		**Dual (Π)**
$\max \quad c \cdot x$	$\Longleftrightarrow$	$\min \quad y \cdot b$
$\text{s.t.} \quad Ax \le b$		$\text{s.t.} \quad yA \ge c$
$x \ge 0$		$y \ge 0$

We can also write the dual linear program in exactly the same canonical form as the primal, by swapping the coefficient vector c and the objective vector b, negating both vectors, and replacing the constraint matrix A with its negative transpose.[3]

Primal (Π)

$$\begin{array}{rl} \max & c \cdot x \\ \text{s.t.} & Ax \le b \\ & x \ge 0 \end{array}$$

$\Longleftrightarrow$

Dual ($\amalg$)

$$\begin{array}{rl} \max & -b^\top \cdot y^\top \\ \text{s.t.} & -A^\top y^\top \le -c^\top \\ & y^\top \ge 0 \end{array}$$

Written in this form, it should be immediately clear that duality is an *involution*: The dual of the dual linear program $\amalg$ is identical to the primal linear program Π. The choice of which linear program to call the "primal" and which to call the "dual" is totally arbitrary.[4]

Dualizing Arbitrary LPs

Given a linear program Π that is not in canonical form, we can mechanically derive its dual linear program by first converting Π into canonical form and then applying the pattern shown above. If we push the symbols around long enough, we eventually find the following general pattern for constructing the dual of any ***maximization*** linear program. (The rules for dualizing minimization programs are the reverse of these.)

- For each constraint in the primal LP *except* sign constraints of the form $x_i \gtrless 0$, there is a corresponding variable in the dual LP.
  - If the primal constraint is an upper bound, the dual variable must be non-negative.
  - If the primal constraint is a lower bound, the dual variable must be non-positive.
  - If the primal constraint is an equation, the sign of dual variable is not constrained.
  - The right side of the primal constraint becomes the dual variable's coefficient in the objective function.
  - The dual linear program *minimizes* its objective value.
- Conversely, for each variable in the primal LP, there is a corresponding constraint in the dual LP.
  - If the primal variable is non-negative, the dual constraint is a lower bound.
  - If the primal variable is non-positive, the dual constraint is an upper bound.
  - If the sign of the primal variable is not constrained, the dual constraint is an equation.

[3]For the notational purists: In these formulations, x and b are column vectors, and y and c are row vectors. This is a somewhat nonstandard choice. Yes, that means the dot in $c \cdot x$ is redundant. Sue me.

[4]For historical reasons, maximization linear programs tend to be called "primal" and minimization linear programs tend to be called "dual". (Or is it the other way around?) This habit is nothing but a pointless and distracting religious tradition.

- Finally, if the primal variable is equal to zero, the dual constraint doesn't actually exist. (If something can't vary, it's not really a variable!)
- The primal variable's coefficient in the objective function becomes the right side of the dual constraint.

Primal	Dual
$\max c \cdot x$	$\min y \cdot b$
$\sum_j a_{ij}x_j \le b_i$	$y_i \ge 0$
$\sum_j a_{ij}x_j \ge b_i$	$y_i \le 0$
$\sum_j a_{ij}x_j = b_i$	–
$x_j \ge 0$	$\sum_i y_i a_{ij} \ge c_j$
$x_j \le 0$	$\sum_i y_i a_{ij} \le c_j$
–	$\sum_i y_i a_{ij} = c_j$
$x_j = 0$	–

Primal	Dual
$\min c \cdot x$	$\max y \cdot b$
$\sum_j a_{ij}x_j \le b_i$	$y_i \le 0$
$\sum_j a_{ij}x_j \ge b_i$	$y_i \ge 0$
$\sum_j a_{ij}x_j = b_i$	–
$x_j \le 0$	$\sum_i y_i a_{ij} \ge c_j$
$x_j \ge 0$	$\sum_i y_i a_{ij} \le c_j$
–	$\sum_i y_i a_{ij} = c_j$
$x_j = 0$	–

Figure H.4. Constructing the dual of an arbitrary linear program.

At first glance, these rules appear to be asymmetric: The inequality directions for primal constraints and dual variables are opposites, but the inequality directions for primal variables and dual constraints are identical. In fact, this asymmetry reflects the switch from a primal *maximization* LP to a dual *minimization* LP. The dual of the dual of any linear program is always (equivalent to) the original linear program.

Shortest Path Example

As an example, let's derive the dual of our very first linear program for shortest paths:

$$\begin{aligned} &\text{maximize} && dist(t) \\ &\text{subject to} && dist(s) = 0 \\ &&& dist(v) - dist(u) \le \ell(u \to v) \quad \text{for every edge } u \to v \end{aligned}$$

This linear program has a variable $dist(v)$ for every vertex v and a constraint for every edge. Thus, the underlying constraint matrix A has a row (constraint) for every edge and a column (variable) for every vertex. Specifically, we have

$$A(u \to v, w) = \begin{cases} -1 & \text{if } u = w, \\ +1 & \text{if } v = w, \\ 0 & \text{otherwise.} \end{cases}$$

(A is the *signed incidence matrix* of the underlying directed graph G.) The coefficients of the objective vector are $+1$ for t and 0 for every other vertex, and the coefficients of the offset vector are the edge lengths.

So, the dual linear program has a constraint for every vertex and a variable for every edge. Let's call the edge variables $x(u{\to}v)$, because why not? Because $dist(s) = 0$, there is no dual constraint corresponding to $dist(s)$ after all. Because the other primal variables $dist(v)$ are not sign-constrained, the dual constraints are all equations. The primal constraints are upper bounds, so the dual variables $x(u{\to}v)$ must be non-negative. Finally, the coefficients of the dual objective vector are the edge lengths, and the coefficients of the dual offset vector are $+1$ for t and 0 for every other vertex.

$$\begin{aligned}
\text{minimize} \quad & \sum_{u\to v} \ell(u{\to}v) \cdot x(u{\to}v) \\
\text{subject to} \quad & \sum_u x(u{\to}t) - \sum_w x(t{\to}w) = 1 \\
& \sum_u x(u{\to}v) - \sum_w x(v{\to}w) = 0 \quad \text{for every vertex } v \neq s, t \\
& x(u{\to}v) \geq 0 \quad \text{for every edge } u{\to}v
\end{aligned}$$

Hey, this looks familiar!

H.5 The Fundamental Theorem

The most important structural observation about linear programming is the following broad generalization of the maxflow-mincut theorem.

The Fundamental Theorem of Linear Programming. *A canonical linear program* Π *has an optimal solution* x^* *if and only if the dual linear program* $\amalg$ *has an optimal solution* y^* *such that* $c \cdot x^* = y^* A x^* = y^* \cdot b$.

The Fundamental Theorem (in a slightly different form) was first conjectured by John von Neumann in May 1947, during a fateful meeting with George Dantzig. Dantzig met with von Neumann to explain his general theory of linear programming and his simplex algorithm (which we'll see in the next chapter). According to Dantzig, after about half an hour into his presentation, von Neumann told him "Get to the point!" When Dantzig then explained his algorithm more concisely, von Neumann replied dismissively "Oh, *that!*" and proceeded to lecture Dantzig for over an hour on convexity, fixed points, and his theory of two-player games, while Dantzig sat (as he later put it) dumbfounded. During his lecture, von Neumann defined the negative-transpose dual of an arbitrary linear program—based on a similar duality in two-player zero-sum games that trivially results from swapping the players—and conjectured correctly that the Fundamental Theorem could be derived from his famous min-max theorem, which he proved in 1928:

Min-Max Theorem. *For any matrix* A*, we have* $\min_x \max_y xAy = \max_y \min_x xAy$.

Von Neumann's min-max theorem is itself a consequence of several equivalent "Theorems of the Alternative" proved by Paul Gordan, Hermann Minkowski, Gyula Farkas, Erich Stiemke, Theodore Motzkin, and others in the late 1800s and early 1900s. The earliest example of such a theorem (that I know of) was proved by Gordan in 1873.

Gordan's Theorem of the Alternative. *For any matrix A, either $Ax = 0$ and $x \geq 0$ for some non-zero vector x, or $yA > 0$ for some vector y, but not both.*

Soon after their 1947 meeting, von Neumann and Dantzig independently wrote up proofs of the Fundamental Theorem, which they each distributed privately for many years. Unfortunately, von Neumann's proof was flawed, although his errors were finally corrected when he finally published the proof in 1963, and Dantzig never published his proof. The first *published* proof was written by Albert Tucker, David Gale, and Harold Kuhn in 1951.

I will defer a proof of the full Fundamental Theorem until after we develop some more intuition, but the following weak form is trivial to prove.

Weak Duality Theorem. *If x is a feasible solution for a canonical linear program Π, and y is a feasible solution for its dual $\amalg$, then $c \cdot x \leq yAx \leq y \cdot b$.*

Proof: Because x is feasible for Π, we have $Ax \leq b$. Since y is non-negative, we can multiply both sides of the inequality to obtain $yAx \leq y \cdot b$. Conversely, y is feasible for $\amalg$ and x is non-negative, so $yAx \geq c \cdot x$. □

The Weak Duality Theorem has two important immediate consequences:

- Let x and y be arbitrary *feasible* solutions to a canonical linear program and its dual, respectively. If $c \cdot x = y \cdot b$, then x and y are *optimal* primal and dual solutions, respectively, and $c \cdot x = yAx = y \cdot b$.
- If a linear program is unbounded, then its dual is infeasible; equivalently, if a linear program is feasible, then its dual is bounded. (It is possible for a linear program and its dual to both be infeasible, but only if both linear programs are degenerate.)

The full version of the Fundamental Theorem implies that that all these implications are actually equivalences.

H.6 Another Duality Example

Before I prove the stronger duality theorem, let me try to provide some intuition about how this duality thing actually works.[5] Consider the following canonical linear

[5]This example is taken from Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions* [Springer, 2001], but the idea appears earlier in Jens Clausen's 1997 paper "Teaching Duality in Linear Programming: The Multiplier Approach".

programming problem:

$$\begin{array}{lrcl} \text{maximize} & 4x_1 + & x_2 + 3x_3 & \\ \text{subject to} & x_1 + 4x_2 & & \le 2 \\ & 3x_1 - & x_2 + x_3 & \le 4 \\ & & x_1, x_2, x_3 & \ge 0 \end{array}$$

Let σ^* denote the optimum objective value for this LP. The feasible solution $x = (1, 0, 0)$ gives us a lower bound $\sigma^* \ge 4$. A different feasible solution $x = (0, 0, 3)$ gives us a better lower bound $\sigma^* \ge 9$. We could play this game all day, finding different feasible solutions and getting ever larger lower bounds. How do we know when we're done? Is there a way to prove an *upper* bound on σ^*?

In fact, there is. Let's multiply each of the constraints in our LP by a new non-negative scalar value y_i:

$$\begin{array}{lrcl} \text{maximize} & 4x_1 + & x_2 + 3x_3 & \\ \text{subject to} & y_1(\ x_1 + 4x_2 & &) \le 2y_1 \\ & y_2(3x_1 - & x_2 + x_3) & \le 4y_2 \\ & & x_1, x_2, x_3 & \ge 0 \end{array}$$

Because each y_i is non-negative, we do not reverse any of the inequalities. Any feasible solution (x_1, x_2, x_3) must satisfy both of these inequalities, so it must also satisfy their sum:

$$(y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \le 2y_1 + 4y_2.$$

Now suppose that the coefficient of each variable x_i in the expression on the left side of this inequality is larger than the corresponding coefficient of the objective function:

$$y_1 + 3y_2 \ge 4, \qquad 4y_1 - y_2 \ge 1, \qquad y_2 \ge 3.$$

This assumption implies an upper bound on the objective value of *any* feasible solution:

$$4x_1 + x_2 + 3x_3 \ \le\ (y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \ \le\ 2y_1 + 4y_2. \qquad (*)$$

In particular, by plugging in the optimal solution (x_1^*, x_2^*, x_3^*) for the original LP, we obtain the following upper bound on σ^*:

$$\sigma^* \ =\ 4x_1^* + x_2^* + 3x_3^* \ \le\ 2y_1 + 4y_2.$$

Now it is natural to ask how tight this upper bound can be. That is, how small can we make the expression $2y_1 + 4y_2$ without violating any of the inequalities we used to

prove the upper bound? This is just another linear programming problem.

$$\begin{aligned} \text{minimize} \quad & 2y_1 + 4y_2 \\ \text{subject to} \quad & y_1 + 3y_2 \geq 4 \\ & 4y_1 - y_2 \geq 1 \\ & y_2 \geq 3 \\ & y_1, y_2 \geq 0 \end{aligned}$$

In fact, the resulting linear program is precisely the dual of our original linear program! Moreover, inequality $(*)$ is just an instantiation of the Weak Duality Theorem.

H.7 Strong Duality

The Fundamental Theorem can be rephrased in the following form:

Strong Duality Theorem. *If x^* is an optimal solution for a canonical linear program Π, then there is an optimal solution y^* for its dual $\amalg$, such that $c \cdot x^* = y^* A x^* = y^* \cdot b$.*

Proof (sketch): I'll prove the theorem only for ***non-degenerate*** linear programs, in which (a) the objective vector is not normal to any constraint hyperplane, and (b) at most d constraint hyperplanes pass through any point. The first assumption implies that the optimal solution to the LP—if it exists—is unique and is therefore a vertex of the feasible region. These non-degeneracy assumptions are relatively easy to enforce in practice and can be removed from the proof at the expense of some technical detail. I will also assume that $n \geq d$; the argument for under-constrained LPs is similar (if not simpler).

To develop some intuition, let's first consider the *very* special case where $x^* = (0, 0, \ldots, 0)$. Let e_i denote the ith standard basis vector, whose ith coordinate is 1 and all other coordinates are 0. Because $x_i^* = 0$ for all i, our non-degeneracy assumption implies the *strict* inequality $a_i \cdot x^* < b_i$ for all i. Thus, any sufficiently small ball around the origin does not intersect any other constraint hyperplane $a_i \cdot x = b_i$. Thus, for all i, and for any sufficiently small $\delta > 0$, the vector δe_i is feasible. Because x^* is the unique optimum, we must have $\delta c_i = c \cdot (\delta e_i) < c \cdot x^* = 0$. We conclude that $c_i < 0$ for all i.

Now let $y = (0, 0, \ldots, 0)$ as well. We immediately observe that $yA \geq c$ and $y \geq 0$; in other words, y is a *feasible* solution for the dual linear program $\amalg$. But $y \cdot b = 0 = c \cdot x^*$, so the weak duality theorem implies that y is an *optimal* solution to $\amalg$, and the proof is complete for this very special case!

Now let's consider the more general case. Let x^* be the optimal solution for the linear program Π; our non-degeneracy assumption implies that this solution is unique, and that it satisfies exactly d of the n linear constraints with equality. Without loss of

generality (by permuting the constraints and possibly changing coordinates), we can assume that these are the first d constraints. Thus, we have

$$\begin{aligned} a_i \cdot x^* &= b_i \qquad \text{for all } i \le d, \\ a_i \cdot x^* &< b_i \qquad \text{for all } i \ge d+1, \end{aligned}$$

where a_i denotes the ith row of A. Let $A_\bullet$ denote the $d \times d$ matrix containing the first d rows of A. Our non-degeneracy assumption implies that $A_\bullet$ has full rank, and thus has a well-defined inverse $V = A_\bullet^{-1}$.

Now define a vector $y \in \mathbb{R}^n$ by setting

$$\begin{aligned} y_j &:= c \cdot v^j \qquad \text{for all } j \le d, \\ y_j &:= 0 \qquad\quad\ \ \text{for all } j \ge d+1, \end{aligned}$$

where v^j denotes the jth column of V. The definition of matrix inverse implies that $a_i \cdot v^i = 1$ for all i, and $a_i \cdot v^j = 0$ for all $i \neq j$.

To simplify notation, let $y_\bullet = (y_1, y_2, \ldots, y_d)$ and let $b_\bullet = (b_1, b_2, \ldots, b_d) = A_\bullet x^*$. Because $y_i = 0$ for all $i \ge d+1$, we immediately have

$$y \cdot b = y_\bullet \cdot b_\bullet = cV b_\bullet = cA_\bullet^{-1} b_\bullet = c \cdot x^*$$

and

$$yA = y_\bullet A_\bullet = cVA_\bullet = cA_\bullet^{-1} A_\bullet = c.$$

The point x^* lies on exactly d constraint hyperplanes; moreover, any sufficiently small ball around x^* intersects *only* those d constraint hyperplanes. Consider the point $\tilde{x} = x^* - \varepsilon v^j$, for some index $1 \le j \le d$ and some sufficiently small $\varepsilon > 0$. We have $a_i \cdot \tilde{x} = a_i \cdot x^* - \varepsilon(a_i \cdot v^j) = b_i$ for all $i \neq j$, and $a_j \cdot \tilde{x} = a_j \cdot x^* - \varepsilon(a_j \cdot v^j) = b_j - \varepsilon < b_j$. Thus, $\tilde{x}$ is a feasible point for Π. Because x^* is the *unique* optimum for Π, we must have $c \cdot \tilde{x} = c \cdot x^* - \varepsilon(c \cdot v^j) < c \cdot x^*$. We conclude that $y_j = c \cdot v^j > 0$ for all j.

We have shown that $yA \ge c$ and $y \ge 0$, so y is a *feasible* solution for the dual linear program $\amalg$. We have also shown that $y \cdot b = c \cdot x^*$, so the Weak Duality Theorem implies that y is an *optimal* solution for $\amalg$, and the proof is complete! □

We can also give a useful geometric interpretation to the dual vector $y_\bullet \in \mathbb{R}^d$. Each linear equation $a_i \cdot x = b_i$ defines a hyperplane in $\mathbb{R}^d$ with normal vector a_i. The normal vectors $a_1, \ldots, a_d$ are linearly independent (by non-degeneracy) and therefore describe a coordinate frame for the vector space $\mathbb{R}^d$. The definition of $y_\bullet$ implies that $c = y_\bullet A_\bullet = \sum_{i=1}^d y_i a_i$. In other words, ***the non-zero dual variables $y_1, \ldots, y_d$ are the coefficients of the objective vector c in the coordinate frame $a_1, \ldots, a_d$.***

In more physical terms, imagine that the objective vector c points downward, and that x^* is the lowest point in the feasible region, which is found by tossing a marble into the feasible region and letting it fall as far as possible without crossing any constraint

hyperplane. Even after the marble reaches the lowest feasible point, gravity is still pulling it downward, but the marble isn't moving, so the constraint hyperplanes must be pushing it upward. Each constraint hyperplane can only push in the direction of its normal vector. ***Each dual variable y_i is the amount of force applied by the ith constraint hyperplane.***

H.8 Complementary Slackness

The Strong Duality Theorem implies a relationship between the optimal solutions of a linear program and its dual that is stronger than just the equality of their objective values. In physical terms, a constraint hyperplane applies a non-zero force to the marble only if it touches the marble; that is, if a dual variable is positive, then the corresponding constraint must be satisfied with equality.

Recall that the theorem states that if x^* is the optimal solution to a canonical linear program Π, then the dual program $\amalg$ has an optimal solution y^* such that $c \cdot x^* = y^* A x^* = y^* \cdot b$. These equations have two immediate consequences for any optimal solution vectors x^* and y^*.

- For any row index i, either $y_i^* = 0$ or $a_i \cdot x^* = b$ (or both).
- For any column index j, either $x_j^* = 0$ or $y^* \cdot a^j = c$ (or both).

Here, a_i and a^j respectively denote the ith row and jth column of A. These consequences of strong duality are called the ***complementary slackness*** conditions.

Call a constraint *tight* if it is satisfied with equality and *loose* otherwise. The complementary slackness conditions can be phrased more colloquially as follows:

- If a primal variable is positive, the corresponding dual constraint is tight.
- If a dual constraint is loose, the corresponding primal variable is zero.
- If a dual variable is positive, the corresponding primal constraint is tight.
- If a primal constraint is loose, the corresponding dual variable is zero.

Complementary Slackness Theorem. *Let x be an arbitrary feasible solution to a canonical linear program Π, and let y be an arbitrary feasible solution to the dual linear program $\amalg$. The following conditions are equivalent:*

- *x and y are optimal solutions to their respective linear programs.*
- $c \cdot x = y \cdot b$
- $c \cdot x = yAx$
- $yAx = y \cdot b$
- *x and y satisfy all complementary slackness conditions.*

The complementary slackness conditions are not necessarily exclusive; it is possible for a constraint to be tight *and* for the corresponding dual variable to be zero. Nevertheless, every bounded and feasible linear program has optimal primal and dual solutions x^* and y^* that satisfy *strict* complementary slackness conditions:

- For any row index i, either $y_i^* > 0$ or $a_i \cdot x^* < b$ (but not both).
- For any column index j, either $x_j^* > 0$ or $y^* \cdot a^j > c$ (but not both).

However, if the linear program is non-degenerate, the primal and dual optimal solutions are both unique and therefore satisfy these stricter conditions.

Exercises

Need more!

★★★

1. A matrix $A = (a_{ij})$ is *skew-symmetric* if and only if $a_{ji} = -a_{ij}$ for all indices $i \neq j$; in particular, every skew-symmetric matrix is square. A canonical linear program $\max\{c \cdot x \mid Ax \leq b; x \geq 0\}$ is *self-dual* if the matrix A is skew-symmetric and the objective vector c is *equal* to the constraint vector b.
   (a) Prove that any self-dual linear program Π is syntactically equivalent to its dual program $\amalg$.
   (b) Show that any linear program Π with d variables and n constraints can be transformed into a self-dual linear program with $n + d$ variables and $n + d$ constraints. The optimal solution to the self-dual program should include both the optimal solution for Π (in d of the variables) and the optimal solution for the dual program $\amalg$ (in the other n variables).

2. (a) Give a linear-programming formulation of the ***bipartite maximum matching*** problem. The input is a bipartite graph $G = (U \cup V; E)$, where $E \subseteq U \times V$; the output is the largest matching in G. Your linear program should have one variable for each edge.
   (b) Now dualize the linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?

3. (a) Give a linear-programming formulation of the ***minimum-cost circulation*** problem. You are given a flow network whose edges have both capacities and costs, and your goal is to find a feasible circulation (flow with value 0) whose total cost is as small as possible.
   (b) Derive the dual of your linear program from part (a).

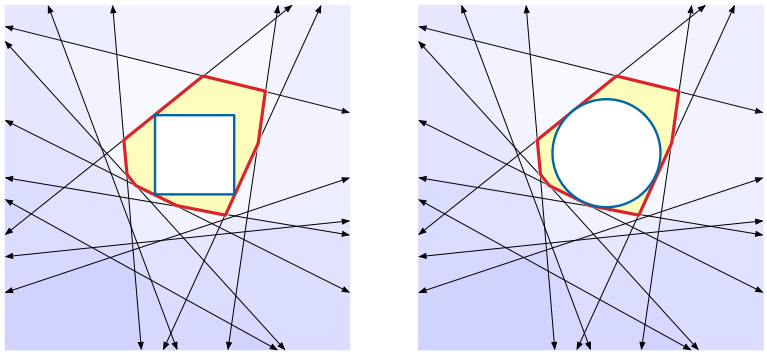
4. Suppose we are given a sequence of n linear inequalities of the form $a_i x + b_i y \leq c_i$. Collectively, these n inequalities describe a convex polygon P in the plane.
   (a) Describe a linear program whose solution describes the largest axis-aligned square that lies entirely inside P.
   (b) Describe a linear program whose solution describes the maximum-perimeter axis-aligned rectangle that lies entirely inside P.

(c) Describe a linear program whose solution describes the largest circle that lies entirely inside P.

(d) Describe a polynomial-time algorithm to compute two interior-disjoint axis-aligned squares with maximum total perimeter that lie entirely inside P. *[Hint: There are exactly two interesting cases to consider; for each case, formulate a corresponding linear program.]*

(e) Describe a polynomial-time algorithm to compute two interior-disjoint axis-aligned rectangles with maximum total perimeter that lie entirely inside P. *[Hint: Again, there are only two interesting cases to consider.]*

In all subproblems, "axis-aligned" means that the edges of the square(s) or rectangles(s) are horizontal and vertical.



5. Given points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in the plane, the ***linear regression problem*** asks for real numbers a and b such that the line $y = ax + b$ fits the points as closely as possible, according to some criterion. The most common fit criterion is minimizing the ***L_2 error***, defined as follows:[6]

$$\varepsilon_2(a, b) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

But there are many other ways of measuring a line's fit to a set of points, some of which can be optimized via linear programming.

(a) The ***L_1 error*** (or *total absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_1(a, b) = \sum_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_1 error.

[6] This measure is also known as *sum of squared residuals*, and the algorithm to compute the best fit is normally called *(ordinary/linear) least squares*.

(b) The ***L_∞ error*** (or *maximum absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_\infty(a, b) = \max_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_∞ error.

6. Let $G = (V, E)$ be an arbitrary directed graph with weighted vertices; vertex weights may be positive, negative, or zero. A ***prefix*** of G is a subset $P \subseteq V$, such that there is no edge $u \to v$ where $u \notin P$ but $v \in P$. A ***suffix*** of G is the complement of a prefix. Finally, an ***interval*** of G is the intersection of a prefix of G and a suffix of G. The weight of a prefix, suffix, or interval is the sum of the weights of its vertices.

   (a) Describe a linear program whose solution describes the maximum-weight prefix of G. Your linear program should have one variable per vertex, indicating whether that vertex is or is not in the chosen prefix.

   (b) Describe an algorithm that computes the maximum-weight prefix of G, by reducing to a standard maximum-flow problem in a related graph. *[Hint: Consider the special case where G is a dag. Haven't we seen this problem before?]*

   (c) Describe an efficient algorithm that computes the maximum-weight prefix of G, when G is a rooted tree with all edges pointing away from the root. *[Hint: This is really easy if you* **don't** *think about linear programming.]*

   (d) Describe a linear program whose solution describes the maximum-weight interval of G.

   (e) Describe an efficient algorithm that computes the maximum-weight interval of G, when G is a rooted tree with all edges pointing away from the root. *[Hint: Again, this is easy if you* **don't** *think about linear programming.]*

   [Hint: Don't worry about the solutions to your linear programs being integral; they will be (essentially by part (b)). If all vertex weights are negative, the maximum-weight interval is empty; if all vertex weights are positive, the maximum-weight interval contains every vertex.]

7. Suppose you are given an arbitrary directed graph $G = (V, E)$ with arbitrary edge weights $\ell : E \to \mathbb{R}$, and two special vertices. Each edge in G is colored either red, white, or blue to indicate how you are permitted to modify its weight:

   - You may increase, but not decrease, the length of any red edge.
   - You may decrease, but not increase, the length of any blue edge.
   - You may not change the length of any black edge.

   Your task is to modify the edge weights—subject to the color constraints—so that *every* path from s to t has exactly the same length. Both the given weights and the

new weights of the edges can be positive, negative, or zero. To keep the following problems simple, assume every edge in G lies on at least one path from s to t, and that G has no isolated vertices.

(a) Describe a linear program that is feasible if and only if it is possible to make every path from s to t the same length. *[Hint: Let* $\text{dist}(v)$ *denote the length of every path from* s *to* v*.]*

(b) Construct the dual of the linear program from part (a). *[Hint: Choose a convenient objective function for your primal LP.]*

♥(c) Give a self-contained description of the combinatorial problem encoded by the dual linear program from part (b), and prove *directly* that it is equivalent to the original path-equalization problem. Do not use the words "linear", "program", or "dual". Yes, you've seen this problem before. *[Hint: The proof is the hard part.]*

(d) Describe and analyze an algorithm to determine *in* $O(EV)$ *time* whether it is possible to make every path from s to t the same length. Do not use the words "linear", "program", or "dual".

(e) Finally, suppose we want to equalize path lengths while modifying the edge weights as little as possible. Specifically, we want to compute new edge weights $\ell'(e)$ that give every path from s to t the same length, such that the sum $\sum_e |\ell'(e) - \ell(e)|$ is as small as possible. Describe and analyze an efficient algorithm to solve this optimization problem. *[Hint: Modify your solutions to parts (a)–(c).]*

♦8. An ***integer program*** is a linear program with the additional constraint that the variables must take only integer values.

(a) Prove that deciding whether an integer program has a feasible solution is NP-complete.

(b) Prove that finding the optimal feasible solution to an integer program is NP-hard.

[Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]

9. A *convex combination* of n points $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$ is weighted average of those points:

$$p = \sum_{i=1}^{n} \lambda_i x_i$$

for some coefficients $\lambda_1, \lambda_2, \ldots, \lambda_n$ such that $\lambda_i \geq 0$ for all i and $\sum_{i=1}^{n} \lambda_i = 1$. The *convex hull* of a set of points X is the set of all convex combinations of points in X. For example, the convex hull of two points is the line segment between them, the convex hull of three points is the (filled) triangle with those three vertices, and so on.

Carathéodory's theorem states that for any set of points $X \subset \mathbb{R}^d$, every point in the convex hull of X lies in the convex hull of $d + 1$ points of X. For example, when

$d = 2$, the theorem states every point inside a convex polygon P is also inside the triangle spanned by three vertices of P.

Prove Carathéodory's theorem. *[Hint: Express "p lies in the convex hull of X" as a system of linear inequalities over the variables λ_i.]*

10. *Helly's theorem* states that for any collection of convex bodies in $\mathbb{R}^d$, if every $d + 1$ of them intersect, then there is a point lying in the intersection of all of them. Prove Helly's theorem for the special case where the convex bodies are halfspaces. Equivalently, show that if a system of linear inequalities $Ax \le b$ is infeasible, then we can select a subset of $d + 1$ inequalities such that the resulting subsystem infeasible. *[Hint: Construct a linear program from the system by imposing a* 0 *objective vector, and consider the dual LP.]*

Simplicibus itaque verbis gaudet Mathematica Veritas, cum etiam per se simplex sit Veritatis oratio. [And thus Mathematical Truth prefers simple words, because the language of Truth is itself simple.]

— Tycho Brahe (quoting Seneca (quoting Euripides))
Epistolarum astronomicarum liber primus (1596)

When a jar is broken, the space that was inside
Merges into the space outside.
In the same way, my mind has merged in God;
To me, there appears no duality.

— Sankara, *Viveka-Chudamani* (c. 700), translator unknown

I

Linear Programming Algorithms

[Read Chapters G and H first.]
Status: Half-finished.

In this chapter I will describe several variants of the *simplex algorithm* for solving linear programming problems, first proposed by George Dantzig in 1947. Although most variants of the simplex algorithm perform well in practice, no deterministic simplex variant is known to run in sub-exponential time in the worst case.[1] However, if the dimension of the problem is considered a constant, there are several variants of the simplex algorithm that run in *linear* time. I'll describe a particularly simple randomized algorithm due to Raimund Seidel.

My approach to describing these algorithms relies much more heavily on geometric intuition than the usual linear-algebraic formalism. This works better for me, but your mileage may vary. For a more traditional description of the simplex algorithm, see Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions*

[1]However, there are *randomized* variants of the simplex algorithm that run in subexponential *expected* time, most notably the RandomFacet algorithm analyzed by Gil Kalai in 1992, and independently by Jiří Matoušek, Micha Sharir, and Emo Welzl in 1996. No randomized variant is known to run in polynomial time. In particular, in 2010, Oliver Friedmann, Thomas Dueholm Hansen, and Uri Zwick proved that the worst-case expected running time of RandomFacet is superpolynomial.

[Springer, 2001], which can be freely downloaded (but not legally printed) from the author's website.

I.1 Bases, Feasibility, and Local Optimality

Consider the canonical linear program $\max\{c \cdot x \mid Ax \le b, x \ge 0\}$, where A is an $n \times d$ constraint matrix, b is an n-dimensional coefficient vector, and c is a d-dimensional objective vector. We will interpret this linear program geometrically as looking for the lowest point in a convex polyhedron in $\mathbb{R}^d$, described as the intersection of $n + d$ halfspaces. As in the last lecture, we will consider only *non-degenerate* linear programs: Every subset of d constraint hyperplanes intersects in a single point; at most d constraint hyperplanes pass through any point; and objective vector is linearly independent from any $d - 1$ constraint vectors.

A ***basis*** is a subset of d constraints, which by our non-degeneracy assumption must be linearly independent. The ***location*** of a basis is the unique point x that satisfies all d constraints with equality; geometrically, x is the unique intersection point of the d hyperplanes. The ***value*** of a basis is $c \cdot x$, where x is the location of the basis. There are precisely $\binom{n+d}{d}$ bases. Geometrically, the set of constraint hyperplanes defines a decomposition of $\mathbb{R}^d$ into convex polyhedra; this cell decomposition is called the ***arrangement*** of the hyperplanes. Every subset of d hyperplanes (that is, every basis) defines a *vertex* of this arrangement (the location of the basis). I will use the words 'vertex' and 'basis' interchangeably.

A basis is ***feasible*** if its location x satisfies all the linear constraints, or geometrically, if the point x is a vertex of the polyhedron. If there are no feasible bases, the linear program is ***infeasible***.

A basis is ***locally optimal*** if its location x is the optimal solution to the linear program with the same objective function and *only* the constraints in the basis. Geometrically, a basis is locally optimal if its location x is the lowest point in the intersection of those d halfspaces. A careful reading of the proof of the Strong Duality Theorem reveals that local optimality is the dual equivalent of feasibility; a basis is locally optimal for a linear program Π if and only if the same basis is feasible for the dual linear program $\amalg$. For this reason, locally optimal bases are sometimes also called ***dual feasible***. If there are no locally optimal bases, the linear program is ***unbounded***.[2]

Two bases are ***neighbors*** if they have $d - 1$ constraints in common. Equivalently, in geometric terms, two vertices are neighbors if they lie on a *line* determined by some $d-1$ constraint hyperplanes. Every basis is a neighbor of exactly dn other bases; to change a basis into one of its neighbors, there are d choices for which constraint to remove and n

[2]For non-degenerate linear programs, the feasible region is unbounded in the objective direction if and only if no basis is locally optimal. However, there are degenerate linear programs with no locally optimal basis that are infeasible.

choices for which constraint to add. The graph of vertices and edges on the boundary of the feasible polyhedron is a subgraph of the basis graph.

The Weak Duality Theorem implies that the value of every feasible basis is less than or equal to the value of every locally optimal basis; equivalently, every feasible vertex is higher than every locally optimal vertex. The Strong Duality Theorem implies that (under our non-degeneracy assumption), if a linear program has an optimal solution, it is the *unique* vertex that is both feasible and locally optimal. Moreover, the optimal solution is both the lowest feasible vertex and the highest locally optimal vertex.

I.2 The Simplex Algorithm

Primal: Falling Marbles

From a geometric standpoint, Dantzig's simplex algorithm is very simple. The input is a set H of halfspaces; we want the lowest vertex in the intersection of these halfspaces.

```
PrimalSimplex(H):
  if ∩H = ∅
      return Infeasible
  x ← any feasible vertex
  while x is not locally optimal
      ⟨⟨pivot downward, maintaining feasibility⟩⟩
      if every feasible neighbor of x is higher than x
          return Unbounded
      else
          x ← any feasible neighbor of x that is lower than x
  return x
```

Let's ignore the first three lines for the moment. The algorithm maintains a feasible vertex x. At each so-called ***pivot*** operation, the algorithm moves to a *lower* vertex, so the algorithm never visits the same vertex more than once. Thus, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the feasible vertex x is locally optimal, and therefore the optimum vertex, or the feasible vertex x is not locally optimal but has no lower feasible neighbor, in which case the feasible region must be unbounded.

Notice that we have not specified *which* neighbor to choose at each pivot. Many different pivoting rules have been proposed, but for almost every known pivot rule, there is an input polyhedron that requires an exponential number of pivots under that rule. *No* pivoting rule is known that guarantees a polynomial number of pivots in the worst case, or even in expectation.[3]

[3] In 1957, Hirsch conjectured that for *any* linear programming instance with d variables and $n + d$ constraints, starting at any feasible basis, there is a sequence of ***at most n*** pivots that leads to the optimal basis. This long-standing conjecture was finally disproved in 2010 by Fransisco Santos, who described an counterexample with 43 variables and 86 constraints, where the worst-case number of required pivots is 44.

Dual: Rising Bubbles

We can also geometrically interpret the execution of the simplex algorithm on the dual linear program Π. Again, the input is a set H of halfspaces, and we want the lowest vertex in the intersection of these halfspaces. By the Strong Duality Theorem, this is the same as the *highest locally-optimal* vertex in the hyperplane arrangement.

```
DUALSIMPLEX(H):
  if there is no locally optimal vertex
      return UNBOUNDED
  x ← any locally optimal vertex
  while x is not feasbile
      ⟨⟨pivot upward, maintaining local optimality⟩⟩
      if every locally optimal neighbor of x is lower than x
          return INFEASIBLE
      else
          x ← any locally-optimal neighbor of x that is higher than x
  return x
```

Let's ignore the first three lines for the moment. The algorithm maintains a locally optimal vertex x. At each pivot operation, it moves to a *higher* vertex, so the algorithm never visits the same vertex more than once. Thus, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the locally optimal vertex x is feasible, and therefore the optimum vertex, or the locally optimal vertex x is not feasible but has no higher locally optimal neighbor, in which case the problem must be infeasible.

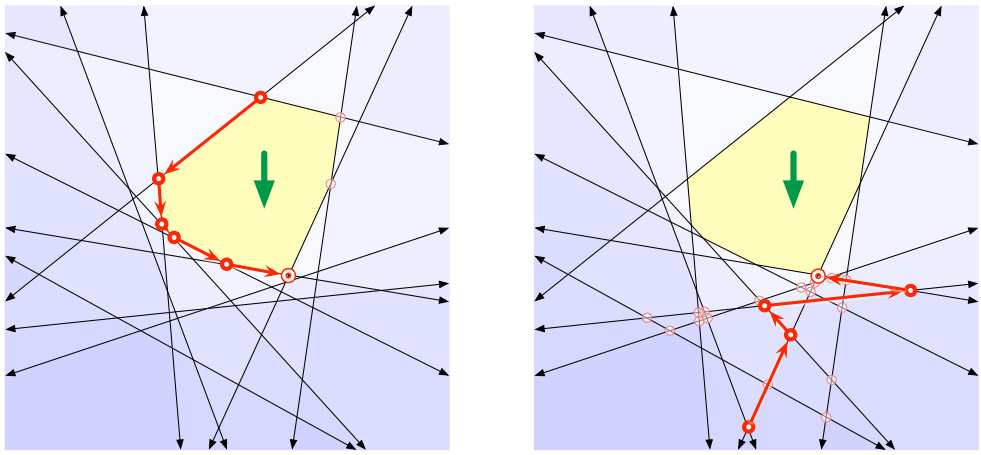


Figure I.1. The primal simplex (falling marble) and dual simplex (rising bubble) algorithms in action.

From the standpoint of linear algebra, there is absolutely no difference between running PRIMALSIMPLEX on any linear program Π and running DUALSIMPLEX on the

dual linear program II. The actual *code* is identical. The only difference between the two algorithms is how we interpret the linear algebra geometrically.

I.3 Computing the Initial Basis

To complete our description of the simplex algorithm, we need to describe how to find the initial vertex x in the third line of PrimalSimplex or DualSimplex. There are several methods to find feasible or locally optimal bases, but perhaps the most natural method uses the simplex algorithm itself. Our approach relies on two simple observations.

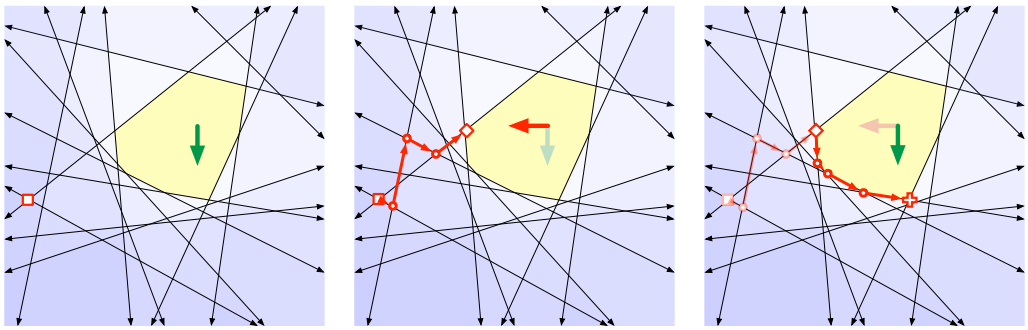
First, the feasibility of a vertex does not depend on the choice of objective vector; a vertex is either feasible for every objective function or for none. Equivalently (by duality), the local optimality of a vertex does not depend on the exact location of the d hyperplanes, but only on their normal directions and the objective function; a vertex is either locally optimal for every translation of the hyperplanes or for none. In terms of the original matrix formulation, feasibility depends on A and b but not c, and local optimality depends on A and c but not b.

Second, *every* basis is locally optimal for *some* objective vector. Specifically, it suffices to choose any vector that has a positive inner product with each of the normal vectors of the d chosen hyperplanes. Equivalently, *every* basis is feasible for some offset vector. Specifically, it suffices to translate the d chosen hyperplanes so that they pass through the origin, and then translate all other halfspaces so that they strictly contain the origin.

Thus, to find an initial feasible vertex for the primal simplex algorithm, we can choose an *arbitrary* vertex x, *rotate* the objective function so that x becomes locally optimal, and then find the optimal vertex for the *rotated* objective function by running the (dual) simplex algorithm. This vertex must be feasible, even after we restore the original objective function!

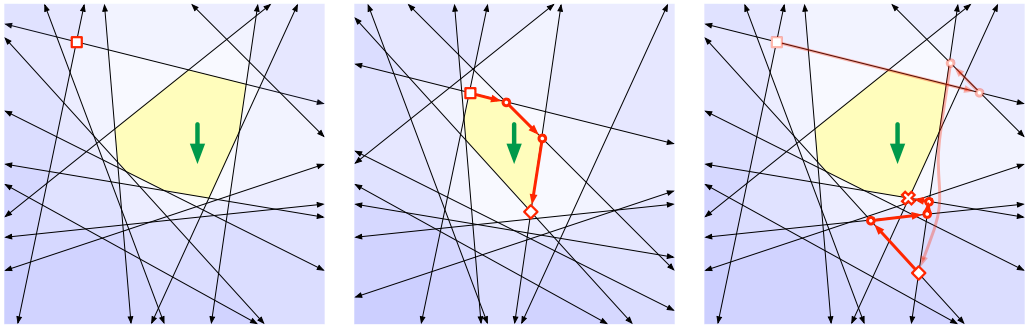
Equivalently, to find an initial locally optimal vertex for the dual simplex algorithm, we can choose an *arbitrary* vertex x, *translate* the constraint hyperplanes so that x becomes feasible, and then find the optimal vertex for the *translated* constraints by running the (primal) simplex algorithm. This vertex must be locally optimal, even after we restore the hyperplanes to their original locations!

Pseudocode for both algorithms is given in Figures I.4 and I.5. As usual, the input to both algorithms is a set H of halfspaces, and the algorithms either return the lowest vertex in the intersection of those halfspaces, report that the linear program is infeasible, or report that the linear program is unbounded.



(a) (b) (c)

Figure I.2. Primal simplex with dual initialization: (a) Choose any basis. (b) Rotate the objective to make the basis locally optimal, and pivot "up" to a feasible basis. (c) Pivot down to the optimum basis for the original objective.



(a) (b) (c)

Figure I.3. Dual simplex with primal optimization: (a) Choose any basis. (b) Translate the constraints to make the basis feasible, and pivot down to a locally optimal basis. (c) Pivot up to the optimum basis for the original constraints.

```
DUALPRIMALSIMPLEX(H):
  x ← any vertex
  H̃ ← any rotation of H that makes x locally optimal

  while x is not feasible
      if every locally optimal neighbor of x is lower (wrt H̃) than x
          return INFEASIBLE
      else
          x ← any locally optimal neighbor of x that is higher (wrt H̃) than x

  while x is not locally optimal
      if every feasible neighbor of x is higher than x
          return UNBOUNDED
      else
          x ← any feasible neighbor of x that is lower than x
  return x
```

Figure I.4. The primal simplex algorithm with dual initialization.

```
PRIMALDUALSIMPLEX(H):
  x ← any vertex
  H̃ ← any translation of H that makes x feasible

  while x is not locally optimal
      if every feasible neighbor of x is higher (wrt H̃) than x
          return UNBOUNDED
      else
          x ← any feasible neighbor of x that is lower (wrt H̃) than x

  while x is not feasible
      if every locally optimal neighbor of x is lower than x
          return INFEASIBLE
      else
          x ← any locally-optimal neighbor of x that is higher than x
  return x
```

Figure I.5. The dual simplex algorithm with primal initializzation.

I.4 Network Simplex

This section needs revision. ★★★

Our first natural examples of linear programming problems were shortest paths, maximum flows, and minimum cuts in edge-weighted graphs. It is instructive to reinterpret the behavior of the abstract simplex algorithm in terms of the original input graphs; this reinterpretation allows for a much more efficient implementation of the simplex algorithm, which is normally called ***network simplex***.

As a concrete working example, I will consider a special case of the minimum-cost flow problem called the ***transshipment problem***. The input consists of a directed graph $G = (V, E)$, a ***balance*** function $b : V \to \mathbb{R}$, and a ***cost*** function $\$: E \to \mathbb{R}$, but no capacities or lower bounds on the edges. Our goal is to compute a ***flow*** function $f : E \to \mathbb{R}$ that is non-negative everywhere, satisfies the balance constraint

$$\sum_{u \to v} f(u \to v) - \sum_{v \to w} f(v \to w) = b(v)$$

at every vertex v, and minimizes the total cost $\sum_e f(e) \cdot \$(e)$.

We can easily express this problem as a linear program with a variable for each edge and constraints for each vertex and edge.

$$\begin{aligned} \text{maximize} \quad & \sum_{u \to v} \$(u \to v) \cdot f(u \to v) \\ \text{subject to} \quad & \sum_{u \to v} f(u \to v) - \sum_{v \to w} f(v \to w) = b(v) \qquad \text{for every vertex } v \neq s \\ & f(u \to v) \geq 0 \qquad \text{for every edge } u \to v \end{aligned}$$

Here I've omitted the balance constraint for some fixed vertex s, because it is redundant; if f is balanced at every vertex except s, then f must be balanced at s as well. By interpreting the balance, cost, and flow functions as vectors, we can write this linear program more succinctly as follows:

$$\boxed{\begin{aligned} \max \quad & \$ \cdot f \\ \text{s.t.} \quad & Af = b \\ & f \geq 0 \end{aligned}}$$

Here A is the ***vertex-edge incidence matrix*** of G; this matrix has one row for each edge and one column for each vertex, and whose entries are defined as follows:

$$A(x{\to}y, v) = \begin{cases} 1 & \text{if } v = y \\ -1 & \text{if } v = x \\ 0 & \text{otherwise} \end{cases}$$

Let $\overline{G} = (V, \overline{E})$ be the undirected version of G, defined by setting $\overline{E} = \{uv \mid u{\to}v \in E\}$. In the following arguments, I will refer to "undirected cycles" and "spanning trees" in G; these phrases are shorthand for the subset of directed edges in G corresponding to undirected cycles and spanning trees in $\overline{G}$.

To simplify the remaining presentation, I will make two non-degeneracy assumptions:

- The cost vector $\$$ is non-degenerate: No residual cycle has cost 0.
- The balance vector is non-degenerate: No non-empty proper subset of vertices has total balance 0.

Because the transshipment LP has E variables, a basis consists of E linearly independent constraints. We call a basis ***balanced*** if it contains all $V-1$ balance constraints; any flow consistent with a balanced basis is balanced at *every* vertex of G. Every balanced basis contains exactly $E - V + 1$ edge constraints, and therefore *omits* exactly $V - 1$ edge constraints. We call an edge *fixed* if its constraint is included in the basis and *free* otherwise. Any flow consistent with a balanced basis is zero on every fixed edge and non-negative on every free edge.

Lemma I.1. *For every balanced basis, the free edges define a spanning tree of G; conversely, for every spanning tree T of G, there is a balanced basis for which T is the set of free edges.*[4]

Proof: First, fix an arbitrary balanced basis, let f be any flow consistent with that basis, and let T be the set of $V-1$ free edges for that basis. (The flow f need not be feasible.)

[4]More generally, every basis (balanced or not) is associated with a spanning *forest* F; the basis contains edge constraints for every edge *not* in F and all but one vertex constraint in each component of F.

For the sake of argument, suppose T contains an undirected cycle. Then by pushing flow around that cycle, we can obtain another (not necessarily feasible) flow f' that is still consistent with our fixed basis. So the basis constraints do not determine a unique flow, which means the constraints are not linearly independent, contradicting the definition of basis. We conclude that T is acyclic, and therefore defines a spanning tree of G.

On the other hand, suppose T is an arbitrary spanning tree of G. We define a function $\mathit{flow}_T : E \to \mathbb{R}$ as follows:

- For each edge $u{\to}v \in T$, we define $\mathit{flow}_T(u{\to}v)$ to be sum of balances in the component of $T \setminus u{\to}v$ that contains v. Our non-degeneracy assumption implies that $\mathit{flow}_T(u{\to}v) \neq 0$.
- For each edge $u{\to}v \notin T$, we define $\mathit{flow}_T(u{\to}v) = 0$.

Routine calculations imply flow_T is balanced at every vertex; moreover, flow_T is the *unique* flow in G that is non-zero only on edges of T. We conclude that the $V-1$ balance constraints and the $E-V+1$ edge constraints for edges not in T are linearly independent; in other words, T is the set of free edges of a balanced basis. □

For any spanning tree T and any edges $u{\to}v \notin T$, let $\boldsymbol{\mathit{cycle}_T(u{\to}v)}$ denote the directed cycle consisting of $u{\to}v$ and the unique residual path in T from v to u. Our non-degeneracy assumption implies that the total cost $\$(\mathit{cycle}_T(u{\to}v))$ of this cycle is not equal to zero. We define the ***slack*** of each edge in G as follows:

$$\mathit{slack}_T(u{\to}v) := \begin{cases} 0 & \text{if } u{\to}v \in T \\ \$(\mathit{cycle}_T(u{\to}v)) & \text{if } u{\to}v \notin T \end{cases}$$

The function $\mathit{flow}_T : E \to \mathbb{R}$ is the location of the balanced basis associated with T; the function $\mathit{slack}_T : E \to \mathbb{R}$ is essentially the location of the corresponding *dual* basis. With these two functions in hand, we can characterize balanced bases as follows:

- The basis associated with any spanning tree T is *feasible* (and thus the dual basis is locally optimal) if and only if $\mathit{flow}_T(e) \geq 0$ (and therefore $\mathit{flow}_T(e) > 0$) for every edge $e \in T$.
- The basis associated with any spanning tree T is *locally optimal* (and thus the dual basis is feasible) if and only if $\mathit{slack}_T(e) \geq 0$ (and therefore $\mathit{slack}_T(e) > 0$) for every edge $e \notin T$.

Notice that the complementary slackness conditions are automatically satisfied: For any edge e, and for any spanning tree T, we have $\mathit{flow}_T(e) \cdot \mathit{slack}_T(e) = 0$. In particular, if T is the optimal basis, then either $\mathit{flow}_T(e) > 0$ and $\mathit{slack}_T(e) = 0$, or $\mathit{flow}_T(e) = 0$ and $\mathit{slack}_T(e) > 0$.

A pivot in the simplex algorithm modifies the current basis by removing one constraint and adding another. For the transshipment LP, a pivot modifies a spanning tree T by adding an edge $e_{\text{in}} \notin T$ and removing an edge $e_{\text{out}} \in T$ to obtain a new spanning tree T'.

- The leaving edge e_{out} must lie in the unique residual cycle in $T + e_{\text{in}}$. The pivot modifies the flow function by pushing flow around the unique residual cycle in $T + e_{\text{in}}$, so that some edge e_{out} becomes empty. In particular, the pivot decreases the overall cost of the flow by $\mathit{flow}_T(e_{\text{out}}) \cdot \mathit{slack}_T(e_{\text{in}})$.
- Equivalently, the entering edge e_{in} must have one endpoint in each component of $T - e_{\text{out}}$. Let S be the set of vertices in the component of $T - e_{\text{out}}$ containing the tail of e_{out}. The pivot subtracts $\mathit{slack}_T(e_{\text{in}})$ from the slack of every edge from S to $V \setminus S$, and adds $\mathit{slack}_T(e_{\text{in}})$ to the slack of every edge from $V \setminus S$ to S.

The primal simplex algorithm starts with an arbitrary feasible basis and then repeatedly pivots to a new feasible basis with smaller cost. For the transshipment LP, we can find an initial feasible flow using the FEASIBLEFLOW algorithm from Chapter F. Each primal simplex pivot finds an edge e_{in} with negative slack and pushes flow around $\mathit{cycle}_T(e_{\text{in}})$ until some edge e_{out} is saturated. In other words, *the primal network simplex algorithm is an implementation of cycle cancellation*.

The dual simplex algorithm starts with an arbitrary locally optimal basis and then repeatedly pivots to a new locally optimal basis with larger cost. For the transshipment LP, the shortest-path tree rooted at any vertex provides a locally optimal basis. Each pivot operation finds an edge e_{out} with negative flow, removes it from the current spanning tree, and then adds the edge e_{in} whose slack is as small as possible.

★★★

I'm not happy with this presentation. I really need to reformulate the dual LP in terms of slacks, instead of the standard "distances", so that I can talk about pushing slack across cuts, just like pushing flow around cycles. This might be helped by a general discussion of cycle/circulation and cut/cocycle spaces of G: (1) orthogonal complementary subspaces of the edge/pseudoflow space of G, (2) generated by fundamental cycles and fundamental cuts of any spanning tree of G. Also, this needs examples/figures.

I.5 Linear Expected Time for Fixed Dimensions

★★★

This section needs careful revision.

In most geometric applications of linear programming, the number of variables is a small constant, but the number of constraints may still be very large.

The input to the following algorithm is a set H of n halfspaces and a set B of b hyperplanes. (B stands for *basis*.) The algorithm returns the lowest point in the intersection of the halfspaces in H and the hyperplanes B. At the top level of recursion, B is empty. I will implicitly assume that the linear program is both feasible and bounded. (If necessary, we can guarantee boundedness by adding a single halfspace to H, and we can guarantee feasibility by adding a dimension.) A point x ***violates*** a constraint h if it is not contained in the corresponding halfspace.

```
SeidelLP(H, B):
  if |B| = d
      return ⋂B
  if |H ∪ B| = d
      return ⋂(H ∪ B)
  h ← random element of H
  x ← SeidelLP(H \ h, B)        (*)
  if x violates h
      return SeidelLP(H \ h, B ∪ ∂h)
  else
      return x
```

The point x recursively computed in line $(*)$ is the optimal solution if and only if the random halfspace h is *not* one of the d halfspaces that define the optimal solution. In other words, the probability of calling $\textsc{SeidelLP}(H, B \cup h)$ is exactly $(d-b)/n$. Thus, we have the following recurrence for the expected number of recursive calls for this algorithm:

$$T(n,b) = \begin{cases} 1 & \text{if } b = d \text{ or } n+b = d \\ T(n-1,b) + \dfrac{d-b}{n} \cdot T(n-1,b+1) & \text{otherwise} \end{cases}$$

The recurrence is somewhat simpler if we write $\delta = d - b$:

$$T(n,\delta) = \begin{cases} 1 & \text{if } \delta = 0 \text{ or } n = \delta \\ T(n-1,\delta) + \dfrac{\delta}{n} \cdot T(n-1,\delta-1) & \text{otherwise} \end{cases}$$

It's easy to prove by induction that $T(n,\delta) = O(\delta!\, n)$:

$$\begin{aligned} T(n,\delta) &= T(n-1,\delta) + \frac{\delta}{n} \cdot T(n-1,\delta-1) \\ &\le \delta!\,(n-1) + \frac{\delta}{n}(\delta-1)! \cdot (n-1) && \text{[induction hypothesis]} \\ &= \delta!\,(n-1) + \delta!\,\frac{n-1}{n} \\ &\le \delta!\, n \end{aligned}$$

At the top level of recursion, we perform one violation test in $O(d)$ time. In each of the base cases, we spend $O(d^3)$ time computing the intersection point of d hyperplanes, and in the first base case, we spend $O(dn)$ additional time testing for violations. More careful analysis implies that the algorithm runs in $\boldsymbol{O(d! \cdot n)}$ ***expected time***.

Exercises

1. Fix a non-degenerate linear program in canonical form with d variables and $n+d$ constraints.
   (a) Prove that every *feasible* basis has exactly d *feasible* neighbors.
   (b) Prove that every *locally optimal* basis has exactly n *locally optimal* neighbors.

2. (a) Give an example of a non-empty polyhedron $Ax \le b$ that is unbounded for *every* objective vector c.
   (b) Give an example of an infeasible linear program whose dual is also infeasible.
   In both cases, your linear program will be degenerate.

3. Describe and analyze an algorithm that solves the following problem in $O(n)$ time: Given n red points and n blue points in the plane, either find a line that separates every red point from every blue point, or prove that no such line exists.

4. In this exercise, we develop another standard method for computing an initial feasible basis for the primal simplex algorithm. Suppose we are given a canonical linear program Π with d variables and $n+d$ constraints as input:

$$\begin{array}{rr} \max & c \cdot x \\ \text{s.t.} & Ax \le b \\ & x \ge 0 \end{array}$$

   To compute an initial feasible basis for Π, we solve a modified linear program Π' defined by introducing a new variable λ and two new constraints $0 \le \lambda \le 1$, and modifying the objective function:

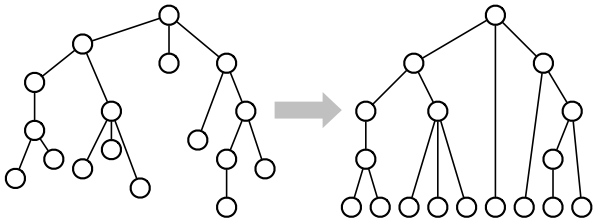
$$\begin{array}{rr} \max & \lambda \\ \text{s.t.} & Ax - b\lambda \le 0 \\ & \lambda \le 1 \\ & x, \lambda \ge 0 \end{array}$$

   (a) Prove that $x_1 = x_2 = \cdots = x_d = \lambda = 0$ is a feasible basis for Π'.
   (b) Prove that Π is feasible if and only if the optimal value for Π' is 1.
   (c) What is the dual of Π'?

5. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time. Your algorithm should return an

optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. *[Hint: Add one variable and one constraint.]*

6. Suppose your are given a rooted tree T, where every edge e has two associated values: a non-negative *length* $\ell(e)$, and a *cost* $\$(e)$ (which may be positive, negative, or zero). Your goal is to add a non-negative *stretch* $s(e) \geq 0$ to the length of every edge e in T, subject to the following conditions:
   - Every root-to-leaf path π in T has the same total stretched length $\sum_{e\in\pi}(\ell(e)+s(e))$
   - The total *weighted* stretch $\sum_e s(e)\cdot\$(\ell)$ is as small as possible.



   (a) Give a concise linear programming formulation of this problem.

   (b) Prove that in any optimal solution to this problem, we have $s(e) = 0$ for every edge on some longest root-to-leaf path in T. In other words, prove that the optimally stretched tree has the same depth as the input tree. *[Hint: What is a basis in your linear program? When is a basis feasible?]*

   (c) Describe and analyze an algorithm that solves this problem in $O(n)$ time. Your algorithm should either compute the minimum total weighted stretch, or report correctly that the total weighted stretch can be made arbitrarily negative.

7. Recall that the single-source shortest path problem can be formulated as a linear programming problem, with one variable d_v for each vertex $v \neq s$ in the input graph, as follows:

$$\begin{aligned}
\text{maximize} \quad & \sum_v d_v \\
\text{subject to} \quad & d_v \leq \ell_{s\to v} && \text{for every edge } s \to v \\
& d_v - d_u \leq \ell_{u\to v} && \text{for every edge } u \to v \text{ with } u \neq s \\
& d_v \geq 0 && \text{for every vertex } v \neq s
\end{aligned}$$

   This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of distances. Assume that the edge weights $\ell_{u\to v}$ are all non-negative and that there is a unique shortest path between any two vertices in the graph.

   (a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?

(b) Show that in the optimal basis, every variable d_v is equal to the shortest-path distance from s to v.

(c) Describe the primal simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

(d) Describe the dual simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

(e) Is Dijkstra's algorithm an instance of network simplex? Is Shimbel-Bellman-Ford? Justify your answers.

(f) Using the results in problem 9, prove that if the edge lengths $\ell_{u\to v}$ are all integral, then the optimal distances d_v are also integral.

8. The maximum (s,t)-flow problem can be formulated as a linear programming problem, with one variable $f_{u\to v}$ for each edge $u \to v$ in the input graph:

$$\begin{aligned}
\text{maximize} \quad & \sum_w f_{s\to w} - \sum_u f_{u\to s} \\
\text{subject to} \quad & \sum_w f_{v\to w} - \sum_u f_{u\to v} = 0 \qquad \text{for every vertex } v \neq s,t \\
& f_{u\to v} \le c_{u\to v} \quad \text{for every edge } u \to v \\
& f_{u\to v} \ge 0 \qquad \text{for every edge } u \to v
\end{aligned}$$

This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of flows.

(a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?

(b) Show that the optimal basis represents a maximum flow.

(c) Describe the primal simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

(d) Describe the dual simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

(e) Is the Ford-Fulkerson augmenting-path algorithm an instance of network simplex? Justify your answer. *[Hint: There is a one-line argument.]*

(f) Using the results in problem 9, prove that if the capacities $c_{u\to v}$ are all integral, then the maximum flow values $f_{u\to v}$ are also integral.

9. A ***minor*** of a matrix A is the submatrix defined by any subset of the rows and any subset of the columns. A matrix A is ***totally unimodular*** if, for every square minor M, the determinant of M is -1, 0, or 1.

   (a) Let A be an arbitrary totally unimodular matrix.

      i. Prove that the transposed matrix $A^\top$ is also totally unimodular.

      ii. Prove that negating any row or column of A leaves the matrix totally unimodular.

      iii. Prove that the block matrix $[A \mid I]$ is totally unimodular.

   (b) Prove that for any totally unimodular matrix A and any integer vector b, the canonical linear program $\max\{c \cdot x \mid Ax \le b, x \ge 0\}$ has an *integer* optimal solution. *[Hint: Cramer's rule.]*

   (c) The ***unsigned incidence matrix*** of an undirected graph $G = (V, E)$ is an $|V| \times |E|$ matrix A, with rows indexed by vertices and columns indexed by edges, where for each row v and column uw, we have

$$A[v, uw] = \begin{cases} 1 & \text{if } v = u \text{ or } v = w \\ 0 & \text{otherwise} \end{cases}$$

   Prove that the unsigned incidence matrix of every *bipartite* graph G is totally unimodular. *[Hint: Each square minor corresponds to a subgraph H of G with k vertices and k edges, for some integer k. Argue that at least one of the following statements must be true: (1) H is disconnected; (2) H has a vertex with degree 1; (3) H is an even cycle.]*

   (d) Prove for every *non*-bipartite graph G that the unsigned incidence matrix of G is *not* totally unimodular. *[Hint: Consider any odd cycle.]*

   (e) The ***signed incidence matrix*** of a directed graph $G = (V, E)$ is also an $|V| \times |E|$ matrix A, with rows indexed by vertices and columns indexed by edges, where for each row v and column $u{\to}w$, we have

$$A[u, v{\to}w] = \begin{cases} 1 & \text{if } v = w \\ -1 & \text{if } v = u \\ 0 & \text{otherwise} \end{cases}$$

   Prove that the signed incidence matrix of every directed graph G is totally unimodular.

Le mieux est l'ennemi du bien. [The best is the enemy of the good.]

— Voltaire, *La Bégueule* (1772)

Who shall forbid a wise skepticism, seeing that there is no practical question on which any thing more than an approximate solution can be had?

— Ralph Waldo Emerson, *Representative Men* (1850)

We dance round in a ring and suppose,
But the secret sits in the middle and knows.

— Robert Frost, "The Secret Sits" (1942)

Now, distrust of corporations threatens our still-tentative economic recovery; it turns out greed is bad, after all.

— Paul Krugman, "Greed is Bad", *The New York Times* (June 4, 2002)

J

Approximation Algorithms

[Read Chapter 12 first.]

Status: Needs revision.

J.1 Load Balancing

On the future viral-hit online reality game show *Grunt Work*, broadcast every Wednesday at 3am on YouTube Green, the contestants are given a series of utterly pointless tasks to perform. Each task has a predetermined time limit; for example, "Sharpen this pencil for 17 seconds," or "Pour pig's blood on your head and sing The Star-Spangled Banner for two minutes," or "Listen to this 75-minute algorithms lecture." The directors of the show want you to assign each task to one of the contestants, so that the last task is completed as early as possible. When your predecessor correctly informed the directors that their problem is NP-hard, he was immediately fired. "Time is money!" they screamed at him. "We don't need perfection. Wake up, dude, this is the *internet*!"

Less facetiously, suppose we have a set of n jobs, which we want to assign to m machines. We are given an array $T[1..n]$ of non-negative numbers, where $T[j]$ is the running time of job j. We can describe an *assignment* by an array $A[1..n]$, where

$A[j] = i$ means that job j is assigned to machine i. The *makespan* of an assignment is the maximum time that any machine is busy:

$$makespan(A) = \max_i \sum_{A[j]=i} T[j]$$

The *load balancing* problem is to compute the assignment with the smallest possible makespan.

It's not hard to prove that the load balancing problem is NP-hard by reduction from PARTITION: The array $T[1..n]$ can be evenly partitioned if and only if there is an assignment to two machines with makespan exactly $\sum_i T[i]/2$. A slightly more complicated reduction from 3PARTITION implies that the load balancing problem is *strongly* NP-hard. If we really need the optimal solution, there is a dynamic programming algorithm that runs in time $O(nM^m)$, where M is the minimum makespan, but that's just horrible.

There is a fairly natural and efficient greedy heuristic for load balancing: consider the jobs one at a time, and assign each job to the machine i with the earliest finishing time $Total[i]$.

```
GREEDYLOADBALANCE(T[1..n], m):
  for i ← 1 to m
      Total[i] ← 0

  for j ← 1 to n
      mini ← arg min_i Total[i]
      A[j] ← mini
      Total[mini] ← Total[mini] + T[j]

  return A[1..m]
```

Theorem J.1. *The makespan of the assignment computed by* GREEDYLOADBALANCE *is at most twice the makespan of the optimal assignment.*

Proof: Fix an arbitrary input, and let *OPT* denote the makespan of its optimal assignment. The approximation bound follows from two trivial observations. First, the makespan of any assignment (and therefore of the optimal assignment) is at least the duration of the longest job. Second, the makespan of any assignment is at least the total duration of all the jobs divided by the number of machines.

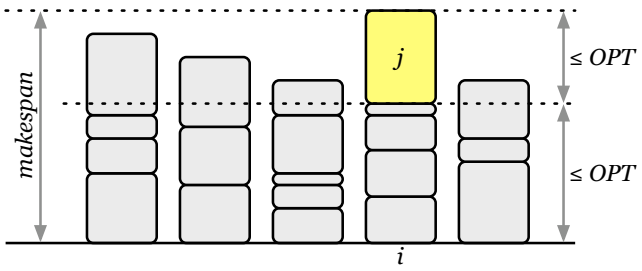
$$OPT \geq \max_j T[j] \qquad \text{and} \qquad OPT \geq \frac{1}{m} \sum_{j=1}^{n} T[j]$$

Now consider the assignment computed by GREEDYLOADBALANCE. Suppose machine i has the largest total running time, and let j be the last job assigned to machine i. Our first trivial observation implies that $T[j] \leq OPT$. To finish the proof, we must show that

$Total[i] - T[j] \le OPT$. Job j was assigned to machine i because it had the smallest finishing time, so $Total[i] - T[j] \le Total[k]$ for all k. (Some values $Total[k]$ may have increased since job j was assigned, but that only helps us.) In particular, $Total[i] - T[j]$ is less than or equal to the *average* finishing time over all machines. Thus,

$$Total[i] - T[j] \le \frac{1}{m}\sum_{i=1}^{m} Total[i] = \frac{1}{m}\sum_{j=1}^{n} T[j] \le OPT$$

by our second trivial observation. We conclude that the makespan $Total[i]$ is at most $2 \cdot OPT$. □



Proof that GreedyLoadBalance is a 2-approximation algorithm

GreedyLoadBalance is an *online* algorithm: It assigns jobs to machines in the order that the jobs appear in the input array. Online approximation algorithms are useful in settings where inputs arrive in a stream of unknown length—for example, real jobs arriving at a real scheduling algorithm. In this online setting, it may be *impossible* to compute an optimum solution, even in cases where the offline problem (where all inputs are known in advance) can be solved in polynomial time. The study of online algorithms could easily fill an entire one-semester course (alas, not this one).

In our original offline setting, we can improve the approximation factor by sorting the jobs before piping them through the greedy algorithm.

```
SortedGreedyLoadBalance(T[1..n], m):
  sort T in decreasing order
  return GreedyLoadBalance(T, m)
```

Theorem J.2. *The makespan of the assignment computed by* SortedGreedyLoadBalance *is at most* $3/2$ *times the makespan of the optimal assignment.*

Proof: Let i be the busiest machine in the schedule computed by SortedGreedyLoadBalance. If only one job is assigned to machine i, then the greedy schedule is actually optimal, and the theorem is trivially true. Otherwise, let j be the last job assigned to machine i. Since each of the first m jobs is assigned to a unique machine, we must have $j \ge m + 1$. As in the previous proof, we know that $Total[i] - T[j] \le OPT$.

In *any* schedule, at least two of the first $m+1$ jobs, say jobs k and ℓ, must be assigned to the same machine. Thus, $T[k] + T[\ell] \le OPT$. Since $\max\{k, \ell\} \le m+1 \le j$, and the jobs are sorted in decreasing order by duration, we have

$$T[j] \le T[m+1] \le T[\max\{k,\ell\}] = \min\{T[k], T[\ell]\} \le OPT/2.$$

We conclude that the makespan $Total[i]$ is at most $3 \cdot OPT/2$, as claimed. □

In fact, neither of these analyses is tight. The exact approximation ratio of GREEDYLOADBALANCE is actually $2 - 1/m$, and the exact approximation ratio of SORTED-GREEDYLOADBALANCE is actually $4/3 - 1/(3m)$.

J.2 Generalities

Consider an arbitrary optimization problem. Let $OPT(X)$ denote the value of the optimal solution for a given input X, and let $A(X)$ denote the value of the solution computed by algorithm A given the same input X. We say that A is an ***α(n)-approximation algorithm*** if and only if

$$\frac{OPT(X)}{A(X)} \le \alpha(n) \quad \text{and} \quad \frac{A(X)}{OPT(X)} \le \alpha(n)$$

for all inputs X of size n. The function $\alpha(n)$ is called the ***approximation factor*** for algorithm A. For any given algorithm, only one of these two inequalities will be important. For maximization problems, where we want to compute a solution whose cost is as small as possible, the first inequality is trivial. For maximization problems, where we want a solution whose value is as large as possible, the second inequality is trivial. A 1-approximation algorithm always returns the exact optimal solution.

Especially for problems where exact optimization is NP-hard, we have little hope of completely characterizing the optimal solution. The secret to proving that an algorithm satisfies some approximation ratio is to find a useful function of the input that provides both lower bounds on the cost of the optimal solution and upper bounds on the cost of the approximate solution. For example, if $OPT(X) \ge f(X)/2$ and $A(X) \le 5f(X)$ for *any* function f, then A is a 10-approximation algorithm. Finding the right intermediate function can be a delicate balancing act.

J.3 Greedy Vertex Cover

Recall that the *vertex color* problem asks, given a graph G, for the smallest set of vertices of G that cover every edge. This is one of the first NP-hard problems introduced in the first week of class. There is a natural and efficient greedy heuristic[1] for computing a small vertex cover: mark the vertex with the largest degree, remove all the edges incident to that vertex, and recurse.

[1] A *heuristic* is an algorithm that doesn't work.

```
GreedyVertexCover(G):
  C ← ∅
  while G has at least one edge
      v ← vertex in G with maximum degree
      G ← G \ v
      C ← C ∪ v
  return C
```

Obviously this algorithm doesn't compute the optimal vertex cover—that would imply P=NP!—but it does compute a reasonably close approximation.

Theorem J.3. *GreedyVertexCover is an $O(\log n)$-approximation algorithm.*

Proof: For all i, let G_i denote the graph G after i iterations of the main loop, and let d_i denote the maximum degree of any node in G_{i-1}. We can define these variables more directly by adding a few extra lines to our algorithm:

```
GreedyVertexCover(G):
  C ← ∅
  G_0 ← G
  i ← 0
  while G_i has at least one edge
      i ← i + 1
      v_i ← vertex in G_{i-1} with maximum degree
      d_i ← deg_{G_{i-1}}(v_i)
      G_i ← G_{i-1} \ v_i
      C ← C ∪ v_i
  return C
```

Let $|G_{i-1}|$ denote the number of edges in the graph G_{i-1}. Let C^* denote the optimal vertex cover of G, which consists of OPT vertices. Since C^* is also a vertex cover for G_{i-1}, we have

$$\sum_{v \in C^*} \deg_{G_{i-1}}(v) \geq |G_{i-1}|.$$

In other words, the *average* degree in G_i of any node in C^* is at least $|G_{i-1}|/OPT$. It follows that G_{i-1} has at least one node with degree at least $|G_{i-1}|/OPT$. Since d_i is the maximum degree of any node in G_{i-1}, we have

$$d_i \geq \frac{|G_{i-1}|}{OPT}$$

Moreover, for any $j \geq i-1$, the subgraph G_j has no more edges than G_{i-1}, so $d_i \geq |G_j|/OPT$. This observation implies that

$$\sum_{i=1}^{OPT} d_i \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{i-1}|}{OPT} \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{OPT}|}{OPT} \;=\; |G_{OPT}| \;=\; |G| - \sum_{i=1}^{OPT} d_i.$$

In other words, the first OPT iterations of GREEDYVERTEXCOVER remove at least half the edges of G. Thus, after at most $OPT \lg|G| \leq 2\,OPT \lg n$ iterations, all the edges of G have been removed, and the algorithm terminates. We conclude that GREEDYVERTEXCOVER computes a vertex cover of size $O(OPT \log n)$. □

So far we've only proved an *upper bound* on the approximation factor of GREEDYVERTEXCOVER; perhaps a more careful analysis would imply that the approximation factor is only $O(\log \log n)$, or even $O(1)$. Alas, no such improvement is possible. For any integer n, a simple recursive construction gives us an n-vertex graph for which the greedy algorithm returns a vertex cover of size $\Omega(OPT \cdot \log n)$. Details are left as an exercise for the reader.

J.4 Set Cover and Hitting Set

The greedy algorithm for vertex cover can be applied almost immediately to two more general problems: ***minimum set cover*** and ***minimum hitting set***. The input for both of these problems is a *set system* $(X, \mathcal{F})$, where X is a finite *ground set*, and $\mathcal{F}$ is a family of subsets of X. A *set cover* of a set system $(X, \mathcal{F})$ is a subfamily of sets in $\mathcal{F}$ whose union is the entire ground set X. A *hitting set* for $(X, \mathcal{F})$ is a subset of the ground set X that intersects every set in $\mathcal{F}$.

An undirected graph can be cast as a set system in two different ways. In one formulation, the ground set X contains the vertices, and each edge defines a set of two vertices in $\mathcal{F}$. In this formulation, a vertex cover is a hitting set. In the other formulation, the *edges* are the ground set, the *vertices* define the family of subsets, and a vertex cover is a set cover.

Here are the natural greedy algorithms for finding a small set cover and finding a small hitting set. Essentially the same analysis as for our greedy algorithm for vertex cover implies that GREEDYSETCOVER finds a set cover whose size is at most $O(\log|\mathcal{F}|)$ times the size of smallest set cover, and that GREEDYHITTINGSET finds a hitting set whose size is at most $O(\log|X|)$ times the size of the smallest hitting set. These are the best approximation ratios possible (up to lower order terms) unless P=NP.

```
GREEDYSETCOVER(X, F):
  C ← ∅
  while X ≠ ∅
      S ← arg max_{S∈F} |S ∩ X|
      X ← X \ S
      C ← C ∪ {S}
  return C
```

```
GREEDYHITTINGSET(X, F):
  H ← ∅
  while F ≠ ∅
      x ← arg max_{x∈X} |{S ∈ F | x ∈ S}|
      F ← F \ {S ∈ F | x ∈ S}
      H ← H ∪ {x}
  return H
```

The similarity between these two algorithms is no coincidence. For any set system $(X, \mathcal{F})$, there is a *dual* set system $(\mathcal{F}, X^*)$ defined as follows. For any element $x \in X$ in the ground set, let x^* denote the subfamily of sets in $\mathcal{F}$ that contain x:

$$x^* = \{S \in \mathcal{F} \mid x \in S\}.$$

Finally, let X^* denote the collection of all subsets of the form x^*:

$$X^* = \{x^* \mid x \in S\}.$$

As an example, suppose X is the set of letters of alphabet and $\mathcal{F}$ is the set of last names of student taking CS 473 this semester. Then X^* has 26 elements, each containing the subset of CS 473 students whose last name contains a particular letter of the alphabet. For example, m^* is the set of students whose last names contain the letter m.

There is a natural bijection between the ground set X and the dual set family X^*. It is a tedious but instructive exercise to prove that the dual of the dual of any set system is isomorphic to the original set system—$(X^*, \mathcal{F}^*)$ is essentially the same as $(X, \mathcal{F})$. It is also easy to prove that a set cover for any set system $(X, \mathcal{F})$ is also a hitting set for the dual set system $(\mathcal{F}, X^*)$, and therefore a hitting set for any set system $(X, \mathcal{F})$ is isomorphic to (or more simply, "is") a set cover for the dual set system $(\mathcal{F}, X^*)$.

J.5 Vertex Cover Redux

The greedy approach doesn't always lead to the best approximation algorithms. Consider the following obviously stupid heuristic for vertex cover:

```
DumbVertexCover(G):
  C ← ∅
  while G has at least one edge
      uv ← any edge in G
      G ← G \ {u, v}
      C ← C ∪ {u, v}
  return C
```

The minimum vertex cover—in fact, *every* vertex cover—contains at least one of the two vertices u and v chosen inside the while loop. It follows immediately that DumbVertexCover is a 2-approximation algorithm! Surprisingly, this "obviously stupid" algorithm is essentially the best polynomial-time approximation algorithm known.

The same stupid idea can be extended to approximate the minimum hitting set for any set system $(X, \mathcal{F})$; the resulting approximation factor is equal to the size of the largest set in $\mathcal{F}$.

J.6 Lightest Vertex Cover: LP Rounding

Now consider a generalization of the minimum vertex cover problem in which every vertex v of the input graph has a non-negative weight $w(v)$, and the goal is to compute a vertex cover C with minimum total weight $w(C) = \sum_{v \in C} w(v)$. Both the greedy and the stupid approximation algorithms can perform arbitrarily badly in this setting. Let G be a path of length 2, where the endpoints have weight 1 and the interior vertex has weight

10^{100}. The lightest vertex cover has weight 2, but the greedy vertex cover has weight 10^{100}, and the stupid vertex cover has weight $10^{100}+1$.

But we can recover a 2-approximation algorithm by casting the problem as an instance of ***integer linear programming***. An integer linear program is just a linear program with the added constraint that all variables are integers. Almost any of the NP-hard problems considered in the previous lecture note can be cast as integer linear programs, which implies that integer linear programming is NP-hard. In particular, the following integer linear program encodes the lightest vertex cover problem. Each feasible solution to this ILP corresponds to some vertex cover; each variable $x(v)$ indicates whether that vertex cover contains the corresponding vertex v.

$$\begin{array}{lrl} \text{minimize} & \displaystyle\sum_v w(v)\cdot x(v) & \\ \text{subject to} & x(u)+x(v)\geq 1 & \text{for every edge } uv \\ & x(v)\in\{0,1\} & \text{for every vertex } v \end{array}$$

Let *OPT* denote the weight of the lightest vertex cover; *OPT* is also the optimal objective value for this integer linear program.

Now we're going to do something weird. Consider the linear program obtained from this ILP by removing the integrality constraint; this linear program is usually called the ***linear relaxation*** of the integer linear program.

$$\begin{array}{lrl} \text{minimize} & \displaystyle\sum_v w(v)\cdot x(v) & \\ \text{subject to} & x(u)+x(v)\geq 1 & \text{for every edge } uv \\ & 0\leq x(v)\leq 1 & \text{for every vertex } v \end{array}$$

Like all linear programs, this one can be solved in polynomial time; let x^* denote its optimal solution, and let $\widetilde{\boldsymbol{OPT}} = \sum_v w(v)\cdot x^*(v)$ denote the denote the optimal objective value. Unfortunately, the optimal solution may not be integral; the "indicator" variables $x^*(v)$ may be rational numbers between 0 and 1. Thus, x^* is often called the optimal ***fractional*** solution. Nevertheless, this solution is useful for two reasons.

First, because every feasible solution to the original ILP is also feasible for its linear relaxation, the optimal fractional solution is a lower bound on the optimal integral solution: $\boldsymbol{OPT} \geq \widetilde{\boldsymbol{OPT}}$. As we've already seen, finding lower bounds on the optimal solution is the key to deriving good approximation algorithms.

Second, we can derive a good approximation to the lightest vertex cover by ***rounding*** the optimal fractional solution. Specifically, for each vertex v, let

$$x'(v) = \begin{cases} 1 & \text{if } x^*(v)\geq 1/2, \\ 0 & \text{otherwise.} \end{cases}$$

For every edge uv, the constraint $x^*(u)+x^*(v) \geq 1$ implies that $\max\{x^*(u), x^*(v)\} \geq 1/2$, and therefore either $x'(u) = 1$ or $x'(v) = 1$. Thus, x' is the indicator function of a vertex cover. On the other hand, we have $x'(v) \leq 2x^*(v)$ for every vertex v, which implies that

$$\sum_v w(v) \cdot x'(v) \;\leq\; 2\sum_v w(v) \cdot x^*(v) \;=\; 2 \cdot \widetilde{OPT} \;\leq\; 2 \cdot OPT.$$

So we've just described a simple polynomial-time 2-approximation algorithm for lightest vertex cover: Compute the optimal fractional solution, and round it as described above.

J.7 Randomized LP Rounding

We can also round the optimal fractional solution *randomly* by interpreting each fractional value $x^*(v)$ as a probability. For each vertex v, independently set

$$x'(v) = \begin{cases} 1 & \text{with probability } x^*(v), \\ 0 & \text{otherwise.} \end{cases}$$

Our usual argument from linearity of expectation immediately implies that

$$\mathrm{E}\left[\sum_v w(v) \cdot x'(v)\right] \;=\; \sum_v w(v) \cdot x^*(v) \;=\; \widetilde{OPT} \;\leq\; OPT.$$

Unfortunately, x' is not necessarily the indicator function of a vertex cover. The probability that any particular edge uv is uncovered is

$$\Pr[x'(u) = 0 \wedge x'(v) = 0] \;=\; (1 - x^*(u))(1 - x^*(v)).$$

Thanks to the constraint $x^*(u) = x^*(v) \geq 1$, the expression $(1 - x^*(u))(1 - x^*(v))$ is minimized when $x^*(v) = x^*(u) = 1/2$. It follows that each edge uv is covered with probability at least $3/4$; equivalently, we expect x' to cover $3/4$ of the edges of the graph.

To construct an actual vertex cover, we can run the randomized rounding algorithm several times in succession, and take the union of the resulting vertex sets.

```
ApproxLightestVertexCover(G):
  C ← ∅
  x* ← optimal fractional solution
  for i ← 1 to 2 lg n
      for all vertices v
          with probability x*(v)
              C ← C ∪ {x}
  return C
```

Because the random choices in each iteration of the main loop are independent, we can simplify the algorithm by changing the rounding criterion as follows, where $N = 2\lg n$:

$$x'(v) = \begin{cases} 1 & \text{with probability } 1-(1-x^*(v))^N, \\ 0 & \text{otherwise,} \end{cases}$$

Let C be the set of vertices returned by this algorithm. Each edge uv is left uncovered by C with probability $(1-x^*(u))^N(1-x^*(v))^N \le 1/4^N = 1/n^4$. It follows that C is a vertex cover with probability at least $1-1/n^2$. On the other hand, linearity of expectation immediately implies that the expected size of C is exactly $N \cdot \widetilde{OPT} \le 2\lg n \cdot OPT$.

If we want to *guarantee* a vertex cover, we can modify the main loop in APPROXLIGHTESTVERTEXCOVER to continue running until every edge is covered. The loop will terminate after $O(\log n)$ iterations with high probability, and the expected size of the resulting vertex cover is still at most $O(\log n) \cdot OPT$. Thus, at least in expectation, we obtain the same $O(\log n)$ approximation ratio (at least in expectation) as our original greedy algorithm.

This randomized rounding strategy can be generalized to the lightest *set* cover problem, where each subset in the input family has an associated weight, and the goal is to find the lightest collection of subsets that covers the ground set. The resulting rounding algorithm computes a set cover whose expected weight is at most $O(\log |\mathcal{F}|)$ times the weight of the lightest set cover, matching the performance of the greedy algorithm for unweighted set cover. The dual of this algorithm is a randomized $O(\log |X|)$-approximation algorithm for weighted hitting set (where every item in the ground set has a weight).

★★★ Derandomization: Greedy set cover with prices with proof via LP duality

J.8 Traveling Salesman: The Bad News

The ***traveling salesman problem***[2] problem asks for the shortest Hamiltonian cycle in a weighted undirected graph. To keep the problem simple, we can assume without loss of generality that the underlying graph is always the complete graph K_n for some integer n; thus, the input to the traveling salesman problem is just a list of the $\binom{n}{2}$ edge lengths.

Not surprisingly, given its similarity to the Hamiltonian cycle problem, it's quite easy to prove that the traveling salesman problem is NP-hard. Let G be an arbitrary undirected graph with n vertices. We can construct a length function for K_n as follows:

$$\ell(e) = \begin{cases} 1 & \text{if } e \text{ is an edge in } G, \\ 2 & \text{otherwise.} \end{cases}$$

[2]This is sometimes bowdlerized into the traveling sales*person* problem. That's just silly. Who ever heard of a traveling salesperson sleeping with the farmer's child?

Now it should be obvious that if G has a Hamiltonian cycle, then there is a Hamiltonian cycle in K_n whose length is exactly n; otherwise every Hamiltonian cycle in K_n has length at least $n+1$. We can clearly compute the lengths in polynomial time, so we have a polynomial time reduction from Hamiltonian cycle to traveling salesman. Thus, the traveling salesman problem is NP-hard, even if all the edge lengths are 1 and 2.

There's nothing special about the values 1 and 2 in this reduction; we can replace them with any values we like. By choosing values that are sufficiently far apart, we can show that even approximating the shortest traveling salesman tour is NP-hard. For example, suppose we set the length of the 'absent' edges to $n+1$ instead of 2. Then the shortest traveling salesman tour in the resulting weighted graph either has length exactly n (if G has a Hamiltonian cycle) or has length at least $2n$ (if G does not have a Hamiltonian cycle). Thus, if we could approximate the shortest traveling salesman tour within a factor of 2 in polynomial time, we would have a polynomial-time algorithm for the Hamiltonian cycle problem.

Pushing this idea to its limits us the following negative result.

Theorem J.4. *For any function $f(n)$ that can be computed in time polynomial in n, there is no polynomial-time $f(n)$-approximation algorithm for the traveling salesman problem on general weighted graphs, unless P=NP.*

J.9 Traveling Salesman: The Good News

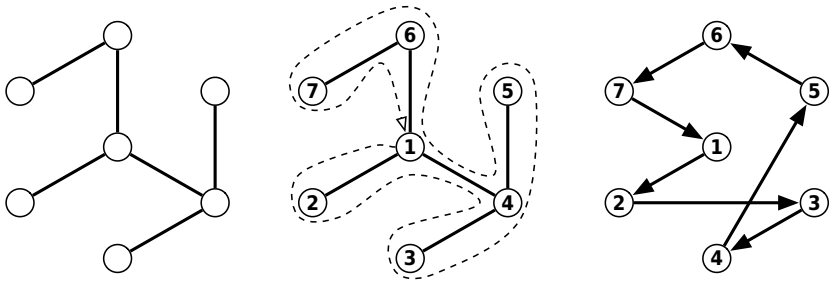
Even though the general traveling salesman problem can't be approximated, a common special case can be approximated fairly easily. The special case requires the edge lengths to satisfy the so-called *triangle inequality*:

$$\ell(u,w) \le \ell(u,v) + \ell(v,w) \quad \text{for any vertices } u, v, w.$$

This inequality is satisfied for *geometric graphs*, where the vertices are points in the plane (or some higher-dimensional space), edges are straight line segments, and lengths are measured in the usual Euclidean metric. Notice that the length functions we used above to show that the general TSP is hard to approximate do not (always) satisfy the triangle inequality.

With the triangle inequality in place, we can quickly compute a 2-approximation for the traveling salesman tour as follows. First, we compute the minimum spanning tree T of the weighted input graph; this can be done in $O(n^2 \log n)$ time (where n is the number of vertices of the graph) using any of several classical algorithms. Second, we perform a depth-first traversal of T, numbering the vertices in the order that we first encounter them. Because T is a spanning tree, every vertex is numbered. Finally, we return the cycle obtained by visiting the vertices according to this numbering.

Theorem J.5. *A depth-first ordering of the minimum spanning tree gives a 2-approximation of the shortest traveling salesman tour.*



A minimum spanning tree T, a depth-first traversal of T, and the resulting approximate traveling salesman tour.

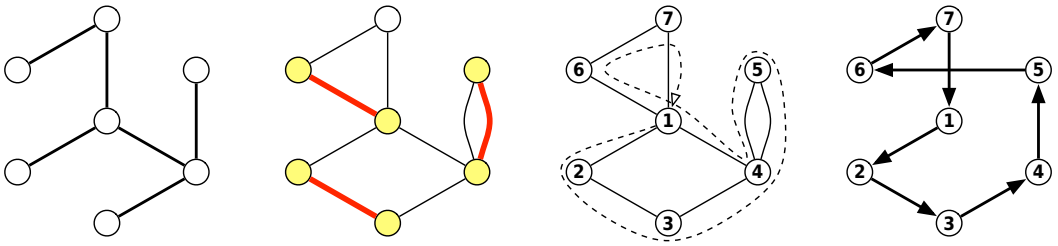
Proof: Let *OPT* denote the cost of the optimal TSP tour, let *MST* denote the total length of the minimum spanning tree, and let A be the length of the tour computed by our approximation algorithm. Consider the 'tour' obtained by walking through the minimum spanning tree in depth-first order. Since this tour traverses every edge in the tree exactly twice, its length is $2 \cdot MST$. The final tour can be obtained from this one by removing duplicate vertices, moving directly from each node to the next *unvisited* node.; the triangle inequality implies that taking these shortcuts cannot make the tour longer. Thus, $A \leq 2 \cdot MST$. On the other hand, if we remove any edge from the optimal tour, we obtain a spanning tree (in fact a spanning *path*) of the graph; thus, $MST \geq OPT$. We conclude that $A \leq 2 \cdot OPT$; our algorithm computes a 2-approximation of the optimal tour. □

We can improve this approximation factor using the following algorithm discovered by Nicos Christofides in 1976. As in the previous algorithm, we start by constructing the minimum spanning tree T. Then let O be the set of vertices with *odd* degree in T; it is an easy exercise (hint, hint) to show that the number of vertices in O is even.

In the next stage of the algorithm, we compute a *minimum-cost perfect matching* M of these odd-degree vertices. A perfect matching is a collection of edges, where each edge has both endpoints in O and each vertex in O is adjacent to exactly one edge; we want the perfect matching of minimum total length. A minimum-cost perfect matching can be computed in polynomial time using maximum-flow techniques, which are described in a different lecture note.

Now consider the multigraph $T \cup M$; any edge in both T and M appears twice in this multigraph. This graph is connected, and every vertex has even degree. Thus, it contains an *Eulerian circuit*: a closed walk that uses every edge exactly once. We can compute such a walk in $O(n)$ time with a simple modification of depth-first search. To obtain the final approximate TSP tour, we number the vertices in the order they first appear in some Eulerian circuit of $T \cup M$, and return the cycle obtained by visiting the vertices according to that numbering.

Theorem J.6. *Given a weighted graph that obeys the triangle inequality, the Christofides heuristic computes a (3/2)-approximation of the shortest traveling salesman tour.*



A minimum spanning tree T, a minimum-cost perfect matching M of the odd vertices in T, an Eulerian circuit of $T \cup M$, and the resulting approximate traveling salesman tour.

Proof: Let A denote the length of the tour computed by the Christofides heuristic; let OPT denote the length of the optimal tour; let MST denote the total length of the minimum spanning tree; let MOM denote the total length of the minimum odd-vertex matching.

The graph $T \cup M$, and therefore any Euler tour of $T \cup M$, has total length $MST + MOM$. By the triangle inequality, taking a shortcut past a previously visited vertex can only shorten the tour. Thus, $A \leq MST + MOM$.

By the triangle inequality, the optimal tour of the odd-degree vertices of T cannot be longer than OPT. Any cycle passing through of the odd vertices can be partitioned into two perfect matchings, by alternately coloring the edges of the cycle red and green. One of these two matchings has length at most $OPT/2$. On the other hand, both matchings have length at least MOM. Thus, $MOM \leq OPT/2$.

Finally, recall our earlier observation that $MST \leq OPT$.

Putting these three inequalities together, we conclude that $A \leq 3 \cdot OPT/2$, as claimed. □

Four decades after its discovery, Christofedes' algorithm is the best approximation algorithm known for metric TSP, although better approximation algorithms are known for interesting special cases. In particular, there have been several recent breakthroughs for the special case where the underlying metric is determined by shortest-path distances in an *unweighted* graph G; this special case is often called the ***graphic*** traveling salesman problem. The best approximation ratio known for this special case is $7/5$, from a 2012 algorithm of Sebő and Vygen.

J.10 k-center Clustering

The *k-center clustering* problem is defined as follows. We are given a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane[3] and an integer k. Our goal to find a collection of k circles that collectively enclose all the input points, such that the radius of the largest circle is as

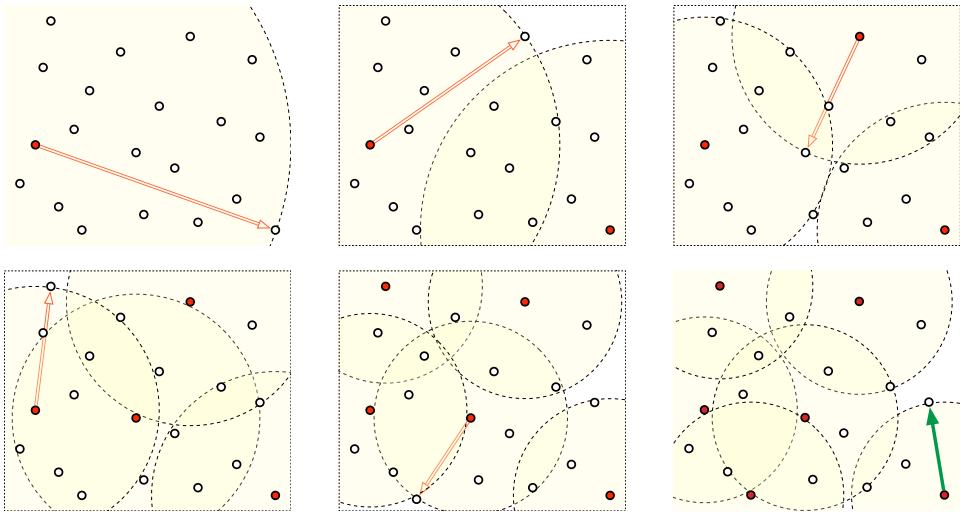
[3]The k-center problem can be defined over *any* metric space, and the approximation analysis in this section holds in any metric space as well. The analysis in the *next* section, however, does require that the points come from the Euclidean plane.

small as possible. More formally, we want to compute a set $C = \{c_1, c_2, \ldots, c_k\}$ of k *center points*, such that the following cost function is minimized:

$$cost(C) := \max_i \min_j |p_i\, c_j|.$$

Here, $|p_i\, c_j|$ denotes the Euclidean distance between input point p_i and center point c_j. Intuitively, each input point is assigned to its closest center point; the points assigned to a given center c_j comprise a *cluster*. The distance from c_j to the farthest point in its cluster is the *radius* of that cluster; the cluster is contained in a circle of this radius centered at c_j. The k-center clustering cost $cost(C)$ is precisely the maximum cluster radius.

This problem turns out to be NP-hard, even to approximate within a factor of roughly 1.8. However, there is a natural greedy strategy, first analyzed in 1985 by Teofilo Gonzalez[4], that is guaranteed to produce a clustering whose cost is at most twice optimal. Choose the k center points one at a time, starting with an arbitrary input point as the first center. In each iteration, choose the input point that is farthest from any earlier center point to be the next center point.



The first six iterations of Gonzalez's k-center clustering algorithm.

In the pseudocode below, d_i denotes the current distance from point p_i to its nearest center, and r_j denotes the maximum of all d_i (or in other words, the cluster radius) after the first j centers have been chosen. The algorithm includes an extra iteration to compute the final clustering radius r_k (and the next center c_{k+1}).

[4]Teofilo F. Gonzalez. Clustering to minimize the maximum inter-cluster distance. *Theoretical Computer Science* 38:293-306, 1985.

```
GonzalezKCenter(P, k):
  for i ← 1 to n
      d_i ← ∞
  c_1 ← p_1
  for j ← 1 to k
      r_j ← 0
      for i ← 1 to n
          d_i ← min{d_i, |p_i c_j|}
          if r_j < d_i
              r_j ← d_i; c_{j+1} ← p_i
  return {c_1, c_2, ..., c_k}
```

GonzalezKCenter clearly runs in $O(nk)$ time. Using more advanced data structures, Tomas Feder and Daniel Greene[5] described an algorithm to compute exactly the same clustering in only $O(n \log k)$ time.

Theorem J.7. *GonzalezKCenter computes a 2-approximation to the optimal k-center clustering.*

Proof: Let OPT denote the optimal k-center clustering radius for P. For any index i, let c_i and r_i denote the ith center point and ith clustering radius computed by GonzalezKCenter.

By construction, each center point c_j has distance at least r_{j-1} from any center point c_i with $i < j$. Moreover, for any $i < j$, we have $r_i \geq r_j$. Thus, $|c_i c_j| \geq r_k$ for all indices i and j.

On the other hand, at least one cluster in the optimal clustering contains at least two of the points $c_1, c_2, \ldots, c_{k+1}$. Thus, by the triangle inequality, we must have $|c_i c_j| \leq 2 \cdot OPT$ for some indices i and j. We conclude that $r_k \leq 2 \cdot OPT$, as claimed. □

♥J.11 Approximation Schemes

With just a little more work, we can compute an arbitrarily close approximation of the optimal k-clustering, using a so-called ***approximation scheme***. An approximation scheme accepts both an instance of the problem and a value $\varepsilon > 0$ as input, and it computes a $(1 + \varepsilon)$-approximation of the optimal output for that instance. As I mentioned earlier, computing even a 1.8-approximation is NP-hard, so we cannot expect our approximation scheme to run in polynomial time; nevertheless, at least for small

[5]Tomas Feder* and Daniel H. Greene. Optimal algorithms for approximate clustering. *Proc. 20th STOC*, 1988. Unlike Gonzalez's algorithm, Feder and Greene's faster algorithm does not work over arbitrary metric spaces; it requires that the input points come from some $\mathbb{R}^d$ and that distances are measured in some L_p metric. The time analysis also assumes that the distance between any two points can be computed in $O(1)$ time.

values of k, the approximation scheme will be considerably more efficient than any exact algorithm.

Our approximation scheme works in three phases:

1. Compute a 2-approximate clustering of the input set P using GONZALEZKCENTER. Let r be the cost of this clustering.
2. Create a regular grid of squares of width $\delta = \varepsilon r/2\sqrt{2}$. Let Q be a subset of P containing one point from each non-empty cell of this grid.
3. Compute an *optimal* set of k centers for Q. Return these k centers as the approximate k-center clustering for P.

The first phase requires $O(nk)$ time. By our earlier analysis, we have $r^* \le r \le 2r^*$, where r^* is the optimal k-center clustering cost for P.

The second phase can be implemented in $O(n)$ time using a hash table, or in $O(n \log n)$ time by standard sorting, by associating approximate coordinates $(\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor)$ to each point $(x, y) \in P$ and removing duplicates. The key observation is that the resulting point set Q is significantly smaller than P. We know P can be covered by k balls of radius r^*, each of which touches $O(r^*/\delta^2) = O(1/\varepsilon^2)$ grid cells. It follows that $|Q| = O(k/\varepsilon^2)$.

Let $T(n, k)$ be the running time of an *exact* k-center clustering algorithm, given n points as input. If this were a computational geometry class, we might see a "brute force" algorithm that runs in time $T(n, k) = O(n^{k+2})$; the fastest algorithm currently known[6] runs in time $T(n, k) = n^{O(\sqrt{k})}$. If we use this algorithm, our third phase requires $(k/\varepsilon^2)^{O(\sqrt{k})}$ time.

It remains to show that the optimal clustering for Q implies a $(1+\varepsilon)$-approximation of the optimal clustering for P. Suppose the optimal clustering of Q consists of k balls $B_1, B_2, \ldots, B_k$, each of radius $\tilde{r}$. Clearly $\tilde{r} \le r^*$, since any set of k balls that cover P also cover any subset of P. Each point in $P \setminus Q$ shares a grid cell with some point in Q, and therefore is within distance $\delta\sqrt{2}$ of some point in Q. Thus, if we increase the radius of each ball B_i by $\delta\sqrt{2}$, the expanded balls must contain every point in P. We conclude that the optimal centers for Q gives us a k-center clustering for P of cost at most $r^* + \delta\sqrt{2} \le r^* + \varepsilon r/2 \le r^* + \varepsilon r^* = (1+\varepsilon)r^*$.

The total running time of the approximation scheme is $O(nk + (k/\varepsilon^2)^{O(\sqrt{k})})$. This is still exponential in the input size if k is large (say $\sqrt{n}$ or $n/100$), but if k and ε are fixed constants, the running time is linear in the number of input points.

♥J.12 An FPTAS for Subset Sum

An approximation scheme whose running time, for any fixed ε, is polynomial in n is called a ***polynomial-time approximation scheme*** or ***PTAS*** (usually pronounced *"pee*

[6] R. Z. Hwang, R. C. T. Lee, and R. C. Chan. The slab dividing approach to solve the Euclidean p-center problem. *Algorithmica* 9(1):1–22, 1993.

taz"). If in addition the running time depends only polynomially on ε, the algorithm is called a ***fully*** polynomial-time approximation scheme or *FPTAS* (usually pronounced *"eff pee taz"*). For example, an approximation scheme with running time $O(n^2/\varepsilon^2)$ is an FPTAS; an approximation scheme with running time $O(n^{1/\varepsilon^6})$ is a PTAS but not an FPTAS; and our approximation scheme for k-center clustering is not a PTAS.

The last problem we'll consider is the SUBSETSUM problem: Given a set X containing n positive integers and a target integer t, determine whether X has a subset whose elements sum to t. The lecture notes on NP-completeness include a proof that SUBSETSUM is NP-hard. As stated, this problem doesn't allow any sort of approximation—the answer is either TRUE or FALSE.[7] So we will consider a related optimization problem instead: Given set X and integer t, find the subset of X whose sum is as large as possible but no larger than t.

We have already seen a dynamic programming algorithm to solve the decision version SUBSETSUM in time $O(nt)$; a similar algorithm solves the optimization version in the same time bound. Here is a different algorithm, whose running time does not depend on t:

```
SUBSETSUM(X[1..n], t):
  S_0 ← {0}
  for i ← 1 to n
      S_i ← S_{i-1} ∪ (S_{i-1} + X[i])
      remove all elements of S_i bigger than t
  return max S_n
```

Here $S_{i-1} + X[i]$ denotes the set $\{s + X[i] \mid s \in S_{i-1}\}$. If we store each S_i in a sorted array, the ith iteration of the for-loop requires time $O(|S_{i-1}|)$. Each set S_i contains all possible subset sums for the first i elements of X; thus, S_i has at most 2^i elements. On the other hand, since every element of S_i is an integer between 0 and t, we also have $|S_i| \le t + 1$. It follows that the total running time of this algorithm is $\sum_{i=1}^{n} O(|S_{i-1}|) = O(\min\{2^n, nt\})$.

Of course, this is only an estimate of worst-case behavior. If several subsets of X have the same sum, the sets S_i will have fewer elements, and the algorithm will be faster. The key idea for finding an approximate solution quickly is to 'merge' nearby elements of S_i—if two subset sums are nearly equal, ignore one of them. On the one hand, merging similar subset sums will introduce some error into the output, but hopefully not too much. On the other hand, by reducing the size of the set of sums we need to maintain, we will make the algorithm faster, hopefully significantly so.

Here is our approximation algorithm. We make only two changes to the exact algorithm: an initial sorting phase and an extra FILTERing step inside the main loop.

[7] Do, or do not. There is no 'try'. (Are old one thousand when years you, alphabetical also in order talk will you.)

```
FILTER(Z[1..k], δ):
  SORT(Z)
  j ← 1
  Y[j] ← Z[i]
  for i ← 2 to k
      if Z[i] > (1 + δ) · Y[j]
          j ← j + 1
          Y[j] ← Z[i]
  return Y[1..j]
```

```
APPROXSUBSETSUM(X[1..n], k, ε):
  SORT(X)
  R_0 ← {0}
  for i ← 1 to n
      R_i ← R_{i−1} ∪ (R_{i−1} + X[i])
      R_i ← FILTER(R_i, ε/2n)
      remove all elements of R_i bigger than t
  return max R_n
```

Theorem J.8. *APPROXSUBSETSUM returns a $(1+\varepsilon)$-approximation of the optimal subset sum, given any ε such that $0 < \varepsilon \le 1$.*

Proof: The theorem follows from the following claim, which we prove by induction:

> For any element $s \in S_i$, there is an element $r \in R_i$ such that $r \le s \le r \cdot (1 + \varepsilon n/2)^i$.

The claim is trivial for $i = 0$. Let s be an arbitrary element of S_i, for some $i > 0$. There are two cases to consider: either $x \in S_{i-1}$, or $x \in S_{i-1} + x_i$.

(1) Suppose $s \in S_{i-1}$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \le s \le r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' \in R_i$, the claim obviously holds. On the other hand, if $r' \notin R_i$, there must be an element $r \in R_i$ such that $r < r' \le r(1 + \varepsilon n/2)$, which implies that

$$r < r' \le s \le r' \cdot (1 + \varepsilon n/2)^{i-1} \le r \cdot (1 + \varepsilon n/2)^i,$$

so the claim holds.

(2) Suppose $s \in S_{i-1} + x_i$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \le s - x_i \le r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' + x_i \in R_i$, the claim obviously holds. On the other hand, if $r' + x_i \notin R_i$, there must be an element $r \in R_i$ such that $r < r' + x_i \le r(1 + \varepsilon n/2)$, which implies that

$$\begin{aligned} r < r' + x_i \le s &\le r' \cdot (1 + \varepsilon n/2)^{i-1} + x_i \\ &\le (r - x_i) \cdot (1 + \varepsilon n/2)^i + x_i \\ &\le r \cdot (1 + \varepsilon n/2)^i - x_i \cdot ((1 + \varepsilon n/2)^i - 1) \\ &\le r \cdot (1 + \varepsilon n/2)^i. \end{aligned}$$

so the claim holds.

Now let $s^* = \max S_n$ and $r^* = \max R_n$. Clearly $r^* \le s^*$, since $R_n \subseteq S_n$. Our claim implies that there is some $r \in R_n$ such that $s^* \le r \cdot (1 + \varepsilon/2n)^n$. But r cannot be bigger than r^*, so $s^* \le r^* \cdot (1 + \varepsilon/2n)^n$. The inequalities $e^x \ge 1 + x$ for all x, and $e^x \le 2x + 1$ for all $0 \le x \le 1$, imply that $(1 + \varepsilon/2n)^n \le e^{\varepsilon/2} \le 1 + \varepsilon$. □

Theorem J.9. *ApproxSubsetSum runs in $O((n^3 \log n)/\varepsilon)$ time.*

Proof: Assuming we keep each set R_i in a sorted array, we can merge the two sorted arrays R_{i-1} and $R_{i-1} + x_i$ in $O(|R_{i-1}|)$ time. Filterin R_i and removing elements larger than t also requires only $O(|R_{i-1}|)$ time. Thus, the overall running time of our algorithm is $O(\sum_i |R_i|)$; to express this in terms of n and ε, we need to prove an upper bound on the size of each set R_i.

Let $\delta = \varepsilon/2n$. Because we consider the elements of X in increasing order, every element of R_i is between 0 and $i \cdot x_i$. In particular, every element of $R_{i-1} + x_i$ is between x_i and $i \cdot x_i$. After Filtering, at most one element $r \in R_i$ lies in the range $(1+\delta)^k \le r < (1+\delta)^{k+1}$, for any k. Thus, at most $\lceil \log_{1+\delta} i \rceil$ elements of $R_{i-1} + x_i$ survive the call to Filter. It follows that

$$\begin{aligned}
|R_i| &= |R_{i-1}| + \left\lceil \frac{\log i}{\log(1+\delta)} \right\rceil \\
&\le |R_{i-1}| + \left\lceil \frac{\log n}{\log(1+\delta)} \right\rceil && [i \le n] \\
&\le |R_{i-1}| + \left\lceil \frac{2 \ln n}{\delta} \right\rceil && [e^x \le 1 + 2x \text{ for all } 0 \le x \le 1] \\
&\le |R_{i-1}| + \left\lceil \frac{n \ln n}{\varepsilon} \right\rceil && [\delta = \varepsilon/2n]
\end{aligned}$$

Unrolling this recurrence into a summation gives us the upper bound $|R_i| \le i \cdot \lceil (n \ln n)/\varepsilon \rceil = O((n^2 \log n)/\varepsilon)$.

We conclude that the overall running time of ApproxSubsetSum is $O((n^3 \log n)/\varepsilon)$, as claimed. □

Exercises

1. (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.
   (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.
   (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.

2. (a) Find the smallest graph (minimum number of edges) for which GreedyVertexCover does not return the smallest vertex cover.
   (b) For any integer n, describe an n-vertex graph for which GreedyVertexCover returns a vertex cover of size $OPT \cdot \Omega(\log n)$.

3. (a) Find the smallest graph (minimum number of edges) for which DUMBVERTEXCOVER does not return the smallest vertex cover.
   (b) Describe an infinite family of graphs for which DUMBVERTEXCOVER returns a vertex cover of size $2 \cdot OPT$.

4. Let R be a set of rectangles in the plane, with horizontal and vertical edges. A *stabbing set* for R is a set S of points such that every rectangle in R contains at least one point in S. The *rectangle stabbing problem* asks for the smallest stabbing set of a given set of rectangles.
   (a) Prove that the rectangle stabbing problem is NP-hard.
   (b) Describe and analyze an efficient approximation algorithm for the rectangle stabbing problem. What is the approximation ratio of your algorithm?

5. Consider the following heuristic for constructing a vertex cover of a connected graph G: return the set of non-leaf nodes in any depth-first spanning tree of G.
   (a) Prove that this heuristic returns a vertex cover of G.
   (b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.
   (c) Describe an infinite family of graphs for which this heuristic returns a vertex cover of size $2 \cdot OPT$.

6. Suppose we have n pieces of candy with weights $W[1..n]$ (in ounces) that we want to load into boxes. Our goal is to load the candy into as many boxes as possible, so that each box contains at least L ounces of candy. Describe an efficient 2-approximation algorithm for this problem. Prove that the approximation ratio of your algorithm is 2. *[Hint: First consider the case where every piece of candy weighs less than L ounces.]*

7. Suppose we want to route a set of N calls on a telecommunications network that consists of a cycle of n nodes, indexed in cyclic order from 0 to $n-1$. Each call can be routed either clockwise or counterclockwise around the cycle from its source node to its destination node. Our goal is to route the calls so as to minimize the overall load on the network. The load L_i on any edge $(i, (i+1) \mod n)$ is the number of calls routed through that edge, and the overall load is $\max_i L_i$. Describe and analyze an efficient 2-approximation algorithm for this problem.

8. The *linear arrangement* problem asks, given an n-vertex directed graph as input, for an ordering $v_1, v_2, \ldots, v_n$ of the vertices that maximizes the number of forward edges: directed edges $v_i \to v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for this problem. (Solving this problem exactly is NP-hard.)

9. A *three-dimensional matching* in an undirected graph G is a collection of vertex-disjoint triangles (cycles of length 3) in G. A three-dimensional matching is *maximal* if it is not a proper subgraph of a larger three-dimensional matching in the same graph.

   (a) Let M and M' be two arbitrary maximal three-dimensional matchings in the same underlying graph G. Prove that $|M| \leq 3 \cdot |M'|$.

   (b) Finding the *largest* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

   (c) Finding the *smallest maximal* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

10. Consider the following optimization version of the PARTITION problem. Given a set X of positive integers, our task is to partition X into disjoint subsets A and B such that $\max\left\{\sum A, \sum B\right\}$ is as small as possible. Solving this problem exactly is clearly NP-hard.

    (a) Prove that the following algorithm yields a $(3/2)$-approximation.

```
GreedyPartition(X[1..n]):
  a ← 0
  b ← 0
  for i ← 1 to n
      if a < b
          a ← a + X[i]
      else
          b ← b + X[i]
  return max{a, b}
```

    (b) Prove that the approximation ratio $3/2$ cannot be improved, even if the input array X is sorted in *non-decreasing* order. In other words, give an example of a sorted array X where the output of GREEDYPARTITION is 50% larger than the cost of the optimal partition.

    (c) Prove that if the array X is sorted in *non-increasing* order, then GREEDYPARTITION achieves an approximation ratio of at most $4/3$.

    ♥(d) Prove that if the array X is sorted in *non-increasing* order, then GREEDYPARTITION achieves an approximation ratio of exactly $7/6$.

11. The *chromatic number* $\chi(G)$ of a graph G is the minimum number of colors required to color the vertices of the graph, so that every edge has endpoints with different colors. Computing the chromatic number exactly is NP-hard.

    Prove that the following problem is also NP-hard: Given an n-vertex graph G, return any integer between $\chi(G)$ and $\chi(G)+31337$. *[Note: This does not contradict the possibility of a constant* ***factor*** *approximation algorithm.]*

12. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.

    (a) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.

    (b) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we assign each vertex in G a *random* color from the set $\{\text{red}, \text{green}, \text{blue}\}$. Prove that the expected number of interesting edges is at least $\frac{2}{3}wow(G)$.

    (c) Prove that with high probability, the expected number of interesting edges is at least $\frac{1}{2}wow(G)$. *[Hint: Use Chebyshev's inequality. But wait... How do we know that we* ***can*** *use Chebyshev's inequality?]*

13. The KNAPSACK problem can be defined as follows. We are given a finite set X, each of whose elements has a non-negative *size* and a non-negative *value*, along with an integer *capacity* c. Our task is to determine the maximum total value among all subsets of X whose total size is at most c. This problem is NP-hard. Specifically, the optimization version of SUBSETSUM is a special case, where each element's value is equal to its size.

    Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```
APPROXKNAPSACK(X, c):
  return max{GREEDYKNAPSACK(X, c), PICKBESTONE(X, c)}
```

```
GREEDYKNAPSACK(X, c):
  Sort X in decreasing order by the ratio value/size
  S ← 0; V ← 0
  for i ← 1 to n
      if S + size(x_i) > c
          return V
      S ← S + size(x_i)
      V ← V + value(x_i)
  return V
```

```
PICKBESTONE(X, c):
  Sort X in increasing order by size
  V ← 0
  for i ← 1 to n
      if size(x_i) > c
          return V
      if value(x_i) > V
          V ← value(x_i)
  return V
```

14. In the *bin packing* problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is

NP-hard; this question asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NextFit(W[1..n]):
  b ← 0
  Total[0] ← ∞

  for i ← 1 to n
      if Total[b] + W[i] > 1
          b ← b + 1
          Total[b] ← W[i]
      else
          Total[b] ← Total[b] + W[i]

  return b
```

```
FirstFit(W[1..n]):
  b ← 0

  for i ← 1 to n
      j ← 1; found ← False

      while j ≤ b and found = False
          if Total[j] + W[i] ≤ 1
              Total[j] ← Total[j] + W[i]
              found ← True
          j ← j + 1

      if found = False
          b ← b + 1
          Total[b] = W[i]

  return b
```

(a) Prove that NextFit uses at most twice the optimal number of bins.

(b) Prove that FirstFit uses at most twice the optimal number of bins.

♥(c) Prove that if the weight array W is initially sorted in decreasing order, then FirstFit uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):

- In the packing computed by FirstFit, every item with weight more than $1/3$ is placed in one of the first OPT bins.
- FirstFit places at most $OPT - 1$ items outside the first OPT bins.

15. Given a graph G with edge weights and an integer k, suppose we wish to partition the vertices of G into k subsets $S_1, S_2, \ldots, S_k$ so that the sum of the weights of the edges that cross the partition (that is, have endpoints in different subsets) is as large as possible.

(a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem.

(b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.

16. The lecture notes describe a $(3/2)$-approximation algorithm for the metric traveling salesman problem, which asks for the minimum-cost Hamiltonian cycle in a graph whose edge weights satisfy the triangle inequality. Here, we consider computing minimum-cost Hamiltonian *paths*. Our input consists of a graph G whose edges have

weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.

(a) If our input includes zero endpoints, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path.

(b) If our input includes one endpoint u, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u.

♥(c) If our input includes two endpoints u and v, describe a $(5/3)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u and ends at v.

17. Suppose we are given a collection of n jobs to execute on a machine containing a row of p identical processors. The ***parallel scheduling problem*** asks us to schedule these jobs on these processors, given two arrays $T[1..n]$ and $P[1..n]$ as input, subject to the following constraints:

- When the ith job is executed, it occupies a contiguous interval of $P[i]$ processors for exactly $T[i]$ seconds.
- No processor works on more than one job at a time.

A valid schedule specifies a non-negative starting time and an interval of processors for each job that meets these constraints. Our goal is to compute a valid schedule with the smallest possible *makespan*, which is the earliest time when all jobs are complete.

(a) Prove that the parallel scheduling problem is NP-hard.

(b) Describe a polynomial-time algorithm that computes a 3-approximation of the minimum makespan of a given set of jobs. That is, if the minimum makespan is M, your algorithm should compute a schedule with make-span at most $3M$. You may assume that p is a power of 2. *[Hint: p is a power of 2.]*

(c) Describe an algorithm that computes a 3-approximation of the minimum makespan of a given set of jobs in $O(n \log n)$ time. Again, you may assume that n is a power of 2.

18. Consider the greedy algorithm for the metric traveling salesman problem: start at an arbitrary vertex u, and at each step, travel to the closest unvisited vertex.

(a) Prove that the approximation ratio for this algorithm is $O(\log n)$, where n is the number of vertices. *[Hint: Argue that the kth least expensive edge in the tour output by the greedy algorithm has weight at most* $\mathrm{OPT}/(n-k+1)$*; try $k=1$ and $k=2$ first.]*

♥(b) Prove that the approximation ratio for this algorithm is $\Omega(\log n)$. That is, describe an infinite family of weighted graphs such that the greedy algorithm returns a

Hamiltonian cycle whose weight is $\Omega(\log n)$ times the weight of the optimal TSP tour.