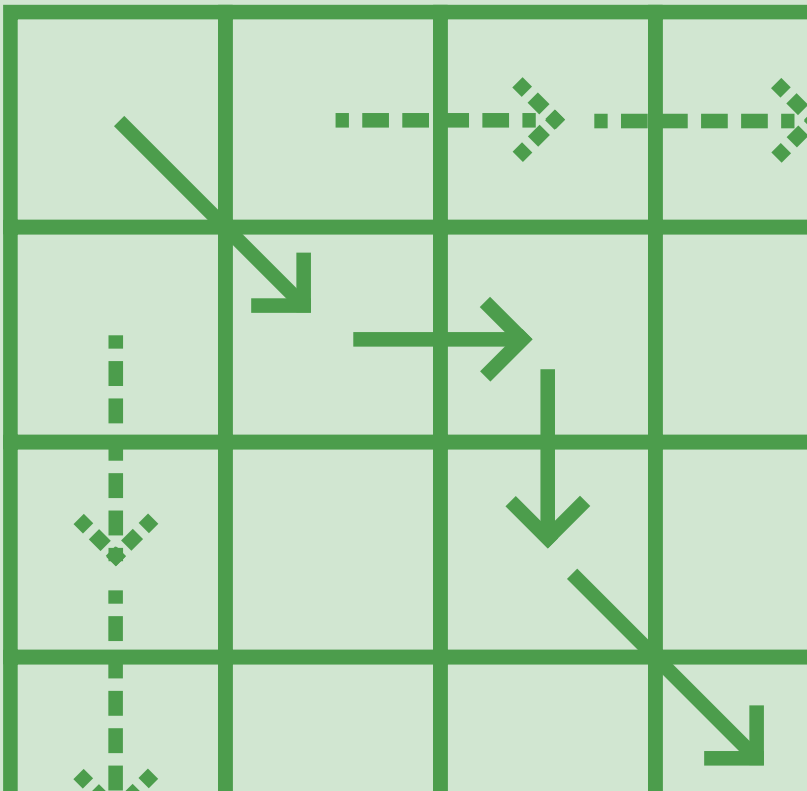
Algorithm Design and Analysis



The cover depicts a dynamic programming matrix used in the edit distance algorithm. Variations of this algorithm such as the Needleman-Wunsch and Smith-Waterman algorithms are frequently used in computational biology to align DNA or protein sequences.

TERRA-INCOGNITA.DEV

Contents

Divide and Conquer

Chapter 1 Divide and Conquer Primer

1.1 Divide and Conquer Paradigm

Most divide and conquer algorithms follow this paradigm.

- Divide up problem into several subproblems. Note that in some cases, we have to generalize the given problem.
- Recursively solving these subproblems.
- Combining the results from subproblems.

In this chapter and the following chapter, we will examine a few examples of divide-and-conquer algorithms, including but not limited to: maximum subarray, Strassen's matrix multiplication algorithm, quicksort, mergesort, median and selection in sorted array, fast Fourier transform (FTT), inversion counting, etc.

1.2 The Master Theorem

We have used the Master Theorem many times in both the introduction to theory of computation and data structure courses. We will once again present the theorem as it appears in CLRS here. For a careful proof of the theorem, see Section 4.6 of CLRS or Tutorial 7 note for CSC240 (Enriched Introduction to Theory of Computation).

Theorem 1.2.1 — The Master Theorem. Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a function. Let $T(n)$ on the nonnegative integers by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $\frac{n}{b}$ is interchangeable with $\lfloor n/b \rfloor$ and $\lceil n/b \rceil$. Then, $T(n)$ has the following asymptotic bounds:

- If $f(n) \in O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.
- If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \lg n)$.
- If $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$.

An equivalent formulation of the theorem is as follows.

Theorem 1.2.2 — The Master Theorem. Suppose that for $n \in \mathbb{Z}^+$.

$$T(n) = \begin{cases} c & \text{if } n < B \\ a_1 T(\lceil n/b \rceil) + a_2 T(\lfloor n/b \rfloor) + dn^i & \text{if } n \geq B \end{cases}$$

where $a_1, a_2, B, b \in \mathbb{N}$.

Let $a = a_1 + a_2 \geq 1$, $b > 1$, and $c, d, i \in \mathbb{R} \cup \{0\}$. Then,

$$T(n) \in \begin{cases} O(n^i \log n) & \text{if } a = b^i \\ O(n^i) & \text{if } a < b^i \\ O(n^{\log_b a}) & \text{if } a > b^i \end{cases}$$

1.3 Counting Inversions

The objective of array inversion problems is to find the number inversions in an unsorted array compared to a sorted array. That is, given an array A, how many pairs (i, j) are there such that $i < j$ and $A[i] > A[j]$. In other words, it counts the number of element-wise swaps needed in order to sort the given array.

For example, given the array $[8,4,2,1]$, the algorithm should answer 6 since the array has these six inversions: $(8,4)$, $(4,2)$, $(8,2)$, $(8,1)$, $(4,1)$, and $(2,1)$. If we follow these inversions, we will get a sorted array.

An naive algorithm for this problem is naturally to examine every pair of elements in the given array, which will take $O(n^2)$ comparisons. However, due to its resemblance to sorting, it is not hard to come up with a divide-and-conquer algorithm similar to mergesort that solves this problem in $O(n \log n)$ time.

At a high level, the algorithm should follow the aforementioned paradigm:

- Divide: split list into two halves A and B
- Conquer: recursively count inversions in each list
- Combine: count inversions (a, b) with $a \in A$ and $b \in B$
- Return the sum of the tree counts

For the combine step, we assume that the two subarrays A and B are sorted. Then, we can scan A and B from left to right in parallel and compare $A[i]$ and $B[j]$. If $A[i] < B[j]$, then $A[i]$ is not inverted with any element in B. If $A[i] > B[j]$ then $B[j]$ is inverted with every element in left in A.

A pseudocode for the algorithm is shown below. Equivalently, instead of explicitly splitting the array, we can perform this in place by passing around the indices p and q, as shown in Section 2.3.1 of CLRS.

```
SORT-AND-COUNT(L)
1  if L.length == 0
2       return (0, L)
3  mid = ⌊(1 + L.length)/2⌋
4  (count-a, A) = SORT-AND-COUNT(A[1...mid])
5  (count-b, B) = SORT-AND-COUNT(A[mid + 1...A.length])
6  (count-ab, L') = MERGE-AND-COUNT(A, B)
7  return (count-a + count-b + count-ab, L')
```

```
MERGE-AND-COUNT(A, B)
 1  count = 0
 2  i, j, k = 1
 3  L = []
 4  while i ≤ A.length and j ≤ B.length
 5       if A[i] ≤ B[j]
 6            L[k] = A[i]
 7            i = i + 1
 8       else
 9            count = count + 1
10            L[k] = B[j]
11            j = j + 1
12       k = k + 1
13  while i ≤ A.length
14       L[k] = A[i]
15       k = k + 1
16  while j ≤ B.length
17       L[k] = B[j]
18       k = k + 1
19  return (count, L)
```

The number of comparisons made by the algorithm is given by this recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{otherwise} \end{cases}$$

By Masters Theorem, $T(n) \in O(n \log n)$.

1.4 Closest Pair

Lemma 1.4.1 Let p be a point in the 2δ strip within δ distance horizontally. There are at most 7 points p such that $|y_p - y_q| \leq \delta$.

Proof. p must lie either in the left or right $\delta \times \delta$ square. Within each square, each point have distance at least k from each other. So, we can pack at most 4 such points into one square, and since we have a left square and right square, we have 8 points in total. Other than p, there are at most 7 points. ■

```
CLOSEST-PAIR(P = p_1, p_2, ..., p_n)
 1  compute a vertical line L such that half the points
    are on each side                                   // O(n log n)
    // consider sorting based on x-axis
 2  δ_1 = CLOSEST-PAIR(P_L)
 3  δ_2 = CLOSEST-PAIR(P_R)
 4  δ = min{δ_1, δ_2}
 5  for p in p_1, p_2, ..., p_n                        // O(n)
 6      if Y-DISTANCE(p, L)
 7          delete p
 8  sort remaining points by y-coordinate              // O(n log n)
 9  for p in p_1, p_2, ..., p_n                        // O(n)
10      for i from 1 to 7
11          p_i = ith neighbor of p
12          if DISTANCE(p, p_i) < δ
13              δ = DISTANCE(p, p_i)
14  return δ
```

The number of operations performed by this algorithm is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n \log n) & \text{otherwise} \end{cases}$$

By Masters Theorem, $T(n) \in O(n \log^2 n)$.

Theorem 1.4.2 — Lower Bound for Closest Pair. In a quadradic decision tree model, the closest pair problem requires at least $\Omega(n \log n)$ quadratic tests.

A quadradic decision tree is a comparison tree whose internal nodes are labeled with quadradic comparisons in the form of $x_i < x_j$ or $(x_i - x_k)^2 - (x_j - x_k)^2 < 0$. More generally, each internal node contains a comparison between a polynomial of degree at most 2 and 0 (2nd-order algebraic decision tree).

A hand-wavy proof of this lower bound follows by reduction from the result that the element uniqueness problem has a $\Omega(n \log n)$ lower bound, first shown by Ben-Or in *Lower Bounds For Algebraic Computation Trees*. Element distinctness reduces to 2D closest pair problem.

1.5 Karatsuba's Integer Multiplication Algorithm

To multiply two n-bit integers a and b, we can follow this recursive procedure:

- add two $\frac{1}{2}n$ bit integers
- multiply three $\frac{1}{2}n$-bit integers recursively
- add, subtract, and shift to obtain the result

$$\begin{aligned} a &= 2^{n/2}a_1 + a_0 \\ b &= 2^{n/2}b_1 + b_0 \\ ab &= 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + a_0 b_1) + a_0 b_0 \\ &= 2^n \underbrace{a_1 b_1} + 2^{n/2}(\underbrace{(a_1 + a_0)(b_1 + b_0)} - \underbrace{a_1 b_1} - \underbrace{a_0 b_0}) + \underbrace{a_0 b_0} \end{aligned}$$

The recursive steps are labeled with underbrace. By Masters Theorem, Karatsuba's algorithm performs

$$T(n) = 3T(\lceil n/2 \rceil) + cn + d \in \Theta(n^{\log_2 3})$$

bit-wise operations.

Chapter 2 Strassen's Algorithm

2.1 Matrix Multiplication

The standard method to multiply two $n \times n$ matrices requires $O(n^3)$ scalar operations (multiplication and addition). The standard algorithm follows from the definition of matrix multiplication that the i, j entry of $C = A \cdot B$ is given by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

SQUARE-MATRIX-MULTIPLY(A, B)

```
1  n = A.rows
2  C = n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ij b_ij
8  return C
```

Clearly, this algorithm runs in $\Theta(n^3)$.

2.2 A Recursive Approach

Without loss of generality, let $n = 2^k$ for some k. Then, we can divide any $n \times n$ matrix into four $n/2 \times n/2$ matrices.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Since matrices are a non-commutative ring, $n \times n$ matrix multiplication can be realized in $7n/2 \times n/2$ matrix multiplications and 18 matrix additions. Matrix addition only takes $O(n^2)$ scalar operations. Then, we have the following recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(n/2) + O(n^2) & \text{otherwise} \end{cases}$$

which implies that $T(n) \in O(n^{\log_2 7})$.

An implementation of this recursive approach is presented in pseudocode below

```
SQUARE-MATRIX-MULTIPLY(A, B)
 1  n = A.rows
 2  C = n × n matrix
 3  if n == 1
 4  else
 5      partition A, B, and C each into four sub-matrices
 6      C11 = SQUARE-MATRIX-MULTIPLY(A11, B11)+
               SQUARE-MATRIX-MULTIPLY(A12, B21)
 7      C12 = SQUARE-MATRIX-MULTIPLY(A11, B12)+
               SQUARE-MATRIX-MULTIPLY(A12, B22)
 8      C21 = SQUARE-MATRIX-MULTIPLY(A21, B11)+
               SQUARE-MATRIX-MULTIPLY(A22, B21)
 9      C22 = SQUARE-MATRIX-MULTIPLY(A21, B12)+
               SQUARE-MATRIX-MULTIPLY(A22, B22)
10  return C
```

However, this particular implementation of the recursive approach still has a $O(n^3)$ running time, as demonstrated by solving the recurrence by Masters Theorem.

2.3 Strassen's Algorithm

Strassen's algorithm is a rather peculiar algorithm that multiplies two square matrices using $O(n^{\log_2 7}) = O(n^{2.81})$ scalar operations. It is debatable how practical asymptotically faster matrix multiplication algorithms (including Strassen's algorithm and Coppersmith-Winograd algorithm) are given that their crossover point (the threshold on input size beyond which such algorithms become actually faster) is often quite large for them to be useful in practical applications.

Strassen's algorithm as described in CLRS works as follows:

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices. This takes $\Theta(1)$ operations.
2. Create 10 submatrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in Step 1. This can be done in $\Theta(n^2)$.
3. Using the submatrices in Step 1 and the 10 matrices in Step 2, recursively compute seven matrix products $P_1, P_2, \ldots, P_7$. Each matrix P_i is $n/2 \times n/2$.
4. Compute the desired $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices. This can also be done in $\Theta(n^2)$ time.

More specifically, in Step 2,

$$\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - B_{21} \\
S_{10} &= B_{11} + B_{12},
\end{aligned}$$

and in Step 3,

$$\begin{aligned}
P_1 = A_{11} \cdot S_1 &= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 = S_2 \cdot B_{22} &= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 = S_3 \cdot B_{11} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 = A_{22} \cdot S_4 &= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 = S_5 \cdot S_6 &= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 = S_7 \cdot S_8 &= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 = S_9 \cdot S_{10} &= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}.
\end{aligned}$$

In Step 4,

$$\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}$$

Chapter 3 Fast Fourier Transform

3.1 Motivation

A polynomial can be represented in various ways. Each representation has its advantages and downsides.

In general, a polynomial $A(x)$ of degree $n-1$ can be written as

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ &= \sum_{k=0}^{n-1} a_k x^k \end{aligned}$$

3.1.1 Coefficient Vector

For a polynomial $A(x)$ of degree $n-1$, the coefficient vector contains the coefficient of the polynomial represented as a vector.

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \quad \leftrightarrow \quad \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle$$

3.1.2 Roots

Equivalently,

$$A(x) = (x - r_0)(x - r_1) \cdots (x - r_{n-1}) \cdot c$$

where $r_0, r_1, \ldots, r_{n-1}$ are the n roots of the polynomial and c is a scale term.

3.1.3 Samples

Given a polynomial, we can take n points (coordinates):

$$(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$$

where $A(x_i) = y_i$ for all $i \in \{0, 1, \ldots, n-1\}$. This representation is also known as the point-value representation.

By the Foundamental Theorem of Algebra, these samples uniquely define a polynomial of degree $n-1$.

Theorem 3.1.1 — The Fundamental Theorem of Algebra. A univariate polynomial of degree n with complex coefficients has exactly n complex roots.

Corollary 3.1.2 — Uniqueness of Polynomial Interpolation. A degree $n-1$ univariate polynomial $A(x)$ is uniquely defined by its evaluation at n distinct values of x.

3.2 Operations on Polynomials

There are three primary operations on polynomials: evaluation, addition, and multiplication. The complexity of these operations varies depending on the representation. There is not a single representation that is efficient for all operations, so we need a way to efficiently convert between the representations.

3.2.1 Coefficient Representation

Evaluation

Evaluation is efficient when the polynomial is represented as the coefficient vector using the Horner's rule:

$$A(x_0) = a_0 + x_0(a_0 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\cdots))$$

which can be written in pseudocode as follows

EVALUATE$(A = \langle a_0, \ldots a_{n-1}\rangle, x)$
1 $y = 0$
2 **for** $i = n-1$ **to** 0
3 $\quad y = a_j + (x \cdot y)$
4 **return** y

Addition

Addition is easy in coefficient representation. We simply add each coefficient to get a new coefficient vector.

ADD$(A = \langle a_0 \ldots a_{n-1}\rangle, B = \langle b_0 \ldots b_{n-1}\rangle)$
1 **for** $j = 0$ **to** $n-1$
2 $\quad c_j = a_j + b_j$
3 **return** $C = \langle c_0, \ldots, c_{n-1}\rangle$

Multiplication

Multiplication is a little bit trickier in coefficient representation. We need to use linear convolution, which takes $O(n^2)$ operations.

$$A(x) \times B(x) = \sum_{j=0}^{2n-2} c_j x^j = \sum_{j=0}^{2n-2} \sum_{k=0}^{j} a_k b_{j-k} x^j$$

MULTIPLY$(A = \langle a_0 \dots a_n \rangle, B = \langle b_0 \dots b_m \rangle)$

```
1  for j = 0 to n + m
2      c_j = 0
3  for j = 0 to n
4      for k = 0 to n
5          c_{j+k} = c_{j+k} + a_j · b_k
6  C = ⟨c_0 ... c_{n+m}⟩
```

3.2.2 Samples

Evaluation

Evaluation can be done in $O(n^2)$ through interpolation using Lagrange's formula.

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)}$$

Addition

Addition is also easy in samples representation.

$$A(x) + B(x) : \ (x_0, y_0 + z_0), (x_1, y_1 + z_1), \dots (x_{n-1}, y_{n-1} + z_{n-1})$$

given that $A(x)$ is represented as $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ and
$B(x)$ is represented as $(x_0, z_0), \dots, (x_{n-1}, z_{n-1})$.

Multiplication

Multiplication is done in a similar fashion as addition.

$$A(x) \times B(x) : \ (x_0, y_0 z_0), (x_1, y_1 z_1), \dots (x_{n-1}, y_{n-1} z_{n-1})$$

given that $A(x)$ is represented as $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ and
$B(x)$ is represented as $(x_0, z_0), \dots, (x_{n-1}, z_{n-1})$.

3.2.3 Roots

Evaluation

Takes $O(n)$ operations by substitution and multiplication.

Addition

Impossible in the general case. There is not a general formula that can converts a coefficient vector back to the roots.

Multiplication

Multiplying two polynomials represented by their roots is equivalent to concatenate the list of roots since the resulting polynomial will have the the same roots of both polynomials.

3.2.4 Summary

	Coefficient	Roots	Samples
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

Table 3.1: Comparison between different representations of polynomial

The following chart outlines the conversions required for efficient multiplication of polynomials. The figure is taken from CLRS pp. 904.

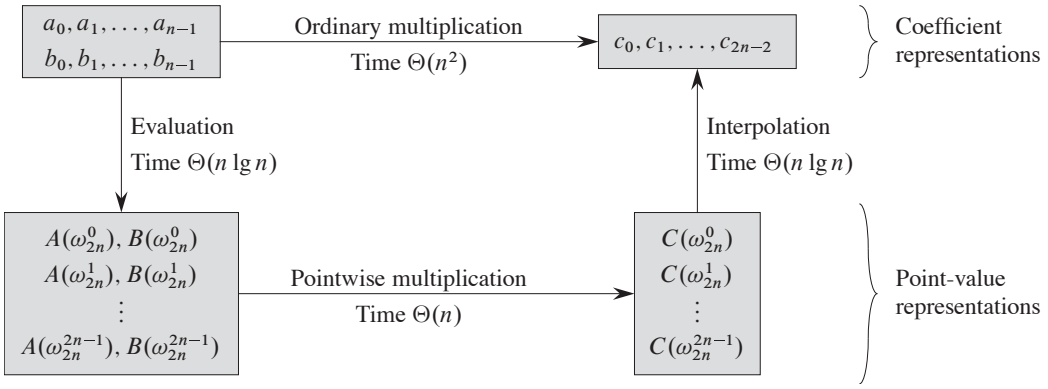


Figure 3.1: Graphical outline of an efficient polynomial multiplication procedure.

3.3 Fast Fourier Transform

3.3.1 Coefficient to Samples (and back)

Conversion from coefficient to samples is similar to evaluation. We fix a set of points $x_0, x_1, \ldots, x_{n-1}$ and evaluate the polynomial at these points. This evaluation is equivalent to the following matrix multiplication

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

where $\mathbf{V}$ is called the Vandermonde matrix with entries $V_{jk} = x_j^k$ and $\mathbf{a}$ is the coefficient vector of the polynomial $A(x)$.

Evaluating this matrix product takes $\Theta(n^2)$ scalar operations. Similarly, to convert from samples back to polynomial (interpolation), we can compute $\mathbf{V}^{-1}$ using Gaussian elimination with $O(n^3)$ operations, and computing $\mathbf{a} = \mathbf{V}^{-1}\mathbf{y}$ (it is rarely a good idea to invert any matrices; "Don't invert that matrix!" – John Cook).

In order to obtain efficient algorithms for manipulating polynomials, we need to somehow get rid of the $O(n^2)$ overhead resulted from the interconversion between coefficient and samples.

Note that we simply said "fix $x_0 \ldots x_{n-1}$" when constructing the Vandermonde matrix without putting any constraints on what values of x to take. By choosing special values for x, we can reduce the conversion overhead to $O(n \log n)$.

3.3.2 Divide and Conquer

We can formulate the evaluation $\mathbf{y} = \mathbf{V}\mathbf{a}$ as a divide and conquer algorithm.

We can come up with an outline of the algorithm by following the divide-and-conquer paradigm

1. Divide: divide the polynomial A into even and odd coefficients (this is equivalent to divide the coefficient vector $\mathbf{a}$ into even and odd entries)

$$A_{even}(x) = \sum_{k=0}^{\lceil \frac{n}{2}-1 \rceil} a_{2k} x^k \quad \leftrightarrow \quad \mathbf{a}_{even} = \langle a_0, a_2, a_4, \ldots \rangle$$

$$A_{odd}(x) = \sum_{k=0}^{\lfloor \frac{n}{2}-1 \rfloor} a_{2k+1} x^k \quad \leftrightarrow \quad \mathbf{a}_{odd} = \langle a_0, a_2, a_4, \ldots \rangle$$

   Note that the degree of the two resulting polynomials is half of the original polynomial.
2. Conquer: recursive conquer $A_{even}(x)$ and $A_{odd}(x)$ for $x \in X^2$ where $X^2 = \{x^2 \mid x \in X\}$.

3. Combine: combine the terms as follows

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2)$$

for $x \in \mathbf{x}$.

It is obvious that the degree of the polynomial n halves, and hence the recurrence

$$T(n,|X|) = 2T\left(\frac{n}{2},|X|\right) + O(n+|X|) \in O(n^2).$$

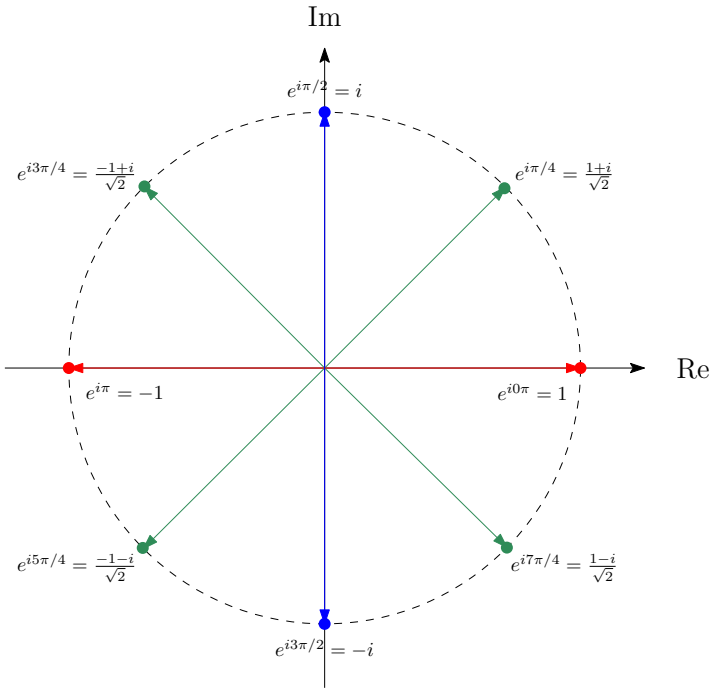
This is no better than the naive approach. The main issue is that we are not halving the size of X. Ideally, we want X to be recursively collapsing.

3.3.3 Complex Roots of Unity

We can construct a collapsing set of x's via square roots. If we only look at real numbers, taking the square root or squaring a number won't give you fewer or more numbers, but if we broaden our view to complex numbers, we notice that starting from 1, every time we take the square root, the size of the set doubles. The nth root of 1 is called the ***complex nth root of unity***. The observation implies that if we start from the nth root of unity and square the elements in the set, the set will collapse every time we square it.

Example: $\{1\} \to \{1,-1\} \to \{1,-1,i,-i\} \to \{1,-1,i,-i,\pm\frac{\sqrt{2}}{2}(1+i),\pm\frac{\sqrt{2}}{2}(-1+i)\}$

The complex roots of unity are spaced equally around the unit circle centered at the origin of the complex plane. These points are of the form $\cos\theta + i\sin\theta = e^{i\theta}$ for $\theta = 0, \frac{2}{n}\pi, \frac{4}{n}\pi, \ldots, \frac{2(n-1)}{n}\pi$.



The nth roots of unity where $n = 2^\ell$ for some integer ℓ form a collapsing set since $(e^{i\theta})^2 = e^{i2\theta} = e^{i(2\theta \bmod 2\pi)}$.

Let us formalize this idea and prove that it works.

> **Lemma 3.3.1 — Halving (Collapsing) Lemma.** If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.

Proof. We know that $\left(e^{\frac{2\pi i}{n}}\right)^{2k} = e^{\frac{4k\pi i}{n}} = \left(e^{\frac{2\pi i}{1/2\cdot n}}\right)^k$ for any nonnegative integer k.

Furthermore, $\left(e^{\frac{2\pi i}{n}}\right)^{2(k+n/2)} = \left(e^{\frac{2\pi i}{n}}\right)^k$. This implies that for every pair of k and $k+n/2$ share the same square, and that if take the square of every element in the set of nth roots of unity, we get the $n/2$ elements and it follows from algebra that the resulting set contains the $(n/2)$th roots of unity. ■

3.3.4 The FFT Algorithm

By choosing the x's in the Vandermonde matrix to be the nth roots of unity, we can make X collapsing along with n. The rest of the algorithm is the same as the divide-and-conquer approach described earlier.

FFT-RECURSIVE(a)
1 $n = a.length$
2 **if** n == 1
3 **return** a
4 $\omega_n = e^{2\pi i/n}$
5 $\omega = 1$
6 $a_{even} = (a_0, a_2, \ldots, a_{n-2})$
7 $a_{odd} = (a_1, a_3, \ldots, a_{n-1})$
8 $y_{even} =$ FFT-RECURSIVE(a_{even})
9 $y_{odd} =$ FFT-RECURSIVE(a_{odd})
10 **for** $k = 0$ to $n/2 - 1$
11 $y_k = y_{even}[k] + \omega y_{odd}[k]$
12 $y_{k+\frac{n}{2}} = y_{even}[k] - \omega y_{odd}[k]$
13 $\omega = \omega\omega_n$
14 **return** $y = (y_0, \ldots, y_{n-1})$

This algorithm takes $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \in O(n \log n)$ operations.

3.4 Inverse Fast Fourier Transform

The inverse discrete fourier transform is an algorithm used to find the coefficients fo a polynomial given a set of samples. The transformation is of the form $\mathbf{a}^* \to \mathbf{V}^{-1}\mathbf{a}^* = \mathbf{a}$. To evaluate this, we need $\mathbf{V}^{-1}$. In fact, this matrix has a very useful property that allows to find it without actually inverting $\mathbf{V}$ (again, don't invert the matrix).

Lemma 3.4.1 Let $\mathbf{V}$ be the Vandermonde matrix constructed from the set of nth roots of unity. Let $\overline{\mathbf{V}}$ denote the complex conjugate of $\mathbf{V}$. Then,

$$\mathbf{V}^{-1} = \frac{1}{n}\overline{\mathbf{V}}$$

Proof. We claim that $\mathbf{P} = \mathbf{V}\overline{\mathbf{V}} = n\mathbf{I}$.

Let p_{jk} be the j,kth entry of $\mathbf{P}$.

$$\begin{aligned} p_{jk} &= (\text{row } j \text{ of } \mathbf{V}) \cdot (\text{column } k \text{ of } \overline{\mathbf{V}}) \\ &= \sum_{m=0}^{n-1} e^{ij2\pi m/n}\overline{e^{ik2\pi m/n}} \\ &= \sum_{m=0}^{n-1} e^{ij2\pi m/n}e^{-ik2\pi m/n} \\ &= \sum_{m=0}^{n-1} e^{i(j-k)2\pi m/n} \end{aligned}$$

Take $j = k$. Then, $p_{jk} = \sum_{m=0}^{n-1} 1 = n$. Otherwise, $p_{jk} = 0$. This means that the diagonal entries are n, and the entires off the diagonal are all 0. Thus, the claim is true.

It follows that $\mathbf{V}^{-1} = 1/n\mathbf{V}$. ■

This lemma implies that for the Inverse Fast Fourier Transform algorithm, we can simply replace $e^{ikr/n}$ with its complex conjugate in the FFT algorithm and divide the final result by 2. The rest of the algorithm is analogous to that for FFT, and it is easy to see that the running of IFFT is also in $O(n\log n)$.

Quite surprisingly, many sources online including lecture notes from numerous institutions and prominant YouTube channels appear to have made some mistakes when presenting the pseudocode for IFFT. For a recursive implementation, the division by n is applied only once at the very end when all recursive calls have terminated, not during the recursive call. It is not correct to simply change the twiddle factor to $w_n = e^{-2\pi i/n}/n$ or to divide by n every time we update a like $a_k = \left(a_{even}[k] + \omega a_{odd}[k]\right)/n$. However, it is correct to divide by 2 every time we update a, as shown below. It is the same implementation as the one in Jeff Erickson's notes.

IFFT-RECURSIVE(y)

```
1  n = y.length
2  if n == 1
3      return y
4  ω_n = e^{-2πi/n}
5  ω = 1
6  y_even = (y_0, y_2, ..., y_{n-2})
7  y_odd = (y_1, y_3, ..., y_{n-1})
8  a_even = IFFT-RECURSIVE(y_even)
9  a_odd = IFFT-RECURSIVE(y_odd)
10 for k = 0 to n/2 - 1
11     a_k = (a_even[k] + ω a_odd[k]) / 2
12     a_{k+n/2} = (a_even[k] - ω a_odd[k]) / 2
13     ω = ω ω_n
14 return a = (a_0, ..., a_{n-1})
```

3.5 Fast Polynomial Multiplication

With FFT, we can implement the procedure outlined in Figure 3.1.

To calculate $A(x) \times B(x)$,

1. Compute $A^* = \text{FFT}(A)$ and $B^* = \text{FFT}(B)$. This step is $O(n \log n)$.
2. Compute $C^* = A^* \cdot B^*$ in sample representation. This step is $O(n)$.
3. Compute $C = \text{IFFT}(C^*)$ to get C in coefficient representation. This step is $O(n \log n)$.

Overall, the algorithm has runtime complexity of $O(n \log n)$.

Chapter 4 Selection in Linear Time

II Greedy Algorithms

Chapter 5 Interval Scheduling

5.1 Interval Scheduling

We begin by considering the interval scheduling problem. The same problem is referred to as the activity selection problem in CLRS.

Consider a set $S = \{a_1, a_2, \ldots, a_n\}$ of jobs/activities, each with a start time s_i and finish time f_i. Two jobs are said to be compatible if they don't overlap. More formally, given activities a_i and a_j, they are compatible if $[s_i, f_i) \cap [s_j, f_j) = \emptyset$. That is, if $s_i \geq f_j$ or $s_j \geq f_i$. The goal of the interval scheduling problem is to find a maximum-size subset of mutually compatible jobs.

Let us consider the greedy strategy for solving the interval scheduling problem. Intuitive, the globally optimal solution should leave resources/time open for as many other jobs as possible. This requires us to consider the jobs in some "natural" order:

- Earliest start time: consider jobs in ascending order of s_i
- Earliest finish time: consider jobs in ascending order of f_i
- Shortest inverval: consider jobs in ascending order of $f_i - s_i$
- Fewest conflicts: for each job a_i, count the remaining number of conflicting jobs c_i and schedule in ascending order of c_i.

Not all of those strategies work. Here are some counterexamples (Figure 5.1).

Therefore, we choose earliest finish time as our greedy strategy, which we can implement into this recursive algorithm

```
INTERVAL-SCHEDULING-RECURSIVE(S, k, n)
1  m = k + 1
2  while m ≤ n and S[m].s < S[k].f
3       m = m + 1
4  if m ≤ n
5       return {S[m]} ∪ INTERVAL-SCHEDULING-RECURSIVE(S, m, n)
6  else
7       return ∅
```

As a precondition, we assume that the array of jobs S is sorted in monotonically increasing order by finish time. $S[i].s$ denotes the starting time of the ith job, and $S[i].f$ denotes the finish time of the ith

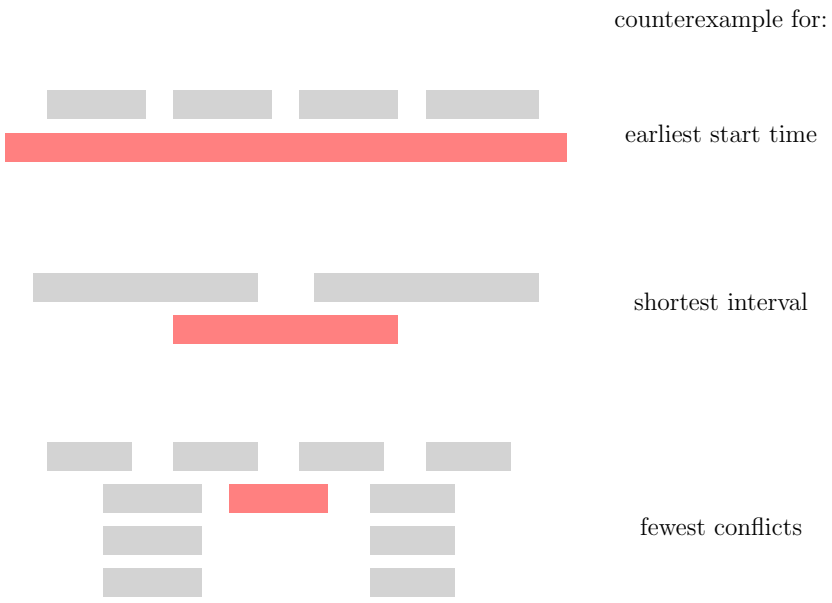


Figure 5.1: Counterexamples for the greedy strategies that do not work for the interval scheduling problem. The interval that satisfies the greedy strategy but leads to incorrect global solution is highlighted in red.

job. k is the job being considered. In each call to the recursive algorithm, we starts from $m = k+1$ and increment m until we find a job with starting time $S[m].s$ strictly lower than the finish time of $S[k]$. This is the job that is selected by our greedy strategy in the current recursive call. After finding this m, we return the union of $\{S[m] = a_m\}$ and the maximum-size subset of S_m returned by the recursive call INTERVAL-SCHEDULING-RECURSIVE(S, m, n).

The initial call that solves the problem globally is INTERVAL-SCHEDULING-RECURSIVE$(S, 0, n)$ where n is the number of jobs to be considered.

This algorithm can be easily converted into an iterative algorithm.

INTERVAL-SCHEDULING(S)

```
1  n = S.length
2  S' = {a_1}
3  k = 1
4  for m = 2 to n
5      if S[m].s ≥ S[k].f
6          S' = S' ∪ {S[m]}
7          k = m
8  return S'
```

Sorting the array of jobs takes $O(n \log n)$ time, and going over the sorted the list and check each job's compatibility takes $O(n)$ in total.

5.2 Proving Optimality

As we have demonstrated earlier in Figure 5.1, greedy strategy does not always work. It is important for us to ensure (prove) that the locally optimal greedy choice leads to a globally optimal solution. Here, we will present three equally valid proofs for the optimality of the greedy algorithm for the interval scheduling problem.

Theorem 5.2.1 The greedy algorithm using earliest finish time is optimal. More formally, the algorithm returns a maximum-size set of disjoint jobs $\{a_1, \ldots, a_m\} \subseteq S$

Proof (by contradiction). Suppose, for contradiction, that the greedy algorithm using earlies finish time is not optimal. Let $i_1, i_2, \ldots, i_k$ be the sequence of jobs selected by the algorithm, and let $j_1, j_2, \ldots, j_m$ be the correct solution where $m > k$. Let r be the largest possible value such that $i_{r+1} \neq j_{r+1}$. That is, the $(r+1)$th job is the first job where the two sequences begin to differ.

Both i_{r+1} and j_{r+1} must be compatible with previous choices of i and j, respectively. Let $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r, i_{r+1}, j_{r+2}, \ldots, j_m$ be a new sequence of jobs. This is the optimal solution with j_{r+1} replaced with i_{r+1}. By assumption, the greedy solution is not optimal so j_{r+1} is not selected by the greedy algorithm and thus $i_{r+1} \leq j_{r+1}$. So, the new sequence of jobs is still feasible because $i_{r+1} \leq j_{r+1}$. The algorithm is still optimal because all m disjoint jobs are scheduled. But then, i_{r+1} is not different from j_{r+1}, which implies that r is not the maximum value such that $i_{r+1} \neq j_{r+1}$. This is a contradiction. Therefore, the greedy algorithm is optimal. ■

Proof (by induction). Let S_k be the subset of jobs selected by the greedy algorithm after considering the first k jobs in increasing order of finish time. We say that $S_k \subseteq S$ is promising if it can be extended to the optimal solution (S_k is part of the optimal solution). More formally, S_k is promising if there exists $T \subseteq \{a_{k+1}, \ldots, a_n\}$ such that $O_k = S_k \cup T$ is optimal.

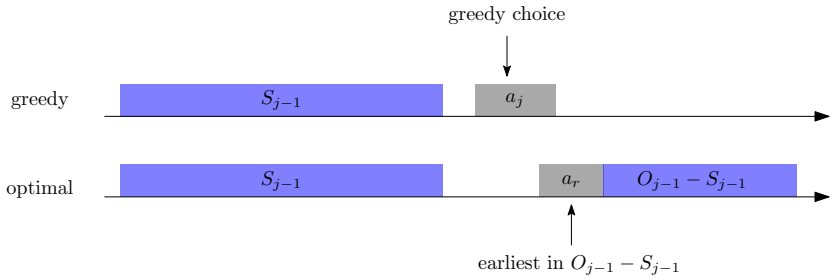
We want to show that for all $k \in \{0, 1, \ldots, n\}$, S_k is promising.

Base case: When $k = 0$, $S_0 = \emptyset$. The claim is vacuously true.

Inductive step: Let $j \in \mathbb{N}$ be arbitrary. Suppose that the claim holds for $k = j - 1$ and that S_{j-1} is promising. For S_j, there are two possibilities:

(1) The greedy algorithm did not select job a_j (i.e. $s_{j+1} < f_j$), so $S_j = S_{j-1}$. This implies that a_j is not compatible with some job in S_{j-1}. Since $S_{j-1} \subseteq O_{j-1}$, O_{j-1} does not include a_j. $O_j = O_{j-1}$ does not include a_j, so S_j can be extended to O_j and S_j is promising.

(2) The greedy algorithm selected job a_j (i.e. $s_{j+1} \geq f_j$), so $S_j = S_{j-1} \cup \{a_j\}$. Consider the earliest job a_r in $O_{j-1} - S_{j-1}$. Consider $O_j = O_{j-1} - \{a_r\} \cup \{a_j\}$. O_j is still feasible since jobs in O_{j-1} are disjoint, a_r is the first to finish, and the finish time of a_j is earlier or equal to the finish time of a_r. Hence, S_j can be extended to O_j.



By induction, the claim is true for all $k \in \{0, 1, \ldots, n\}$. ■

The previous two proofs use the technique known as the exchange argument. They both rely on the argument that if the greedy solution after j iterations can be extended to an optimal solution, then the greedy solution after $j+1$ iterations can also be extended to an optimal solution by exchanging some interval with one chosen based on the greedy strategy.

The last proof uses what we call an "***charging argument***". In this case, the charging argument charges each interval of an optimal solution (or arbitrary solution) to a unique interval in the greedy solution. Charging arguments are also used in approximation algorithms (for example, to prove that an algorithm is a k-approximation).

Proof (by charging argument). Given a set of jobs $S = \{a_1, \ldots, a_n\}$, let O be an optimal solution of the interval scheduling problem. Let S' be the solution given by the greedy algorithm using earliest finish time. We want to find a one-to-one function $h : O \to S'$. For any job $j \in O$, define $h(j)$ as the interval $j' \in S'$ that intersects j and has the earliest finish time amongst intervals in S' intersecting j.

We claim that the h is well-defined so $h(j)$ must exists for all j. This is proved by contradiction. Suppose, for contradiction, that there exists a $j \in O$ where $h(j)$ as we defined does not exist. By definition, this means no interval in S' intersects with j. This implies that j is compatible with every interval in S', so the greedy strategy would have selected j and j would be part of S'. But then if $j \in S'$, j intersects with itself. This is a contradiction. $h(j)$ **exists** for all $j \in O$. Furthermore, $h(j)$ is **unique** because all intervals in S' are mutually disjoint, and every interval in S' that intersects j have distinct finish time.

We then show that h is **injective**, similarly by contradiction. Assume that there are two intervals $j_1, j_2 \in O$ such that $h(j_1) = h(j_2) = j' \in S'$. Without loss of generality, suppose that the finish time of j_1 is earlier than j_2. j_1 and j_2 are disjoint because they are both in O, which implies that $f_1 \leq s_2 < f_2$. Since $j' \in S'$, the greedy algorithm must have encountered j' before j_1 and j_2. Thus, $f' \leq f_1$. But then, this implies that $f' \leq f_1 \leq s_2 < f_2$. That is, j' and j_2 do not overlap. This is a contradiction because if they do not intersect, $h(j_2) \neq j'$. Therefore, h is injective.

Since there exists an one-to-one function $h : O \to S'$, by the charging argument, the greedy algorithm for the interval scheduling problem is optimal. ■

More generally, we have shown that $|O| \leq |S'|$, which implies the optimality of the algorithm (that the

algorithm returns the maximum-size subset).

5.3 Interval Coloring (Interval Partition)

Let us know consider a modified version of the original interval scheduling problem. Suppose that we are given a set of intervals. We want to color all intervals so that intervals with the same color do not intersect while using the minimum number of colors. This problem is also known as the interval partitioning problem.

Similar to interval scheduling, let's take a look at the few possible choices for our greedy strategy:

- Earliest start time
- Earliest finish time
- Shortest interval
- Fewest conflicts

We can show using counterexamples that the last three heuristics, earliest finish time, shortest interval, and fewest conflicts, do not work. We will prove that the earliest start time greedy choice gives us a globally optimal solution for the interval partitioning problem.

The proof for this is somewhat similar to the charging argument that we used to prove the optimality of interval scheduling. We attempt to bound the size of the solution set in order to show that it is optimal (miminal). In the charging argument, we bound the size by showing that there exists an one-to-one function. In this proof, we will skip that part and instead bound the size directly.

Lemma 5.3.1 Given a set of intervals, let d be the maximum number of intersecting intervals at any time. The number of partitions (colors) given by any algorithms that solve the interval partitioning problem must be at least d.

Proof. By contradiction. ■

Lemma 5.3.2 Let d be defined as in the previous lemma. The greedy algorithm using earliest start time produces a solution with at most d partitions.

Proof. Let d' be the number of partitions produced by the greedy algorithm. Suppose for contradiction that the algorithm used more than d partitions. Consider the first time that the greedy algorithm used $d+1$ partitions. Suppose this happens when the algorithm is trying to assign a partition to the interval j. This implies that there are d intervals intersecting j. Let s_j be the starting time of j. These d intervals must contain s_j. This is because all the previous d intervals have starting time earlier than s_j. So it must be that case that s_j is after or at the starting time of the other d intervals. But then, this implies that there are $d+1$ overlapping intervals at s_j, contradicting the fact that there are at most d intersecting intervals at any time. ■

Corollary 5.3.3 The greedy algorithm produces a solution with exactly d partitions.

Proof. Follows immediately from the Lemma 5.3.1 and 5.3.2. ■

Hence the theorem:

Theorem 5.3.4 The greedy algorithm using earliest starting time as greedy choice is optimal.

This greedy algorithm is implemented as follows. Suppose, as a precondition, that S is sorted in increasing order of starting time, and that elements in Q contains objects of INTERVALs that are indexed by finish time.

```
INTERVAL-PARTITIONING(S)
 1  d = 0
    // initialize PQ using finish time as key
 2  Q = PRIORITY-QUEUE(key = f)
 3  for i = 1 to S.length
 4      k = EXTRACT-MIN(Q)
 5      if k == NIL
 6          d = d + 1
 7          interval = new INTERVAL(i.f)
 8          INSERT(Q, interval)
 9      if i.s > MIN(Q)
            // if i is compatible with k, allocate i to k
10          k.f = i.f
11          INSERT(Q, k)
12      else
            // otherwise, create new classroom d + 1
13          d = d + 1
14          INSERT(Q, k)
15          interval = new INTERVAL(i.f)
16          INSERT(Q, interval)
17  return d
```

This algorithm runs in $O(n \log n)$ time. This is the case regardless of whether or not we consider sorting as part of the algorithm because the n priority queue operation is going to cost $O(n \log n)$ anyway, assuming the priority is implemented as a min-heap.

5.4 Greedy Strategy

The greedy strategy can be generalized as follows:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing (typically by contradiction or induction) that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

The important property that requires us to prove when implementing a greedy algorithm is the ***greedy property***. It tells us that we can assemble a globally optimal solution from locally optimal choices. The exchange argument for the proof (the first two proofs that we have seen for interval scheduling) argues that we can modify the globally optimal solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem.

Chapter 6 Graph Algorithms Using Greedy

6.1 Interval Scheduling as Graph Problems

There is a natural way to view the interval scheduling and partitioning problems as graph problems.

Let I be a set of intervals. We can construct the ***intersection graph*** $G(I) = (V, E)$ where $V = I$ and $(u, v) \in E$ if and only if the intervals corresponding to u and v intersect.

Any graph that is the intersection graph of a set of intervals is called an ***interval graph***. The interval scheduling and interval partition problem can be viewed as the ***maximum independent set problem*** for the class of interval graphs.

Definition 6.1.1 — Maximum Independent Set. Let $G = (V, E)$ be a graph. A subset U of V is an independent set (stable set) in G if for all $u, v \in U$, $(u, v) \notin E$.

The maximum independent set of G is an independent set of G with maximum cardinality.

! Maximal and maximum are ***not*** the same. It is important to distinguish between "maximum" and "maximal" sets, and between "minimum" and "minimal" sets. A maximal set is a set that is not a proper subset of any other sets, while a maximum set is simply a set of maximum cardinality among all sets.

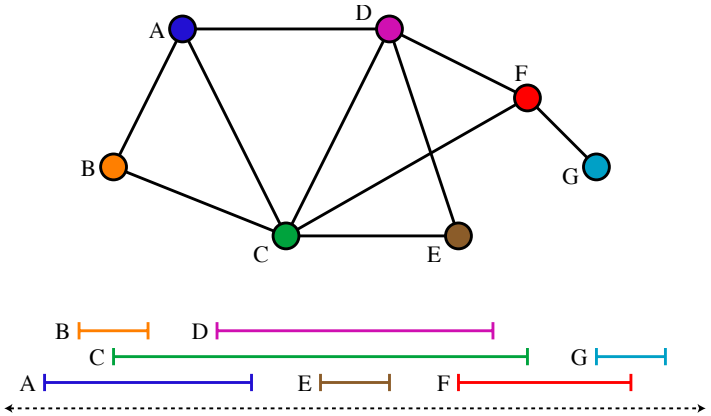


Figure 6.1: Intervals represented as an interval graph. Image from Wikipedia.

Definition 6.1.2 — Graph Coloring. Let $G = (V, E)$ be a graph. A function $c : V \to \{1, \ldots, k\}$ is a valid coloring of G if $c(u) \neq c(v)$ for all $(u, v) \in E$.

The graph coloring problem is to find a coloring c that minimizes the number of colors k. Such number is called the chromatic number of G, denoted $\chi(G)$.

To model the interval scheduling and partition problem as a graph theory problem, we first need a way to efficiently interconvert between the interval graph representation and the set representation.

Given a set I of intervals, it is easy to construct the interval graph $G(I)$. For conversion back from interval graph, the following theorem claims that it can also be done efficiently.

Theorem 6.1.1 Given any graph G, there is a linear-time algorithm to decide if G is an interval graph, and if so, to construct an interval representation.

The maximum independent set problem for intersection graph can be efficiently solved, so the interval scheduling and partition problem can also be efficiently solved. The graph theoretical explanation for this is that interval graphs are chordal graphs with a perfect elimination ordering, making it possible to use the greedy approach (or more generally, a priority based approach). However, in the general case, the maximum independent set problem is known to be NP-hard.

To formallly prove that the maximum independent set problem can be efficiently solved for interval graphs, we need to introduce a few additional concepts.

6.1.1 Interval Graph is Chordal Graph

We first show that an interval graph is also a chordal graph (the converse is not true).

Definition 6.1.3 — Chordal Graph. Let G be an undirected graph. G is a ***chordal graph*** or ***rigid circuit graph*** if for any cycles of 4 or more vertices, there exists a pair of vertices u, v on the cycle such that $\{u, v\} \in E$ but $\{u, v\}$ is not part of the cycle. The edge $\{u, v\}$ is called a ***chord*** of the cycle.

Theorem 6.1.2 An interval graph is also a chordal graph.

Proof. Let $G = (V, E)$ be an interval graph with a cycle $C_k = v_1 v_2 \ldots v_k v_1$ where $k \geq 4$ and each v_i represents an interval $[a_i, b_i)$. The case where $k < 4$ is vacuously true. Suppose the first k vertices of the cycle are ordered in increasing order by the left endpoints of the intervals they represent. Consider the k vertex on the cycle. $\{v_{k-1}, v_k\} \in E$, so $[a_{k-1}, b_{k-1}) \cap [a_k, b_k) \neq \emptyset$. Since the vertices are sorted by left endpoints, $a_k \in [a_{k-1}, b_{k-1})$. Since C_k is a cycle, $\{v_k, v_1\} \in E$, so $[a_1, b_1) \cap [a_k, b_k) \neq \emptyset$. Again, because the intervals are sorted by left endpoints, $a_1 < a_k$. It follows that $a_k \in [a_1, b_1)$. a_k is in both the interval $[a_{k-1}, b_{k-1})$ and $[a_1, b_1)$, so $[a_1, b_1) \cap [a_{k-1}, b_{k-1}) \neq \emptyset$, so $\{v_1, v_{k-1}\} \in E$ by definition of interval graphs. $\{v_1, v_{k-1}\}$ is a chord, so G is a chordal graph. ■

6.1.2 Perfect Elimination Ordering and Cliques

The next step is to show that there exists an ordering of the vertices of the interval graph, from which we can derive a greedy algorithm.

Definition 6.1.4 — Simplicial Vertex. In a graph $G = (V, E)$, we say $v \in V$ is ***simplicial*** if the subgraph induced by the $\{v\} \cup N(v)$ is a complete graph (forms a clique).

Definition 6.1.5 — Perfect Elimination Ordering. A graph $G = (V, E)$ has a ***perfect elimination ordering (P.E.O.)*** if there is an ordering $(v_1, \ldots, v_n)$ of V such that v_i is simplicial vertex in the subgraph induced by $\{v_i, \ldots, v_n\}$.

Definition 6.1.6 — Separator and Minimal Separator. Given a graph $G = (V, E)$, we define the ***separator*** S in G to be the subset of V such that it partitions V into three disjoint sets A, S, B such that $V = A \cup S \cup B$ and for all $a \in A$ and $b \in B$, $(a, b) \notin E$. Intuitively, this means there is no edge going directly from a vertex in A to a vertex in B, so the subgraph induced by $V - S$ has two disjoint connected components (induced by A and B).

Given two non-adjcent vertices $a, b \in V$ such that $(a, b) \notin E$, we say S is an $(\mathbf{a}, \mathbf{b})$-***separator*** if S partitions V into disjoint subsets A, S, B such that $a \in A$ and $b \in B$. A ***minimal*** (a, b)-separator is an (a, b)-separator S such that no subset of S is an (a, b)-separator.

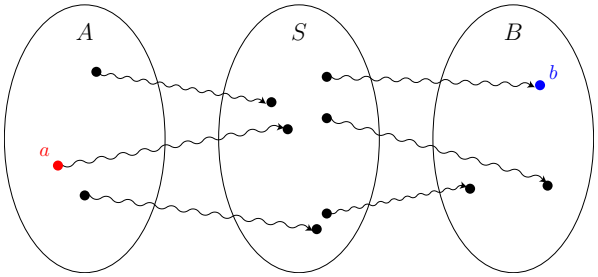


Figure 6.2: Example of an (a, b)-separator

Lemma 6.1.3 Given a chordal graph $G = (V, E)$ and two non-adjacent vertices $a, b \in V$ such that $(a, b) \notin V$, any minimal (a, b)-separator induces a clique.

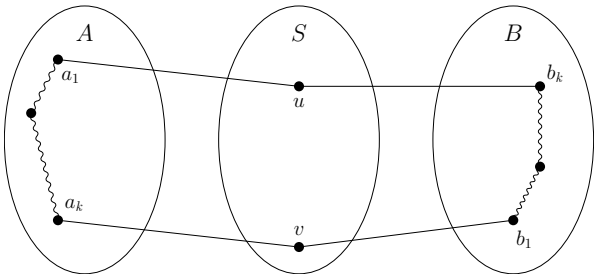
Proof. By contradiction.

Let S be a minimal (a, b)-separator. Let A, B be the two connected components separated by S, where $a \in A$ and $b \in B$. Suppose, for contradiction, that S does not induce a clique, so there exists $u, v \in S$ such that $\{u, v\} \notin E$. Since S is minimal, there are edges from u and v to the subgraphs induced by A and B. Otherwise, $S - \{u, v\}$ would have been a smaller (a, b)-separator.

Let p_a be the shortest path from u to v, passing through vertices in A. Similarly, let p_b be the shortest path from v to u, passing through vertices in B. p_a and p_b each has path length of at least 2 because u, v are not adjacent. It follows that the union of the two paths p_a and p_b is a cycle

$u \to a_1 \rightsquigarrow a_k \to v \to b_1 \rightsquigarrow b_k \to u$ of length at least 4. Since G is chordal, this cycle must contain a chord. But since this is the shortest cycle, there is no edge from u, v to vertices in A, B (otherwise, we can form a shorter cycle using this edge). There is no edge from vertices in A to vertices in B because the subgraphs they induce are disjoint. So, the chord must be between u and v. However, this contradicts the assumption that $\{u, v\} \notin E$.

Hence, for every $u, v \in E$, $\{u, v\} \in E$.



■

Lemma 6.1.4 — Dirac 1961. Given a graph $G = (V, E)$, if all for all $a, b \in V$, the minimal (a, b)-separator induces a clique, then G has a perfect elimination ordering.

Proof. By induction on the number of vertices in G.

Base case: When $n = 1$, the lemma trivially holds.

Inductive step: Let $n \in \mathbb{N}$. Assume that the lemma holds for all $k < n$.

If G is a complete graph, the whole graph is a clique, and we are done since every ordering is a perfect elimination ordering.

Otherwise, there exists $a, b \in V$ such that $\{a, b\} \notin E$. Let S be a minimal (a, b)-separator in G. S separates V into disjoint subsets A, S, B. By assumption, S induces a clique. Since the size of A is strictly smaller than V, by induction hypothesis, we know that A has a perfect elimination ordering. This implies that there exists $u \in A$ such that $\{u\} \cup N(u)$ induces a clique on the subgraph induced by A.

Since S separates A and B, there is no edge going directly from vertices in A to vertices in B. There is an edge connecting every pair of vertices in S, so if there are edges going between u and vertices in S, the resulting subgraph is also a clique. Hence, $\{u\} \cup N(u)$ also induces a clique in G because S induces a clique.

Consider the graph G' obtained by removing vertex u from G. $|V - \{u\}| < |V| = n$. By induction hypothesis, G' has a perfect elimination ordering. Suppose the ordering is $(v_1, \ldots, v_{n-1})$. We can let

$v_n = u$. By adding v_n to the ordering, we get a new valid perfect elimination ordering because v_n and its neighbors induces a clique in G.

By induction, the lemma holds for graph of any size. ■

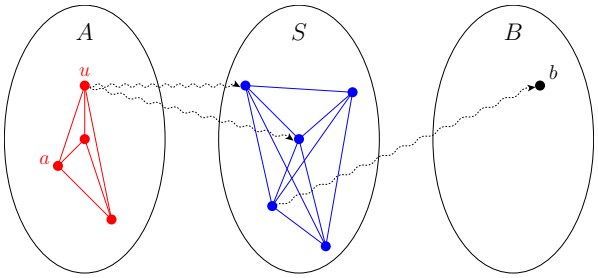


Figure 6.3: Proof idea for Lemma 6.1.4

Theorem 6.1.5 — Fulkerson and Gross 1965. A graph is a chordal graph if and only if it has a perfect elimination order.

Proof. We will prove the theorem by proving the chain of implications that

(1) P.E.O. $\Rightarrow$ (2) chordal graph $\Rightarrow$ (3) minimal separator induces a clique $\Rightarrow$ (1) P.E.O.

(1) $\Rightarrow$ (2): Let C be a cycle of length at least 4 in G. Assume G has a perfect elimination ordering. Then, take the perfect elimination ordering, and remove vertices by the ordering until we reach a vertex c on the cycle C. Once we remove c, $N(c)$ induces a clique by definition of perfect elimination ordering. Therefore, there exists an edge between every two vertices in $C - \{c\}$. In other words, C contains chords. Hence, G is a chordal graph.

(2) $\Rightarrow$ (3): By Lemma 6.1.3.

(3) $\Rightarrow$ (1): By Lemma 6.1.4. ■

Tarjan and Yannakakis designed a linear-time algorithm called the maximum cardinality search for computing the perfect elimination ordering given a chordal graph [19].

COMPUTE-PEO(G)

```
1  for all v ∈ V
2      v.label = 0
3      σ(v) = −1
4  for i = |V| downto 1
5      u = argmax_{v∈V}{v.label | σ(v) = −1}    // unvisited vertex with largest label
6      σ(u) = i    // assign an order
7      for neighbor w of v
8          if σ(w) == −1
9              w.label = w.label + 1
10 return σ
```

The algorithm starts by initializing the labels and numbers for each vertex. Then, it iteratively select unnumbered vertex with largest label to assign an order. With an naive implementation, Line 5 of the algorithm takes $O(|V|)$ time. However, this can be improved by using a disjoint set data structure. Let S_i be the sets of unnumbered vertices with label i for $0 \leq i \leq |E|-1$. Each S_j is implemented using a linked list, and $[S_0, S_1, \ldots S_i]$ is a direct access array indexed by i. Every time we increment the label of w, we move w from S_i to S_{i+1} where in this case, i is the $w.label$. It is clear that this improved implementation runs in $O(|V|+|E|)$ time.

Finally, as promised, we will show an algorithm that solves the maximum independent set problem on an interval (chordal) graph.

```
MIS(G)
1  σ = COMPUTE-PEO(G)
2  I = ∅
3  for i = 1 to |V|
4       v = σ^{-1}(i)
5       N = {v ∈ N(v) | σ^{-1}(v) < i}
6       if N ∩ I == ∅
7            I = I ∪ {v}
8  return I
```

The proofs in this section are based on the papers *Incidence matrices and interval graphs* by Fulkerson and Gross [13], and *On rigid circuit graphs* by Dirac [9]. The presentation of the proofs is loosely based on *Advanced Topics in Graph Algorithms* by Ron Shamir [18].

6.2 Minimum Spanning Tree

A tree (V,T) is a connected graph with no cycles (connected acyclic graph). Every node is reachable from every other node in exactly one way. If (V,T) is connected, then (V,T) is acyclic if and only if $|T| = |V|-1$.

Definition 6.2.1 — Spanning Forest and Spanning Tree. Let $G = (V,E)$ be an undirected graph. A spanning forest of $G = (V,E)$ is an acyclic graph (V,F) with $F \subseteq E$. It is a set of tree on disjoint sets of vertices.

A spanning tree of G is a connected spanning forest of G. It contains $|V|-1$ edges.

Kruskal's algorithm for MST proceeds by interatively merging subtrees of the input graph by picking the minimum weighted edge between the two subtrees. Prim's algorithm proceeds by starting from a single vertex and iteratively growing a tree from the vertex, again, by picking the minimum weighted edge as we expand the growing frontier of the current subtree.

6.2.1 Kruskal's Algorithm

KRUSKAL(V,w)
1 $A = \emptyset$
2 $Q =$ PRIORITY-QUEUE(all edges e where priority is $w(e)$)
3 **for** $v \in V$
4 MAKE-SET(v)
5 **while** $|A| < n-1$
6 $\{u,v\} =$ EXTRACT-MIN(Q)
7 $u' =$ FIND-SET(u)
8 $v' =$ FIND-SET(v)
9 // if u and v are in different components of (V,A)
10 **if** $u' \neq v'$
11 add $\{u,v\}$ to A
12 LINK(u',v')
13 **return** (V,A)

The optimality of Kruskal's algorithm can be proved using an exchange argument similar to the one that we used for the interval selection (job scheduling) problem.

Theorem 6.2.1 — Correctness of Kruskal's Algorithm. If $G = (V,E)$ is a undirected connected graph with weight function $w : E \to \mathbb{R}$, when we run KRUSKAL on G, the algorithm returns a minimum spanning tree (V,A).

Proof. Let A_i be the set of edges in A after the ith iteration of the outer loop at Line 5. We will prove by induction the claim that for all $i \in \{0,\ldots,n-1\}$, there exists a set $T_i \subseteq \{e_{i+1},\ldots,e_n\}$ ordered by the weight of its elements, such that $A_i \cup T_i \subseteq A_{opt} \subseteq A_i \cup \{e_{i+1},\ldots,e_n\}$ where A_{opt} is an optimal solution.

Base case: When $i = 0$, $A_0 = \emptyset$. Since the graph is connected, there must exists a minimum spanning tree A_{opt} and $A_0 \subseteq A_{opt} \subseteq A_0 \cup \{e_{i+1},\ldots,e_n\}$.

Inductive step: Let $i \in \mathbb{N}$ be arbitrary. Suppose that the claim holds for i. This means $A_i \subseteq A_{opt} \subseteq A_i \cup \{e_{i+1},\ldots,e_n\}$. Let $e_{i+1} = \{u,v\}$ be the edge with $(i+1)$th smallest weight. Consider the following cases:

(1) Adding $e_{i+1} = \{u,v\}$ forms a cycle. In this case, e_{i+1} is not added to A and $A_{i+1} = A_i \subseteq A_{opt} \subseteq A_i \cup \{e_{i+2},\ldots,e_n\}$.

(2) Adding $e_{i+1} = \{u,v\}$ does not form a cycle and $e_{i+1} \in A_{opt}$. In this case, e_{i+1} is added to A. $A_{i+1} = A_i \cup \{e_{i+1}\}$. But since e_{i+1} is already in A_{opt}, $A_{i+1} = A_i \cup \{e_{i+1}\} \subseteq A_{opt} \cup \{e_{i+1}\} = A_{opt} \subseteq A_i \cup \{e_{i+2},\ldots,e_n\}$.

(3) Adding $e_{i+1} = \{u,v\}$ does not form a cycle and $e_{i+1} \notin A_{opt}$. e_{i+1} is added to A. Since A_{opt} is a spanning tree, it covers every vertices in V so adding e_{i+1} to A_{opt} forms a cycle between u and v. This implies that there exists an edge $e_j \in A_{opt} - A_{i+1}$ on the cycle since A_{i+1} is acyclic.

Removing e_j from $A_{opt} \cup e_{i+1}$ breaks the cycle, and $A'_{opt} = A_{opt} - \{e_j\} \cup \{e_{i+1}\}$ is a spanning tree. Since $A_{opt} \subseteq \{e_{i+1}, \ldots, e_n\}$ by inductive hypothesis, and $e_{i+1} \notin A_{opt}$, we have $j > i+1$. Because $\{e_{i+1}, \ldots, e_n\}$ is sorted in nondecreasing order by edge weight, we can conclude that $w(e_{i+1}) \leq e_j$. It follows that $w(A'_{opt}) \leq w(A_{opt})$. Since A_{opt} is already optimal, $w(A'_{opt}) = w(A_{opt})$ and thus A'_{opt} is also optimal. Furthermore, $A'_{opt} \subseteq A_{i+1} \cup \{e_{i+2}, \ldots, e_n\}$.

By induction, the claim holds for all $\{0, \ldots, n-1\}$. It follows that when the algorithm terminates after the final iteration, $A_n \subseteq A_{opt} \subseteq A_n \cup \emptyset = A_n$, so $A_n = A_{opt}$. ■

In fact, this proof also shows the correctness of Prim's algorithm.

6.2.2 Prim's Algorithm

```
PRIM(V, r)
 1  for v ∈ V − {r}
 2      v.priority = ∞
 3      v.parent = NIL
 4  Q = PRIORITY-QUEUE(V − {r})
 5  u = r
 6  while Q ≠ ∅
 7      for neighbor v of u
 8          if v ∈ Q and w({u, v}) < v.priority
 9              v.priority = w({u, v})
10              DECREAE-PRIORITY(Q, v, priority = w({u, v}))
11              v.parent = u
12      u = EXTRACT-MIN(Q)
13      add {u, u.parent} to A
14  return (V, A)
```

6.3 Single-Source Shortest Path

Given a weighted directed graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$, we define the weight of a path $p = v_0, v_1, \ldots, v_k$ to be the sum of all edges on that path

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Additionally, if there is a path from u to v in a graph G, let $\delta(u, v)$ denote the minimum weight of any of such path. Otherwise, define $\delta(u, v) = \infty$, meaning that v is not reachable from u.

A path p from u to v is a shortest path if $w(p) = \delta(u, v)$.

6.3.1 Dijkstra's Algorithm

The idea of ***Dijkstra's algorithm*** is to construct a set of vertices V' whose shortest path from s has been determined. Each vertex $v \in V' - \{s\}$ has its predecessor $v.parent$ on this path and $v.d$ is the weight of this path. One important limitation of Dijkstra's algorithm is that the graph **does not contain any edge with negative weight**.

Line 10-13 of the algorithm is known as "edge relaxation". Similar to Prim and Kruskal's algorithm, Dijkstra's algorithm is a greedy algorithm. It orders the nodes based on their distances (or more precisely, the current estimate of distances at a given stage of the algorithm, d) from s. At each step of the algorithm (the outer while loop), the algorithm looks at the node u with the smallest d and perform edge relaxation between u and its neighbors.

DIJKSTRA(G,s)

```
1  for v ∈ V − {s}
2      v.d = ∞
3      v.parent = NIL
4  s.d = 0
5  s.parent = s
6  Q = PRIORITY-QUEUE(V, key = d)
7  while Q ≠ ∅
8      u = EXTRACT-MIN(Q)
9      for neighbor v of u
10         if v ∈ Q and v.d > u.d + w({u,v})
11             v.d = u.d + w({u,v})
12             DECREASE-PRIORITY(Q, v, priority = u.d + w({u,v}))
13             v.parent = u
```

Chapter 7 Huffman Encoding

Chapter 8 Generalizing and Formalizing Greedy

8.1 Priority Model

For a given problem, we assume that the input items belong to some set $\mathcal{J}$. For any execution of the algorithm, the input is a finite subset $\mathcal{I} \subset \mathcal{J}$.

Let $f : \mathcal{J} \to \mathbb{R}$ be a function. We do not place restriction on the complexity or computability of such function. For an input set $\mathcal{I} = \{I_1, \ldots, I_n\}$, the function induces a total ordering $\preceq$ on $\mathcal{I}$, breaking ties using certain rules (for example, by input ID).

For a fixed order priority algorithm, f and $\preceq$ are set initially before the algorithm observes the input set. For adaptive order, the algorithm computes f and $\preceq$ dynamically. More formally, there is a different function f_i and ordering $\preceq_i$ associated with each iteration i where f_i and $\preceq$ depends on the items $\{I_1, \ldots, I_j\}$ considered in prior iterations $j < i$.

Let us first consider fixed order algorithm under this priority model. In each iteration k for $1 \leq k \leq n$, the algorithm observes input element $I_k \in \mathcal{I}$ and based on this input and all previous inputs and decisions, the algorithm makes an irrevocable decision D_k about this input item (this typically involves whether to include or discard the current item).

This model gives us the following template for fixed order priority algorithms

```
1  J = set of all possible inputs
2  ≼ = a total ordering on J (typically induced by a function f)
3  I ⊂ J = actual input to the algorithm
4  S = ∅                                  // items already examined by the algorithm
5  i = 0
6  while I − S ≠ ∅
7      i = i + 1
8      I = I − S
9      I_i = min_≼{I ∈ I}                 // select min element based on the ordering ≼
10     make an irrevocable decision D_i concerning I_i
11     S = S ∪ {I_i}
```

This template can be modified to allow the algorithm to determine the total ordering dynamically based on the elements it has observed so far.

```
1  J = set of all possible inputs
2  I ⊂ J = actual input to the algorithm
3  S = ∅                                  // items already examined by the algorithm
4  i = 0
5  while I − S ≠ ∅
6      i = i + 1
7      ⪯_i = a total ordering on J (typically induced by a function f_i)
8      I = I − S
9      I_i = min_{⪯_i}{I ∈ I}             // select min element based on the ordering ⪯_i
10     make an irrevocable decision D_i concerning I_i
11     S = S ∪ {I_i}
12     J = J − {I ∈ I | I ⪯_i I_i}
```

For greedy algorithm, the algorithm does not have knowledge of the input other than the elements it has observed so far. Sometimes, we allow the algorithm to have some easily computed global information such as the size of the input. The input is said to be chosen by an adversary.

This generalization applies to a broader category of algorithms known as priority-based algorithms. Informally, greedy algorithms always assume that the current iteration could be the last one and the current item being considered might as well be the last item. Hence, an optimal decision is immediate when the algorithm finishes. A more general priority algorithm that are not necessarily greedy does not have such restriction

This formalization was first given by Allan Borodin *et al.* in *(Incremental) Priority Algorithms* (Thirteen Annual ACM-SIAM Symposium on Discrete Algorithms, January 2002) [3].

8.2 Global Optimality of Greedy Algorithms

(From Allan Borodin's notes)

Most greedy algorithms are not opitmal. The method we use to show that a greedy algorithm is optimal (when it is) known as the exchange argument often proceeds as follws. At each stage i, we define our partial solution to be promising if it can be extended to an optimal solution by using elements that haven't been considered yet by the algorithm; that is, a partial solution is promising after stage i if there exists an optimal solution that is consistent with all the decisions made through stage i by our partial solution. We prove the algorithm is optimal by fixing the input problem, and proving by induction on $i \geq 0$ that after stage i is performed, the partial solution obtained is promising. The base case of $i = 0$ is usually completely trivial: the partial solution after stage O is what we start with, which is usually the empty partial solution, which of course can be extended to an optimal solution. The hard part is always the induction step, which we prove as follmvs. Say that stage $i+1$ occurs, and that the partial solution after stage i is S_i and that the partial solution after stage $i+1$ is S_{i+1}, and we know that there is an optimal solution S_{opt} that extends S_i; we want to prove that there is an optimal solution S'_{opt} that extends S_{i+1}. S_{i+1} extends S_i by taking only one decision; if S_{opt} makes the same decision, then it also extends S_{i+1}, and wve can just let $S'_{opt} = S_{op}$ and we are done. The hard part of the induction

step is if S_{opt} does not extend S_{i+1}, In this case, we have to show either that S_{opt} could not have been optimal (implying that this case cannot happen), or we show how to change some parts of S_{opt} to create a solution S'_{opt} such that

- S'_{opt} extends S_{i+1} and
- S'_{opt} has value (cost, profit, etc.) at least as good as S_{opt}, so the fact that S_{opt} is optimal implies that S'_{opt} is optimal.

For most greedy algorithms, when it ends, it has constructed a solution that cannot be extended to any solution other than itself. Therefore, if we have proven the above, we know that the solution constructed by the greedy algorithm must be optimal.

8.3 Greedy v.s. DP

In the next part, we will examine dynamic programming, which is another powerful algorithm design paradigm for solving many optimization problem. In fact, greedy and dynamic programming shares a lot of similarities, especially in the sense that both approaches exploit the optimal substructure property where the globally optimal solution can be constructed from optimal solutions of its subproblems.

Unlike greedy, which makes the locally optimal choice, dynamic programming solves potentially overlapping subproblems and makes an informed choice.

The major differences can be summarized using this chart

	Greedy	DP
Decision	starts from minimal subproblem, construct solution based on best option for the current step	starts from minimal subproblem, recursively construct solution based on all previously considered subproblems; use memoization for overlapping subproblems
Optimality	not necessarily	yes
Space complexity	better	requires an array to store solutions previously considered subproblems
Time complexity	usually better	usually polynomial
Examples	Kruskal and Prim; Dijkstra; interval scheduling (selection) and partition (coloring)	Bellman-Ford; 0-1 knapsack; longest common subsequence

Table 8.1: Differences between greedy and dynamic programming

Another way to view greedy and dynamic programming is to consider the problem and the possible solutions as a set and compare how the two approaches reduce the size of problem and narrow down the solution space. This is an illustration that I came up completely by myself. Although it seems to make sense, it is not presented in any research articles or texts. It should only be used as an intuition

to understand the similarity and differences of the two approaches and should not in any way be considered a rigorous formulation or analysis.

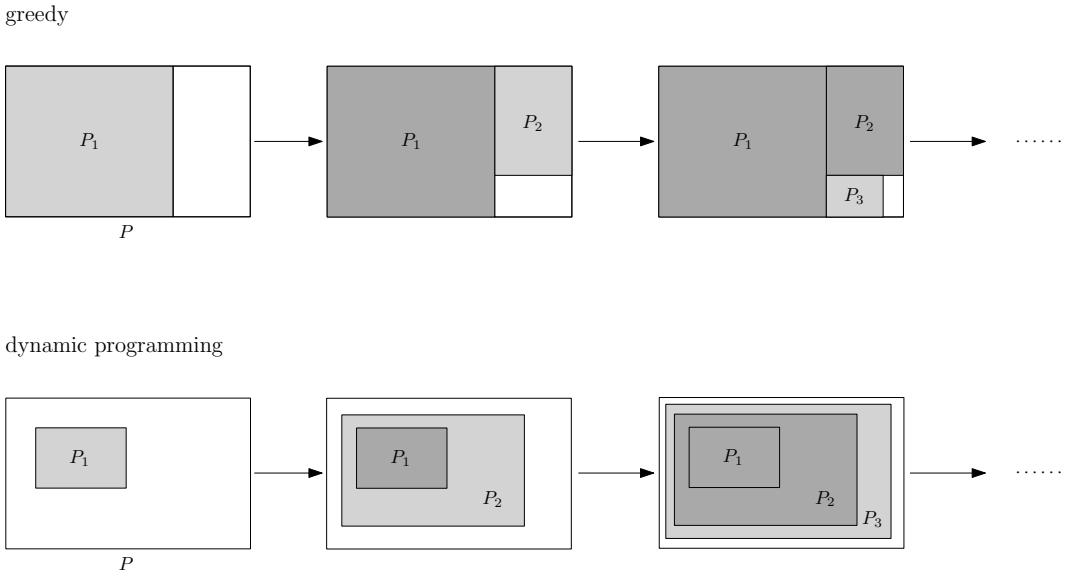


Figure 8.1: Another way to view the differences between greedy and DP. The rectangle represents the overall problem. Shaded region of the rectangle represents the subproblems being considered, and heavily shaded regions of the rectangle represent the subproblems that have already been solved. Note that greedy expands the solution (and reduces the size of the problem remaining) by looking at the locally optimal choice without going back; while DP often involves overlapping subproblems where a subproblem or subproblems can be contained within another subproblem (this is why memoization is important for DP).

III Dynamic Programming

Chapter 9 Weighted Interval Selection

9.1 Dynamic Programming

Dynamic programming (DP) is a method for solving complex problem by breaking it down to a collection of simpler subproblems, solving each of those subproblems, and storing their solutions. The next time the same subproblem occurs, instead of recomputing the solution, the algorithm can just use the previously computed solution. Each subproblem is indexed in some way to facilitate the lookup. The technique of storing solutions to subproblems is called **memoization**.

Let's consider the weighted interval selection problem.

9.2 Weighted Interval Selection

The objective of the weighted interval selection problem is to find a non-intersecting set of intervals so that the sum of the weights of intervals in the set is minimized. More formally, given a set of jobs/activities $\{a_1, \ldots, a_n\}$ with start time s_i and finish time f_i and weight w_i, we want to find a set S of mutually compatible jobs with highest total weight $\sum_{i \in S} w_i$

Note that when $w_i = 1$ for all $i = \{1, \ldots, n\}$, the weighted interval selection problem is the same as simple interval scheduling.

We first consider greedy algorithms. Possible choices for the greedy choice include: weights, interval length, etc. Unfortunately, all the possible ways of ordering the input items will not only fail to give an optimal solution, but can in fact produce arbitrarily bad solutions for some instances. With a generalized greedy approach, it can be proved that no greedy algorithm can produce a good solution. With certain modifications/extensions to greedy, the algorithm can produce a good approximation (by allowing revocable decisions) and even optimal solution (using local ratio/primal dual algorithms with a reverse delete phase).

9.3 Computing the Optimal Weight

9.3.1 The Dynamic Programming Approach

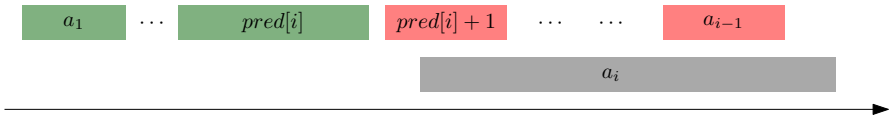
Assume that the jobs (intervals) have been sorted by non-decreasing finish time. Then, in an optimal solution O, either the last interval I_n was selected, or it was not. If not, then we must be using an

optimal solution for the first $n-1$ intervals. If a_n is in O, then no interval in O can end after the starting time of a_n.

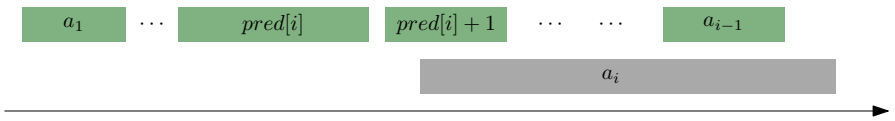
Let us now make this intuition more concrete. Given that the jobs are sorted by non-decreasing finish time, let $pred(i)$ be the largest index $j < i$ such that $f_j < s_j$. This means that jobs $a_1, \ldots, a_j$ are compatible with a_i, but $a_{i+1}, \ldots, a_{j-1}$ are not compatible.

Let O be an optimal solution. For each job a_i, there are two possibilities:

- Job $a_i \in O$: This implies that $\{pred[i]+1, \ldots, a_{i-1}\}$ are all incompatible with a_i. So, we must select jobs from $\{a_1, \ldots, pred[i]\}$.



- Job $a_i \notin O$: This implies that we must select jobs from $\{a_1, \ldots, a_{i-1}\}$.



Let define $V : \{a_1, \ldots, a_n\} \to \mathbb{R}$ be defined such that

$$V(i) = \text{max total weight of compatible jobs from } \{a_1, \ldots, a_i\}$$

More formally, V can be expressed as this recurrence

$$V(i) = \begin{cases} 0 & i = 0 \\ \max\{V(i-1), w_i + V(pred[i])\} & i > 0 \end{cases}$$

Lemma 9.3.1 The two definitions of V are equivalent.

Proof. By induction. ■

Sometimes, the first definition of V in English is represented as an array indexed by i and referred to as a ***semantic array***. The second equation of V is called the ***Bellman equation***.

This recurrence gives us a clear recursive solution for the weighted interval selection problem.

Consider the time complexity of the naive recursive implementation. $pred[i]$ can be computed using binary search in $O(\log n)$ time, and for all of the n values to be considered, this takes $O(n \log n)$ time. However, the time spent on computing $pred$ turns out to be rather insignificant compared to the running time of the main algorithm.

$$T(n) = T(n-1) + T(pred[n]) \leq T(n-1) + T(n-2) \in \Theta(\varphi^n)$$

where $\varphi \approx 1.618$ is the golden ratio. This is a Fibonacci recurrence.

This is clearly bad. Some solutions are being computed many times unnecessarily. Recall that one of the most important component of dynamic programming is memoization. In many cases, without memoization, dynamic programming simply becomes divide and conquer, or even worse, recurrence that runs in exponential time like this one. Memoization allows us to remember the results that we have already computed and reuse them if needed.

9.3.2 Top-Down DP

In our first dynamic programming implementation, we will store the previously computed values in the array M indexed by the index of each interval.

Assume that prior to calling COMPUTE-OPT-DP(n), the intervals are sorted in nondecreasing order based on finish time, *pred* has been precomputed in $O(n \log n)$ time, and M is a global array of size n with every entry set to 0 initially.

```
COMPUTE-OPT-DP(j)
1  if M[j] == NIL
2      M[j] = max{COMPUTE-OPT-DP(j − 1), w_j + COMPUTE-OPT-DP([j])}
3  return M[j]
```

For each $i = \{1, \ldots, n\}$, there is only one call to COMPUTE-OPT-DP(i) because previously computed values are stored in the array M. Therefore, there are at most $O(n)$ calls to the recursive procedure. Sorting by finish time and computing *pred* takes $O(n \log n)$ time, so the overall time complexity of this implementation is $O(n \log n)$. Much better than our original exponential implementation!

9.3.3 Bottom-Up DP

The bottom-up approach differs from the top-down approach in that values of M are precomputed. Instead of using recursive call and checking if $M[i]$ == NIL, we can simply reference to indices of the array where previously computed values are stored. Assume the same preconditions are met prior to calling COMPUTE-OPT-DP-BOTTON-UP.

```
COMPUTE-OPT-DP-BOTTON-UP()
1  for j = 1 to n
2      M[j] = max{M[j − 1], w_j + M[pred[j]]}
3  return M[n]
```

It can be shown using a simple loop invariant and induction that at the beginning of the ith iteration, $M[j]$ is already computed for all $j < i$ and thus we will not get a null pointer error.

This approach has the same time complexity as the top-down approach.

9.3.4 Comparison of Two Approaches

Top-down may be preferred when not all sub-solutions need to be computed on some inputs. This helps us save some time.

Bottom-up may be preferred when all sub-solutions will always need to be computed for all inputs. The bottom-up approach could be faster as it prevents unnecessary recursive calls which results in unnecessary random memory access. This is because even though recursive dynamic programming does not compute repeated results, there could still be redundant recursive calls, resulting in a large overhead.

9.4 Computing the Optimal Solution

In the previous section, we come up with a dynamic programming algorithm that computes the optimal weight of the solution set for the weighted interval selection problem. Now, we will extend that algorithm so that we get the actual solution set (subset of S that maximizes weight). We only need a really simple modification from the original definition of V. Recall that V is defined as

$$V(i) = \begin{cases} 0 & i = 0 \\ \max\{V(i-1), w_i + V(pred[i])\} & i > 0 \end{cases}$$

Instead of calculating values, we want a subset of the input set.

$$S(i) = \begin{cases} \emptyset & i = 0 \\ S(i-1) & \text{if } V(i) = V(i-1) \\ S(pred[i]) \cup \{a_i\} & \text{otherwise} \end{cases}$$

We can either precompute V and uses it directly in the implementation of S. Or alternatively, we can compute V and S simultaneously.

Chapter 10 Knapsack Problem

10.1 Knapsack

In the knapsack problem, we are given a set of n items $I_1, \ldots, I_n$ and a size bound B where each item $I_j = (w_j, v_j)$ with w_j being the weight of the item and v_j the value of the item.

A subset of items S is feasible if the sum of the weights of items in S is at most B. The goal of the knapsack problem is to find a feasible set S that maximizes the sum of the values of items in S.

10.2 DP Algorithm for 0-1 Knapsack

We define the semantic array

$$V(i,b) = \text{max profit possible using the first } i \text{ items within weight } b$$

with corresponding Bellman equation

$$V(i,b) = \begin{cases} 0 & \text{if } i = 0 \text{ or } b = 0 \\ \max\{C, D\} & \text{if } w_i \leq b \end{cases}$$

where $C = V(i-1,b)$ and $D = V(i-1,b-w_i) + v_i$.

C corresponds to the decision to not include the ith item in the knapsack, and D correspond to the decision to include the ith item, in which case we deduce w_i from the remaining weight of the knapsack and add v_i to the total profit. Like all DP problems, we need to prove that the semantic array is equivalent to the Bellman equation.

```
KNAPSACK(S, B)
 1  n = S.length
 2  for b = 0 to B
 3      M[0, b] = 0
 4  for i = 1 to n
 5      for b = 0 to B
 6          if S[i].w > b
 7              M[i, b] = M[i − 1, b]
 8          else
 9              M[i, b] = max{M[i − 1, b], S[i].v + M[i − 1, b − 1]}
10  M[n, B]
```

The input to the algorithm: S is the set of all items to be considered, and B is the weight bound of the knapsack. This approach makes an important assumption that the weights are integer.

This DP algorithm solving the 0-1 knapsack problem with n items and weight bound B runs in $\Theta(nB)$ time and uses $\Theta(nB)$ space. As we can see, the running of this algorithm is, unfortunately, not polynomial in the input size. It is ***pseudo-polynomial***. It is polynomial in $\log B + \sum_{i=1}^{n}(\log v_i + \log w_i)$.

If B is small (a polynomial in n), then the algorithm runs in polynomial time. No known exact algorithm runs in polynomial time. The 0-1 knapsack problem is NP-complete.

10.3 A Different DP Algorithm

Consider version of the knapsack problem with a different restriction that the values v_i are integer and small ($V = \sum_{i=1}^{n} v_i \ll B$). In this case, it makes more sense to derive an algorithm in terms of v instead of B.

The semantic array is defined as follows

$$W(i,v) = \begin{cases} \text{minimum weight required to obtain at least profit } v \text{ using a subset of } \{I_1,\ldots,I_i\} & \text{if possible} \\ \infty & \text{otherwise} \end{cases}$$

The goal is to compute $\max\{v \mid W(n,v) \leq B\}$.

The corresponding Bellman equation is

$$W(i,v) = \begin{cases} \infty & \text{if } i = 0 \text{ and } v > 0 \\ 0 & \text{if } i \leq 0 \text{ or } b \leq 0 \\ \max\{C,D\} & \text{otherwise} \end{cases}$$

where $C = W(i-1,v)$ and $D = W(i-1,v-v_i) + w_i$.

This algorithm is still pseudo-polynomial but the complexity is now $O(nV)$ where $V = v_1 + \cdots + v_n$. This is more efficient when $V \ll B$.

10.4 FPTAS Approximation for Knapsack

Because 0-1 knapsack is NP-complete, it is also of theoretical interest to find an efficient approximation algorithm. In the case of the knapsack problem, it is possible to find an algorithm that approximates the result within a specific degree on all possible inputs.

The idea behind the algorithm is as follows: The high order bits/digits of the value v can be used to determine an approximate solution. The fewer high order bits we use, the faster the algorithm runs, but also the worse the approximation.

The goal is to scale the values v using a parameter (rounding factor) ε so that a $(1+\varepsilon)$ approximation is obtained with time complexity polynomial in n and $1/\varepsilon$.

More precisely, let $\tilde{v} = \lceil v_i/b \rceil b$. This ensures that the constraint that v must be integer for the second algorithm is satisfied. By rounding the values, we get an approximation $\tilde{v}$ of v that is divisible by b. This means we can divide all values of v by ε and get an equivalent problem. Let $\hat{v} = \tilde{v}/b$. This allows us to reduce the size of v so that it becomes smaller than n (or at least bounded by a polynomial in n).

KNAPSACK-APPROX(ε, S)
1 $b = (\frac{\varepsilon}{2n}) \cdot \max_i v_i$
2 **for** i in S
3 $\quad \tilde{v} = \lceil v_i/b \rceil b$
4 $\quad \hat{v} = \tilde{v}/b$
5 $\quad i.v = \hat{v}$
6 KNAPSACK-V$(S, \max_i\{v_i\})$

where KNAPSACK-V is the second DP algorithm discussed above.

KNAPSACK-APPROX is a **FPTAS** (fully polynomial time approximation scheme) algorithm for the 0-1 knapsack problem.

Definition 10.4.1 — FPTAS and PTAS.
An **FPTAS** (Fully Polynomial Time Approximation Scheme) algorithm is one that is polynomial in the encoding of the input and $1/\varepsilon$.

A **PTAS** (Polynomial Time Approximation Scheme) algorithm is one that is polynomial to the encoding of the algorithm but can have any complexity in terms of $1/\varepsilon$ ($1/\varepsilon$ term may not be polynomial).

Chapter 11 Graph Algorithms Using DP

11.1 Single-Source Shortest Path

We have discussed single-source shortest path when we talked about greedy algorithms.

Previously, we have restricted our discussion to graphs without negative weights/cycles. In this chapter, we will talk about algorithms that can be used to general graphs (even one with negative cycles).

Recall that we store the weight of the path from the source s to each node u at $u.d$. Consider a shortest path p from s to a vertex v. If there is a vertex u immediately preceding v on such path, then the path from s to u must also be a shortest path.

One caveat we must consider is that if there is a negative cycle in the graph, one can keep traversing through that negative cycle and yield a path with lower weight. If this happens, the algorithm can get stuck in a negative cycle and d values may never converge. Hence, we need to put one additional constraint: only consider path within certain path length. Let $\delta_k(s,v)$ denote the shortest path from s to v using at most k edges. Similarly, for a vertex v, let $v.d_k$ be the weight of the path from s to v using at most k edges.

In addition, we will define some notations to aid our discussion. If v is immediately reachable from u via one edge, we write $u \to v$. If v is reachable from u via multiple edges, we write $u \rightsquigarrow v$.

11.1.1 Bellman-Ford

An obvious dynamic programming solution follows from this Bellman equation

$$v.d_k = \begin{cases} 0 & \text{if } k = 0 \text{ and } v = s \\ \infty & \text{if } k = 0 \text{ and } v \neq s \\ \min\{A, B\} & \text{otherwise} \end{cases}$$

where $A = v.d_{k-1}$ and $B = \min\{u.d_{k-1} + w(u,v) \mid (u,v) \in E\}$. To prove the correctness of this equation, we can use induction on the length of the path k.

For a simple shortest path, there are at most $|V|-1$ edges. There will be $O((|V|-1)|V|) \in O(|V|^2)$ because we may need to update the d field of all $|V|$ vertices. Each call takes $O(|E|)$ time in order to evaluate $B = \min\{u.d_{k-1} + w(u,v) \mid (u,v) \in E\}$. Therefore, the overall running time of the algorithm is in $O(|V|^2|E|)$.

The dynamic programming algorithm is a somewhat worse version of the Bellman-Ford algorithm, which runs in $O(|V| \cdot |E|)$ time with space complexity $O(|V| + |E|)$.

This rather unfortunate $O(|V|^2|E|)$ running time partly originates from the fact that we are updating the d field of each v for all $v \in |V|$. But if for every v, we are taking $O(|E|)$ time to examine all the edges to update $v.d$ anyway, we might as well just update all the d fields whenever we run a pass through E. This gives us an algorithm with running time $O(|V| \cdot |E|)$. The $|V|$ comes from the $|V| - 1$ iterations to optimize ("relax") each edge on the simple shortest path of at most $|V| - 1$ edges, and the $|E|$ comes from the pass through E during each of the $|V| - 1$ iterations to update the d fields.

Using the following set of lemmas, we will show that it is safe to do so.

Lemma 11.1.1 — Triangle Inequality. For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Proof. By contradiction.

Suppose the claim does not hold. In particular, let $\delta(s, v) > \delta(s, u) + w(u, v)$. This implies that the shortest path from s to v has a higher weight than the path $s \rightsquigarrow u \to v$. But this contradicts the fact that $\delta(s, v)$ is the shortest path. ■

Lemma 11.1.2 — Upper-bound Property. We always have $v.d \geq \delta(s, v)$ for all verticies $v \in V$, and once $v.d$ is equal to $\delta(s, v)$, it never changes.

Proof. By induction on the number of relaxations.

Base case: $v.d \geq \delta(s, v)$ after initialization since $v.d = \infty$ for all $v \in V - \{s, v\}$, and for $s.d = 0 = \delta(s, s)$.

Inductive step: Consider the relaxation of an arbitrary edge (u, v). Assume that $x.d \geq \delta(s, x)$ for all $x \in V$ prior to relaxation. The relaxation of (u, v) can only affect $v.d$. If $v.d$ is updated, then

$$\begin{aligned} v.d &= u.d + w(u, v) \\ &\geq \delta(s, u) + w(u, v) && \text{inductive hypothesis} \\ &\geq \delta(s, v) && \text{triangle inequality} \end{aligned}$$

The invariant is maintained after relaxing (u, v).

Once $v.d = \delta(s, v)$, $v.d$ cannot decrease because $v.d \geq \delta(s, v)$ always holds and relaxation cannot increase $v.d$. ■

Lemma 11.1.3 — No-path Property. If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Proof. Follows immediately from the upper-bound property. ■

Lemma 11.1.4 — Convergence Property. If $s \rightsquigarrow u \to v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s,u)$ at any time prior to relaxing edge (u,v), then $v.d = \delta(s,v)$ at all times afterward.

Proof. By upper bound property, once $u.d = \delta(s,u)$, it no longer changes. In particular, after relaxing (u,v), we have

$$\begin{aligned} v.d &\leq u.d + w(u,v) && \text{Lemma 24.13, CLRS} \\ &= \delta(s,u) + w(u,v) \\ &= \delta(s,v) && \text{Lemma 24.1, CLRS} \end{aligned}$$

(Note that Lemma 24.1 was proved in the chapter covering Dijkstra's algorithm in my notes on data structures, CSC265 Notes)

By the upper bound property, $v.d \geq \delta(s,v)$. It follows that $v.d = \delta(s,v)$. ■

Lemma 11.1.5 — Path-relaxation Property. If $p = v_0, v_1, \ldots, v_k$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0,v_1), (v_1,v_2), \ldots, (v_{k-1},v_k)$, then $v_k.d = \delta(s,v_k)$. This property holds regardless of other relaxations that occur, including those intermixed with relaxations of edges in p.

Proof. Proof by induction on the ith edge of p that is relaxed.

Base case: $i = 0$. This is before any edge of p have been relaxed. From initialization, $v_0.d = \delta(s,s) = 0$. By upper bound property, $v_0.d$ does not change once it converges to $\delta(s,v_0)$

Inductive step: Assume that $v_{i-1}.d = \delta(s,v_{i-1})$. We relax edge (v_{i-1},v_i). By convergence property, after relaxation of this edge, $v_i.d = \delta(s,v_i)$, and by upper bound property, this equality is maintained thereafter. ■

From the path-relaxation property, we can prove the correctness of Bellman-Ford on a graph without negative-weight cycles.

Theorem 11.1.6 — Correctness of Bellman-Ford (without negative cycle). Let BELLMAN-FORD be run on a weighted directed graph $G = (V,E)$ without negative cycles, with source s and weight function $w : E \to \mathbb{R}$. Then, the algorithm terminates and when it does, $v.d = \delta(s,v)$ for all vertices $v \in V$, from which we can construct a predecessor subgraph G_π that is a shortest path tree.

Proof. We first prove that at termination, $v.d = \delta(s,v)$ for all vertices $v \in V$. If v is not reachable from s, the claim follows from the no-path property. If v is reachable from s, then consider a shortest path $p = v_0, v_1, \ldots, v_k$ where $v_0 = s$ and $v_k = v$. Because the shortest path is simple in a graph without negative cycle, so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the outer loop of the algorithm relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration in the $|V| - 1$ total iterations, is (v_{i-1},v_i). By path-relaxation property, $v.d = v_k.d = \delta(s,v_k) = \delta(s,v)$ even if other relaxation steps are intermixed

within the sequence of relaxations of edges $(v_0, v_1), \ldots, (v_{k-1}, v_k)$. Hence, the claim holds when v is reachable from s.

By the predecessor-subgraph property (Lemma 24.17, CLRS), G_π is a shortest-path tree.

Termination of the algorithm follows from the fact that the loop-counters of both loops are strictly increasing. ■

Let us finalliy examine how Bellman-Ford detects negative cycle.

Theorem 11.1.7 — Correctness of Bellman-Ford (negative cycles). If $v.d$ for some vertex v fails to converge after $|V|-1$ passes, then there exists a negative-weight cycle reachable from s.

Proof. After $|V|-1$ passes, if we find an edge that can be relaxed, it means that the current path from s to some vertex is not simple and vertices are repeated on that path. Since this cyclic path has less weight than any simple path, the cycle must be a negative-weight cycle. ■

The proofs give a clear picture of how the algorithm should look like.

```
BELLMAN-FORD(G = (V, E), w, s)
 1  for v ∈ V − {s}
 2      v.d = ∞
 3      v.parent = NIL
 4  s.d = 0
 5  for i = 0 to |V| − 1
 6      for each edge (u, v) ∈ E
 7          if v.d > u.d + w(u, v)
 8              v.d = u.d + w(u, v)
 9              v.parent = u
10  for each edge (u, v) ∈ E
11      if v.d > u.d + w(u, v)
12          return negative cycle
13  return no negative cycle, G_π
```

Note that G_π is constructed by following the parent pointer (if exists) of each vertex to s.

As we alluded to earlier, this improved version of the Bellman-Ford algorithm runs in $O(|V| \cdot |E|)$ time.

11.2 Maximum Length Path Fails

Bellman-Ford solves the single-source shortest path problem on a general graph efficiently. It is natual to ask if it is possible to find a ***longest simple path*** on scuh general graph. It might be tempting to

simply modify the min dynamic programming function used Bellman-Ford to max. Although this approach works find on a DAG, the algorithm does not guarantee that the resulting path is a simple path when run on a graph with cycles and the algorithm may get stuck in a positive cycle.

This problem is NP-hard, for which no known polynomial-time algorithm exists. A special case of this problem is the Hamiltonian path problem: does a graph $G = (V, E)$ has a simple path of length $|V| - 1$. The Hamiltonian path problem is a variant of the NP-hard ***traveling salesman problem*** (TSP).

11.3 All-Pairs Shortest Paths

11.3.1 Matrix Multiplication

11.3.2 Floyd-Warshall

11.3.3 Johnson's Algorithm

11.4 Traveling Salesman Problem

Chapter 12 Bioinformatics

12.1 Dynamic Programming in Computational Biology

Dynamic programming is one of the most commonly used algorithm design techniques in bioinformatics and computational biology. In this chapter, we discuss several problems of practical interest in computational biology that can be solve using dynamic programming. Dynamic programming is also the foundation of many more powerful algorithms such as BLAST (basic local alignment search tool). BLAST is probably one of the most used tool among biologists. It allows one to use a query string to search against a database of biological sequences, and return sequences that contains the query or are closely related to the query sequence. The core of any such algorithm is sequence alignment, which is solved using dynamic programming.

Bioinformatics is a subject as much about biology as it is about strings. Strings arise naturally as biological sequences. One of the most common thing to do with strings is comparing them. By comparing two strings and locating the common subsequence, we can find conserved protein motifs, infer evolutionary relationships, and even predicting function of newly discovered genes/proteins. We will look at DP algorithms that solve the so-called pairwise sequence alignment problem.

12.2 Longest Common Subsequence

Formally, given a sequence $X = x_1, x_2, \ldots, x_m$, the sequence $Z = z_1, z_2, \ldots, z_k$ is a subsequence of X if there exists a strictly increasing sequence $i_1, i_2, \ldots, i_k$ of indicies such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.

12.3 Edit Distance

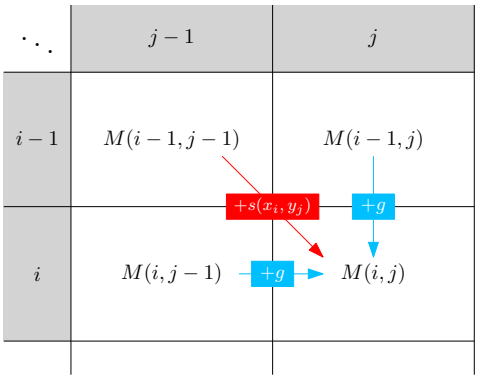


Figure 12.1: The dynamic programming matrix when computing the edit distance. Going diagonally corresponds to a match or substitution; going horizontally and vertically corresponds to insertion and/or deletion.

IV Network Flow

Chapter 13 Ford-Fulkerson

13.1 Flow Network

13.1.1 Definitions

We can interpret a graph as a "flow network" and use it to model the flow of materials such as water through pipes, electricity through wires, or goods through supply chain. Each flow network will have a source where everything originates from and a sink (destination) where everything arrives at. We first formally define a ***flow network***.

Definition 13.1.1 — Flow Network. A ***flow network*** $G = (V, E)$ is a directed network in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$. Additionally, if $(u, v) \in E$, then $(v, u) \notin E$. If $(u, v) \notin E$, then $c(u, v) = 0$.

In particular, in a flow network, there is a ***source*** s and ***sink*** t, and every vertex lies on some path from from s to t. That is, $\forall v \in V$, there exists a path p such that $p = s \rightsquigarrow v \rightsquigarrow t$.

Definition 13.1.2 — Flow. Let $G = (V, E)$ be a flow network with capacity function $c : V \times V \to \mathbb{R}$, and let s be the source and t be the sink of the network. A ***flow*** in G is a function $f : V \times V \to \mathbb{R}$ that satisfies the following properties:

- Capacity constraint: for all $u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$.
- Flow conservation: for all $u \in V - \{s, t\}$, $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$. That is, the flow into u must equal the flow out from u. When $(u, v) \notin E$, $f(u, v) = 0$.

We define the ***value*** of a flow f to be

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

Sometimes, we abuse notations a little bit and omit the summation by writing $f(s, V)$ where V is the set of vertices. Some also uses the notation $f_{in}(v)$ and $f_{out}(v)$ to denote the flow into and out from v, respectively.

■ **Example 13.1 — Example of a Flow Network.** Below is an example of flow network. On each edge $(u, v) \in E$, we write $f(u, v)/c(u, v)$. The slash is simply a separator and does not mean division.

■

The goal of the maximum flow problem is to find a flow f that maximizes the value of the flow $|f|$. This is also equivalent to finding the maximum flow into the sink t.

The max-flow problem is interesting not only because its practical applications, but also its relation to other problems. Many other problems are closely related to the max-flow problem can be polynomial-time reduced to the max-flow problem.

13.1.2 Antiparallel Edges and Multiple Sources/Sinks

In our definition of a flow network, we requires there to be exactly only one source and one sink, and we prohibit antiparallel edges (i.e. if $(u,v) \in E$, then $(v,u) \notin E$).

However, it is easy to convert a graph with multiple sources and/or sinks and antiparallel edges into a flow network that fit our original definition.

Theorem 13.1.1 Suppose that a graph G contains an edge (u,v). We can create a new flow network $G' = (V', E')$ where $V' = V \cup \{x\}$ and $E' = E - \{(u,v)\} \cup \{(u,x),(x,v)\}$. G' is obtained by creating a new vertex x and replacing (u,v) by $(u,x),(v,x)$. We set the capacity $c(u,x) = c(x,v) = c(u,v)$. The maximum flow in G' is the same as the maximum flow in G.

Proof. ■

Theorem 13.1.2 Let G be a graph with multiple source vertices and multiple sink verticies. Let G' be constructed as follows. Suppose G has sources $S = \{s_1, \ldots, s_i\}$ and sinks $T = \{t_1, \ldots, t_j\}$. We add a supersource s and edges (s, s_i) with capacity $c(s, s_i) = \infty$ for all $s_i \in S$. We add a supersink t and edges (t_j, t) for all $t_j \in T$ with capacity $c(t_j, t) = \infty$. More formally, $G' = (V', E')$ such that $V' = V \cup \{s\} \cup \{t\}$ and $E' = E \cup \{(s, s_i) \mid s_i \in S\} \cup \{(t_j, t) \mid t_j \in T\}$. G' has the same maximum flow as G.

Proof. ■

This theorem will become useful later when we look at the maximum bipartite matching problem.

13.2 Ford-Fulkerson Method

Ford-Fulkerson method is a general approach to solve the maximum flow problem. It uses an incremental improve approach by gradually increasing the flow until we reach the maximum. More generally, Ford-Fulkerson can be viewed as a greedy local search algorithm. It is called a method or a scheme because most parts of it can be implemented differently but the general approach remains the same.

The table below summarizes some of the common implementation of the Ford-Fulkerson method. At the end of this chapter, we will discuss the Edmonds-Karp algorithm, and in a later chapter, we will

discuss a different algorithm known as the push-relabel algorithm, which uses a different approach than Ford-Fulkerson.

Algorithm	Implementation	Complexity
Edmonds-Karp (1970)	BFS for finding augmenting path	$O(\|E\|^2\|V\|)$
Dinitz (1970)	BFS + DFS for finding augmenting path	$O(\|V\|^2\|E\|)$
Capacity Scaling (Dinitz, 1972)	Capacity scaling heuristic	$O(\|E\|^2 \log c_{max})$
Dinitz-Gabow (1973)	Capacity scaling heuristic	$O(\|V\|\|E\| \log c_{max})$

Table 13.1: Comparison of multiple algorithms based on the Ford-Fulkerson. c_{max} in capacity scaling denotes the largest capacity in the network.

13.2.1 Idea Behind Ford-Fulkerson

The idea behind Ford-Fulkerson is quite simple. For each iteration of the algorithm, we find a path from s to t, increase the flow along the path until we hit a bottleneck at one of the edges on the path (the edge with the smallest remaining capacity). This is called augmenting flow, and the path is called augmenting path. When augmenting the flow along the augmenting path, we may also want to decrease the flow along each residue (i.e. reverse) edge so that we can "undo" bad augmentation choices. We will show that when the algorithm terminates, it correctly returns the max-flow.

Note that this is more precisely, only the partial correctness. Surprisingly, depending on the implementation and capacities, the algorithm may not terminate. In particular, the Ford-Fulkerson method may fail to terminate if edge capacities are irrational [21]. In his 1993 paper, Zwick showed a minimal example flow network (with 6 vertices and 8 edges) on which Ford-Fulkerson fails to terminate.

FORD-FULKERSON(G,s,t)
1 $f = 0$ for all $(u,v) \in E$
2 **while** there exists an augmenting path p in residue network G_f
3 augment flow f along p
4 **return** f

13.3 Correctness of Ford-Fulkerson

13.3.1 Residue Network and Augmentation

Let us make the idea more concrete by formalizing it and proving the correctness of the Ford-Fulkerson method. We first define what a residue network is.

Given a flow network G, the residue network G_f contains edges with capacities that represent how much more flow we can push through each edge. For each edge (u,v), we will also add backward edges (v,u) to the residue network that can at most cancel out the flow through the edge (u,v).

Definition 13.3.1 — Residue Capacity. Let $G = (V, E)$ be a flow network with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. The ***residue capacity*** $c_f(u, v)$ is defined as

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in E \\ f(v,u) & (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

By assumption, since G is a flow network, $(u, v) \in E$ IFF $(v, u) \notin E$, exactly one case in the definition applies.

Definition 13.3.2 — Residue Network. Let $G = (V, E)$ be a flow network and a flow f. The ***residue network*** of G induced by f, denoted G_f is $G_f = (V, E_f)$ where $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$.

A residue network contains at most twice many edges as the original flow network. If an edge does not exist in G (has zero capacity), it does not exist in G_f either.

Definition 13.3.3 — Augmentation. Let $G = (V, E)$ be a flow network with flow f. G_f is the residue network of G induced by f. Suppose that f' is a flow in G_f. We define the ***augmentation*** of f by f', denoted $f \uparrow f'$, to be a function $(f \uparrow f') : V \times V \to \mathbb{R}$ such that

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Note that because we allow backward edges in a residue network, we need to subtract $f'(v, u)$ when augmenting f and f' on edge (u, v). The notion of pushing flow through a reverse edge is called cancellation.

Lemma 13.3.1 Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residue network of G induced by f, and let f' be a flow in G_f. Then $f \uparrow f'$ is a flow in G and $|f \uparrow f'| = |f| + |f'|$.

Proof. To show that $f \uparrow f'$ is a flow in G, it suffices to show the capacity constraint and flow conservation properties hold.

Capacity constraint: Let $u, v \in V$. Then,
$(f \uparrow f')(u,v) = f(u,v) + f'(u,v) - f'(v,u) \leq f(u,v) + f'(u,v) \leq f(u,v) + c_f(u,v) = f(u,v) + c(u,v) - f(u,v) = c(u,v)$.

Flow conservation: Fix $u \in V$. Let $In(u) = \{v \in V \mid (u,v) \in E\}$ and $Out(u) = \{v \in V \mid (u,v) \in E\}$. That is, $In(u)$ is the set of vertices with edges to u and $Out(u)$ is the set of vertices with edges from u.

$$\sum_{v \in Out(u)} (f \uparrow f')(u,v) - \sum_{v \in In(u)} (f \uparrow f')(v,u) = \sum_{v \in Out(u)} f(u,v) + \sum_{v \in Out(u)} f'(u,v) - \sum_{v \in Out(u)} f'(v,u) - \sum_{v \in In(u)} f(v,u) - \sum_{v \in In(u)} f'(v,u) + \sum_{v \in In(u)} f'(u,v)$$

Regrouping the terms yields
$\sum_{v\in Out(u)} f(u,v) - \sum_{v\in In(u)} f(v,u) + [\sum_{v\in Out(u)} f'(u,v) + \sum_{v\in In(u)} f'(u,v)]$
$- [\sum_{v\in Out(u)} f'(v,u) + \sum_{v\in In(u)} f'(v,u)]$,

which is equivalent to $\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u) + \sum_{v\in V} f'(u,v) - \sum_{v\in V} f'(v,u)$. We can extend the summation from $In(u)$ and $Out(u)$ to V because all the additional terms are 0 by definition of flow and the fact that there $(u,v) \notin E$ for $v \in V - Out(u)$ and $(v,u) \notin E$ for $v \in V - In(u)$.

By flow conservation of f and f' individually,
$\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u) + \sum_{v\in V} f'(u,v) - \sum_{v\in V} f'(v,u) = 0$,
which implies that $\sum_{v\in Out(u)} (f \uparrow f')(u,v) = \sum_{v\in In(u)} (f \uparrow f')(u,v)$.

To show that $|f \uparrow f'| = |f| + |f'|$, set $u = s$ in the previous derivation, and we have

$$\begin{aligned} |f \uparrow f'| &= \sum_{v\in Out(s)} (f \uparrow f')(s,v) - \sum_{v\in In(s)} (f \uparrow f')(v,s) \\ &= \sum_{v\in V} f(s,v) - \sum_{v\in V} f(v,s) + \sum_{v\in V} f'(s,v) - \sum_{v\in V} f'(v,s) \\ &= |f| + |f'|. \end{aligned}$$

■

13.3.2 Augmenting Paths

Definition 13.3.4 — Augmenting Path. Given a flow network $G = (V,E)$ with source s and sink t and flow f, an ***augmenting path*** p is a simple path from s to t in the residue graph G_f of G induced by f.

Definition 13.3.5 — Residue Capacity of an Augmenting Path. Let p be an augmenting path for the flow network G. Then, the ***residue capacity*** of p is defined as the maximum amount by which we can increase the flow on each edge in the p, or more formally,

$$c_f(p) = \min\{c_f(u,v) \mid (u,v) \text{ is on } p\}$$

Definition 13.3.6 — Flow of an Augmenting Path. Let $G = (V,E)$ be a flow network, f be a flow in G, and let p be an augmenting path in G_f. Define the flow of the augmenting path $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u,v) = \begin{cases} c_f(p) & (u,v) \text{ on } p \\ 0 & \text{otherwise} \end{cases}$$

Lemma 13.3.2 The flow f_p on an augmenting path p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

Proof. We show that f_p satisfies the capacity constraint and flow conservation.

Capacity constraint: If (u,v) is on p, then $f_p(u,v) = c_f(p) \leq c_f(u,v)$ by definition. If (u,v) is not on p, then $f_p(u,v) = 0 \leq c_f(p)$.

Flow conservation: Fix $u \in V - \{s,t\}$. First, consider the case where u is on p. Say, edges (w,u) and (u,v) are on p. Then, since p is a simple path and $u \in V - \{s,t\}$, there is exactly one edge going into and out from u. $\sum_{v \in In(u)} f_p(v,u) = f_p(w,u) = c_f(p)$ and $\sum_{v \in Out(u) f_p(u,v)} = f_p(u,v) = c_f(p)$. If (u,v) is not on p, both the flow into u and flow out from u are 0. In both cases, flow conservation is satisfied.

Finally, to show that $|f_p| = c_f(p)$, consider the flow out from s. Since p is a simple path, there is no flow input s, so $|f_p| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) = c_f(p) - 0 = c_f(p)$ ■

Immediately following from Lemma 13.3.2 and 13.3.1, we have the following corollary.

Corollary 13.3.3 Let $G = (V,E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be a flow on the augmenting path defined previously. Then, $f \uparrow f_p$ is a flow in G with value

$$|f \uparrow f_p| = |f| + |f_p| > |f|$$

Proof. By Lemma 13.3.1, $|f_p| = c_f(p) > 0$. By Lemma 13.3.2, $f \uparrow f_p$ is a flow and has value $|f| + |f_p| > |f|$. ■

As a result of Corollary 13.3.3, we know that augmenting a flow with an augmenting path yields a flow with value strictly greater than the flow prior to augmentation.

13.3.3 Max-Flow Min-Cut Theorem

Our analyses so far shows a promising picture of the correctness of the Ford-Fulkerson method. By iteratively augmenting f with an augmenting path p, we gradually increases the flow in the flow network, and when the procedure halts, we should expect to see that f is equal to the maximum flow. The proof utilizes the max-flow min-cut theorem and shows that if there is no augmenting path, there exists a cut in the flow network that implies that f is the max-flow.

Definition 13.3.7 — Cut and Net Flow Across a Cut. A ***cut*** (S,T) of flow network $G = (V,E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the ***net flow*** $f(S,T)$ across the cut (S,T) is defined as

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

The ***capacity of the cut***, denoted $c(S,T)$ is

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

A ***minimum cut*** is a cut whose capacity is minimum over all cuts of the flow network. The definition of a cut and minimum cut will help us prove the important max-flow min-cut theorem. The proof roughly follows the outline shown in Figure 13.1.

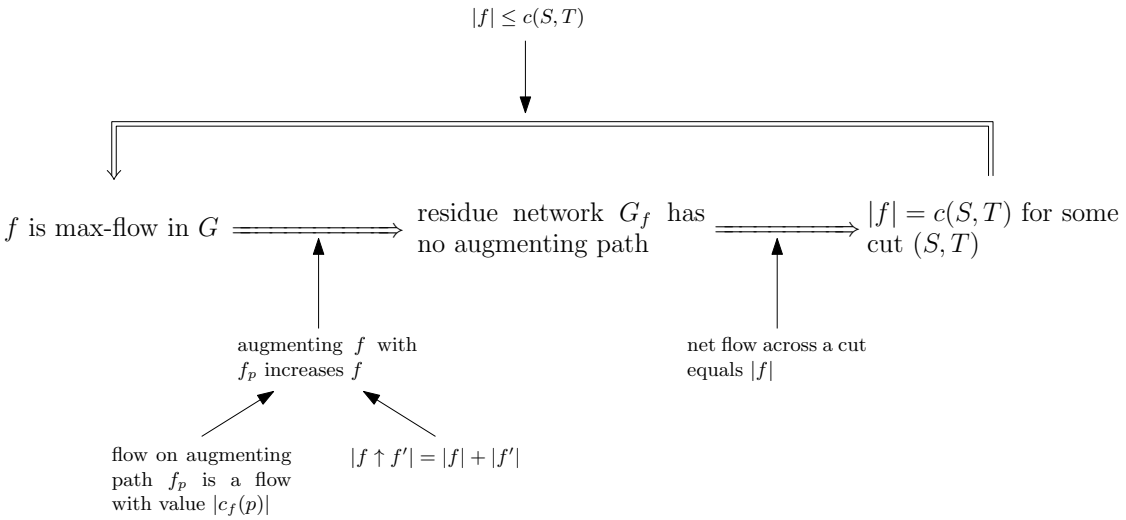


Figure 13.1: Proof outline for the partial correctness of the Ford-Fulkerson method.

Lemma 13.3.4 Let f be a flow in a flow network G with source s and sink t. Let (S,T) be a cut of G. Then, the net flow across the cut (S,T) is $f(S,T) = |f|$.

Proof. Let (S,T) be a cut in G. Consider the following net flows:

- $f(S,T)$: the net flow across the cut (S,T)
- $f(S,V)$: the net flow for edges starting in S
- $f(S,S)$: the net flow within S
- $f(\{s\},V)$: the net flow from s (i.e. $|f|$)
- $f(S-\{s\},V)$: the net flow of edges that start from the partition S but not the source s

Fix $u \in V - \{s,t\}$. By flow conservation, we have $\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) = 0$. This is true for all $u \in V - \{s,t\}$, so

$$\sum_{u \in S-\{s\}} \left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) \right) = 0.$$

This summation can be expanded to

$$\sum_{u \in S-\{s\}} \sum_{v \in V} f(u,v) - \sum_{u \in S-\{s\}} \sum_{v \in V} f(v,u) = 0$$

by linearity. Note this expression is equivalent to $f(S-\{s\},V)$ by definition. So, $f(S-\{s\},V) = 0$.

Now consider $f(S,S)$. By definition, we can rewrite $f(S,S)$ as

$$\sum_{u \in S} \sum_{v \in S} f(u,v) - \sum_{u \in S} \sum_{v \in S} f(v,u) = 0$$

because for all vertices $x, y \in S$, the term $f(x,y)$ appears exactly once in each summation. Hence, $f(S,S) = 0$.

Since $V = S \cup T$ and $S \cap T = \emptyset$,

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) \\ &= f(\{s\},V) + f(S - \{s\},V) \\ &= f(\{s\},V) \\ &= |f| \end{aligned}$$

■

This proof is inspired by the proof presented in the second edition of CLRS. The proof in the third edition employs a more rigorous argument using properties of summation, but I personally find it less intuitive.

A corollary to this lemma shows that cut capacity can be used to bound the value of a flow.

Corollary 13.3.5 The value of any flow f in a flow network G is bounded above by the capacity of any cut of G.

Proof. Let (S,T) be a cut of G and let f be a flow. By Lemma 13.3.4 and capacity constraint,

$$\begin{aligned} |f| = f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\ &\leq \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) && \text{capacity constraint} \\ &= c(S,T) && \text{definition} \end{aligned}$$

■

Now we finally have everything we need to prove the max-flow min-cut theorem. We will follow the proof outline to prove the equivalences (iff).

Theorem 13.3.6 — Max-Flow Min-Cut Theorem. If f is a flow in a flow network $G = (V,E)$ with source s and sink t, then the following conditions are equivalent

1. f is a maximum flow in G
2. The residue network G_f contains no augmenting paths
3. $|f| = c(S,T)$ for some cut (S,T) of G

Proof.

(1) $\Rightarrow$ (2): Suppose, for contradiction, that f is a maximum flow in G, but G_f has a new augmenting path p. Then, by Corollary 13.3.3, the flow obtained by augmenting f and f_p has a strictly greater value and $f \uparrow f_p$ is a flow in G. This contradicts the assumption that f is a maximum flow.

(2) $\Rightarrow$ (3): Assume (2) holds. Then, there is no path from s to t in the residue network G_f. Let $S = \{u \in V \mid \text{there exists a path from } s \text{ to } u \text{ in } G_f\}$, and let $T = V - S$. Clearly, (S,T) is a cut (this can be trivially proved by showing $s \in S$ and $t \notin S$ because there is no path from s to t). We claim that for each $u \in S$ and $v \in T$, $(u,v) \in E$, $f(u,v) = c(u,v)$. To see why this is true, we use proof by contradiction. Suppose the claim is false. By capacity constraint $f(u,v) \leq c(u,v)$. If $f(u,v) \neq c(u,v)$, $f(u,v) < c(u,v)$ and $(u,v) \in E_f$ because it would have positive residue capacity. But then, this means there would be a path from u to v and $v \in S$, contradicting the assumption that $v \in T$. The other case where $(v,u) \in E$ follows from the same argument with u and v swapped.

(3) $\Rightarrow$ (1): Assume (3) holds. Then, there exists a cut (S,T) such that $|f| = c(S,T)$. By Corollary 13.3.5, $|f| \leq c(S,T)$. This implies that every other flow has value less than f, so f is a maximum flow. ■

This theorem shows that if Ford-Fulkerson terminates, then the resulting flow is the maximum flow. The proof of correctness also suggests the following algorithm that elaborates on the Ford-Fulkerson method outline described earlier.

FORD-FULKERSON($G = (V,E), s, t$)

```
1  for each (u,v) ∈ E
2      (u,v).f = 0                                   // initialize flow to 0
3  while there exists a path p from s to t in the residue network G_f
4      c_f(p) = min{c_f(u,v) | (u,v) is on p}        // residue capacity of path p
5      for each (u,v) on p                           // augment flow
6          if (u,v) ∈ E
7              (u,v).f = (u,v).f + c_f(p)
8          else (v,u).f = (v,u).f − c_f(p)
```

This gives us an upper bound on the running time of the Ford-Fulkerson method. Without loss of generality, we can assume that the capacities are integral (otherwise we can rescale them to integers). Then, we know that each iteration of the while loop increases the flow value by at least one unit. Hence, the while loop of Line 3-8 runs at most $|f^*|$ iterations where f^* is the maximum flow. We can find the augmenting path p using either BFS or DFS, which can be done in $O(|E|)$ time. Since we need to find an augmenting path for every iteration of the while loop and the while loop runs at most $|f^*|$ iterations, we can conclude that the running time of the Ford-Fulkerson method is upper bounded by $O(|f^*||E|)$.

A better upper bound on the running time of Ford-Fulkerson's algorithm/method depends on the way we implement the procedure for finding augmenting path p. In the next chapter, we will look at Edmonds-Karp's algorithm, which implements this operation using BFS.

13.4 Irrational Capacity

The Ford-Fulkerson method may fail to terminate on a flow network with irrational capacities. We will present an example of such network, first discovered by Zwick in 1993 [21].

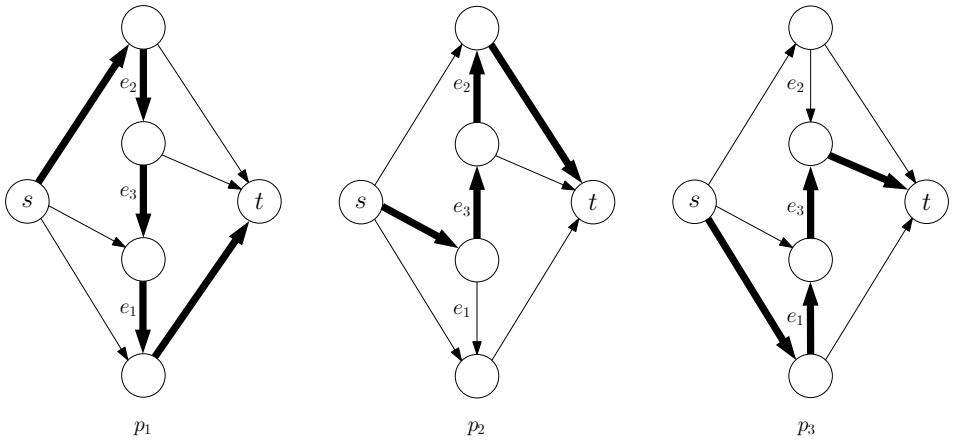


Figure 13.2: 3 augmenting paths for the nonterminating example

Chapter 14 Edmonds-Karp Algorithm

14.1 Edmonds-Karp

In the last chapter, we saw the Ford-Fulkerson method for finding the maximum flow. We have left out one important aspect of the implementation – how to find the augmenting path. The ***Edmonds-Karp algorithm*** finds augmenting paths using breadth-first search (BFS). We will first take a look at the pseudocode for Edmonds-Karp. It follows the same structure as Ford-Fulkerson.

```
EDMONDS-KARP(G = (V,E),s,t)
 1  for each (u,v) ∈ E
 2      (u,v).f = 0                                   // initialize flow to 0
 3  p = BFS(G_f,s,t)          // find shortest path from s to t on residue network
 4  while p ≠ NIL
 5      c_f(p) = min{c_f(u,v) | (u,v) is on p}        // residue capacity of path p
 6      for each (u,v) on p                           // augment flow
 7          if (u,v) ∈ E
 8              (u,v).f = (u,v).f + c_f(p)
 9          else (v,u).f = (v,u).f − c_f(p)
10      p = BFS(G_f,s,t)
```

The correctness of the Edmonds-Karp algorithm follows immediately from the correctness of Ford-Fulkerson.

14.2 Running Time of Edmonds-Karp Algorithm

We will show that the Edmonds-Karp algorithm runs in $O(|V||E|^2)$ time. It is a rather subtle proof involving properties of shortest paths and how augmenting a path on the residue network affects the shortest path from s to t.

Lemma 14.2.1 Let $\delta_f(s,v)$ be the shortest-path distance from s to v in the residue network G_f. After each flow augmentation, $\delta_f(s,v)$ does not decrease for all $v \in V - \{s,t\}$.

Proof. Suppose for contradiction, that there exists some vertex $v \in V - \{s,t\}$ such that $\delta_f(s,v)$ is decreased after certain flow augmentation. Let f be the flow before the augmentation, and f' be the

flow immediately after the augmentation. Let v be the vertex with **minimum** $\delta_{f'}(s,v)$ whose distance was decreased by flow augmentation. Further, let u be the vertex immediately before v on the shortest path from s to v in $G_{f'}$. By our assumption, $\delta_{f'}(s,v) < \delta_f(s,v)$.

Since we have chosen v to be the vertex with minimum $\delta_{f'}(s,v)$ whose flow was decreased, we know that the distance from s to u (the vertex immediately before v on the shortest path from s to v in $G_{f'}$) did not decrease. Hence,

$$\delta_f(s,u) \leq \delta_{f'}(s,u)$$

Claim. $(u,v) \notin G_f$.

We can prove this claim by contradiction. Suppose not. Then, since $\delta_f(s,v)$ is the shortest-path distance from s to v in G_f, by the triangle inequality,

$$\delta_f(s,v) \leq \delta_f(s,u) + 1 \leq \delta_{f'}(s,u) + 1 = \delta_{f'}(s,v)$$

which is a contradiction to our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$.

The only way $(u,v) \notin G_f$ and $(u,v) \in G_{f'}$ after one augmentation step is if the flow through (v,u) is increased during the augmentation (creating the reverse edge (u,v)). Since Edmonds-Karp algorithm uses BFS to select augmenting paths, this implies that (v,u) is on the shortest path selected by the algorithm. But then,

$$\delta_f(s,v) = \delta_f(s,u) - 1 \leq \delta_{f'}(s,u) - 1 = \delta_{f'}(s,v) - 1 - 1$$

which implies that $\delta_f(s,v) < \delta_{f'}(s,v)$, contradicting our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$. ■

We say an edge is ***critical*** on an augmenting path p if $c_f(p) = c_f(u,v)$ (i.e. (u,v) is the bottleneck). Based on our algorithm, during an augmentation of the path p, (u,v) is removed and (v,u) is added to the residue network.

Lemma 14.2.2 Between any two steps in which (u,v) is critical, $\delta(s,u)$ increases by at least 2.

Proof. Suppose (u,v) was cirtical in G_f. So, (u,v) is removed after augmentation. Further, since (u,v) is selected by the algorithm, it must be on the shortest path from s to t, so

$$\delta_f(s,v) = \delta_f(s,u) + 1$$

Once (u,v) is removed, it can only reappear in a residue network if the flow from u to v is decreased, which is only possible when (v,u) is in the network. Let f' be the flow immediately before the augmentation that added (u,v) back. Then, the algorithm must have selected an augmenting path containing (v,u). By Lemma 14.2.1 on v, $\delta_f(s,v) \leq \delta_{f'}(s,v)$. It follows that

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 \geq \delta_f(s,v) + 1 = \delta_f(s,u) + 2$$

Hence, between the two steps in which (u,v) is critical, the distance from s to u increases by at least 2. ■

Corollary 14.2.3 Every edge can become critical at most $|V|/2$ times.

Proof. Immediate from 14.2.2. After the first time an edge (u, v) became critical, the distance of the vertex from s to u increases by at least 2 every time, and there are $|V| - 2$ vertices that can show up on the shortest path from s to u. This can happen at most $(|V| - 2)/2$ times. Along with the first time (u, v) becmoes critical, we have in total at most $|V|/2$ times. ■

Theorem 14.2.4 EDMONDS-KARP runs in $O(|V||E|^2)$ time.

Proof. As shown by Corollary 14.2.3, there can be at most $O(|V||E|)$ augmentations steps because there are $|E|$ edges, and every edge can be critical for at most $|V|/2 \in O(|V|)$ times. Each augmentation step takes $O(|E|)$ using breadth-first search and hence the running time $O(|V||E|^2)$. ■

Chapter 15 Maximum Bipartite Matching and Applications of Network Flow

15.1 Maximum Bipartite Matching

Given a ***bipartite graph*** $G = (V, E)$ where $V = V_1 \cup V_2$ for some $V_1 \cap V_2 = \emptyset$ and $E \subseteq V_1 \times V_2$, a matching is a set of edges going from vertices in V_1 to vertices in V_2. The objective of the ***maximum bipartite matching problem*** is to find a bipartite matching of the maximum cardinality.

There is a natural way to solve this problem using max flow. Given such bipartite graph, we can add a source node s and target node t and for all $u \in V_1$, add edge (s, u) with capacity 1 and for all $v \in V_2$, add edge (v, t) with capacity 1. Assign a capacity of 1 to all other edges already in E (note that technically we need to turn those undirected edges into direct edges going from V_1 to V_2 but the construction for this is trivial so we won't spend time describing it formally).

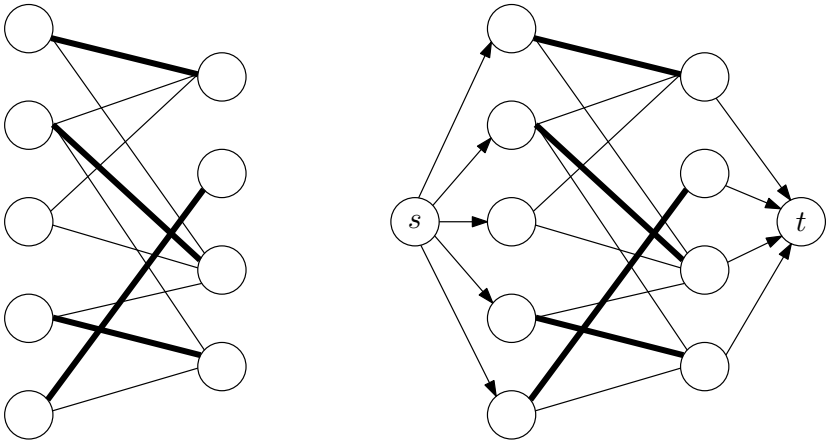


Figure 15.1: Maximum bipartite matching and its reduction to maximum flow problem.

Once we apply this construction to obtain a flow network, we can run any max-flow algorithms and the result gives rise to a matching with the maximum cardinality.

Theorem 15.1.1 — Integrality Theorem. If the capacities in a flow network are integers, then the value of the maximum flow produced by the Ford-Fulkerson method is also an integer.

Proof. By induction on the number of augmentations. ■

Once we have the integrality theorem, we are able to prove the corrctness of the reduction from maxi-

mum bipartite matching to max flow. We do so by showing that there is an one-to-one correspondence between matchings of size k in the original graph and flows of value k in the flow network.

Proof.

(matching $\Rightarrow$ integral flow): Let $M = \{(u_1, v_1), \ldots, (u_k, v_k)\}$ be a matching of size k. We construct a flow f such that for all $i = 1, \ldots, k$, $f(s, u_i) = f(u_i, v_i) = f(v_i, t) = 1$. It is easy to verify that f is indeed a flow (by showing that it satisfies capacity constraint and flow conservation). The value of the flow $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} (v, s) = k$.

(integral flow $\Rightarrow$ matching): Let f be a flow with value k. Let M be the matching such that

$$M = \{(u, v) \in E \mid u \in V_1, v \in V_2, f(u, v) = 1\}.$$

Since a flow of k comes out of s, there must be k edges each with flow of 1 going from s to distinct vertices in V_1. From each vertex in V_1, there must also be k edges with flow 1 going into vertices in V_2 in order to satisfy the flow conservation property of a flow. Hence, $|M| = k$. ■

15.2 Hall's Theorem

We now consider the problem of finding a perfect matching.

Definition 15.2.1 — Perfect Matching. A ***perfect matching*** is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, the neighborhood of X, denoted $N(X)$ is

$$N(X) = \{y \in V \mid (x, y) \in E \text{ for some } x \in X\}$$

The question is: when does a bipartite graph have a perfect matching. This is exactly what Hall's theorem answers. Hall's theorem establishes the necessary and sufficient condition for a perfect matching in bipartite graphs.

Theorem 15.2.1 — Hall's Theorem. Let G be a bipartite graph where $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$. G contains a perfect matching if and only if $|N(S)| \geq |S|$ for all $S \subseteq V_1$.

Proof. By induction on the size of V_1.

Base case: $|V_1| = 1$. The theorem trivially holds.

Inductive step: Let V_1 be a set of vertices such that $|V_1| = k$ for some $k \geq 2$. Assume that for all vertex sets of size smaller than k, the theorem holds. Suppose bipartite graph $G = (V_1 \cup V_2, E)$ satisfies Hall's condition.

Case 1: For all $S \subsetneq V_1$, $|N(S)| \geq |S| + 1$. Let (a, b) be an edge where $a \in V_1$ and $b \in V_2$. Let G' be the subgraph induced by $V - \{a, b\}$. Clearly, $|V_1 - \{a\}| \leq |N(V_1 - \{a\})|$. Here's a more careful argument

of why G' satisfies Hall's condition. Let $S' \subseteq V_1 - \{a\}$, and let $N'(S')$ denote the neighborhood of S' in the graph G' induced by $V - \{a,b\}$. Further, $S' \subseteq V_1 - \{a\} \subset V_1$, so by assumption that G satisfies Hall's condition,

$$|N(S')| - 1 \geq |S'|$$

Since only b has been removed from the induced subgraph G', we also have $|N'(S')| \geq |N(S')| - 1$. It follows that $|N'(S')| \geq |S'|$ and Hall's condition holds for G'.

By inductive hypothesis, G' contains a perfect matching M'. Since (a,b) connects $a \in V_1$ with $b \in V_2$, $\{(a,b)\}$ is a perfect matching. Hence, $M = M' \cup \{(a,b)\}$ is a perfect matching in G.

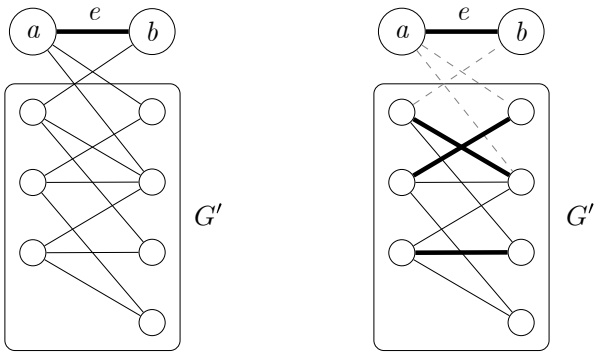


Figure 15.2: Case 1. Note that $N(S')$ can have one fewer vertex than $N'(S')$ (this happens when there is an edge from some vertex in S' to b, shown as dashed lines). $N(S')$ can also have the same number of vertices as $N'(S')$ if there is no edge going from vertices in S' to b. Hence the inequality $|N'(S')| \geq |N(S')| - 1$ holds.

Case 2: There exists some $S \subsetneq V_1$ such that $|S| = |N(S)|$. Since S is a proper subset of V_1, $|S| < |V_1|$. Let G_1 be the subgraph induced by $S \cup N(S)$. S is a proper subset of V_1, and V_1 satisfies Hall's condition. It follows that any subset of S must also be a subset of V_1 and hence satisfies Hall's hypothesis. By induction hypothesis, G_1 has a perfect matching M_1.

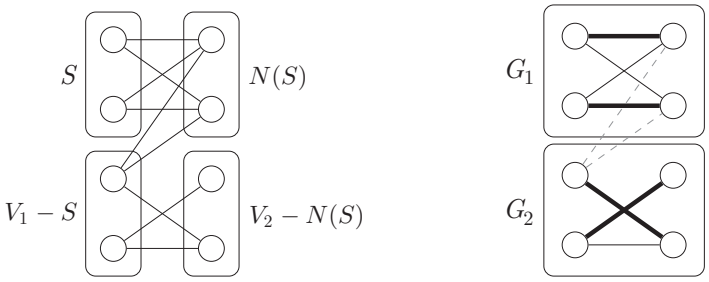


Figure 15.3: Case 2. We partition V_1 into S and $V_1 - S$. The reasoning behind the construction $(S \cup S') \cup (N(S) \cup N(S'))$ when showing that G_2 satisfies Hall's condition is that there any neighboring vertices of S' in G not included in $N_{G_2}(S')$ should be included in $N(S)$, which allows us to derive a contradiction if G_2 does not satisfy Hall's condition.

Let G_2 be the subgraph induced by $(V_1 - S) \cup (V_2 - N(S))$. We claim that G_2 also has a perfect matching. It suffices to show that G_2 satisfies Hall's condition. Suppose not, then there exists some $S' \subseteq (V_1 - S)$ such that $|S'| > |N_{G_2}(S')|$ where $N_{G_2}(S')$ denotes the neighborhood of S' in the subgraph G_2. More precisely, $N_{G_2}(S') = N(S) \cap (V_2 - N(S))$. Consider the subgraph induced by $(S \cup S') \cup (N(S) \cup N(S'))$.

Since $S \cup S'$ is a subset of V_1 and G satisfies Hall's condition

$$|N(S \cup S')| \geq |S \cup S'|$$

Since $N(S)$ and $N_{G_2}(S')$ are disjoint,

$$\begin{aligned} |N(S \cup S')| &= |N(S) \cup N_{G_2}(S')| \\ &= |N(S)| + |N_{G_2}(S')| \\ &= |S| + |N_{G_2}(S')| \\ &< |S| + |S'| \\ &= |S \cup S'| \end{aligned}$$

which contradicts the assumption that G satisfies Hall's condition. So, G_2 must also satisfy Hall's condition and by induction hypothesis, have a perfect matching M_2.

M_1 and M_2 are perfect matchings within their individual subgraphs that are disjoint. Taking the union of M_1 and M_2 yields a perfect matching M in G.

By induction, Hall's theorem holds for bipartite graphs of all sizes. ■

15.3 Disjoint Paths

Given a graph (directed or undirected) $G = (V, E)$, two non-adjacent nodes s and t, we say two paths p and p' from $s \rightsquigarrow t$ is are ***edge-disjoint*** if the two paths do not share a common edge. The ***maximum edge-disjoint paths*** problem wants to find the maximum number of edge-disjoint $s \rightarrow t$ paths. There is a simlilar but different notion of disjoint paths known as ***interally disjoint paths*** or ***vertex disjoint paths*** where instead of defining two paths p and p' as disjoint if they do not share an edge, we say two paths are disjoint only if they don't share a vertex. The problem of finding the maximum-size edge-disjoint paths is often useful in the context of communication networks, where we may want to evaluate the ***fautlt tolerance*** of a network and ensure that there are sufficiently many edge-disjoint paths between two given nodes in the network.

Menger's theorem states that the maximum number of edge-disjoint $s-t$ paths is equal to the minimum number of edges in an $s-t$ cut. We will present the theorem for both edge-disjoint paths and vertex disjoint paths. For the edge-based version, we will present an algorithm using max-flow to find the maximum number of edge-disjoint paths, and Menger's theorem follows immediately from the algorithmic construction and the max flow-min cut theorem.

For the vertex-based version, however, things are a little bit more complicated. It is easy to show the vertex-based version of Menger's theorem on a directed graph since we can follow the same construction as in the edge-based version. For undirected graphs, simply making every undirected edge a bidirectional directed edge is not enough, and we need some additional modifications to the original construction. We will also see another not quite constructive proof of Menger's theorem that does not directly rely on the max flow-min cut theorem.

15.3.1 Edge-Disjoint Paths

We first consider the problem of finding the maximum number of edge-disjoint paths on a directed graph. The construction is really simple. We simply take every edge in the graph and assign a capacity of 1. After this, we run compute the max flow on the graph, and the value of the final flow is the maximum number of edge-disjoint paths. This simple construction allows us to prove Menger's theorem for edge-disjoint paths.

Theorem 15.3.1 — Menger's Theorem (edge). Let $G = (V,E)$ be a graph and let $a,b \in V$ be two distinct vertices. The minimum number of **edges** in a a,b separating cut equals the maximum number of **pairwise edge disjoint** $a \rightsquigarrow b$ paths in G.

Proof.

(edge-disjoint path $\Rightarrow$ flow): Let $\{p_1,\ldots,p_k\}$ be a set of k edge disjoint paths from $s \rightsquigarrow t$. For all $e \in p$ for all $p \in \{p_1,\ldots,p_k\}$, let $f(e) = 1$. Since the paths are edge-disjoint, flow conservation and capacity constraints are clearly satisfied. Running the max flow algorithm on the flow network yields a flow with value of k.

(flow $\Rightarrow$ edge-disjoint path): Let f be a flow of value k. By construction of the flow network, there must be k edges from s with unit flow. For each edge $(s,u) \in E$, there must be an outgoing flow from u in order to satisfy flow conservation. We can inductively build a path from s to t by picking an edge with unit flow. Repeat this for all k outgoing edges from s, and we will have k edge-disjoint paths.

There is a one-to-one correspondence between the number of edge-disjoint paths and the value of the flow. It follows that when the value of the flow is maximized, we also have the maximum number of edge-disjoint paths. And by max flow-min cut, the value of the maximum flow is equal to the capacity of some (S,T) cut. Since all edges have unit capacity, the number of edges separated by this cut is equal to the capacity of the cut. Therefore, the number of edges separated by this minimal cut is equal to the maximum number of edge-disjoint paths. ■

15.3.2 Vertex-Disjoint Paths

Let $G = (V,E)$ be an undirected graph. We construct a directed graph $G' = (V',E')$ where:

$$V' = \{s,t\} \cup \{v^-, v^+ \mid v \in V - \{s,t\}\}$$

$$\begin{aligned} E' = &\{(s,v^-) \mid \{s,v\} \in E\} \cup \{(v^+,t) \mid \{v,t\} \in E\} \\ &\cup \{(u^+,v^-) \mid \{u,v\} \in E\} \\ &\cup \{(v^-,v^+) \mid v \in V\} \end{aligned}$$

$$c(x,y) = \begin{cases} 1 & \text{if } x = v^-,\ y = v^+ \text{ for some } v \in V \\ \infty & \text{otherwise} \end{cases}$$

Notice that every negative $(-)$ vertex has exactly one outgoing edge connecting it to a positive $(+)$ vertex. Conversely, every positive vertex has exactly one edge coming into it. Additionally, edges

between negative and positive vertices have capacity of 1. This makes sure every internal vertex (that is not the source or sink) can only be visited once in the maximum flow.

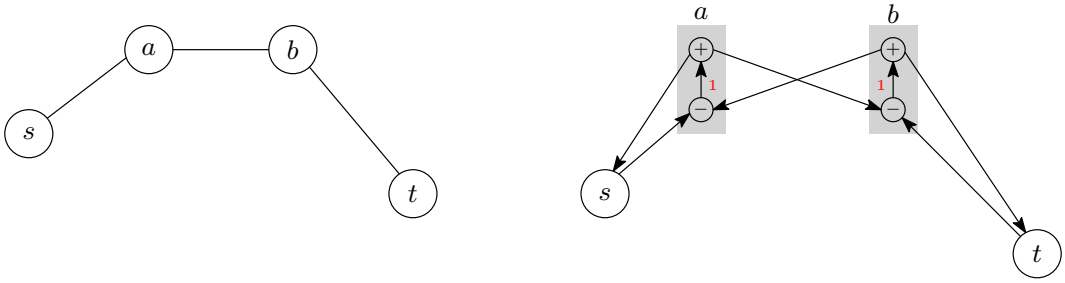


Figure 15.4: The construction converting an undirected graph to a directed graph when finding the maximum number of vertex disjoint paths on an undirected graph. The edges added between the vertices labeled $+$ and $-$ have capacity of 1, and were added to ensure every vertex can only be visited once because as soon as a vertex is visited, the edge become saturated and cannot admit more flow.

This construction converts an undirected graph into a directed graph, and we can prove Menger's theorem using an argument similar to how we proved it for the edge-based case.

Theorem 15.3.2 — Menger's Theorem (vertex). Let $G = (V, E)$ be a graph and let $a, b \in V$ be two **non-adjacent** distinct vertices. The minimum number of **vertices** in a a, b separating cut equals the maximum number of **internally vertex disjoint** $a \rightsquigarrow b$ paths in G.

Proof.

(flow $\Rightarrow$ vertex-disjoint path): Let f be a maximum flow in the flow network constructed as shown above. For each negative vertex $v^- \in V'$, if there is a flow into v^-, then the value of the flow through v^- must be 1 because $c(v^-, v^+) = 1$. This flow must have originated from s and arrived at t. Following this unit flow path gives us a path from s to t. Since there are $|f|$ many unit flow paths from s to t and that the construction prevents a flow of value more than 1 through vertices other than s and t, it follows that there are $|f|$ internally vertex disjoint $s \rightsquigarrow t$ paths in the original graph.

(vertex-disjoint path $\Rightarrow$ flow): Let (S, T) be a minimum cut with capacity k in the flow network G' constructed as shown above. We claim that every edge crossing the cut must be of the form (v^-, v^+). Suppose not. Then, we must have an edge (u, v) crossing the cut with infinite capacity. It follows that $c(S, T) = \infty$, which clearly contradicts the minimality of the cut. Finally, we show that the minimum cut (S, T) corresponds to a cut in the original graph G. Let $U = \{u \in V \mid u^- \in S, u^+ \in T\}$. We have $|U| = c(S, T) = k$. Let $p = sv_1v_2 \dots v_t t$ be an undirected path from s to t. We can construct an equivalent directed path $p' = sv_1^- v_1^+ v_2^- \dots v_t^+ t$ in the flow network G'. The directed path p' must contain an edge going from a negative vertex to a positive vertex, which are the edges crossing the cut (S, T) by the claim proved earlier. Removing the edges crossing the cut disconnects s from t in G'. Then, by construction of U, removing vertices in U will also disconnect s from t in G. Hence U is a cut in G. Since k is the capacity of the minimum cut, by max flow-min cut, we have $|U| = k = \text{max flow} = |f|$. ■

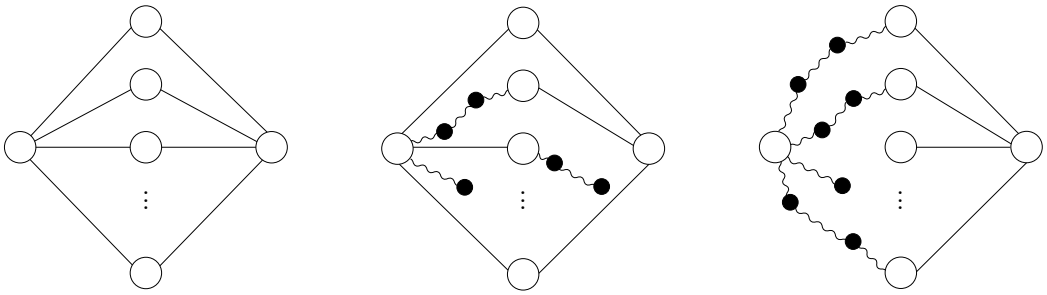


Figure 15.5: The three possible cases in the inductive proof of Menger's theorem (vertex version). Straight lines indicate an edge connecting two vertices, and squiggly lines denote a path containing at least 3 vertices (including the initial and terminal vertices).

Chapter 16 Push-Relabel Algorithm

V

Complexity

VI Linear Programming

Chapter 17 Linear Programming

17.1 Linear Programming Optimization

Linear programming is a commonly used techniques in optimization. In linear programming, we want to optimize (maximize or minimize) a linear function, subject to a set of linear inequalities (called linear constraints).

Definition 17.1.1 — Linear Constraint. Given a set of real numbers $a_1, a_2, \ldots, a_n$ and a set of variables $x_1, x_2, \ldots, x_n$, a ***linear function*** f on those variables is

$$f(x_1, \ldots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j$$

If b is a real number and f is a linear function, then $f(x_1, \ldots, x_n) = b$ is a linear equality ,and $(x_1, \ldots, x_n) \leq b$ and $(x_1, \ldots, x_n) \geq b$ are linear inequalities. Both linear equalities and linear inequalities are called ***linear constraints***.

Geometrically, a linear function forms a line in $\mathbb{R}^n$. A linear inequality forms a half-space in $\mathbb{R}^n$. All possible solutions to a linear program lies in a convex region called the ***feasible region***. If no such region exsits (no solution satisfies all the contraints), we say the optimization problem is ***infeasible***. The function we wish to optimize is called the ***objective function***, and the value of the objective function evaluated at a particular point is called the ***objective value*** at that point. The goal of the linear programming problem is to find a feasible solution that maximizes/minimizes the objective value.

Theorem 17.1.1 — Feasible Region for Linear Programming is Convex. Let $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ be two feasible solutions to the linear programming problem, satisfying some particular constraints. Then, for all $\lambda \in [0,1]$, $\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2$ is also a solution that satisfies the same constraints.

Proof. Without loss of generality, suppose that the constraints are given as a system of linear equations $\mathbf{Ax} = \mathbf{b}$. Let $\lambda \in [0,1]$ be arbitrary.

$$\begin{aligned}\mathbf{A}(\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) &= \lambda\mathbf{Ax}_1 + \mathbf{Ax}_2 - \lambda\mathbf{Ax}_2 \\ &= \lambda\mathbf{b} + \mathbf{b} - \lambda\mathbf{b} \\ &= \mathbf{b}\end{aligned}$$

Hence, $\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2$ is also a solution to the system of linear equation and thus is a feasible solution. ■

Figure 17.1 shows a geometric representation of a linear programming problem with the objective function $x_1 + x_2$.

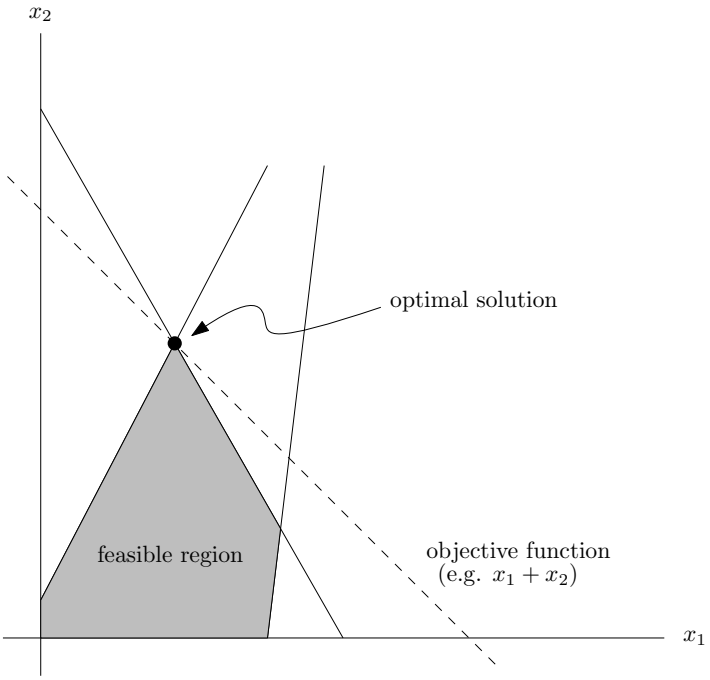


Figure 17.1: A linear programming problem with 2 variables, 4 linear constraints (solid lines), and objective function $x_1 + x_2$ (dashed line).

17.2 Standard and Slack Forms

17.2.1 Standard Form

In standard form, we are given n real numbers $c_1, \ldots, c_n$, m real numbers $b_1, \ldots, b_m$, and mn real numbers a_{ij} for $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$. We wish to

Maximize $\sum_{j=1}^{n} c_j x_j$
Subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_1 \quad \text{for } i = 1, \ldots, m$$

$$x_j \geq 0 \quad \text{for } j = 1, \ldots, n$$

or, equivalently, in vectorized form

Maximize $\mathbf{c}^\top \mathbf{x}$
Subject to $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$,

where $\mathbf{c}$ is an n-vector, $\mathbf{x}$ is an n-vector, $\mathbf{b}$ is an m-vector, and $\mathbf{A}$ is an $m \times n$ matrix. This can also be written in as a tuple $(\mathbf{A}, \mathbf{b}, \mathbf{c})$ as a shorthand.

This vectorized form is often used in machine learning because vectorized operations can be carried out more quickly using the GPU.

17.2.2 Conversion From Non-standard Form to Standard Form

A linear programming problem might not be in standard form, but it is easy to convert from non-standard form to standard form.

If

- the objective is minimization rather than maximization $\Rightarrow$ flip the sign of coefficients;
- there are variables without nonnegativity constraints $\Rightarrow$ say a variable x_j does not have nonnegativity constraints, replace x_j with $x'_j - x''_j$ and add constraints $x'_j \geq 0$ and $x''_j \geq 0$;
- the constraints are equality constraints rather than $\leq \Rightarrow$ replace the constraint with two non-strict inequality constraints ($\leq$ and $\geq$);
- the constraints are in the opposite direction ($\geq$ instead of $\leq$) $\Rightarrow$ multiply both sides by -1

17.2.3 Slack Form

Maximize $z = v + \sum_{j=1}^{n} c_j x_j$

Subject to

$$s_i = b_1 - \sum_{j=1}^{n} a_{ij} x_j \qquad \text{for } i = 1, \ldots, m$$

$$x_j, s_i \geq 0 \qquad \text{for } j = 1, \ldots, n \text{ and } i = 1, \ldots, m$$

or, equivalently, in vectorized form

Maximize $z = v + \mathbf{c}^\top \mathbf{x}$
Subject to $\mathbf{s} = \mathbf{b} - \mathbf{A}\mathbf{x}$ and $\mathbf{x}, \mathbf{s} \geq \mathbf{0}$.

We call the variables on the left-hand side of the equality constraints the ***basic variables***, and the ones on the right-hand side the ***non-basic variables***.

Similarly, a slack from can be concisely defined by a tuple $(N, B, \mathbf{A}, \mathbf{b}, \mathbf{c}, v)$, where N is the set of indices of the non-basic variables, B is the set of indices of the basic variables such that $|N| = n$, $|B| = m$ and $N \cup B = \{1, 2, \ldots, n+m\}$.

It is called the slack from because the variable s_i (or in vectorized form $\mathbf{s}$) measures remaining "slack" or difference between two sides of the original inequality constraints. This is related to the simplex

algorithm where we increase the non-basic variables as much as possible until we hit a bottleneck – having no more slack for that non-basic variable.

17.3 Geometry of Linear Programming

Consider the following system of inequalities

$$\begin{aligned} x_1 + x_2 + x_3 &\leq 1 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

This is the type of inequalities that we often encounter in linear programming problems. The points that satisfy these inequalities forms a 3D polytope.

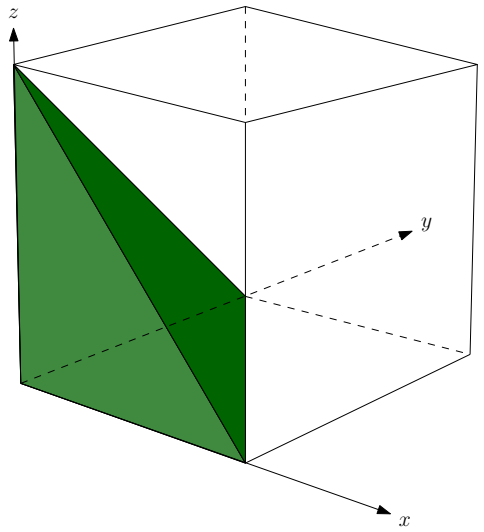


Figure 17.2: A 3D polytope.

17.3.1 Hyperplane and Halfspace

In general, the set $\{x \in \mathbb{R}^n \mid a^\top x = b\}$ is called a hyperplane. For all lines in the ***hyperplane*** $H = \{x \in \mathbb{R}^n \mid a^\top = b\}$ that intersects the line $\ell = \{ta \mid t \in \mathbb{R}\}$, the line is also perpendicular to ℓ. As a simple example, consider the two-dimensional case where $a = [1 \quad -1]^\top$.

The set $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}^\top \mathbf{x} \leq b\}$ is called a ***halfspace***, and $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a}^\top \mathbf{x} = b\}$ is called the ***supporting hyperplane***.

17.3.2 Polyhedron and Polytope

A set that is the intersection of halfspaces is called a ***polyhedron***. A polyhedron can be defined as $\{x \in \mathbb{R}^n \mid \mathbf{Ax} \leq \mathbf{b}\}$ for some matrix $\mathbf{A}$ and vector $\mathbf{b}$.

A polyhedron $P \subseteq \mathbb{R}^n$ is unbounded if there exists a point $\mathbf{x} \in P$ and a direction $\mathbf{v} \in \mathbb{R}^n$ such that for every $t \geq 0$, $\mathbf{x} + t\mathbf{v} \in P$. If a polyhedron is bounded, we call it a ***polytope***.

A ***face*** of a polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ is a set $F = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \leq b\} \cap \{\mathbf{x} \in \mathbb{R}^n \mid \forall i \in S.\, \mathbf{a}_i\mathbf{x} = \mathbf{b}_i\}$ where $\mathbf{a}_i$ is a row vector denoting the ith row of matrix $\mathbf{A}$, and S is some subset of the rows of $\mathbf{A}$. Intuitively, a face of a polyhedron is just one of its surface.

Assuming that $F \neq \emptyset$, we define S_F to be the set of all $i \in S$ such that $\mathbf{a}_i\mathbf{x} = \mathbf{b}_i$ for all $\mathbf{x} \in F$ (we can intutively think this as the rows of constraints that created the face F). Let $\mathbf{A}_F$ be the submatrix of $\mathbf{A}$ containing rows of A indexed by S_F. If the rank of $\mathbf{A}_F$ is $n - j$, we say F is a face of ***dimension*** j, or a j-face. The dimension of a polytope can be defined similarly. Faces of dimension one lower than the dimension of P are called ***facets***, faces of dimension 1 are called ***edges***, faces of dimension 0 are called ***vertices***.

17.3.3 Convex Hull

A polytope can be determined by its vertices.

Definition 17.3.1 — Convex Hull. The ***convex hull*** of the points $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^n$ is defined as the set

$$\operatorname{conv}\{\mathbf{v}_1, \ldots, \mathbf{v}_n\} = \{\lambda_1\mathbf{v}_1 + \cdots + \lambda_n\mathbf{v}_n \mid \forall i \in \{1, \ldots, n\}.\, \lambda_i \geq 0,\ \lambda_1 + \cdots + \lambda_n = 1\}$$

Theorem 17.3.1 A polytope P with vertices $\mathbf{v}_1 \ldots \mathbf{v}_n$ is equal to the convex hull of the vertices.

$$P = \operatorname{conv}\{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$$

Proof. TODO. (by showing $\operatorname{conv}\{\mathbf{v}_1, \ldots, \mathbf{v}_n\} \subseteq P$ and $P \subseteq \operatorname{conv}\{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$) ■

An important implication of Theorem 17.3.1 is that we can always find an optimal solution to a linear program at a vertex of the feasible region. This will become the basis of the simplex algorithm in the coming chapter.

Theorem 17.3.2 — Foundamental Theorem of Linear Programming. If $\max\{\mathbf{c}^\top\mathbf{x} \in \mathbb{R} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ is a linear program whose feasible region $\{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ is a polytope, then the linear program has an optimal solution at a vertex of P.

Proof. Let $\mathbf{x}$ be an optimal solution to the linear program. By Theorem 17.3.1, $\mathbf{x}$ can be expressed as a convex combination.

$$\lambda_1\mathbf{v}_1 + \cdots + \lambda_n\mathbf{v}_n \quad \forall i \in \{1, \ldots, n\}.\, \lambda_i \geq 0,\ \lambda_1 + \cdots + \lambda_n = 1$$

where $\mathbf{v}_1, \ldots, \mathbf{v}_n$ are the vertices of P. It follows by substitution that

$$\mathbf{c}^\top\mathbf{x} = \mathbf{c}^\top\lambda_1\mathbf{v}_1 + \cdots + \mathbf{c}^\top\lambda_n\mathbf{v}_n \leq \lambda_1 \max_{i=1}^{n}(\mathbf{c}^\top\mathbf{v}_i) + \cdots + \lambda_n \max_{i=1}^{n}(\mathbf{c}^\top\mathbf{v}_i) = \max_{i=1}^{n}(\mathbf{c}^\top\mathbf{v}_i)$$

Therefore, a maximum of $\mathbf{c}^\top \mathbf{v}_i$ is also a maximum of $\mathbf{c}^\top \mathbf{x}$. ■

Chapter 18 Simplex Algorithm for LP

18.1 Geometric Intuition

The geometry behind the simplex algorithm is actually quite intuitive, especially after we have proved that there is always at least one optimal solution lying on a vertex of the feasible region. The simplex algorithm starts at an initial vertex, follows the edges of the polytope to neighboring vertices, and checks each vertices along the way. The algorithm terminates when the objective function cannot be improved anymore.

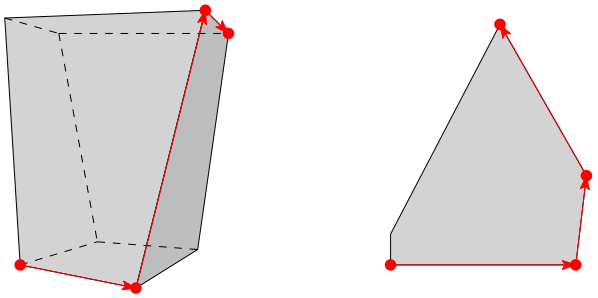


Figure 18.1: The geometric intuition behinds the simplex algorithm. The figure shows the simplex algorithm trying to find the optima on a 3-dimensional and 2-dimensional polytope.

18.2 Pivoting

The way the simplex algorithm navigates through the polytope and jumps between vertices in through an operation called ***pivoting***.

During pivoting, we select a non-basic variable x_e as the ***entering variable***, and a basic variable x_l as the ***leaving variable***. The pivot operation switches the roles of the entering and leaving variable. x_e becomes a new basic variable, and x_l becomes a non-basic variable. Because of this, we also need to update the expression of other basic variables as well as the objective function. Geometrically, this corresponds to following an edge of the feasible to a neighboring vertex.

Pivoting not only affects the entering and leaving variables. Because it changes the roles of the variables, we need to update the entire linear program so that x_e no longer appears as a non-basic variable in any expressions. Similarly, we also want x_l to appear as a non-basic variable in every expressions. Such updates also need to be applied to the objective function.

18.2.1 Update Coefficient Matrix for the Entering Variable

Suppose we have selected x_e as the entering variable and x_l as the leaving variable. Currently, x_e is a non-basic variable, and x_l is a basic variable where

$$x_l = b_l - a_{l1}x_1 - \cdots - a_{lj}x_j - a_{le}x_e$$

Move the entering variable to the left-hand side and move the leaving variable to the right-hand side.

$$a_{le}x_e = b_l - a_{l1}x_1 - \cdots - a_{lj}x_j - x_l$$

Divide both sides by a_{le}, the coefficient of the entering variable.

$$x_e = \frac{b_l}{a_{le}} - \frac{a_{l1}}{a_{le}}x_1 - \cdots - \frac{a_{lj}}{a_{le}}x_j - \frac{1}{a_{le}}x_l$$

Now, x_e has become a basic variable, and x_l has become a non-basic variable.

18.2.2 Update Coefficient Matrix for Other Basic Variables

Since we have made x_e a basic variable, expressed in terms of the non-basic variables, we need to update the expressions of the remaining basic variables to reflect this change.

Consider the ith basic variable x_i, with the expression

$$x_i = b_1 - a_{i1}x_1 - \cdots - a_{ij}x_j - a_{ie}x_e$$

We can substitute x_e with its new expression in terms of the non-basic variables

$$x_i = b_1 - a_{i1}x_1 - \cdots - a_{ij}x_j - a_{ie}\left(\frac{b_l}{a_{le}} - \frac{a_{l1}}{a_{le}}x_1 - \cdots - \frac{a_{lj}}{a_{le}}x_j - \frac{1}{a_{le}}x_l\right)$$

Combine like terms.

$$x_i = \left(b_i - a_{ie}\frac{b_l}{a_{le}}\right) - \left(a_{i1} - a_{ie}\frac{a_{l1}}{a_{le}}\right)x_1 - \cdots - \left(a_{ij} - a_{ie}\frac{a_{lj}}{a_{le}}\right)x_j - \left(-a_{ie}\frac{1}{a_{le}}\right)x_l$$

18.2.3 Update Objective Function

The objective function

$$z = v + \sum_{j=1}^{n} c_j x_j$$

also needs to be updated.

In particular, we will substitute x_e with its new expression in terms of the non-basic variables. This is similar to how we update the basic variables.

$$z = v + c_1x_1 + \cdots + c_jx_j + c_e\left(\frac{b_l}{a_{le}} - \frac{a_{l1}}{a_{le}}x_1 - \cdots - \frac{a_{lj}}{a_{le}}x_j - \frac{1}{a_{le}}x_l\right)$$

Again, we combine like terms, yielding

$$z = \left(v + c_e\frac{b_l}{a_{le}}\right) + \cdots + \left(c_j - c_e\frac{a_{lj}}{a_{le}}\right)x_j + \left(-c_e\frac{1}{a_{le}}\right)x_l$$

18.2.4 Pivoting Algorithm

Based on our derivation above, we can implement the pivot operation as follows. The procedure PIVOT takes in the tuple (N, B, A, b, c, v) representing a linear program in slack form, along with the index of the leaving and entering variables l and e, and outputs a new tuple $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$.

Recall that N is the set of indices of the non-basic variables, B is the set of indices of the basic variables, A is the coefficient matrix, b and c are column vectors. In the algorithm, we assume we can access the i, j-entry of a matrix A using a_{ij} representing the linear program after pivoting.

PIVOT(N, B, A, b, c, v, l, e)
1 // Initialize $m \times n$ matrix
2 $\hat{A} = [[]]$
3 // Calculate coefficients for new basic variable x_e
4 $\hat{b}_e = b_l / a_{le}$
5 **for** $j \in N - \{e\}$
6 $\quad \hat{a}_{ej} = a_{lj} / a_{le}$
7 $\hat{a}_{el} = 1 / a_{le}$
8 // Calculate coefficients for the remaining constraints
9 **for** $i \in B - \{l\}$
10 $\quad \hat{b}_i = b_i - a_{ie}\hat{b}_e$
11 $\quad$ **for** $j \in N - \{e\}$
12 $\quad\quad \hat{a}_{ij} = a_{ij} - a_{ie}\hat{a}_{ej}$
13 $\quad \hat{a}_{il} = -a_{ie}\hat{a}_{el}$
14 // Calculate the new objective function
15 $\hat{v} = v + c_e\hat{b}_e$
16 **for** $j \in N - \{e\}$
17 $\quad \hat{c}_j = c_j - c_e\hat{a}_{ej}$
18 $\hat{c}_l = -c_e\hat{a}_{el}$
19 // Update new sets of basic and non-basic variables
20 $\hat{N} = N - \{e\} \cup \{l\}$
21 $\hat{B} = B - \{l\} \cup \{e\}$
22 **return** $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$

18.3 Simplex Algorithm

18.3.1 Basic Feasible Solution

The simplex algorithm follows the intuitive that we have developed throughout the chapter. We start from a vertex (say, the origin) keep moving to a neighboring vertex if such move can increase the value of the objective function. We have seen that this can be done through the pivot operation. To make this idea more concrete, let's define a basic solution and a basic feasible solution.

Definition 18.3.1 — Basic Solution. A solution $\mathbf{x}$ of $\mathbf{Ax} = \mathbf{b}$ is a ***basic solution*** if the set $\{\mathbf{a}_i \mid x_i \neq 0\}$ (the columns of A corresponding to a non-zero variable) is linearly independent. Equivalently, this means that there are n linearly independent constraints.

Definition 18.3.2 — Basic Feasible Solution. A basic solution that is also a feasible solution is called a basic feasible solution.

In slack form, a basic solution can be obtained by setting all non-basic variables to 0.

The goal of the simplex algorithm is to, in each iteration, reformulate the linear program so that the basic solution leads to a greater objective value. This is done by selecting the tightest basic variable as the leaving variable, a non-basic variable whose coefficient in the objective function is positive as the entering variable, and performing the pivot operation.

18.3.2 Example of Simplex Algorithm

As an example, let's examine the linear program on page 865 of CLRS.

$$\begin{aligned}
\text{maximize} \quad & 3x_1 + x_2 + 2x_3 \\
\text{subject to} \quad & x_1 + x_2 + 3x_3 \leq 30 \\
& 2x_1 + 2x_2 + 5x_3 \leq 24 \\
& 4x_1 + x_2 + 2x_3 \leq 36 \\
& x_1, x_2, x_3 \geq 0
\end{aligned}$$

Convert it to slack form

$$\begin{aligned}
\text{maximize} \quad & z = 3x_1 + x_2 + 2x_3 \\
\text{such that} \quad & x_4 = 30 - x_1 - x_2 - 3x_3 \\
& x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \\
& x_6 = 36 - 4x_1 - x_2 - 2x_3 \\
& x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{aligned}$$

Set all non-basic variables x_1, x_2, x_3 to 0 and obtain the basic solution $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 30, x_5 = 24, x_6 = 36$. The objective value for this basic solution is 0. (Starts from the origin of the feasible region.)

Select x_1 as entering variable because it has a positive coefficient in the objective function. The tightest constraint is x_6 since x_1 can only increase by 9 while maintaining $x_6 \geq 0$. Perform pivot operation, and

we get

$$\begin{aligned}
\text{maximize} \quad & z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
\text{such that} \quad & x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
& x_4 = 21 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
& x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \\
& x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{aligned}$$

The basic solution is currently $x_1 = 9, x_2 = 0, x_3 = 0, x_4 = 21, x_5 = 6, x_6 = 0$ with $z = 27$. Select x_3 as the next entering variable. x_5 has the tightest constraint, so x_5 is the leaving variable. Perform pivot operation.

$$\begin{aligned}
\text{maximize} \quad & z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
\text{such that} \quad & x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
& x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
& x_4 = \frac{69}{4} + \frac{3x_2}{16} - \frac{5x_5}{8} - \frac{x_6}{16} \\
& x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{aligned}$$

The basic solution is currently $x_1 = \frac{33}{4}, x_2 = 0, x_3 = \frac{3}{2}, x_4 = \frac{69}{4}, x_5 = 0, x_6 = 0$ with $z = \frac{111}{4}$. Select x_2 as the next entering variable as it is the only variable with a positive coefficient in the objective function. x_3 has the tightest constraint while x_4 puts no constraint on how much we can increase x_2. x_3 will be the next leaving variable.

$$\begin{aligned}
\text{maximize} \quad & z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
\text{such that} \quad & x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
& x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} \frac{x_6}{3} \\
& x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} + 0x_6 \\
& x_1, x_2, x_3, x_4, x_5, x_6 \geq 0
\end{aligned}$$

Observe that at this stage, none of the nonbasic variables in the objective function has positive coefficient, so it cannot be optimized anymore. The optimal solution to the original linear program is $x_1 = 8, x_2 = 4, x_3 = 0$ with $z = 28$.

Going back to our geometric intuition for the simplex algorithm, the algorithm can be understood as follows. We start from the origin $(0,0,0)$. Increase x_1 until it hits the barrier set forth by x_6 (tightest constraint). Increase x_3 until it hits the barrier set forth by x_5 (tightest constraint). Increase x_2 until it hits the barrier by x_3 (tightest constraint). Indeed, just as described in our geometric interpretation, the simplex algorithm travels along the edges of the feasible region to neighboring vertices and finds an optimal solution at one of vertices of the feasible region. See Figure 18.2.

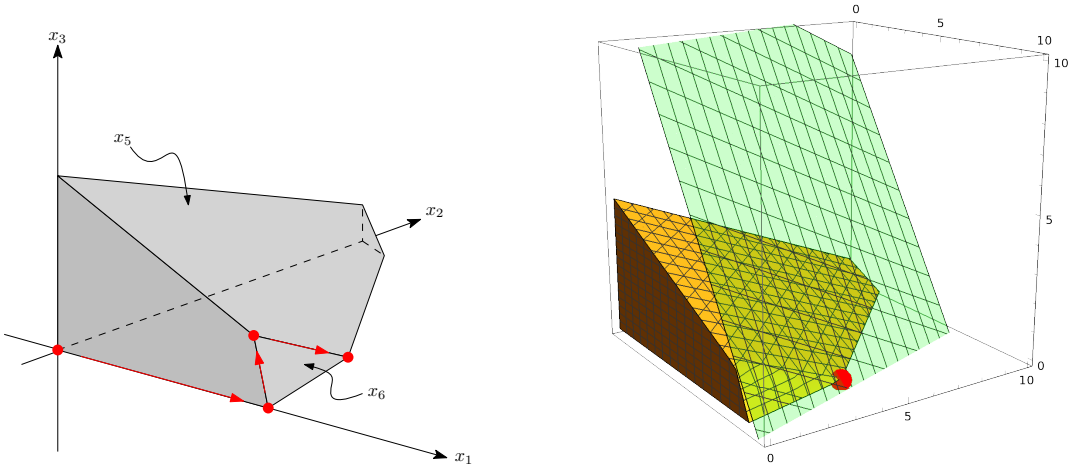


Figure 18.2: Running the simplex algorithm on the example linear program. The figure on the right shows the feasible region in orange with the objective function as a green plane, and the optimum as a red dot in the graph.

18.3.3 Implementing the Simplex Algorithm

SIMPLEX(A,b,c)
1 $(N,B,A,b,c,v) =$ INIT-SIMPLEX(A,b,c) // init and convert to slack form
2 $\Delta = []$ // vector of length m
3 **while** $\exists j \in N.c_j > 0$
4 // break ties by choosing variable with smallest index
5 $e = \{e \in N \mid c_e > 0\}[0]$
6 **for** $i \in B$
7 **if** $a_{ie} > 0$
8 $\Delta_i = b_i/a_{ie}$
9 **else** $\Delta_i = \infty$
10 // find tightest constraint, break ties by choosing variable with smallest index
11 $l = \text{argmin}_{i \in B}\{\Delta_i\}$
12 **if** Δ_i == ∞
13 **return** unbounded
14 **else** $(N,B,A,b,c,v) =$ PIVOT(N,B,A,b,c,v,l,e) // perform pivot
15 **for** $i = 1$ **to** n
16 **if** $i \in B$
17 $\bar{x}_i = b_i$
18 **else** $\bar{x}_i = 0$
19 **return** $(\bar{x}_1,\ldots,\bar{x}_n)$

18.3.4 Finding the Tightest Constraint

The tightest constraint can be easily identified using a method known as the ***minimum ratio test***. Once we have selected a non-basic variable as entering variable, say x_e, we iterate over the basic variables, and select the ith basic variable that minimizes b_i/a_{ie} if a_{ie} is negative. If a_{ie} is positive, then the

constraint does not limit how much we can increase x_e. In more concise notation,

$$l = \operatorname*{argmin}_{i \in \{b \in B | a_{be} > 0\}} \frac{b_i}{a_{ie}}.$$

If there are multiple variables with the same smallest b_i/a_{ie}, we break ties by choosing the one with the smallest index. This rule is known as ***Bland's rule***. The overall idea of the minimum ratio test should be quite intuitive. Consider the expression of a basic variable with nonnegativity constraint of the form

$$x_i = b_i - a_{i1}x_1 - \cdots - a_{ie}x_e \geq 0$$

x_e is bounded by b_i/a_{ie}, so to find the tightest constraint on how much we can increase x_e, we want to minimize the ratio b_i/a_{ie}.

18.4 Time Complexity of the Simplex Algorithm

At any given point during the execution of the simplex algorithm, there are m basic variables. It can be shown that the set B of basic variables uniquely determines a slack form. There are $n+m$ variables. Hence, there are at most $\binom{n+m}{m}$ unique slack forms. If the algorithm terminates (which can be achieved using Bland's rule to prevent cycling), it terminates within $\binom{n+m}{m}$ iterations. Otherwise, it does not terminate and we say the algorithm ***cycles***. Cycling is usually due to ***degeneracy***, in which the simplex algorithm fails to change the objective value. Each iteration of the simplex algorithm takes $O(mn)$ time due to the PIVOT operation.

In the worst case, the simplex algorithm has a time complexity of $\Theta(2^d)$ where d is the dimension of the feasible region, and Klee & Minty proved the upper bound on worst-case time complexity and provided an example linear program, now known as the ***Klee-Minty cube*** on which the algorithm runs in exponential time. However, in practice, the simplex algorithm is surprisingly efficient, and it often runs in polynomial time. It often beats asymptotically more efficient algorithms like the ellipsoid algorithm in practice. This phenomenon is explained by Spielman & Teng in their paper, *Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time*.

Chapter 19 Duality

19.1 Proving Optimality

Given a maximization linear program, it is easy to show that the optimal objective value has a certain lower bound by providing a feasible solution that achieves the value. However, this is not sufficient for proving the upper bound of the given linear program. A single feasible solution is not enough because the optimal solution is the maximum over all feasible solutions.

Chapter 20 Linear Programming Problems

Chapter 21 Integer Programming

Chapter 22 Ellipsoid Algorithm for LP

VII Approximation Algorithms

VIII Randomized Algorithms

IX

Algorithms on Strings

Appendix

Commonly Used Axioms & Theorems

Rules of Inference

Axiom 1 — Modus Ponens. $(P \wedge (P \text{ IMPLIES } Q)) \text{ IMPLIES } Q$

Axiom 2 — Modus Tollens. $(\neg Q \wedge (P \text{ IMPLIES } Q)) \text{ IMPLIES } \neg P$

Axiom 3 — Hypothetical Syllogism (transitivity).

$((P \text{ IMPLIES } Q) \wedge (Q \text{ IMPLIES } R)) \text{ IMPLIES } (P \text{ IMPLIES } R)$

Axiom 4 — Disjunctive Syllogism. $((P \vee Q) \wedge \neg P) \text{ IMPLIES } Q$

Axiom 5 — Addition. $P \text{ IMPLIES } (P \vee Q)$

Axiom 6 — Simplification. $(P \wedge Q) \text{ IMPLIES } P$

Axiom 7 — Conjunction. $((P) \wedge (Q)) \text{ IMPLIES } (P \wedge Q)$

Axiom 8 — Resolution. $((P \vee Q) \wedge (\neg P \vee R))$ IMPLIES $(Q \vee R)$

Laws of Logic

Axiom 9 — Implication Law. $(P$ IMPLIES $Q) \equiv (\neg P \vee Q)$

Axiom 10 — Distributive Law.

$$(P \wedge (Q \vee R)) \equiv ((P \wedge Q) \vee (P \wedge R))$$
$$(P \vee (Q \wedge R)) \equiv ((P \vee Q) \wedge (P \vee R))$$

Axiom 11 — De Morgan's Law.

$$\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$$
$$\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$$

Axiom 12 — Absorption Law.

$$(P \vee (P \wedge Q)) \equiv P$$
$$(P \wedge (P \vee Q)) \equiv P$$

Axiom 13 — Commutativity of AND. $A \wedge B \equiv B \wedge A$

Axiom 14 — Associativity of AND. $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$

Axiom 15 — Identity of AND. $\mathbf{T} \wedge A \equiv A$

Axiom 16 — Zero of AND. $\mathbf{F} \wedge A \equiv \mathbf{F}$

Axiom 17 — Idempotence for AND. $A \wedge A \equiv A$

Axiom 18 — Contradiction for AND. $A \wedge \neg A \equiv \mathbf{F}$

Axiom 19 — Double Negation. $\neg(\neg A) \equiv A$

Axiom 20 — Validity for OR. $A \vee \neg A \equiv \mathbf{T}$

Induction

Axiom 21 — Well Ordering Principle. Every nonempty set of nonnegative integers has a smallest element. i.e., For any $A \subset \mathbb{N}$ such that $A \neq \emptyset$, there is some $a \in A$ such that $\forall a' \in A. a \leq a'$.

Recurrences

Theorem 22 — The Master Theorem. Suppose that for $n \in \mathbb{Z}^+$.

$$T(n) = \begin{cases} c & \text{if } n < B \\ a_1 T(\lceil n/b \rceil) + a_2 T(\lfloor n/b \rfloor) + dn^i & \text{if } n \geq B \end{cases}$$

where $a_1, a_2, B, b \in \mathbb{N}$.

Let $a = a_1 + a_2 \geq 1$, $b > 1$, and $c, d, i \in \mathbb{R} \cup \{0\}$. Then,

$$T(n) \in \begin{cases} O(n^i \log n) & \text{if } a = b^i \\ O(n^i) & \text{if } a < b^i \\ O(n^{\log_b a}) & \text{if } a > b^i \end{cases}$$

Linear Recurrences:

A linear recurrence is an equation

$$f(n) = \underbrace{a_1 f(n-1) + a_2 f(n-2) + \cdots + a_d f(n-d)}_{\text{homogeneous part}} + \underbrace{g(n)}_{\text{inhomogeneous part}}$$

along with some boundary conditions.

The procedure for solving linear recurrences are as follows:

1. Find the roots of the characteristic equation. Linear recurrences usually have exponential solutions (such as x^n). Such solution is called the **homogeneous solution**.

   $$x^n = a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_{k-1} x + a_k$$

2. Write down the homogeneous solution. Each root generates one term and the homogeneous solution is their sum. A non-repeated root r generates the term cr^n, where c is a constant to be determined later. A root with r with multiplicity k generates the terms

   $$d_1 r^n \quad d_2 n r^n \quad d_3 n^2 r^n \quad \cdots \quad d_k n^{k-1} r^n$$

   where $d_1, \cdots, d_k$ are constants to be determined later.

3. Find a **particular solution** for the full recurrence including the inhomogeneous part, but without considering the boundary conditions.
   If $g(n)$ is a constant or a polynomial, try a polynomial of the same degree, then of one higher degree, then two higher. If $g(n)$ is exponential in the form $g(n) = k^n$, then try $f(n) = ck^n$, then $f(n) = (bn+c)k^n$, then $f(n) = (an^2+bn+c)k^n$, etc.
4. Write the **general solution**, which is the sum of homogeneous solution and particular solution.
5. Substitute the boundary condition into the general solution. Each boundary condition gives a linear equation. Solve such system of linear equations for the values of the constants to make the solution consistent with the boundary conditions.

Basic Prerequisite Mathematics

Basic Prerequisite Mathematics

SET THEORY

Common Sets

- $\mathbb{N} = \{0, 1, 2, \dots\}$: the natural numbers, or non-negative integers. The convention in computer science is to include 0 in the natural numbers.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$: the integers
- $\mathbb{Z}^+ = \{1, 2, 3, \dots\}$: the positive integers
- $\mathbb{Z}^- = \{-1, -2, -3, \dots\}$: the negative integers
- $\mathbb{Q}$ the rational numbers, $\mathbb{Q}^+$ the positive rationals, and $\mathbb{Q}^-$ the negative rationals.
- $\mathbb{R}$ the real numbers, $\mathbb{R}^+$ the positive reals, and $\mathbb{R}^-$ the negative reals.

Notation

For any sets A and B, we will use the following standard notation.

- $x \in A$: "x is an element of A" or "A contains x"
- $A \subseteq B$: "A is a subset of B" or "A is included in B"
- $A = B$: "A equals B" (Note that $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.)
- $A \subsetneq B$: "A is a proper subset of B"
  (Note that $A \subsetneq B$ if and only if $A \subseteq B$ and $A \neq B$.)
- $A \cup B$: "A union B"
- $A \cap B$: "A intersection B"
- $A - B$: "A minus B" (*set* difference)
- $|A|$: "cardinality of A" (the number of elements of A)
- $\varnothing$ or $\{\}$: "the empty set"
- $\mathcal{P}(A)$ or 2^A: "powerset of A" (the set of all subsets of A)
  If $A = \{a, 34, \triangle\}$, then $\mathcal{P}(A) = \big\{\{\}, \{a\}, \{34\}, \{\triangle\}, \{a, 34\}, \{a, \triangle\}, \{34, \triangle\}, \{a, 34, \triangle\}\big\}$.
  $S \in \mathcal{P}(A)$ means the same as $S \subseteq A$.
- $\{x \in A \mid P(x)\}$: "the set of elements x in A for which $P(x)$ is true"
  For example, $\{x \in \mathbb{Z} \mid \cos(\pi x) > 0\}$ represents the set of integers x for which $\cos(\pi x)$ is greater than zero, *i.e.*, it is equal to $\{\dots, -4, -2, 0, 2, 4, \dots\} = \{x \in \mathbb{Z} \mid x \text{ is even}\}$.

- $A \times B$: "the cross product or Cartesian product of A and B"
  $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$.
  If $A = \{1, 2, 3\}$ and $B = \{5, 6\}$, then $A \times B = \{(1, 5), (1, 6), (2, 5), (2, 6), (3, 5), (3, 6)\}$.)
- A^n: "the cross product of n copies of A"
  This is set of all sequences of $n \geq 1$ elements, each of which is in A.
- B^A or $A \to B$: "the set of all functions from A to B."
- $f : A \to B$ or $f \in B^A$: "f is a function from A to B"
  f associates one element $f(x) \in B$ to every element $x \in A$.

NUMBER THEORY

For any two natural numbers a and b, we say that a *divides* b if there exists a natural number c such that $b = ac$. In such a case, we say that a is a *divisor* of b (*e.g.*, 3 is a divisor of 12 but 3 is not a divisor of 16). Note that any natural number is a divisor of 0 and 1 is a divisor of any natural number. A number a is *even* if 2 divides a and is *odd* if 2 does not divide a.

A natural number p is *prime* if it has exactly two positive divisors (*e.g.*, 2 is prime since its positive divisors are 1 and 2 but 1 is **not** prime since it only has one positive divisor: 1). There are an infinite number of prime numbers and any integer greater than one can be expressed in a unique way as a finite product of prime numbers (*e.g.*, $8 = 2^3$, $77 = 7 \times 11$, $3 = 3$).

Inequalities

For any integers m and n, $m < n$ if and only if $m + 1 \leq n$ and $m > n$ if and only if $m \geq n + 1$. For any real numbers w, x, y, and z, the following properties always hold (they also hold when $<$ and $\leq$ are exchanged throughout with $>$ and $\geq$, respectively).

- if $x < y$ and $w \leq z$, then $x + w < y + z$
- if $x < y$, then $\begin{cases} xz < yz & \text{if } z > 0 \\ xz = yz & \text{if } z = 0 \\ xz > yz & \text{if } z < 0 \end{cases}$
- if $x \leq y$ and $y < z$ (or if $x < y$ and $y \leq z$), then $x < z$

Functions

Here are some common number-theoretic functions together with their definitions and properties of them. (Unless noted otherwise, in this section, x and y represent arbitrary real numbers and k, m, and n represent arbitrary positive integers.)

- $\min\{x, y\}$: "minimum of x and y" (the smallest of x or y)

  Properties: $\min\{x, y\} \leq x$
  $\min\{x, y\} \leq y$

- $\max\{x, y\}$: "maximum of x and y" (the largest of x or y)

  Properties: $x \leq \max\{x, y\}$
  $y \leq \max\{x, y\}$

- $\lfloor x \rfloor$: "floor of x" (the greatest integer less than or equal to x, *e.g.*, $\lfloor 5.67 \rfloor = 5$, $\lfloor -2.01 \rfloor = -3$)

  Properties: $x - 1 < \lfloor x \rfloor \leq x$
  $\lfloor -x \rfloor = -\lceil x \rceil$
  $\lfloor x + k \rfloor = \lfloor x \rfloor + k$
  $\lfloor \lfloor k/m \rfloor / n \rfloor = \lfloor k/mn \rfloor$
  $(k - m + 1)/m \leq \lfloor k/m \rfloor$

- $\lceil x \rceil$: "ceiling of x" (the least integer greater than or equal to x, *e.g.*, $\lceil 5.67 \rceil = 6$, $\lceil -2.01 \rceil = -2$)

  Properties: $x \leq \lceil x \rceil < x + 1$
  $\lceil -x \rceil = -\lfloor x \rfloor$
  $\lceil x + k \rceil = \lceil x \rceil + k$
  $\lceil \lceil k/m \rceil / n \rceil = \lceil k/mn \rceil$
  $\lceil k/m \rceil \leq (k + m - 1)/m$

  Additional property of $\lfloor\ \rfloor$ and $\lceil\ \rceil$: $\lfloor k/2 \rfloor + \lceil k/2 \rceil = k$.

- $|x|$: "absolute value of x" ($|x| = x$ if $x \geq 0$; $-x$ if $x < 0$, *e.g.*, $|5.67| = 5.67$, $|-2.01| = 2.01$)

  BEWARE! The same notation is used to represent the cardinality $|A|$ of a set A and the absolute value $|x|$ of a number x so be sure you are aware of the context in which it is used.

- $m \operatorname{div} n$: "the quotient of m divided by n" (integer division of m by n, *e.g.*, $5 \operatorname{div} 6 = 0$, $27 \operatorname{div} 4 = 6$, $-27 \operatorname{div} 4 = -6$)

  Properties: If $m, n > 0$, then $m \operatorname{div} n = \lfloor m/n \rfloor$
  $(-m) \operatorname{div} n = -(m \operatorname{div} n) = m \operatorname{div} (-n)$

- $m \operatorname{rem} n$: "the remainder of m divided by n" (*e.g.*, $5 \operatorname{rem} 6 = 5$, $27 \operatorname{rem} 4 = 3$, $-27 \operatorname{rem} 4 = -3$)

  Properties: $m = (m \operatorname{div} n) \cdot n + m \operatorname{rem} n$
  $(-m) \operatorname{rem} n = -(m \operatorname{rem} n) = m \operatorname{rem} (-n)$

- $m \bmod n$: "m modulo n" (*e.g.*, $5 \bmod 6 = 5$, $27 \bmod 4 = 3$ $-27 \bmod 4 = 1$)

  Properties: $0 \leq m \bmod n < n$
  n divides $m - (m \bmod n)$.

- $\gcd(m, n)$: "greatest common divisor of m and n" (the largest positive integer that divides both m and n)

  For example, $\gcd(3, 4) = 1$, $\gcd(12, 20) = 4$, $\gcd(3, 6) = 3$

- $\operatorname{lcm}(m, n)$: "least common multiple of m and n" (the smallest positive integer that m and n both divide)

  For example, $\operatorname{lcm}(3, 4) = 12$, $\operatorname{lcm}(12, 20) = 60$, $\operatorname{lcm}(3, 6) = 6$

  Properties: $\gcd(m, n) \cdot \operatorname{lcm}(m, n) = m \cdot n$.

CALCULUS

Limits and Sums

An infinite sequence of real numbers $\{a_n\} = a_1, a_2, \ldots, a_n, \ldots$ *converges* to a limit $L \in \mathbb{R}$ if, for every $\varepsilon > 0$, there exists $n_0 \geq 0$ such that $|a_n - L| < \varepsilon$ for every $n \geq n_0$. In this case, we write $\lim_{n\to\infty} a_n = L$ Otherwise, we say that the sequence *diverges*.

If $\{a_n\}$ and $\{b_n\}$ are two sequences of real numbers such that $\lim_{n\to\infty} a_n = L_1$ and $\lim_{n\to\infty} b_n = L_2$, then

$$\lim_{n\to\infty}(a_n + b_n) = L_1 + L_2 \quad \text{and} \quad \lim_{n\to\infty}(a_n \cdot b_n) = L_1 \cdot L_2.$$

In particular, if c is any real number, then

$$\lim_{n\to\infty}(c \cdot a_n) = c \cdot L_1.$$

The sum $a_1 + a_2 + \cdots + a_n$ and product $a_1 \cdot a_2 \cdots a_n$ of the finite sequence $a_1, a_2, \ldots, a_n$ are denoted by

$$\sum_{i=1}^{n} a_i \text{ and } \prod_{i=1}^{n} a_i.$$

If the elements of the sequence are all different and $S = \{a_1, a_2, \ldots, a_n\}$ is the set of elements in the sequence, these can also be denoted by

$$\sum_{a\in S} a \text{ and } \prod_{a\in S} a.$$

Examples:

- For any $a \in \mathbb{R}$ such that $-1 < a < 1$, $\lim_{n\to\infty} a^n = 0$.
- For any $a \in \mathbb{R}^+$, $\lim_{n\to\infty} a^{1/n} = 1$.
- For any $a \in \mathbb{R}^+$, $\lim_{n\to\infty} (1/n)^a = 0$.
- $\lim_{n\to\infty}(1 + 1/n)^n = e = 2.71828182845904523536\ldots$

- For any $a, b \in \mathbb{R}$, the *arithmetic* sum is given by:

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

- For any $a, b \in \mathbb{R}^+$, the *geometric* sum is given by:

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

EXPONENTS AND LOGARITHMS

Definition: For any $a, b, c \in \mathbb{R}^+$, $a = \log_b c$ if and only if $b^a = c$.

Notation: For any $x \in \mathbb{R}^+$, $\ln x = \log_e x$ and $\lg x = \log_2 x$.

For any $a, b, c \in \mathbb{R}^+$ and any $n \in \mathbb{Z}^+$, the following properties always hold.

- $\sqrt[n]{b} = b^{1/n}$
- $b^a b^c = b^{a+c}$
- $(b^a)^c = b^{ac}$
- $b^a / b^c = b^{a-c}$
- $b^0 = 1$
- $a^b c^b = (ac)^b$
- $b^{\log_b a} = a = \log_b b^a$
- $a^{\log_b c} = c^{\log_b a}$
- $\log_b(ac) = \log_b a + \log_b c$
- $\log_b(a^c) = c \cdot \log_b a$
- $\log_b(a/c) = \log_b a - \log_b c$
- $\log_b 1 = 0$
- $\log_b a = \log_c a / \log_c b$

BINARY NOTATION

A *binary number* is a sequence of bits $a_k \cdots a_1 a_0$ where each bit a_i is equal to 0 or 1. Every binary number represents a natural number in the following way:

$$(a_k \cdots a_1 a_0)_2 = \sum_{i=0}^{k} a_i 2^i = a_k 2^k + \cdots + a_1 2 + a_0.$$

For example, $(1001)_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 1 = 9$, $(01110)_2 = 8 + 4 + 2 = 14$.

Properties:

- If $a = (a_k \cdots a_1 a_0)_2$, then $2a = (a_k \cdots a_1 a_0 0)_2$, *e.g.*, $9 = (1001)_2$ so $18 = (10010)_2$.
- If $a = (a_k \cdots a_1 a_0)_2$, then $\lfloor a/2 \rfloor = (a_k \cdots a_1)_2$, *e.g.*, $9 = (1001)_2$ so $4 = (100)_2$.
- The smallest number of bits required to represent the positive integer n in binary is called the *length* of n and is equal to $\lceil \log_2(n+1) \rceil$.

Make sure you know how to add and multiply two binary numbers. For example, $(1111)_2 + (101)_2 = (10100)_2$ and $(1111)_2 \times 101)_2 = (1001011)_2$.

Proof Templates

Proof Outlines

LINE NUMBERS: Only lines that are referred to have labels (for example, L1) in this document. For a formal proof, all lines are numbered. Line numbers appear at the beginning of a line. You can indent line numbers together with the lines they are numbering or all line numbers can be unindented, provided you are consistent.

INDENTATION: Indent when you make an assumption or define a variable. Unindent when this assumption or variable is no longer being used.

1. **Implication**: Direct proof of A IMPLIES B.

    L1. Assume A.
    ⋮
    L2. B
   A IMPLIES B; direct proof: L1, L2

2. **Implication**: Indirect proof of A IMPLIES B.

    L1. Assume $NOT(B)$.
    ⋮
    L2. $NOT(A)$
   A IMPLIES B; indirect proof: L1, L2

3. **Equivalence**: Proof of A IFF B.

    L1. Assume A.
    ⋮
    L2. B
   L3. A IMPLIES B; direct proof: L1, L2
    L4. Assume B.
    ⋮
    L5. A
   L6. B IMPLIES A; direct proof: L4, L5
   A IFF B; equivalence: L3, L6

4. **Proof by contradiction** of A.

    L1. To obtain a contradiction, assume $NOT(A)$.
    ⋮
        L2. B
    ⋮
        L3. $NOT(B)$
    L4. This is a contradiction: L2, L3
   Therefore A; proof by contradiction: L1, L4

5. **Modus Ponens**.

   ⋮
   L1. A
   ⋮
   L2. A IMPLIES B
   B; modus ponens: L1, L2

6. **Conjunction:** Proof of A AND B:

   ⋮
   L1. A
   ⋮
   L2. B
   A AND B; proof of conjunction; L1, 2

7. **Use of Conjunction**:

   ⋮
   L1. A AND B
   A; use of conjunction: L1
   B; use of conjunction: L1

8. **Implication with Conjunction**: Proof of (A_1 AND A_2) IMPLIES B.

      L1. Assume A_1 AND A_2.
      A_1; use of conjunction, L1
      A_2; use of conjunction, L1
      ⋮
      L2. B
   (A_1 AND A_2) IMPLIES B; direct proof, L1, L2

9. **Implication with Conjunction**: Proof of A IMPLIES (B_1 AND B_2).

      L1. Assume A.
      ⋮
      L2. B_1
      ⋮
      L3. B_2
      L4. B_1 AND B_2; proof of conjunction: L2, L3
   A IMPLIES (B_1 AND B_2); direct proof: L1, L4

10. **Disjunction**: Proof of A OR B and B OR A.

    ⋮
    L1. A
    A OR B; proof of disjunction: L1
    B OR A; proof of disjunction: L1

11. **Proof by cases**.

    L1. C OR $NOT(C)$ tautology
    L2. Case 1: Assume C.
    $\vdots$
    L3. A
    L4. C IMPLIES A; direct proof: L2, L3
    L5. Case 2: Assume $NOT(C)$.
    $\vdots$
    L6. A
    L7. $NOT(C)$ IMPLIES A; direct proof: L5, L6
    A proof by cases: L1, L4, L7

12. **Proof by cases** of A OR B.

    L1. C OR $NOT(C)$ tautology
    L2. Case 1: Assume C.
    $\vdots$
    L3. A
    L4. A OR B; proof of disjunction, L3
    L5. C IMPLIES (A OR B); direct proof, L2, L4
    L6. Case 2: Assume $NOT(C)$.
    $\vdots$
    L7. B
    L8. A OR B; proof of disjunction, L7
    L9. $NOT(C)$ IMPLIES (A OR B); direct proof: L6, L8
    A OR B; proof by cases: L1, L5, L9

13. **Implication with Disjunction**: Proof by cases of (A_1 OR A_2) IMPLIES B.

    L1. Case 1: Assume A_1.
    $\vdots$
    L2. B
    L3. A_1 IMPLIES B; direct proof: L1,L2
    L4. Case 2: Assume A_2.
    $\vdots$
    L5. B
    L6. A_2 IMPLIES B; direct proof: L4, L5
    (A_1 OR A_2) IMPLIES B; proof by cases: L3, L6

14. **Implication with Disjunction**: Proof by cases of A IMPLIES (B_1 OR B_2).

    L1. Assume A.
    L2. C OR NOT(C) tautology
    L3. Case 1: Assume C.
    $\vdots$
    L4. B_1
    L5. B_1 OR B_2; disjunction: L4
    L6. C IMPLIES (B_1 OR B_2); direct proof: L3, L5
    L7. Case 2: AssumeNOT(C).
    $\vdots$
    L8. B_2
    L9. B_1 OR B_2; disjunction: L8
    L10. NOT(C) IMPLIES (B_1 OR B_2); direct proof: L7, L9
    L11. B_1 OR B_2; proof by cases: L2, L6, L10
    A IMPLIES (B_1 OR B_2): direct proof. L1, L11

15. **Substitution of a Variable in a Tautology**:
    Suppose P is a propositional variable, Q is a formula, and R' is obtained from R by replacing *every* occurrence of P by (Q).

    L1. R tautology
    R'; substitution of all P by Q: L1

16. **Substitution of a Formula by a Logically Equivalent Formula**:
    Suppose S is a subformula of R and R' is obtained from R by replacing *some* occurrence of S by S'.

    L1. R
    L2. S IFF S'
    L3. R'; substitution of an occurrence of S by S': L1, L2

17. **Specialization**:

    L1. $c \in D$
    L2. $\forall x \in D.P(x)$
    $P(c)$; specialization: L1, L2

18. **Generalization**: Proof of $\forall x \in D.P(x)$.

    L1. Let x be an arbitrary element of D.
    $\vdots$
    L2. $P(x)$
    Since x is an arbitrary element of D,
    $\forall x \in D.P(x)$; generalization: L1, L2

19. **Universal Quantification with Implication**: Proof of $\forall x \in D.(P(x)$ IMPLIES $Q(x))$.

    L1. Let x be an arbitrary element of D.
        L2. Assume $P(x)$
        $\vdots$
        L3. $Q(x)$
    L4. $P(x)$ IMPLIES $Q(x)$; direct proof: L2, L3
    Since x is an arbitrary element of D,
    $\forall x \in D.(P(x)$ IMPLIES $Q(x))$; generalization: L1, L4

20. **Implication with Universal Quantification**: Proof of $(\forall x \in D.P(x))$ IMPLIES A.

    L1. Assume $\forall x \in D.P(x)$.
    $\vdots$
    L2. $a \in D$
    $P(a)$; specialization: L1, L2
    $\vdots$
    L3. A
    Therefore $(\forall x \in D.P(x))$ IMPLIES A; direct proof: L1, L3

21. **Implication with Universal Quantification**: Proof of A IMPLIES $(\forall x \in D.P(x))$.

    L1. Assume A.
        L2. Let x be an arbitrary element of D.
        $\vdots$
        L3. $P(x)$
    Since x is an arbitrary element of D,
    L4. $\forall x \in D.P(x)$; generalization, L2, L3
    A IMPLIES $(\forall x \in D.P(x))$; direct proof: L1, L4

22. **Instantiation**:

    L1. $\exists x \in D.P(x)$
    Let $c \in D$ be such that $P(c)$; instantiation: L1
    $\vdots$

23. **Construction**: Proof of $\exists x \in D.P(x)$.

    L1. Let $a = \cdots$
    $\vdots$
    L2. $a \in D$
    $\vdots$
    L3. $P(a)$
    $\exists x \in D.P(x)$; construction: L1, L2, L3

24. **Existential Quantification with Implication**: Proof of $\exists x \in D.(P(x)$ IMPLIES $Q(x))$.

    L1. Let $a = \cdots$

    $\vdots$

    L2. $a \in D$

        L3. Suppose $P(a)$.

        $\vdots$

        L4. $Q(a)$

    L5. $P(a)$ IMPLIES $Q(a)$; direct proof: L3, L4

    $\exists x \in D.(P(x)$ IMPLIES $Q(x))$; construction: L1, L2, L5

25. **Implication with Existential Quantification**: Proof of $(\exists x \in D.P(x))$ IMPLIES A.

    L1. Assume $\exists x \in D.P(x)$.

        Let $a \in D$ be such that $P(a)$; instantiation: L1

        $\vdots$

        L2. A

    $(\exists x \in D.P(x))$ IMPLIES A; direct proof: L1, L2

26. **Implication with Existential Quantification**: Proof of A IMPLIES $(\exists x \in D.P(x))$.

    L1. Assume A.

        L2. Let $a = \cdots$

        $\vdots$

        L3. $a \in D$

        $\vdots$

        L4. $P(a)$

    L5. $\exists x \in D.P(x)$; construction: L2, L3, L4

    A IMPLIES $(\exists x \in D.P(x))$; direct proof: L1, L5

27. **Subset**: Proof of $A \subseteq B$.

    L1. Let $x \in A$ be arbitrary.

    $\vdots$

    L2. $x \in B$

    The following line is optional:

    L3. $x \in A$ IMPLIES $x \in B$; direct proof: L1, L2

    $A \subseteq B$; definition of subset: L3 (or L1, L2, if the optional line is missing)

28. **Weak Induction**: Proof of $\forall n \in N.P(n)$

    Base Case:
    $\vdots$
    L1. $P(0)$
        L2. Let $n \in N$ be arbitrary.
            L3. Assume $P(n)$.
            $\vdots$
            L4. $P(n+1)$
        The following two lines are optional:
        L5. $P(n)$ IMPLIES ($P(n+1)$; direct proof of implication: L3, L4
    L6. $\forall n \in N.(P(n)$ IMPLIES $P(n+1))$; generalization L2, L5
    $\forall n \in N.P(n)$ induction; L1, L6 (or L1, L2, L3, L4, if the optional lines are missing)

29. **Strong Induction**: Proof of $\forall n \in N.P(n)$

        L1. Let $n \in N$ be arbitrary.
            L2. Assume $\forall j \in N.(j < n$ IMPLIES $P(j))$
            $\vdots$
            L3. $P(n)$
        The following two lines are optional:
        L4. $\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)$; direct proof of implication: L2, L3
    L5. $\forall n \in N.[\forall j \in N.(j < n$ IMPLIES $P(j))$ IMPLIES $P(n)]$; generalization: L1, L4
    $\forall n \in N.P(n)$; strong induction: L5 (or L1, L2, L3, if the optional lines are missing)

30. **Structural Induction**: Proof of $\forall e \in S.P(e)$, where S is a recursively defined set

    Base case(s):
        L1. For each base case e in the definition of S
        L2. $P(e)$.
    Constructor case(s):
        L3. For each constructor case e of the definition of S,
            L4. assume $P(e')$ for all components e' of e.
            $\vdots$
            L5. $P(e)$
    $\forall e \in S.P(e)$; structural induction: L1, L2, L3, L4, L5

31. **Well Ordering Principle**: Proof of $\forall e \in S.P(e)$, where S is a well ordered set, i.e. every nonempty subset of S has a smallest element.

    L1. To obtain a contradiction, suppose that $\forall e \in S.P(e)$ is false.
    L2. Let $C = \{e \in S \mid P(e) \text{ is false}\}$ be the set of counterexamples to P.
    L3. $C \neq \phi$; definition: L1, L2
        L4. Let e be the smallest element of C; well ordering principle: L2, L3
            Let $e' = \cdots$
            $\vdots$
            L5. $e' \in C$
            $\vdots$
            L6. $e' < e$.
    L7. This is a contradiction: L4, L5, L6

$\forall e \in S.P(e)$; proof by contradiction: L1, L7

Index

Index

Bibliography

Courses

[1] Allan Borodin. *CSC364 Computational Complexity and Computability*. University of Toronto. 2004.

[2] Allan Borodin. *CSC373S Algorithm Design, Analysis & Complexity, Spring 2011*. University of Toronto. 2011.

[4] Allan Borodin and Sara Rahmati. *CSC373S Algorithm Design, Analysis & Complexity, Spring 2020*. University of Toronto. 2020.

[6] Erik Demaine and Srini Devadas. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. 2011.

[7] Erik Demaine, Srini Devadas, and Nancy Lynch. *6.046J Design and Analysis of Algorithms*. Massachusetts Institute of Technology. 2015.

[10] Faith Ellen. *CSC240S Enriched Introduction to the Theory of Computation, Winter 2021*. University of Toronto. 2021.

[11] Faith Ellen. *CSC265F Enriched Data Structures and Analysis Fall 2021*. University of Toronto. 2021.

[14] Mauricio Karchmer, Anand Natarajan, and Nir Shavit. *6.006 Introduction to Algorithms*. Massachusetts Institute of Technology. 2021.

[16] Aleksandar Nikolov. *CSC473S Advanced Algorithms, Winter 2020*. University of Toronto. 2020.

[17] Debmalya Panigrahi and Kevin Sun. *COMPSCI 230: Discrete Mathematics for Computer Science (Course Notes)*. Duke University. 2019.

[20] Nathan Wiebe and Karan Singh. *CSC373S Algorithm Design, Analysis & Complexity, Spring 2022*. University of Toronto. 2022.

Books

[5] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[8] Reinhard Diestel. *Graph Theory*. 5th. Springer, 2017. ISBN: 3662536218.

[12] Jeff Erickson. *Algorithms*. 1st. 2019. ISBN: 978-1792644832.

[15] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 1st. Pearson, 2005. ISBN: 978-0321295354.

[18] Ron Shamir. *Advanced Topics in Graph Algorithms*. Tel-Aviv University. 1994 (cited on page 48).

Journal Articles

[3] Allan Borodin, Morten N. Nielsen, and Charles Rackoff. "(Incremental) Priority Algorithms". In: *Algorithmica* 37.4 (2003), pages 295–326. DOI: 10.1007/s00453-003-1036-3 (cited on page 56).

[9] G. A. Dirac. "On rigid circuit graphs". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 25.1 (1961), pages 71–76. DOI: 10.1007/BF02992776. URL: https://doi.org/10.1007/BF02992776 (cited on page 48).

[13] D. R. Fulkerson and O. A. Gross. "Incidence matrices and interval graphs." In: *Pacific Journal of Mathematics* 15.3 (1965), pages 835 –855. DOI: pjm/1102995572. URL: https://doi.org/ (cited on page 48).

[19] Robert E. Tarjan and Mihalis Yannakakis. "Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs". In: *SIAM Journal on Computing* 13.3 (1984), pages 566–579. DOI: 10.1137/0213035. eprint: https://doi.org/10.1137/0213035. URL: https://doi.org/10.1137/0213035 (cited on page 47).

[21] Uri Zwick. "The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate". In: *Theoretical Computer Science* 148.1 (1995), pages 165–170. ISSN: 0304-3975. DOI: 10.1016/0304-3975(95)00022-O (cited on pages 81, 88).