Notes for Operations Research & More

Lan Peng, PhD Student

Department of Industrial and Systems Engineering
University at Buffalo, SUNY
lanpeng@buffalo.edu

February 18, 2021

February 18, 2021

To My Beloved Motherland China

Contents

Part I

Preliminary Topics

Chapter 1

Introduction to Optimization

1.1 Optimization Model

The following is the basic forms of terminology:

$$\begin{aligned} (\mathrm{P}) \quad \min \quad & f(x) && (1.1)\\ \text{s.t.} \quad & g_i(x) \leq 0, \quad i = 1, 2, ..., m && (1.2)\\ & h_j(x) = 0, \quad j = 1, 2, ..., l && (1.3)\\ & x \in X && (1.4) \end{aligned}$$

We have

- - $x \in R^n \to X \subseteq R^m$
- $g_i(x)$ are called inequality constraints
- $h_j(x)$ are called equality constraints
- X is the domain of the variables (e.g. cone, polygon, $\{0,1\}^n$, etc.)
- Let F be the feasible region of (P):
  - x^0 is a feasible solution iff $x^0 \in F$
  - x^* is an optimized solution iff $x^* \in F$ and $f(x^*) \leq f(x^0), \forall x^0 \in F$ (for minimized problem)

Notice: Not every (P) has a feasible region, we can have $F = \emptyset$. Even if $F \neq \emptyset$, there might not be an solution to P, e.g. unbounded. If (P) has optimized solution(s), it could be 1) Unique, or 2) Infinite number of solution, or 3) Finite number of solution

Types of Optimization Problem

- $m = l = 0$, $x \in R^n$, unconstrained problem
- $m + l > 0$, constrained problem
- $f(x), g_i(x), h_j(x)$ are linear, Linear Optimization
  - If $X = R^n$, Linear Programming
  - If X is discrete, Discrete Optimization
  - If $X \subseteq Z^n$, Integer Programming
  - If $X \in \{0,1\}^n$, Binary Programming
  - If $X \in Z^n \times R^m$, Mixed Integer Programming

1.2 Linear Programming Formulation Skills

1.2.1 Absolute Value

Consider the following model statement:

$$\min \quad \sum_{j\in J} c_j|x_j|, \quad c_j > 0 \tag{1.5}$$

$$\text{s.t.} \quad \sum_{j\in J} a_{ij}x_j \gtreqless b_i, \quad \forall i \in I \tag{1.6}$$

$$x_j \quad \text{unrestricted}, \quad \forall j \in J \tag{1.7}$$

Modeling:

$$\min \quad \sum_{j\in J} c_j(x_j^+ + x_j^-), \quad c_j > 0 \tag{1.8}$$

$$\text{s.t.} \quad \sum_{j\in J} a_{ij}(x_j^+ - x_j^-) \gtreqless b_i, \quad \forall i \in I \tag{1.9}$$

$$x_j^+, x_j^- \geq 0, \quad \forall j \in J \tag{1.10}$$

1.2.2 A Minimax Objective

Consider the following model statement:

$$\min \quad \max_{k\in K} \sum_{j\in J} c_{kj}x_j \tag{1.11}$$

$$\text{s.t.} \quad \sum_{j\in J} a_{ij}x_j \gtreqless b_i, \quad \forall i \in I \tag{1.12}$$

$$x_j \geq 0, \quad \forall j \in J \tag{1.13}$$

Modeling:

$$\min \quad z \tag{1.14}$$

$$\text{s.t.} \quad \sum_{j\in J} a_{ij}x_j \gtreqless b_i, \quad \forall i \in I \tag{1.15}$$

$$\sum_{j\in J} c_{kj}x_j \leq z, \quad \forall k \in K \tag{1.16}$$

$$x_j \geq 0, \quad \forall j \in J \tag{1.17}$$

1.2.3 A Fractional Objective

Consider the following model statement:

$$\min \quad \frac{\sum_{j\in J} c_j x_j + \alpha}{\sum_{j\in J} d_j x_j + \beta} \tag{1.18}$$

$$\text{s.t.} \quad \sum_{j\in J} a_{ij}x_j \gtreqless b_i, \quad \forall i \in I \tag{1.19}$$

$$x_j \geq 0, \quad \forall j \in J \tag{1.20}$$

Modeling:

$$\min \quad \sum_{j\in J} c_j x_j t + \alpha t \tag{1.21}$$

$$\text{s.t.} \quad \sum_{j\in J} a_{ij} x_j \gtreqless b_i, \quad \forall i \in J \tag{1.22}$$

$$\sum_{j\in J} d_j x_j t + \beta t = 1 \tag{1.23}$$

$$t > 0 \tag{1.24}$$

$$x_j \geq 0, \quad \forall j \in J \tag{1.25}$$

$$(t = \frac{1}{\sum_{j\in J} d_j x_j + \beta}) \tag{1.26}$$

For the following statement:

$$\min \quad z^P = \frac{\mathbf{c}^\top \mathbf{x} + d}{\mathbf{e}^\top \mathbf{x} + f} \tag{1.27}$$

$$\text{s.t.} \quad \mathbf{G}\mathbf{x} \leq \mathbf{h} \tag{1.28}$$

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1.29}$$

Modeling

$$\min \quad z^R = \mathbf{c}^\top \mathbf{y} + dz \tag{1.30}$$

$$\text{s.t.} \quad \mathbf{G}\mathbf{y} - \mathbf{h}z \leq 0 \tag{1.31}$$

$$\mathbf{A}\mathbf{y} - \mathbf{b}z = 0 \tag{1.32}$$

$$\mathbf{e}^\top \mathbf{y} + fz = 1 \tag{1.33}$$

$$z \geq 0 \tag{1.34}$$

1.2.4 A Range Constraint

Consider the following model statement:

$$\min \quad \sum_{j\in J} c_j x_j \tag{1.35}$$

$$\text{s.t.} \quad d_i \leq \sum_{j\in J} a_{ij} x_j \leq e_i, \quad \forall i \in I \tag{1.36}$$

$$x_j \geq 0, \quad \forall j \in J \tag{1.37}$$

Modeling:

$$\min \quad \sum_{j\in J} c_j x_j, \quad c_j > 0 \tag{1.38}$$

$$\text{s.t.} \quad u_i + \sum_{j\in J} a_{ij} x_j = e_i, \quad \forall i \in I \tag{1.39}$$

$$x_j \geq 0, \quad \forall j \in J \tag{1.40}$$

$$0 \leq u_i \leq e_i - d_i, \quad \forall i \in I \tag{1.41}$$

1.3 Integer Programming Formulation Skills

1.3.1 A Variable Taking Discontinuous Values

In algebraic notation:

$$x = 0, \quad \text{or} \quad l \leq x \leq u \tag{1.42}$$

Modeling:

$$x \leq uy \tag{1.43}$$
$$x \geq ly \tag{1.44}$$
$$y \in \{0,1\} \tag{1.45}$$

where

$$y = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } l \leq x \leq u \end{cases} \tag{1.46}$$

1.3.2 Fixed Costs

In algebraic notation:

$$C(x) = \begin{cases} 0 & \text{for } x = 0 \\ k + cx & \text{for } x > 0 \end{cases} \tag{1.47}$$

Modeling:

$$C^*(x,y) = ky + cx \tag{1.48}$$
$$x \leq My \tag{1.49}$$
$$x \geq 0 \tag{1.50}$$
$$y \in \{0,1\} \tag{1.51}$$

where

$$y = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } x \geq 0 \end{cases} \tag{1.52}$$

1.3.3 Either-or Constraints

In algebraic notation:

$$\sum_{j \in J} a_{1j} x_j \leq b_1 \text{ or } \sum_{j \in J} a_{2j} x_j \leq b_2 \tag{1.53}$$

Modeling:

$$\sum_{j \in J} a_{1j} x_j \leq b_1 + M_1 y \tag{1.54}$$
$$\sum_{j \in J} a_{2j} x_j \leq b_2 + M_1 (1 - y) \tag{1.55}$$
$$y \in \{0,1\} \tag{1.56}$$

where

$$y = \begin{cases} 0, & \text{if } \sum_{j \in J} a_{1j} x_j \leq b_1 \\ 1, & \text{if } \sum_{j \in J} a_{2j} x_j \leq b_2 \end{cases} \tag{1.57}$$

Notice that the sign before M is determined by the inequality $\geq$ or $\leq$, if it is "$\geq$", use "$-$", if it "$\leq$", use "$+$".

1.3.4 Conditional Constraints

If constraint A is satisfied, then constraint B must also be satisfied

$$\text{If } \sum_{j \in J} a_{1j} x_j \leq b_1 \text{ then } \sum_{j \in J} a_{2j} x_j \leq b_2 \tag{1.58}$$

The key part is to find the opposite of the first condition. We are using $A \Rightarrow B \Leftrightarrow \neg B \Rightarrow \neg A$
Therefore it is equivalent to

$$\sum_{j \in J} a_{1j} x_j > b_1 \text{ or } \sum_{j \in J} a_{2j} x_j \leq b_2 \tag{1.59}$$

Furthermore, it is equivalent to

$$\sum_{j\in J} a_{1j}x_j \geq b_1 + \epsilon \text{ or } \sum_{j\in J} a_{2j}x_j \leq b_2 \tag{1.60}$$

Where ϵ is a very small positive number.
Modeling:

$$\sum_{j\in J} a_{1j}x_j \geq b_1 + \epsilon - M_2 y \tag{1.61}$$

$$\sum_{j\in J} a_{2j}x_j \leq b_2 + M_2(1-y) \tag{1.62}$$

$$y \in \{0,1\} \tag{1.63}$$

1.3.5 Special Ordered Sets

Out of a set of yes-no decisions, at most one decision variable can be yes. Also known as SOS1.

$$x_1 = 1, x_2 = x_3 = \cdots = x_n = 0 \tag{1.64}$$

$$\text{or} \tag{1.65}$$

$$x_2 = 1, x_1 = x_3 = \cdots = x_n = 0 \tag{1.66}$$

$$\text{or ...} \tag{1.67}$$

Modeling:

$$\sum_i x_i = 1, \quad i \in N \tag{1.68}$$

Out of a set of binary variables, at most two variables can be nonzero. In addition, the two variables must be adjacent to each other in a fixed order list. Also known as SOS2. Modeling: If $x_1, x_2, ..., x_n$ is a SOS2, then

$$\sum_{i=1}^{n} x_i \leq 2 \tag{1.69}$$

$$x_i + x_j \leq 1, \forall i \in \{1,2,...,n\}, j \in \{i+2, i+3, ..., n\} \tag{1.70}$$

$$x_i \in \{0,1\} \tag{1.71}$$

There is another type of definition, that is out of a set of nonnegative variables **not binary here**, at most two variables can be nonzero. In addition, the two variables must be adjacent to each other in a fixed order list. All variables summing to 1.
This definition of SOS2 is used in Piecewise Linear Formulations.

1.3.6 Piecewise Linear Formulations

The objective function is a sequence of line segments, e.g. $y = f(x)$, consists $k-1$ linear segments going through k given points $(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)$.
Denote

$$d_i = \begin{cases} 1, & x \in (x_i, x_{i+1}) \\ 0, & \text{otherwise} \end{cases} \tag{1.72}$$

Then the objective function is

$$\sum_{i\in\{1,2,...,k-1\}} y = d_i f_i(x) \tag{1.73}$$

Modeling: Given that objective function as a piecewise linear formulation, we can have these constraints

$$\sum_{i\in\{1,2,...,k-1\}} d_i = 1 \tag{1.74}$$

$$d_i \in \{0,1\}, i \in \{1,2,...,k-1\} \tag{1.75}$$

$$x = \sum_{i\in\{1,2,...,k\}} w_i x_i \tag{1.76}$$

$$y = \sum_{i\in\{1,2,...,k\}} w_i y_i \tag{1.77}$$

$$w_1 \leq d_1 \tag{1.78}$$

$$w_i \leq d_{i-1} + di, i \in \{2,3,...,k-1\} \tag{1.79}$$

$$w_k \leq d_{k-1} \tag{1.80}$$

In this case, $w_i \in SOS2$ (second definition)

1.3.7 Conditional Binary Variables

Choose at most n binary variable to be 1 out of $x_1, x_2, ...x_m, m \geq n$. If $n = 1$ then it is SOS1.
Modeling:

$$\sum_{k\in\{1,2,...,m\}} x_k \leq n \tag{1.81}$$

Choose exactly n binary variable to be 1 out of $x_1, x_2, ...x_m, m \geq n$
Modeling:

$$\sum_{k\in\{1,2,...,m\}} x_k = n \tag{1.82}$$

Choose x_j only if $x_k = 1$
Modeling:

$$x_j = x_k \tag{1.83}$$

"and" condition, iff $x_1, x_2, ..., x_m = 1$ then $y = 1$
Modeling:

$$y \leq x_i, i \in \{1,2,...,m\} \tag{1.84}$$

$$y \geq \sum_{i\in\{1,2,...,m\}} x_i - (m-1) \tag{1.85}$$

1.3.8 Elimination of Products of Variables

For variables x_1 and x_2,

$$y = x_1 x_2 \tag{1.86}$$

Modeling: If x_1, x_2 are binary, it is the same as "and" condition of binary variables.
If x_1 is binary, while x_2 is continuous and $0 \leq x_2 \leq u$, then

$$y \leq u x_1 \tag{1.87}$$

$$y \leq x_2 \tag{1.88}$$

$$y \geq x_2 - u(1 - x_1) \tag{1.89}$$

$$y \geq 0 \tag{1.90}$$

If both x_1 and x_2 are continuous, it is non-linear, we can use Piecewise linear formulation to simulate.

Chapter 2

Basic Concepts of Linear Algebra

2.1 Vector Spaces

2.1.1 Field

Definition 2.1.1 (Field)**.** Let F denote either the set of real numbers or the set of complex numbers.

- Addition is commutative: $x + y = y + x, \forall x, y \in F$
- Addition is associative: $x + (y + z) = (x + y) + z, \forall x, y, z \in F$
- Element 0 exists and unique: $\exists 0, x + 0 = x, \forall x \in F$
- To each $x \in F$ there corresponds a unique element $(-x) \in F$ such that $x + (-x) = 0$
- Multiplication is commutative: $xy = yx, \forall x, y \in F$
- Multiplication is associative: $x(yz) = (xy)z, \forall x, y, z \in F$
- Element 1 exists and unique: $\exists 1, x1 = x, \forall x \in F$
- To each $x \neq 0 \in F$ there corresponds a unique element $x^{-1} \in F$ that $xx^{-1} = 1$
- Multiplication distributes over addition: $x(y + z) = xy + xz, \forall x, y, z \in F$

Suppose one has a set F of objects $x, y, z, \cdots$ and two operations on the elements of F as following:

- (Addition) associates with each pair of elements $x, y \in F$ an element $(x + y) \in F$,
- (Multiplication) associates with each pair x, y an element $xy \in F$,

and these two operations satisfy all conditions above. The set F, together with these two operations, is then called a **field**.

Definition 2.1.2 (Subfield)**.** A **subfield** of the field C is a set F of complex numbers which itself is a field.

Example. The set of rational numbers is a field.

Example. The set of integers is **not** a field.

Example. The set of all complex numbers of the form $x + y\sqrt{2}$ where x and y are rational, is a subfield of $\mathbb{C}$.

2.1.2 Vector Space and Subspace

Definition 2.1.3 (Vector space)**.** A **vector space** consists of the following:

- A field F of scalars;
- A set V of vectors;

- An addition operation, which associated with each pair of vectors $\alpha, \beta \in V$ a vector $\alpha + \beta$ in V called the **sum** of α and β, in such a way that
  - $\alpha + \beta = \beta + \alpha$;
  - $\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$;
  - $\forall \alpha \in V, \alpha + 0 = \alpha$
  - $\forall \alpha \in V, \alpha + (-\alpha) = 0$
- A multiplication operation, which associated with each scalar $c \in F$ and a vector $\alpha \in V$ a vector $c\alpha \in V$ called the **product** of c and α, in such a way that
  - $1\alpha = \alpha, \forall \alpha \in V$
  - $(c_1 c_2)\alpha = c_1(c_2\alpha)$
  - $c(\alpha + \beta) = c\alpha + c\beta$
  - $(c_1 + c_2)\alpha = c_1\alpha + c_2\alpha$

we may simply denote the vector space as V, which is the same notation as the set of vectors, if the field F of scalars needs to be specified, we shall say V is a vector space over the field F.

Definition 2.1.4 (Subspace)**.** Let V be a vector space over the field F. A **subspace** of V is a subset H of V which is itself a vector space over F with the operations of vector addition and scalar multiplication on H

The definition of subspace implies $\lambda x \in H, \forall \lambda \in F$ and if $x, y \in V, x + y \in H$.
The following is equivalent:

- $H \subseteq \mathbb{R}^n$ is a subspace
- There is an $m \times n$ matrix A such that $H = \{x \in \mathbb{R}^n | Ax = 0\}$
- There is a $k \times n$ matrix B such that $H = \{x \in \mathbb{R}^n | x = uB, u \in \mathbb{R}^k\}$

Particularly,

Definition 2.1.5 (Orthogonal subspace)**.** For a subspace H, then $\{x \in \mathbb{R}^n | xy = 0, y \in H\}$ is a **orthogonal subspace** and denoted by $H^\perp$

Proposition 1. *If $H = \{x \in \mathbb{R}^n | Ax = 0\}$, with A being an $m \times n$ matrix, then $H^\perp = \{x \in \mathbb{R}^n | x = A^\top u, u \in \mathbb{R}^m\}$*

2.1.3 Linear, Conic, Affine, and Convex Combinations

Proposition 2. *The following statements are equivalent:*

- $x^1, x^2, \cdots, x^k \in \mathbb{R}^n$ *are affinely independent*
- $x^2 - x^1, x^3 - x^1, \cdots, x^k - x^1$ *are linearly independent*
- $\begin{bmatrix} x^1 \\ 1 \end{bmatrix}, \begin{bmatrix} x^2 \\ 1 \end{bmatrix}, \cdots, \begin{bmatrix} x^k \\ 1 \end{bmatrix}$ *are linearly independent*

2.2 Determinants

2.2.1

2.3 Inner Products

Definition 2.3.1 (Inner Product)**.** Let F be the field of real numbers or the field of complex numbers, and V a vector space over F. An **inner product** on V is a function which assigns to each ordered pair of vectors α, β in V a scalar $< \alpha | \beta >$ in F in such a way that $\forall \alpha, \beta, \gamma \in V, c \in \mathbb{R}$ that

- $< \alpha + \beta | \gamma > = < \alpha | \gamma > + < \beta | \gamma >$

- $< c\alpha|\beta >= c < \alpha|\beta >$
- $< \alpha|\beta >= \overline{< \beta|\alpha >}$
- $< \alpha|\alpha > \geq 0$, $< \alpha|\alpha >= 0$ iff $\alpha = \mathbf{0}$

Furthermore, the above properties imply that

- $< \alpha|c\beta + \gamma >= \bar{c} < \alpha|\beta > + < \alpha|\gamma >$

Definition 2.3.2. On F^n there is an inner product which we call the **standard inner product**. It is defined on $\alpha = (x_1, x_2, ..., x_n)$ and $\beta = (y_1, y_2, ..., y_n)$ by

$$< \alpha|\beta >= \sum_j x_j \bar{y_j} \tag{2.1}$$

For $F = \mathbb{R}^n$

$$< \alpha|\beta >= \sum_j x_j y_j \tag{2.2}$$

In the real case, the standard inner product is often called the dot product and denoted by $\alpha \cdot \beta$

Example. For $\alpha = (x_1, x_2)$ and $\beta = (y_1, y_2)$ in $\mathbb{R}^2$, the following is an inner product.

$$< \alpha|\beta >= x_1 y_1 - x_2 y_1 - x_1 y_2 + 4 x_2 y_2 \tag{2.3}$$

Example. For $\mathbb{C}^{n\times n}$,

$$< \mathbf{A}|\mathbf{B} >= trace(\mathbf{B}^* \mathbf{A}) \tag{2.4}$$

is an inner product, where

$$\mathbf{A}^*_{ij} = \bar{\mathbf{A}}_{ji} \quad (\textbf{conjugate transpose}) \tag{2.5}$$

For $\mathbb{R}^{n\times n}$,

$$< \mathbf{A}|\mathbf{B} >= trace(\mathbf{B}^T \mathbf{A}) = \sum_j (AB^T)_{jj} = \sum_j \sum_k A_{jk} B_{jk} \tag{2.6}$$

2.4 Norms

Definition 2.4.1 (Norms)**.** A **norm** on a vector space $\mathcal{V}$ is a function $\| \cdot \| : \mathcal{V} \to \mathbb{R}$ for which the following three properties hold for all point $\mathbf{x}, \mathbf{y} \in \mathcal{V}$ and scalars $\lambda \in \mathbb{R}$

- (Absolute homogeneity) $\|\lambda \mathbf{x}\| = |\lambda| \|\mathbf{x}\|$
- (Triangle inequality) $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$
- (Positivity) Equality $\|\mathbf{x}\| = 0$ holds iff $\mathbf{x} = 0$

Definition 2.4.2 (L_p-norms)**.** Let $p \geq 1$ be a real number. We define the p-norm of vector $\mathbf{v} \in \mathbb{R}^n$ as:

$$\|\mathbf{x}\|_p = (\sum_{i=1}^{n} |v_i|^p)^{\frac{1}{p}} \tag{2.7}$$

Particularly

$$\|\mathbf{v}\|_1 = \sum_{i=1}^{n} |v_i| \tag{2.8}$$

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2} \tag{2.9}$$

$$\|\mathbf{v}\|_\infty = \max_{i=1}^{n} |v_i| \tag{2.10}$$

Definition 2.4.3 (Frobenius norm)**.** $\mathbf{X} \in \mathbb{R}^{m\times n}$, the **Frobenius norm** is defined as

$$\|\mathbf{X}\|_F = \sqrt{trace(\mathbf{X}^\top \mathbf{X})} \tag{2.11}$$

Definition 2.4.4 (Dual norm)**.** For an arbitrary norm $\|\cdot\|$ on Euclidean space $\mathbf{E}$, the **dual norm** $\|\cdot\|^*$ on $\mathbf{E}$ is defined by

$$\|\mathbf{v}\|^* = \max\{< \mathbf{v}|\mathbf{x} > | \|\mathbf{x}\| \leq 1\} \tag{2.12}$$

For $p, q \in [1, \infty]$, the l_p and l_q norms on $\mathbb{R}^n$ are dual to each other whenever $\frac{1}{p} + \frac{1}{q} = 1$.

2.5 Eigenvectors and Eigenvalues

Definition 2.5.1. If $\mathbf{A}$ is an $n \times n$ matrix, then a nonzero vector $\mathbf{x} \in \mathbb{R}^n$ is called an **eigenvector** of $\mathbf{A}$ if $\mathbf{Ax}$ is a scalar multiple of $\mathbf{x}$, i.e.

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{2.13}$$

for some scalar λ. The scalar λ is called **eigenvalue** of $\mathbf{A}$ and the vector $\mathbf{x}$ is said to be an **eigenvector corresponding to** λ

Theorem 2.1 (Characteristic Equation)**.** *If* $\mathbf{A}$ *is an* $n \times n$ *matrix, then* λ *is an eigenvalue of* $\mathbf{A}$ *iff*

$$\det(\lambda I - A) = 0 \tag{2.14}$$

Corollary 2.1.1.

$$\sum \lambda_A = tr(\mathbf{A}) \tag{2.15}$$

Corollary 2.1.2.

$$\prod \lambda_A = \det(\mathbf{A}) \tag{2.16}$$

Notice: Gaussian elimination changes the eigenvalues.

2.6 Decompositions

Chapter 3

Basic Concepts of Convex Analysis

3.1 Convex Sets

Definition 3.1.1 (convex set, convex combination)**.** A set X in $\mathbb{R}^n$ is called a **convex set** if given any two points $\mathbf{x}_1 \in X$ and $\mathbf{x}_2 \in X$, then $\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \in X, \forall\lambda \in [0,1]$. Any point of the form $\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2$ where $0 \leq \lambda \leq 1$ is called a **convex combination** of $\mathbf{x}_1$ and $\mathbf{x}_2$. If $\lambda \in (0,1)$, then the convex combination is called **strict**.

Example. $S = \{(x_1, x_2) | 3x_1^2 + 4x_2^2 \leq 1\}$

Example. $X = \{\mathbf{x} | \mathbf{A}_{m\times n}\mathbf{x} = \mathbf{b}_m\}$

The following are some families of convex sets.

Example. Empty set is by convention considered as convex.

Example. Polyhedrons are convex sets.

Example. Let $P = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{x}^\top \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ where $\mathbf{A} \in \mathbb{S}_+^{n\times n}$ and $\mathbf{b} \in \mathbb{R}_+$. The set P is a convex subset of $\mathbb{R}^n$.

Example. Let $\|.\|$ be any norm in $\mathbb{R}^n$. Then, the unit ball $B = \{\mathbf{x} \in \mathbb{R}^n | \|\mathbf{x}\| \leq b, b > 0\}$ is convex.

Let S_1, S_2 be convex set, then:

- $S_1 \cap S_2$ is convex set
- $S_1 \oplus S_2$ (Minkowski addition) is convex set, where

$$S_1 \oplus S_2 = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{x} = \mathbf{x_1} + \mathbf{x_2}, \mathbf{x_1} \in S_1, \mathbf{x_2} \in S_2\} \tag{3.1}$$

- $S_1 \ominus S_2$ is convex set, where

$$S_1 \oplus S_2 = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{x} = \mathbf{x_1} - \mathbf{x_2}, \mathbf{x_1} \in S_1, \mathbf{x_2} \in S_2\} \tag{3.2}$$

- $f(S_1)$ is convex iff $f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}, \mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{b} \in \mathbb{R}^m$

Definition 3.1.2 (convex hull)**.** Let S be an arbitrary set in $\mathbb{R}^n$. The **convex hull** of S, denoted by $conv(S)$ is the collection of all convex combinations of elements in S

$$\mathbf{x} \in conv(S) \iff \mathbf{x} = \sum_{j=1}^{k} \lambda_j \mathbf{x}_j \tag{3.3}$$

$$\sum_{j=1}^{k} \lambda_j = 1 \tag{3.4}$$

$$\lambda_j \leq 0, \quad j = 1, \cdots, k \tag{3.5}$$

$$\mathbf{x}_1, \mathbf{x_2}, \cdots, \mathbf{x_k} \in S \tag{3.6}$$

Lemma 3.1. *Let S be arbitrary set in $\mathbb{R}^n$. Then $conv(S)$ is the smallest convex set containing S, which means $conv(S)$ is the intersection of all convex sets containing S.*

Theorem 3.2 (Carathéodory's Theorem)**.** *Let $S \subseteq \mathbb{R}^n$. Then $\forall \mathbf{x} \in conv(S)$, there exists $\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^p \in S$, where $p \leq n+1$ such that $\mathbf{x} \in conv\{\mathbf{x}^1, \mathbf{x}^2, ...\mathbf{x}^p\}$.*

Notice: This theorem means, any point $\mathbf{x} \in \mathbb{R}^n$ in a convex hull of S, i.e., $conv(S)$, can be included in a convex subset $S' \subseteq conv(S)$ that has $n+1$ extreme points.

Theorem 3.3. *Let S be a convex set with nonempty interior. Let $\mathbf{x}_1 \in cl(S)$ and $\mathbf{x}_2 \in int(S)$, then $\mathbf{y} = \lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \in int(S), \forall \lambda \in (0,1)$*

3.2 Convex Functions

Definition 3.2.1. Let $C \in \mathbb{R}^n$ be a convex set. A function $f : C \to \mathbf{R}$ is (resp. strictly) convex if

$$f(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2) \tag{3.7}$$
$$\forall \mathbf{x}_1, \mathbf{x}_2 \in C, \forall \lambda \in (0,1) \tag{3.8}$$

(resp.)

$$f(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) < \lambda f(\mathbf{x}_1) + (1-\lambda) f(\mathbf{x}_2) \tag{3.9}$$
$$\forall \mathbf{x_1} \neq \mathbf{x}_2 \in C, \forall \lambda \in (0,1) \tag{3.10}$$

Notice: When calling a function convex, we imply that its domain is convex.

Example. Given any norm $\|.\|$ on $\mathbb{R}^n$, the function $f(x) = \|x\|$ is convex over $\mathbb{R}^n$.

Definition 3.2.2. Let S be a nonempty convex subset of $\mathbb{R}^n$, $f : S \to \mathbb{R}$ is (resp. strictly) **concave** if $-f(x)$ is (resp. strictly) convex.

Notice: A function may be neither convex nor concave.

Theorem 3.4. *Consider $f : \mathbb{R}^n \to \mathbb{R}$. $\forall \bar{\mathbf{x}} \in \mathbb{R}^n$ and a nonzero direction $\mathbf{d} \in \mathbb{R}^n$. Define $F_{\bar{\mathbf{x}},d}(\lambda) = f(\bar{\mathbf{x}} + \lambda \mathbf{d}$. Then f is (resp. strictly) convex iff $F_{\bar{\mathbf{x}},d}(\lambda)$ is (resp. strictly) convex for all $\bar{\mathbf{x}} \in \mathbb{R}^n, \forall \mathbf{d} \in \mathbb{R}^n \setminus \{0\}$.*

Definition 3.2.3 (Level-set)**.** Given a function $f : \mathbb{R}^n \to \mathbb{R}$ and a scalar $\alpha \in \mathbb{R}$, we refer to the set $S_\alpha = \{\mathbf{x} \in S | f(\mathbf{x}) \leq \alpha\} \subseteq \mathbb{R}^n$ as the **α-level-set** of f.

Lemma 3.5. *Let S be a nonempty convex set in $\mathbb{R}^n$. Let $f : S \to \mathbb{R}^n$ be a convex function, then the **α-level-set** of f is a convex set for each value of $\alpha \in \mathbb{R}$.*

Notice: The converse is not necessarily true.

Definition 3.2.4 (Epigraphs, Hypographs)**.** Let $S \in \mathbb{R}^n$ be such that $S \neq \emptyset$. The **epigraph** of f, denoted by $epi(f)$ is

$$epi(f) = \{(\mathbf{x}, y) \in S | \mathbf{x} \in S, y \in \mathbb{R}, y \geq f(x)\} \in \mathbb{R}^{n+1} \tag{3.11}$$

The **hypograph** of f, denoted by $hypo(f)$ is

$$hypo(f) = \{(\mathbf{x}, y) \in S | \mathbf{x} \in S, y \in \mathbb{R}, y \leq f(x)\} \in \mathbb{R}^{n+1} \tag{3.12}$$

Theorem 3.6. *Let S be a nonempty convex subset in $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$. Then f is convex iff $epi(f)$ is convex.*

Theorem 3.7. *Let S be a nonempty convex subset in $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$ be a convex function on S. Then f is continuous in $int(S)$.*

3.3 Subgradients and Subdifferentials

Definition 3.3.1 (Subgradient)**.** Let S be a nonempty convex set in $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$ be a convex function, then ξ is a **subgradient** of f at $\bar{\mathbf{x}}$ if

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \xi^\top(\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \in S \tag{3.13}$$

Definition 3.3.2 (Subdifferential)**.** The set of all subgradients of f at $\bar{\mathbf{x}}$ is called **subdifferential** of f at $\bar{\mathbf{x}}$, denoted as $\partial f(\bar{\mathbf{x}})$

Theorem 3.8. *Let S be a nonempty convex set in $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$ be a convex function. Then for $\bar{\mathbf{x}} \in int(S)$, there exists a vector ξ such that*

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \xi^\top(\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \in S \tag{3.14}$$

In particular, the hyperplane

$$\mathcal{H} = \{(\mathbf{x}, y) | y = f(\bar{\mathbf{x}}) + \xi^\top(\mathbf{x} - \bar{\mathbf{x}})\} \tag{3.15}$$

is a supporting plane of $epi(f)$ at $(\bar{\mathbf{x}}, f(\bar{\mathbf{x}}))$

Theorem 3.9. *Let S be a nonempty convex set in $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$ be a convex function. Suppose that for each $\bar{\mathbf{x}} \in S$, there exists ξ such that*

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \xi^\top(\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \in S \tag{3.16}$$

Then f is convex on $int(S)$

Notice: Not all convex functions are continuous, it has to be continuous in its interior, but it may not be continuous at the boundary.

3.4 Differentiable Functions

Definition 3.4.1 (Differentiable Functions)**.** Let S be a nonempty subset of $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$.Then f is said to be **differentiable** at $\bar{\mathbf{x}} \in int(S)$ if there exists a vector $\nabla f(\bar{\mathbf{x}})$ and a function $\alpha : \mathbb{R}^n \to \mathbb{R}$ such that

$$f(\mathbf{x}) = f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top(\mathbf{x} - \bar{\mathbf{x}}) + \alpha(\bar{\mathbf{x}}, \mathbf{x} - \bar{\mathbf{x}})\|\mathbf{x} - \bar{\mathbf{x}}\| \tag{3.17}$$

for all $\mathbf{x} \in S$ where $\lim_{\mathbf{x} - \bar{\mathbf{x}}} \alpha(\bar{\mathbf{x}}, \mathbf{x} - \bar{\mathbf{x}}) = 0$

Remark. If function f is differentiable, then $\nabla f(\bar{\mathbf{x}}) = (\frac{\partial f(\bar{\mathbf{x}})}{\partial \mathbf{x}_1}, \frac{\partial f(\bar{\mathbf{x}})}{\partial \mathbf{x}_2}, \cdots, \frac{\partial f(\bar{\mathbf{x}})}{\partial \mathbf{x}_n})$, and the gradient is unique.

Lemma 3.10. *Let $S \neq \emptyset$ be a convex set of $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$ be convex. If f is differentiable at $\bar{\mathbf{x}} \in int(S)$, then the subdifferential of f at $\bar{\mathbf{x}}$ is the singleton, $\{\nabla f(\bar{\mathbf{x}})\}$*

Theorem 3.11. *Let S be a nonempty subset of $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$ be differentiable on S. Then f is (resp. strictly) convex on S iff $\forall \bar{\mathbf{x}} \in S$*

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \in S \tag{3.18}$$

(resp.)

$$f(\mathbf{x}) > f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \neq \bar{\mathbf{x}} \in S \tag{3.19}$$

Theorem 3.12 (Mean-value Theorem)**.** *Let S be a nonempty subset of $\mathbb{R}^n$. Let $f : S \to \mathbb{R}$ be differentiable on S. Then for all $\mathbf{x}_1, \mathbf{x}_2 \in S$, there exists $\lambda \in (0, 1)$ such that*

$$f(\mathbf{x}_2) = f(\mathbf{x}_2) + \nabla f(\hat{\mathbf{x}})(\mathbf{x}_2 - \mathbf{x}_1) \tag{3.20}$$

where

$$\hat{\mathbf{x}} = \lambda \mathbf{x}_1 + (1 - \lambda)\mathbf{x}_2 \tag{3.21}$$

3.5 Convex and Affine Hulls

3.6 Interior and Closure

3.7 Cones

3.8 Hyperplanes

Chapter 4

Basic Concepts of Polyhedral Theory

4.1 Extreme Points

Definition 4.1.1 (extreme points)**.** Let $S \subseteq \mathbb{R}^n$, an **extreme point** of S is a point $x \in S$ such that

$$\begin{cases} x = \sum_{i=1}^m \lambda_i x^i \\ \sum_{i=1}^m \lambda_i = 1 \\ \lambda_i > 0, i = 1, \cdots, m \\ x^i \in S, i = 1, \cdots, m \end{cases} \quad \text{implies} \quad x^i = x, i = 1, \cdots, m \tag{4.1}$$

which means, if x is an extreme point, it cannot be written as a convex combination of points in S except the copies of x itself.

Theorem 4.1. *Let P be a polyhedron in $\mathbb{R}^n$ and $v \in P$. Then v is an extreme point of P iff $dim(span\{a^i | (a^i)^t opv = b_i\}) = n$*

This theorem means, at an extreme point of a n-dimensional polyhedron, there will be n active inequalities. A direct result will be the equivalence between basic feasible solutions and extreme points in Linear Programming.

4.2 Polar Cones

4.3 Polyhedral and Dimension

4.3.1 Polyhedral

Definition 4.3.1 (polyhedron)**.** A **polyhedron** is a set of the form $\{x \in \mathbb{R}^n | Ax \leq b\} = \{x \in \mathbb{R}^n | a^i x \leq b^i, \forall i \in M\}$, where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$

Proposition 3. *A polyhedron is a convex set.*

Definition 4.3.2 (polytope)**.** A polyhedron $P \subset \mathbb{R}^n$ is **bounded** if there exists a constant K such that $|x_i| < K, \forall x \in P, \forall i \in [1, n]$, in this case the polyhedron is call **polytope**. The lower-bound of K is called **diagonal** denoted by d

Definition 4.3.3 (cone)**.** $C \subseteq \mathbb{R}^n$ is a **cone** if $x \in C$ implies $\lambda x \in C, \forall \lambda \in \mathbf{R}_+$

4.3.2 Dimension of Polyhedral

Definition 4.3.4 (dimension)**.** A polyhedron P is **dimension** k, denoted $dim(P) = k$, if the maximum number of affinely independent points in P is $k + 1$

Definition 4.3.5 (full-dimensional)**.** A polyhedron $P \subseteq \mathbb{R}^n$ is **full-dimensional** if $dim(P) = n$

Proposition 4. *If $P \subseteq \mathbb{R}^n$, then $dim(P) = n - rank(A^=, b^=)$*

To proof a constraint $(A^=, b^=)$ is an equality constraint, we need to proof all point in the closure of P satisfied the constraint, to proof it is not an equality constraint, we need to find one point that is not in the hyperplane.

Definition 4.3.6 (inner point, interior point)**.** $x \in P$ is called an **inner point** of P if $a^i x < b_i, \forall i \in M^{\leq}$, $x \in P$ is called an **interior point** of P if $a^i x < b_i, \forall i \in M$

Corollary 4.1.1. *Every nonempty polyhedron has at least one inner point.*

Corollary 4.1.2. *A polyhedron has an interior point iff P is full-dimensional, i.e., there is no equality constraint.*

4.4 Face and Facet

4.4.1 Valid Inequalities and Faces

The inequality denoted by (π, π_o) is called a **valid inequality** for P if $\pi x \leq \pi_0, \forall x \in P$. Note that (π, π_0) is a valid inequality iff P lies in the half-space $\{x \in \mathbb{R}^n | Ax \leq b\}$
- If (π, π_0) is a valid inequality for P and $F = \{x \in P | \pi x = x_0\}$, F is called a **facet** of P and we say that (π, π_0)

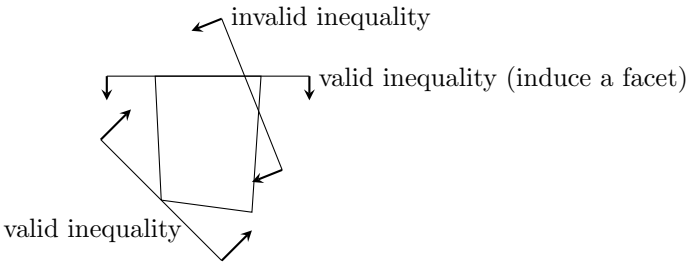


Figure 4.1: Example of valid/invalid inequality

represents or **defines** F
- A face is said to be **proper** if $F \neq \emptyset$ and $F \neq P$
- The face represented by (π, π_0) is nonempty iff $\max\{\pi x | x \in P\} = \pi_0\}$
- If the face F is nonempty, we say it **supports** P
- Let P be a polyhedron with equality set $M^=$. If

$$F = \{x \in P | \pi^T x = \pi_0\} \tag{4.2}$$

is not empty, then F is a polyhedron. Let

$$M^= \subseteq M_F^=, M_F^{\leq} = M \setminus M_F^= \tag{4.3}$$

then

$$F = \{x | a_i^T x = b_i, \forall i \in M_F^=, a_i^T x \leq b_i, \forall i \in M_f^{\leq}\} \tag{4.4}$$

4.4.2 Facet

- A face F is said to be a **facet** of P if $dim(F) = dim(P) - 1$
- Facets are all we need to describe polyhedral
- If F is a facet of P, then in any description of P, there exists some inequality representing F
- Every inequality that represents a face that is not a facet is unnecessary in the description of P - Every full-dimensional polyhedron P has a unique (up to scalar multiplication) representation that consists of one inequality representing each facet of P
- If $dim(P) = n - k$ with $k > 0$, then P is described by a maximal set of linearly independent rows of $(A^=, b^=)$, as well as one inequality representing each facet of P

4.4.3 Proving Facet

To prove an inequality $\sum_i a_i x_i \leq b_i$ is facet inducing for a D dimensional polyhedral, we need to prove there are D affinely independent vectors in $\sum_i a_i x_i = b_i$

4.4.4 Domination

$\Pi x \leq \Pi_0$ dominates $Mx \leq M_0$ if

$$\begin{cases} \Pi \geq \mu M, \mu > 0 \\ \Pi_0 \leq \mu M_0, \mu > 0 \\ (\Pi, \Pi_0) \neq (M, M_0) \end{cases} \tag{4.5}$$

Chapter 5

Basic Concepts of Real Analysis

5.1 The Real Number System

5.2 Open Sets and Closed Sets

Definition 5.2.1 (neighborhood)**.** $N_\epsilon = \{y \in \mathbb{R}^n | \|y - x\| < \epsilon\}$ as the **neighborhood** of $x \in \mathbb{R}^n$

Definition 5.2.2 (interior)**.** Given $S \subseteq \mathbb{R}^n$, x belongs to the **interior** of S, denoted by $int(S)$ if there is $\epsilon > 0$ such that $N_\epsilon(x) \subseteq S$

Definition 5.2.3 (boundary)**.** x belongs to the **boundary** ∂S if $\forall \epsilon > 0$, $N_\epsilon(x)$ contains at least one point in S and a point not in S

Definition 5.2.4 (closure)**.** $x \in S$ belongs to the **closure** of S, denoted $cl(s)$ if $\forall \epsilon > 0$, $N_\epsilon(x) \cap S = \emptyset$

Definition 5.2.5 (metric space)**.** A **metric space** is a set X where we have a notion of distance. That is, if $x, y \in X$, then $d(x, y)$ is the distance between x and y. The particular distance function must satisfy the following conditions:

- $d(x, y) > 0, \forall x, y \in X$
- $d(x, y) = 0 \iff x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$

Definition 5.2.6 (ball)**.** Let X be a metric space. A **ball** B of radius r around a point $x \in X$ is

$$B = \{y \in X | d(x, y) < r\} \tag{5.1}$$

Definition 5.2.7 (open set)**.** A subset $O \subseteq X$ is **open** if $\forall x \in O, \exists r, B = \{x \in X | d(x, y) < r\} \subseteq O$

S is said to be an **open set** iff $S = int(S)$

Theorem 5.1. *The union of any collection if open sets is open.*

Proof. Sets $S_1, S_2, ..., S_n$ are open sets, let $S = \cup_{i=1}^n S_i$, then $\forall i, S_i \subseteq S$. $\forall x \in S, \exists i, x \in S_i$. Given that S_i is an open set, then for x, $\exists r$ that $B = \{x \in S_i | d(x, y) < r\} \subseteq S_i \subseteq S$, therefore S is an open set. □

Theorem 5.2. *The intersection of any finite number of open sets is open.*

Proof. Sets $S_1, S_2, ..., S_n$ are open sets, let $S = \cap_{i=1}^n S_i$, then $\forall i, S \subseteq S_i$. $\forall x \in S, x \in S_i$. For any i, we can define an r_i, such that $B_i = \{x \in S_i | d(x, y) < r_i\} \subseteq S_i$. Let $r = \min_i\{r_i\}$. Noticed that $\forall i, B' = \{x \in S_i | d(x, y) < r\} \subseteq B_i \subseteq S_i$. Therefore S is an open set. □

Remark. The intersection of infinite number of open sets is not necessarily open.

Here we find an example that the intersection of infinite number of open sets can be closed.

Example. Let $A_n \in \mathbb{R}$ and $B_n \in \mathbb{R}$ be two infinite series, with the following properties. First, $\forall n, A_n < a, \lim A_n = a$, second, $\forall n, B_n > b, \lim B_n = b$, third $a < b$. Then we define infinite number of sets S_i, the ith set is defined as

$$S_i = (A_i, B_i) \subset \mathbb{R} \tag{5.2}$$

Then

$$S = \cap_{i=1}^{\infty} S_i = [a, b] \subset \mathbb{R} \tag{5.3}$$

and S is a closed set.

Definition 5.2.8 (limit point)**.** A point z is a **limit point** for a set A if every open set U that $z \in U$ intersects A in a point other than z.

Notice: z is not necessarily in A.

Definition 5.2.9 (closed set)**.** A set C is **closed** iff it contains all of its limit points.

S is called **closed** iff $S = cl(S)$

Theorem 5.3. *$S \in \mathbb{R}^n$ is closed $\iff \forall \{x_k\}_{k=1}^{\infty} \in S, \lim_{k\to\infty} \{x_k\}_{k=1}^{\infty} \in S$*

Theorem 5.4. *Every intersection of closed sets is closed.*

Theorem 5.5. *Every finite union of closed sets is closed.*

Remark. The union of infinite number of closed sets is not necessarily closed.

Theorem 5.6. *A set C is a closed set if $X \setminus C$ is open*

Proof. Let S be an open set, $x \notin S$, for any open set S_i that $x \in S_i$, we can find a correspond $r_i > 0$, such that $B_i = \{x \in S_i | d(x, y) < r_i\}$. Take $r = \min_{\forall i}\{r_i\}$, set $B = \{x \notin S | d(x, y) < r\} \neq \emptyset$. Which means for any $x \notin S$, we can find at least one point $x' \in B$ that for all open set S_i, $x' \in S_i$, which makes x a limit point of the complement of the open set. Notice that x is arbitrary, then the collection of x, i.e., the complement of S is a closed set. □

Remark. The empty set is open and closed, the whole space X is open and closed.

5.3 Functions, Sequences, Limits and Continuity

5.4 Differentiation

5.5 Integration

5.6 Infinite Series of Constants

5.7 Power Series

5.8 Uniform Convergence

5.9 Arcs and Curves

5.10 Partial Differentiation

5.11 Multiple Integrals

5.12 Improper Integrals

5.13 Fourier Series

Chapter 6

Basic Concepts of Probability Theory

6.1 Relationship between Some Random Variables

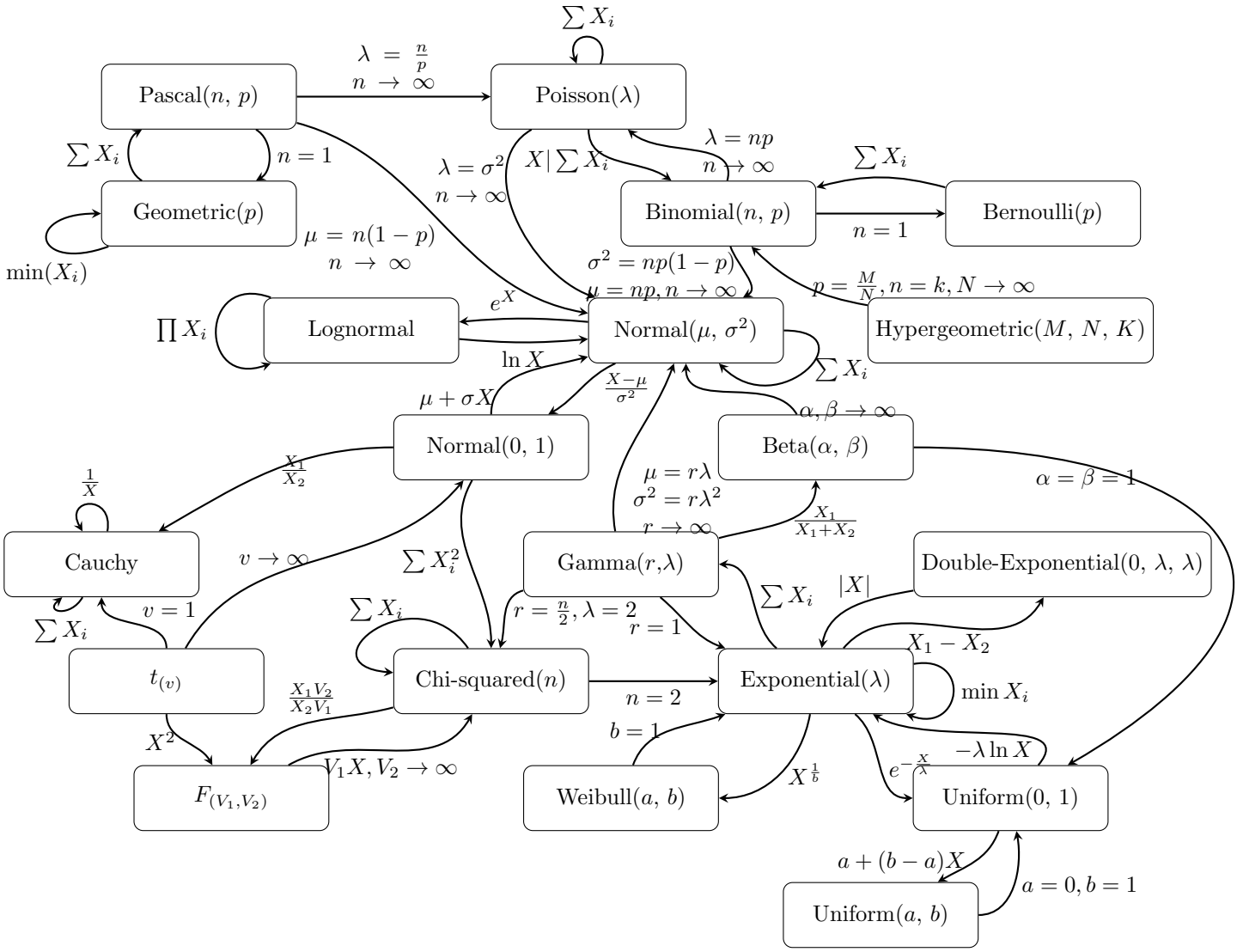


Figure 6.1: Relationship between Some Random Variables

6.2 Discrete Random Variables

Distribution	PMF	CDF	Exp.	Var.	MGF
Uniform(a,b)	$\frac{1}{b-a+1}$ $x=a,a+1,...,b$	$\frac{x-a+1}{b-a+1}$ $x=a,a+1,...,b$	$\frac{b-a}{2}$	$\frac{(b-a+1)^2-1}{12}$	$\frac{e^{at}-e^{(b+1)t}}{(b-a+1)(1-e^t)}$ $t\in\mathbb{R}$
Bernoulli(p)	$p^x(1-p)^{1-x}$ $x\in\{0,1\}$	$\begin{cases}0, & x<0\\ 1-p, & 0\le x\le 1\\ 1, & x>1\end{cases}$	p	$p(1-p)$	$1-p+pe^t$ $t\in\mathbb{R}$
Binomial(n,p)	$\binom{n}{x}p^x(1-p)^{n-x}$ $x=0,1,...,n$	$\sum_{k=0}^{x}\binom{n}{k}p^k(1-p)^{n-k}$ $x=0,1,...,n$	np	$np(1-p)$	$(1-p+pe^t)^n$ $t\in\mathbb{R}$
Poisson(μ)	$\frac{\mu^x e^{\mu}}{x!}$ $x=0,1,...,n,...$	$\frac{\Gamma(x+1,\mu)}{\Gamma(x+1)}$ $x=0,1,...,n,...$	μ	μ	$e^{\mu(e^t-1)}$ $t\in\mathbb{R}$
Geometric(p)	$p(1-p)^x$ $x=0,1,...,n,...$	$1-(1-p)^{x+1}$ $x=0,1,...,n,...$	$\frac{1-p}{p}$	$\frac{1-p}{p^2}$	$\frac{p}{1-(1-p)e^t}$ $t<-\ln(1-p)$
Pascal(n,p)	$\binom{n-1+x}{x}p^n(1-p)^x$ $x=0,1,2,...,n,...$	$1-I_p(k+1,n)$ $x=0,1,2,...,n,...$	$\frac{n(1-p)}{p}$	$\frac{n(1-p)}{p^2}$	$(\frac{p}{1-(1-p)e^t})^n$ $t<-\ln(1-p)$

(6.1)

6.3 Continuous Random Variables

Distribution	PDF	CDF	Exp.	Var.	MGF
Uniform(a,b)	$\frac{1}{b-a}$ $x=[a,b]$	$\frac{x-a}{b-a}$ $x=[a,b]$	$\frac{b-a}{2}$	$\frac{(b-a)^2}{12}$	$\begin{cases}1, & t=0\\ \frac{e^{bt}-e^{at}}{t(b-a)}, & t\neq 0\end{cases}$
Normal(μ,σ)	$\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ $x\in\mathbb{R}$	$\int_{-\infty}^{x}\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ $x\in\mathbb{R}$	μ	σ^2	$e^{\frac{t(t\sigma^2+2\mu)}{2}}$ $t\in\mathbb{R}$
Exponential(λ)	$\lambda e^{-\lambda x}$ $x>0$	$1-e^{\lambda x}$ $x>0$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$	$\frac{1}{1-\frac{t}{\lambda}}$ $t<\lambda$
Erlang(n,λ)	$\frac{\lambda^n x^{n-1}e^{-\lambda x}}{(n-1)!}$ $x>0$	$1-\sum_{i=0}^{n-1}\frac{\lambda^n x^n e^{-\lambda x}}{n!}$ $x>0$	$\frac{n}{\lambda}$	$\frac{n}{\lambda^2}$	$\frac{1}{(1-\frac{t}{\lambda})^n}$ $t<\lambda$

(6.2)

Chapter 7

Basic Concepts of Computational and Complexity Theory

7.1 Finite Automata and Regular Languages

7.2 Pushdown Automata and Context-Free Languages

7.3 Turing Machines and the Church-Turing Thesis

7.3.1 Turing Machine

7.3.2

7.4 Computational Complexity

7.4.1 Asymptotic Notation

Definition 7.4.1 (asymptotically positive function)**.** $f : \mathbb{N} \to \mathbb{R}$ is an asymptotically positive function if $\exists n_0 > 0$ such that $\forall n > n_0$ we have $f(n) > 0$

Definition 7.4.2 (O-Notation)**.** For a function $g(n)$,

$$O(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0\}$$

O-Notation also known as asymptotic upper bound.

Definition 7.4.3 (Ω-Notation)**.** For a function $g(n)$,

$$\Omega(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \geq cg(n), \forall n \geq n_0\}$$

Ω-Notation also known as asymptotic lower bound.

Definition 7.4.4 (Θ-Notation)**.** For a function $g(n)$,

$$\Theta(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

Ω-Notation and Θ-Notation are not used very often when we talk about running times.

Definition 7.4.5 (o-Notation)**.** For a function $g(n)$,

$$o(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } f(n) < cg(n), \forall n \geq n_0\}$$

Definition 7.4.6 (ω-Notation)**.** For a function $g(n)$,

$$\omega(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } f(n) > cg(n), \forall n \geq n_0\}$$

Notice: $O(g)$, $\Omega(g)$ and $\Theta(g)$ are sets, we use "=" to represent "$\in$". In here "=" is asymmetric. Equality such as $O(n^3) = n^3 + n$ is incorrect.

Example. The following are some examples

$f(n)$	$g(n)$	O	Ω	Θ
$4n^2 + 3n$	$n^3 - 2n + 3$	Yes	No	No
$\lg^{10} n$	$n^{0.1}$	Yes	No	No
$\log_{10} n$	$\lg(n^3)$	Yes	Yes	Yes
$\lceil\sqrt{10n + 100}\rceil$	n	Yes	No	No
$n^3 - 100n$	$10n^2 \lg n$	No	Yes	No
2^n	$2^{\frac{n}{2}}$	No	Yes	No
$\sqrt{n}$	$n^{\sin n}$	No	No	No

(7.1)

Theorem 7.1. *Let f and g be two functions that*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$$

Then $f(n) = \Theta(g(n))$

Proof. Since $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and positive, there is some n_0 beyond which the ratio is always between $\frac{1}{2}c$ and 2c. Thus,

$$\forall n > n_0, f(n) \le 2cg(n) \Rightarrow f(n) = O(g(n)) \tag{7.2}$$
$$\forall n > n_0, f(n) \ge \frac{1}{2}cg(n) \Rightarrow f(n) = \Omega(g(n)) \tag{7.3}$$

□

A set of properties:

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \tag{7.4}$$
$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } g(n) = O(f(n)) \tag{7.5}$$

Another set of properties:

$$f = O(g), g = O(h) \Rightarrow f = O(h) \tag{7.6}$$
$$f = \Omega(g), g = \Omega(h) \Rightarrow f = \Omega(h) \tag{7.7}$$
$$f = \Theta(g), g = \Theta(h) \Rightarrow f = \Theta(h) \tag{7.8}$$

Theorem 7.2. *If $f_i = O(h)$, for finite number of $i \in K$, then $\sum_{i\in K} f_i = O(h)$*

Proof is trivia.

Notice: Function f and g not necessarily have relation $f = O(g)$ or $g = O(f)$. E.g., for $f = \sqrt{n}$ and $g = n^{\sin n}$, $f \notin O(g)$ and $g \notin O(f)$

7.4.2 Common Running Times

The following are examples of common running times.

Running Time	Examples
$O(n)$	Scan through a list to find a element matching with input
$O(\lg n)$	Binary search
$O(n^2)$	Scan through every pair of elements, $\binom{n}{2}$
$O(n^3)$	Matrix multiplication by definition
$O(n \lg n)$	Many divide and conquer algorithm which in each step iteratively divide the problem into two part and solve the subproblem, for example mergesort.
$O(n!)$	Enumerate all permutation, for example Hamiltonian Cycle Problem
$O(c^n)$	Enumerate all elements in power set. ($O(2^n)$)
$O(n^n)$	Enumerate all combinations. (Can't find good example yet)

Here is a comparison between running times: $\lg n < \sqrt{n} < n < n \lg n < n^2 < n^{\sqrt{n}} < 2^n < e^n < n! < n^n$

7.5 NP and Computational Intractability

7.5.1 P, NP and Co-NP

Definition 7.5.1 (Decision Problem)**.** A problem X is a **decision problem** if the output is either 0 or 1 (yes/no).

The input of a problem can always be encoded into a binary string. The size of an input is the length of the encoded string s for the input.

Notice: For optimization problem X, we can always define a decision version X', by giving a threshold and ask if the objective function can satisfy that threshold or not. If that decision version X' can be solved in polynomial time, we can solve the original problem X in polynomial time.

Definition 7.5.2 (Polynomial running time)**.** An algorithm A has a **polynomial running time** if there is a polynomial function $p(\cdot)$ such that $\forall s$, the algorithm A terminates on s in at most $p(|\cdot|)$ steps.

Definition 7.5.3 (P)**.** The complexity class P is the set of decision problems X that can be solved in polynomial time.

Example. Shortest path, minimum spanning tree, determine if an integer is prime number, those are in P.

Definition 7.5.4 (Certificate, Certifier)**.** In order to check the an algorithm A, for a binary string s, such that $s \in A$, there is another separated algorithm B uses s and t, a separated string that contains the evidence that s is a "yes" instance of X, as input string and output (another) "yes". Then this string t is called **certificate**, this separated algorithm $B(s, t)$ is called **certifier**.

Notice: Certificate is like a solution, and certifier is the algorithm to prove solution is correct. But both with fancy terminology

Example. For Independent set problem. The input (s) is a graph, the certificate (t) will be a set of size k, and the certifier will be an algorithm to check the given set is an independent set.

Example. For 3-SAT problem. The input s will be the 3-CNF (conjunctive normal form), the certificate t will be the assignment of true values for each terms in the 3-CNF. The certifier will be the algorithm calculates the true value of the clause given the true values in t.

Definition 7.5.5 (Efficient Certifier)**.** B is an efficient certifier for problem X if

- B runs in polynomial time with input s and t. (s is the input of origin algorithm and t is the certificate)

- There is a polynomial function $p(|\cdot|)$ such that for every string s, we have $s \in X$ (s is a "yes" solution for X) and $B(s,t) = yes$ (the certifier returns "yes")

Definition 7.5.6 (NP)**.** The complexity class NP is the set of all problems for which there exists an efficient certifier.

From the definition of NP we can immediately know that $P \subseteq NP$. Because all the problems in P can satisfy the definition of NP. But is there a problem that $X \in NP$ and $X \notin P$? Solve that problem and you will win 100 million dollars and the chance to be remember forever.

Notice: NP stands for **Non-deterministic Polynomial** time

To prove if a problem is NP, we need to have a certifier that can output "yes" given the certificate in polynomial time. Otherwise it is not in NP.

Example. Given a graph $G = (V, E)$ and an integer $t > 0$, whether the minimum vertex cover of G has size at least t. This problem is **unlikely** in NP. Reason is, the "yes" instance is "all minimum vertex cover has size at least t", that is not **likely to be** solvable in polynomial time for it need to find all minimum vertex covers.

Definition 7.5.7 (Co-NP)**.** For a problem X, the problem $\bar{X}$ is the problem such that $s \in \bar{X} \iff s \notin X$. Then **Co-NP** is the set of decision problems X such that $\bar{X} \in NP$.

Example (Tautology Problem)**.** Given a boolean formula, determine whether the formula is always evaluates to 1. This is a problem in Co-NP. Because we can have a polynomial time certifier to confirm that an instance is not a tautology.

Example. Given two boolean formulas, to determine whether or not they are equivalent. This is a problem in Co-NP. Because if we have one instance such that the output is "no", then we can easily prove there are counter examples in origin algorithm.

Notice: If the instance is "easy" to prove to be true, then it is in NP, if the counter instance is "easy" to prove to be false, then it is in Co-NP

Relation between P, NP and $Co - NP$ is as following

- $P \subseteq NP$
- $P \subseteq Co - NP$
- $P = NP$? is not known
- $P = Co - NP$? is not known
- $NP = Co - NP$? is not known
- If $P = NP$ then $P = Co - NP$

7.5.2 Polynomial-Time Reductions

Definition 7.5.8 (Polynomial-time reducible)**.** Given an algorithm A that solves problem Y, if any instance of problem X can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to algorithm A, then we say X is **polynomial-time reducible** to Y, denoted as

$$X \leq_P Y \tag{7.9}$$

In a more intuitive way, $X \leq_P Y$ means X can't be more difficult than Y. Solving X can be "transforming" X into an equivalent Y in polynomial number of steps and then solve it by calling Y polynomial number of times, usually one time. Thus, if $X \leq_P Y$,

- If Y can be efficiently solved, X can be efficiently solved.
- If X cannot be efficiently solve, Y cannot be efficiently solved.

Notice: To prove $X \leq_P Y$, usually we already have an algorithm for Y, could be polynomial or not.

Lemma 7.3. *Hamiltonian-Path $\leq_P$ Hamiltonian-Cycle*

Proof. For graph $G = (V, E)$, s and t are two vertices that in V, define a new graph $G' = (V \cup \{v\}, E \cup \{(u, s)\} \cup \{(t, u)\}$. To solve the Hamilton-Path from s to t in graph G, say p_{st}, is equivalent to solving Hamilton-Cycle problem in G', i.e., find $sp_{st}te_{tu}ue_{us}$. □

Lemma 7.4. *Hamiltonian-Cycle $\leq_P$ Hamiltonian-Path*

Proof. For vertex s, make a copy and denote it as s', s' is connected to all the vertices that s connected. Solving the Hamiltonian-Cycle problem is equivalent to solving the Hamiltonian-Path problem from s to s' □

Lemma 7.5. *Hamiltonian-Path $\leq_P$ degree-3 spanning tree*

Proof. In graph G, for vertex s and t, add vertices s', s'', t', t'' and edges (s, s'), (s, s''), (t, t'), (t, t''). For all the other vertex $u \in V \setminus \{s\} \setminus \{t\}$, add vertices u' and edge (u, u') to the graph. Then solving Hamiltonian-Path problem is equivalent to solve the degree spanning tree problem in this new G. □

Lemma 7.6. *Vertex Cover $\leq_P$ Set Cover*

Proof. □

Lemma 7.7. *Set Cover $\leq_P$ Vertex Cover*

Proof. □

Lemma 7.8. *Clique $\leq_P$ Independent Set*

Proof. S is a clique in $G = (V, E)$ iff S is an independent set in $\bar{G} = (V, \bar{E})$ □

Lemma 7.9. *Independent Set $\leq_P$ Clique*

Proof. S is an independent set in $G = (V, E)$ iff S is a clique in $\bar{G} = (V, \bar{E})$ □

Lemma 7.10. *Vertex-Cover $\leq_P$ Independent Set*

Proof. S is a vertex-cover of $G = (V, E)$ iff $V \setminus S$ is an independent set of G □

Lemma 7.11. *3-Coloring $\leq_P$ 4-Coloring*

Proof. For a graph $G = (V, E)$, define a new graph $G' = (V \cup \{u\}, E \cup \{(u, v) | \forall\ v \in V\})$. Solving the 3-Coloring problem is equivalent to solving the 4-Coloring problem in G' □

Lemma 7.12. *Independent Set $\leq_P$ Set Packing*

7.5.3 NP-Completeness

Definition 7.5.9 (NP-Completeness)**.** A problem X is called **NP-Complete** if

- $X \in NP$, and
- $Y \leq_P X, \forall\ Y \in NP$

An intuitive explanation will be, we can regard problems that is NP-Complete to be the most difficult problems in NP. If any of those can be solved in polynomial time, then all problems in NP can be solved in polynomial time.

Theorem 7.13 (Cook's Theorem)**.** *3-SAT is in NP-Complete.*

Proof. □

Theorem 7.14. *2-SAT is in P*

Algorithm 1 Clear(S, x)

Initialize $S' \leftarrow S$
for Each clause c in S' **do**
    if c is a singleton x **then**
        Remove singleton clause x in S'
    else if c is a singleton $\neg x$ **then**
        return null
    else if c is a 2-clause with x **then**
        Remove singleton clause c in S'
    else if c is a 2-clause with $\neg x$ **then**
        Remove $\neg x$ in clause c in S'
return S'

Algorithm 2 2-SAT(S)

while Length of S is greater than 0 **do**
    if S has a singleton x (or $\neg x$) **then**
        $S \leftarrow \text{Clear}(S, x)$ (or $S \leftarrow \text{Clear}(S, \neg x)$)
        if $S == null$ **then**
            return False
    else
        Denote $S = (x_1 \wedge x_2) \vee S_r$
        $S_1 \leftarrow x_1 \vee S_r$
        $S_2 \leftarrow x_2 \vee S_r$
        if 2-SAT(S_1) **then**
            return True
        else if 2-SAT(S_2) **then**
            return True
        else
            return False
return True

Proof. The following process $Clear(S, x)$ can be executed within $O(n)$, in which S is a CNF with at least one clause being singleton x (or $\neg x$), where n is the length of CNF S, if it returns S', S' will be a CNF without x (or $\neg x$) in any of clause in S. Otherwise, if it returns null, then S is not satisfiable.
To determine a 2-SAT CNF satisfiability, the following process $2 - SAT(S)$ can be executed within $O(n)$, where n is the length of CNF S. It returns True if S is satisfiable, False otherwise.
General idea is, if there are singletons in CNF S, if there are conflicts between singletons, the CNF is not satisfiable, otherwise, those singletons can be assigned True values, or False values, one singleton at a time. For the rest of clauses, either it becomes a singleton, or it remains a 2-clause. When we cannot find any singleton in CNF, we generate two sub-CNF, by separating the first 2-clause and keep the remaining of CNF. Then both two sub-CNF will have at least one singleton, we can perform the previous process to those two sub-CNF. If at least one of those sub-CNF is satisfiable, the original CNF is satisfiable, otherwise, if both sub-CNF are not satisfiable, the original CNF is not satisfiable. Therefore 2-SAT CNF Satisfiability is a problem in P. □

Theorem 7.15. *If X is NP-Complete and $X \in P$, then $P = NP$*

Proof. Direct result from Cook's theorem. □

7.5.4 NP-Complete Problems

Part II

Linear Programming

Chapter 8

The Simplex Method - Basic

8.1 Basic Feasible Solutions and Extreme Points

Definition 8.1.1 (Basic Feasible Solutions)**.** Consider the system $\{\mathbf{A_{m\times n}x = b_m}, \mathbf{b_m \geq 0}\}$, suppose $rank(\mathbf{A}, \mathbf{b}) = rank(\mathbf{A}) = m$, we can rearrange the columns of $\mathbf{A}$ so that we have a partition of $\mathbf{A}$. Let $\mathbf{A} = \begin{bmatrix}\mathbf{B} & \mathbf{N}\end{bmatrix}$ where $\mathbf{B}$ is an $m \times m$ invertible matrix, and $\mathbf{N}$ is an $m \times (n-m)$ matrix. The solution $\mathbf{x} = \begin{bmatrix}\mathbf{x_B} \\ \mathbf{x_N}\end{bmatrix}$ to the equation $\mathbf{Ax = b}$, where $\mathbf{x_B = B^{-1}b}$ and $\mathbf{x_N = 0}$ is called **basic solution** of system. If $\mathbf{x_B \geq 0}$, it is called **basic feasible solution** or **B.F.S.**. If $\mathbf{x_B > 0}$ it is called **non-degenerate basic feasible solution**. For $\mathbf{x_B \geq 0}$, if some $x_j = 0$, those components are called **degenerated basic feasible solution**. $\mathbf{B}$ is called the **basic matrix**, $\mathbf{N}$ is called **nonbasic matrix**

Theorem 8.1. $\mathbf{x}$ *is an extreme point* $\iff$ $\mathbf{x}$ *is a basic feasible solution.*

Proof. This proof is lack of details. FIXME
Denote $\mathcal{S}$ as feasible region.
($\Rightarrow$) First, Let $\mathbf{x}$ be a B.F.S., Suppose $\mathbf{x} = \lambda\mathbf{u} + (1-\lambda)\mathbf{v}$, for $\mathbf{u}, \mathbf{v} \in \mathcal{S}, \lambda \in (0,1)$. Let $I = \{i : x_i > 0\}$ be the set of index where the inequality constraint are not tight. Then for $i \notin I$, $x_i = 0$, which implies $u_i = v_i = 0$. $\mathbf{u}, \mathbf{v} \in \mathcal{S} \Rightarrow \mathbf{Au} = \mathbf{Av} = \mathbf{b} \Rightarrow \mathbf{A}(\mathbf{u} - \mathbf{v}) = \mathbf{0} \Rightarrow \sum_{i=1}^{n}(u_i - v_i)a_i = 0$.

-

- if $i \notin I$ then $x_i = 0$, which implies $u_i = v_i = 0$ - $\because \mathbf{Au} = \mathbf{Av} = \mathbf{b}$, $\therefore \mathbf{A}(\mathbf{u} - \mathbf{v}) = \mathbf{0} \Rightarrow \sum_{i=1}^{n}(u_i - v_i)a_i = 0$, $\because u_i = v_i = 0$, for $i \notin I$, it implies $u_i = v_i$ for $i \in I$, Hence $u = v$, x is E.P.

($\Leftarrow$) Second, suppose $\mathbf{x}$ is not B.F.S., i.e. $\{a_i : i \in I\}$ are linearly dependent.
Then there $\exists \mathbf{u} \neq \mathbf{0}, u_i = 0, i \notin I$ such that $\mathbf{Au} = \mathbf{0}$.
Hence, for a small ϵ, $\mathbf{x} = \frac{1}{2}(\mathbf{x} + \epsilon\mathbf{u}) + \frac{1}{2}(\mathbf{x} - \epsilon\mathbf{u})$, $\mathbf{x}$ is not E.P. □

8.2 The Simplex Method

8.2.1 Key to Simplex Method

Cost Coefficient

The cost coefficient can be derived from the following

$$
\begin{aligned}
z &= cx && (8.1)\\
&= c_B x_B + c_N x_N && (8.2)\\
&= c_B(B^{-1}b - B^{-1}Nx_N) + c_N x_N && (8.3)\\
&= c_B B^{-1}b - \sum_{j\in N}(c_B B^{-1}a_j - c_j)x_j && (8.4)\\
&= c_B B^{-1}b - \sum_{j\in N}(z_j - c_j)x_j && (8.5)
\end{aligned}
$$

We denote $z_0 = c_B B^{-1} b$, $z_j = c_B^{-1} a_j$, $\bar{b} = B^{-1} b$ and $y_j = B^{-1} a_j$ for all nonbasic variables.
The formulation can be transformed into

$$\min \quad z = z_0 - \sum_{j \in N} (z_j - c_j) x_j \tag{8.6}$$

$$\text{s.t.} \quad \sum_{j \in N} y_j x_j + x_B = \bar{b} \tag{8.7}$$

$$x_j \geq 0, j \in N \tag{8.8}$$

$$x_B \geq 0 \tag{8.9}$$

In the above formulation, $z_j - c_j$ is the cost coefficient. If $\exists j$ and $z_j - c_j > 0$, it means the objective function can still be optimized. (If $\forall j$, $z_j - c_j \leq 0$, then $z \geq z_0$ for any feasible solution, z is the optimal solution)

Pivot

After finding the most violated $z_j - c_j$, we find a variable, say x_k, where $z_k - c_k = \min\{z_j - c_j\}$ to be the variable leaving the basis.
If there are degenerated variables, we can perform different method to choose variable to enter basis.

Minimum Ratio

$$x_{B_i} = \bar{b}_i - y_{ik} x_k \geq 0 \tag{8.10}$$

Therefore we have the minimum ratio rule

$$x_k = \min_{i \in B} \{ \frac{\bar{b}_i}{y_{ik}}, y_{ik} > 0 \} \tag{8.11}$$

If for the that column all $y_{ik} \leq 0$, unbounded.

8.2.2 Simplex Method Algorithm

The pseudo-code of Simplex Method is given as following:

8.3 Tableau Method for Simplex Method

The following is an example of using tableau to solve simplex method. Initial tableau:

	z	x_1	x_2	x_3	x_4	x_5	RHS
z	1	1	3	0	0	0	0
x_3	0	1	-2	1	0	0	0
x_4	0	-2	$\boxed{1}$	0	1	0	4
x_5	0	5	3	0	0	1	15

(8.12)

Last tableau:

	z	x_1	x_2	x_3	x_4	x_5	RHS
z	1	0	0	0	$-\frac{12}{11}$	$-\frac{7}{11}$	$-\frac{153}{11}$
x_3	0	0	1	1	$\frac{13}{11}$	$\frac{3}{11}$	$\frac{97}{11}$
x_2	0	1	0	0	$\frac{5}{11}$	$\frac{2}{11}$	$\frac{50}{11}$
x_1	1	0	0	0	$-\frac{3}{11}$	$\frac{1}{11}$	$\frac{3}{11}$

(8.13)

- The optimal basic variables are x_3, x_2, x_1. The optimal basis is the columns in the initial tableau with correspond columns

$$B = \begin{pmatrix} \frac{13}{11} & \frac{3}{11} & \frac{97}{11} \\ \frac{5}{11} & \frac{2}{11} & \frac{50}{11} \\ -\frac{3}{11} & \frac{1}{11} & \frac{3}{11} \end{pmatrix} \tag{8.14}$$

Algorithm 3 Simplex Method

Require: Given a basic feasible solution with basis B
Ensure: Optimal objective value $\min z = cx$
Set **B** for basic variables, **N** for nonbasic variables
B $\leftarrow$ all slack variables
N $\leftarrow$ all variables excepts slack variables
for $\forall j$ **do**
    $z_j = c_B B^{-1} a_j = 0$
while $\exists z_j - c_j > 0$ **do**
    $z_j = w a_j - c_j = c_B B^{-1} a_j - c_j$
    $z_k - c_k = \max_{j \in \mathbf{N}} \{z_j - c_j\}$
    $y_k = B^{-1} a_k$
    if $\exists y_{ik} > 0$ **then**
        $\theta_r = \min_{i \in \mathbf{B}} \{\theta_i = \frac{\bar{b}_i}{y_{ik}} : y_{ik} > 0\}$
        B $\leftarrow$ **B**$\backslash\{k\}$
        N $\leftarrow$ **N** $\cup \{k\}$
        B $\leftarrow$ **B** $\cup \{r\}$
        N $\leftarrow$ **N**$\backslash\{r\}$
    else
        Unbounded
$x_B^* = B^{-1} b = \bar{b}$
$x_N = 0$
$z^* = c_B B^{-1} b = c_B \bar{b} \mathbf{a_{B_k}}$

- From the initial tableau, we can see the initial basis is built from slack variables x_3, x_4, x_5. The B^{-1} is the correspond columns in final tableau.

$$B = \begin{pmatrix} 1 & -2 & 1 \\ 0 & 1 & -2 \\ 0 & 3 & 5 \end{pmatrix} \tag{8.15}$$

- The optimal basic variables are x_3, x_2, x_1. Find c_B in the initial tableau.

$$c_B = \begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix} \tag{8.16}$$

- Find $w = c_B B^{-1}$ from the final tableau, correspond to the slack variable.

$$w = c_B B^{-1} = \begin{pmatrix} 0 \\ -\frac{12}{11} \\ -\frac{7}{11} \end{pmatrix} \tag{8.17}$$

8.4 Initial Basis

If some of the constraint is not in $\sum_{i=1}^{n} a_i x_i \leq 0$ form, we cannot add a positive slack variable. In this case, we add an artificial variable other than slack variable.

$$\sum_{i=1}^{n} a_i x_i \geq (or =) 0 \Rightarrow \sum_{j=1}^{n} a_i x_i + x_a = 0 \tag{8.18}$$

Notice that in an optimal solution, $x_a = 0$, otherwise it is not valid.
Artificial variables are only a tool to get the simplex method started.

8.4.1 Two-Phase Method

Two-Phase Method

For **Phase I:**
Solve the following program start with a basic feasible solution $x = 0, x_a = b$, i.e., the artificial variable forms the basis.

$$\begin{aligned} \min \quad & \mathbf{1}x_a && (8.19)\\ \text{s.t.} \quad & Ax + x_a = b && (8.20)\\ & x \geq 0 && (8.21)\\ & x_a \geq 0 && (8.22) \end{aligned}$$

If the optimal $\mathbf{1}x_a \neq 0$, infeasible, stop. Otherwise proceed Phase II. For **Phase II:**
Remove the columns of artificial variables, replace the objective function with the original objective function, proceed to solve simplex method.

Discussion

Case A: $x_a \neq 0$
Infeasible.
Case B.1: $x_a = 0$ and all artificial variables are out of the basis
At the end of Phase I, we derive

x_0	x_B	x_N	x_a	RHS
1	0	0	-1	0
0	I	$B^{-1}N$	B^{-1}	$B^{-1}b$

(8.23)

We can discard x_a columns, (or we can leave it because it keeps track of B^{-1}), and then we do the Phase II

z	x_B	x_N	RHS
1	0	$c_B B^{-1}N - c_N$	$c_B B^{-1}b$
0	I	$B^{-1}N$	$B^{-1}b$

(8.24)

Case B.2: Some artificial variables are in the basis at zero values
This is because of degeneracy. We pivot on those artificial variables, once they leave the basis, eliminate them.

8.4.2 Big M Method

8.4.3 Single Artificial Variable

8.5 Degeneracy and Cycling

8.5.1 Degeneracy

Degeneracy in Simplex Method

If the basic variable x_B is not strictly > 0, i.e. if some basic variable equals to 0, we call it degenerate.

Degeneracy for Bounded Variables

If some basic variables are at their upper bound or lower bound, we call it degenerate.

8.5.2 Cycling

In the degenerate case, pivoting by the simplex rule does not always give a strict decrease in the objective function value, because it may have $b_r = 0$. It is possible that the tableau may repeat if we use the simplex rule.
Geometrically speaking, it means that at the same point - extreme point - it corresponds to more than one feasible solutions, so when we are pivoting, we stays at the same place.
In computer algorithm, we rarely care about cycling because the data in computer is not precise, it is very hard to get into cycling.

8.5.3 Cycling Prevent

Lexicographic Rule

- For entering variable, same as simplex rule
- For leaving variable, if there is a tie, choose the variable with the smallest $\frac{y_{r1}}{y_{rk}}$.

Bland's Rule

- For entering variable, choose the variable with smallest index where $z_j - c_j \leq 0$
- For leaving variable, if there is a tie, choose the variable with smallest index.

Successive Ratio Rule

- Select the pivot column as any column k where $z_k - c_k \leq 0$
- Given k, select the pivot row r as the minimum successive ratio row associated with column k.

In other words, for pivot columns where there is no tie in the usual minimum ratio, the successive ratio rule reduces to the simplex rule

8.6 As a Search Algorithm

8.6.1 Improving Search Algorithm

A simplex method is a search algorithm, for each iteration it finds a not-worse solution, which can be represented as:

$$x^t = x^{t-1} + \lambda_{t-1} d^{t-1} \tag{8.25}$$

Where
- x^t is the solution of the tth iteration
- λ_t is the step length of tth iteration
- d^t is the direction of the tth iteration

For each iteration, it contains three steps:
- Optimality test
- Find direction
- Find the step length

8.6.2 Optimality Test

$$z = cx \tag{8.26}$$
$$= \begin{bmatrix} c_B & c_N \end{bmatrix} \begin{bmatrix} x_B \\ x_N \end{bmatrix} \tag{8.27}$$
$$= c_B x_B + c_N x_N \tag{8.28}$$
$$\text{and } \because Ax = b \tag{8.29}$$
$$\therefore Bx_B + Nx_N = b, x_B \geq 0, x_N \geq 0 \tag{8.30}$$
$$\therefore x_B = B^{-1}b - B^{-1}Nx_N \tag{8.31}$$
$$z = c_B B^{-1} b - c_B B^{-1} N x_N + c_N x_N \tag{8.32}$$

for current solution $\hat{x} = \begin{bmatrix} \hat{x_B} \\ 0 \end{bmatrix}$, $\hat{z} = c_B B^{-1} b$, then

$$z - \hat{z} = \begin{bmatrix} 0 & c_N - c_B B^{-1} N \end{bmatrix} \begin{bmatrix} x_B \\ x_N \end{bmatrix} \tag{8.33}$$

The $c_N - c_B B^{-1} N$ is the reduced cost, for a minimized problem, if $c_N - c_B B^{-1} N > 0$ means $z - \hat{z} \geq 0$, it reaches the optimality because we cannot find a solution less than $\hat{z}$.

8.6.3 Find Direction

Suppose we choose x_k as a candidate to pivot into Basis

$$x = \begin{bmatrix} B^{-1}b - B^{-1}a_k x_k \\ 0 + e_k x_k \end{bmatrix} = \begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix} + \begin{bmatrix} -B^{-1}a_k \\ e_k \end{bmatrix} x_k \tag{8.34}$$

In this form, we can see: x is the result after tth iteration, $\begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix}$ is the result after $(t-1)$th iteration. $\begin{bmatrix} -B^{-1}a_k \\ e_k \end{bmatrix}$ is the iteration direction, x_k is the step length.
The only requirement of x_k is $r_k < 0$ where $r_k = c_k - z_k$ is reduce cost, which is the kth entry of $c_N - c_B B^{-1} N$.
Generally speaking, we usually take $r_k = \min\{c_j - z_j\}$ (which in fact can not guarantee the efficient of the algorithm.)

8.6.4 Find the Step Length

We need to guarantee the non-negativity, so for each iteration, we need to make sure $x \geq 0$. Which means

$$B^{-1}b - B^{-1}a_k x_k \geq 0 \tag{8.35}$$

Denote $B^{-1}b$ as $\bar{b}$, denote $B^{-1}a_k$ as y_k
If $y_k \leq 0$, we can have x_k as large as infinite, which means unboundedness.
If $y_k > 0$ now we can use the minimum ratio to guarantee non-negativity.
Remember hit the bound, basic variable leave the basis and become non-basic variable.

Chapter 9

The Simplex Method - Improved

9.1 Revised Simplex Method

9.1.1 Key to Revised Simplex Method

The procedure of Simplex Method is (almost) exactly the same as original simplex method. However, notice that we don't need to use N so for the revised simplex method, we don't calculate any matrix related to N
The original matrix:

z	x_B	x_N	RHS
1	0	$c_B B^{-1} N - c_N$	$c_B B^{-1} b$
0	I	$B^{-1} N$	$B^{-1} b$

(9.1)

The revised matrix:

Basic Inverse	RHS
$w = c_B B^{-1}$	$c_B \bar{b} = c_B B^{-1} b$
B^{-1}	$\bar{b} = B^{-1} b$

(9.2)

For each pivot iteration, calculate $z_j - c_j = wa_j - c_j = c_B B^{-1} a_j - c_j, \forall j \in N$, pivot rules are the same as simplex method, each time find a variable x_k to enter basis

B^{-1}	RHS
w	$c_B \bar{b}$
B^{-1}	$\bar{b}$

x_k
$z_k - c_k$
y_k

(9.3)

Do the minimum ratio rule to find the variable x_r to leave the basis

B^{-1}	RHS
w	$c_B \bar{b}$
B^{-1}	$\bar{b_1}$
	$\bar{b_2}$
	...
	$\bar{b_r}$
	...
	$\bar{b_m}$

x_k
$z_k - c_k$
y_{1k}
y_{2k}
...
y_{rk}(pivot at here)
...
y_{mk}

(9.4)

9.1.2 Comparison between Simplex and Revised Simplex

Advantage of Revised Simplex

- Save storage memory
- Don't need to calculate N (including $B^{-1}N$ and $c_B B^{-1} N$)
- More accurate because round up errors will not be accumulated

Disadvantage of Revised Simplex

- Need to calculate wa_j for all $j \in N$ (in fact don't need to calculated it for the variable just left the basis)

Computation Complexity

Method	Type	Operations
Simplex	$\times$	$(m+1)(n-m+1)$
	$+$	$m(n-m+1)$
Revised Simplex	$\times$	$(m+1)^2+m(n-m)$
	$+$	$m(m+1)+m(n-m)$

(9.5)

When to use?

- When $m >> n$, do revise simplex method on the dual problem
- When $m \simeq n$, revise simplex method is not as good as simplex method
- When $m << n$ perfect for revise simplex method.

9.1.3 Decomposition of B inverse

Let $B = \{a_{B_1}, a_{B_2}, ..., a_{B_r}, ..., a_{B_m}\}$ and B^{-1} is known. If a_{B_r} is replaced by a_{B_k}, then B becomes $\bar{B}$. Which means a_{B_r} enters the basis and a_{B_k} leaves the basis.
Then $\bar{B}^{-1}$ can be represent by B^{-1}. Noting that $a_k = By_k$ and $a_{B_i} = Be_i$, then

$$\bar{B} = (a_{B_1}, a_{B_2}, ..., a_{B_{r-1}}, a_k, a_{B_{r+1}}, a_m) \tag{9.6}$$
$$= (Be_1, Be_2, ..., Be_{r-1}, By_k, Be_{r+1}, ..., Be_m) \tag{9.7}$$
$$= BT \tag{9.8}$$

where T is

$$T = \begin{bmatrix} 1 & 0 & ... & 0 & y_{1k} & 0 & ... & 0 \\ 0 & 1 & ... & 0 & y_{2k} & 0 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 1 & y_{r-1,k} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & y_{rk} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & y_{r+1,k} & 1 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 0 & y_{mk} & 0 & ... & 1 \end{bmatrix} \tag{9.9}$$

and

$$E = T^{-1} = \begin{bmatrix} 1 & 0 & ... & 0 & \frac{-y_{1k}}{y_{rk}} & 0 & ... & 0 \\ 0 & 1 & ... & 0 & \frac{-y_{2k}}{y_{rk}} & 0 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 1 & \frac{-y_{r-1,k}}{y_{rk}} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & \frac{1}{y_{rk}} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & \frac{-y_{r+1,k}}{y_{rk}} & 1 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 0 & \frac{-y_{mk}}{y_{rk}} & 0 & ... & 1 \end{bmatrix} \tag{9.10}$$

For each iteration, i.e. one variable enters the basis and one leaves the basis, $\bar{B}^{-1} = T^{-1}B^{-1} = EB^{-1}$. Given that the first iteration starts from slack variables, the first basis B_1 is I, then we have

$$B_t^{-1} = E_{t-1}E_{t-2}\cdots E_2E_1I \tag{9.11}$$

Using E in calculation can simplify the product of matrix where

$$cE = c_1, c_2, ..., c_m \begin{bmatrix} 1 & 0 & ... & g_1 & ... & 0 \\ 0 & 1 & ... & g_2 & ... & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & ... & g_m & ... & 1 \end{bmatrix} \tag{9.12}$$

$$= (c_1, c_2, ..., c_{r-1}, cg, c_{r+1}, ..., c_m) \tag{9.13}$$

and

$$Ea = \begin{bmatrix} 1 & 0 & ... & g_1 & ... & 0 \\ 0 & 1 & ... & g_2 & ... & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & ... & g_m & ... & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \tag{9.14}$$

$$= \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{r-1} \\ 0 \\ a_{r+1} \\ \vdots \\ a_m \end{bmatrix} + a_r \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_{r-1} \\ g_r \\ g_{r+1} \\ \vdots \\ g_m \end{bmatrix} \tag{9.15}$$

$$= \bar{a} + a_r g \tag{9.16}$$

Then we can calculate w, y_k and $\bar{b}$

$$w = c_B B^{-1} = c_B E_{t-1} E_{t-2} ... E_2 E_1 \tag{9.17}$$

$$y_k = B^{-1} a_k = E_{t-1} E_{t-2} ... E_2 E_1 a_k \tag{9.18}$$

$$\bar{b} = B_{t+1}^{-1} b = E_t E_{t-1} E_{t-2} ... E_2 E_1 b \tag{9.19}$$

9.2 Dual Simplex Method

Maintain dual feasibility, i.e. primal optimality, and complementary slackness and work towards primal feasibility.
Tip: The RHS become new $z_j - c_j$, the old $z_j - c_j$ become new RHS. We are actually solving the dual problem.

9.3 Simplex with Equations

9.4 Simplex with Bounded Variables

9.4.1 Bounded Variable Formulation

$$\min \quad cx \tag{9.20}$$

$$\text{s.t.} \quad Ax = b \tag{9.21}$$

$$l \leq x \leq b \tag{9.22}$$

Reason why we don't the following formulation

$$\begin{aligned} \min \quad & cx && (9.23)\\ \text{s.t.} \quad & Ax = b && (9.24)\\ & x - Ix_l = l && (9.25)\\ & x + Ix_u = u && (9.26)\\ & x \geq 0 && (9.27)\\ & x_l \geq 0 && (9.28)\\ & x_u \geq 0 && (9.29) \end{aligned}$$

is that this formulation increase the number of variable from n to $3n$, and the number of constraint from m to $m + 2n$, the size in increase significantly.

9.4.2 Basic Feasible Solution

Consider the system $Ax = b$ and $l \leq x \leq b$, where A is a $m \times n$ matrix of rank m, the solution $\bar{x}$ is a **basic feasible solution** if A can be partition into $[B, N_l, N_u]$ where the solution x can be partition into $x = (x_B, x_{N_l}, x_{N_u})$, in which $\bar{x}_{N_l} = l_{N_l}$ and $\bar{x}_{N_u} = u_{N_u}$, therefore

$$\bar{x}_B = B^{-1}b - B^{-1}N_l x_{N_l} - B^{-1}N_u x_{N_u} \tag{9.30}$$

Furthermore, similar to definition of nonnegative variables, if $l_B \leq x_B \leq u_B$, x_B is a basic feasible solution, if $l_B < x_B < u_B$, x_B is a non-degenerate basic feasible solution.

9.4.3 Improving Basic Feasible Solution

The basic variables and the objective function can be derived as following:

$$\begin{aligned} x_B &= B^{-1}b - B^{-1}N_l x_{N_l} - B^{-1}N_u x_{N_u} && (9.31)\\ z &= c_B x_B + c_{N_l} x_{N_l} + c_{N_u} x_{N_u} && (9.32)\\ &= c_B(B^{-1}b - B^{-1}N_l x_{N_l} - B^{-1}N_u x_{N_u}) && (9.33)\\ &\quad + c_B x_B + c_{N_l} x_{N_l} + c_{N_u} x_{N_u} && (9.34)\\ &= c_B B^{-1}b + (c_{N_l} - c_B B^{-1}N_l)x_{N_l} && (9.35)\\ &\quad + (c_{N_u} - c_B B^{-1}N_u)x_{N_u} && (9.36)\\ &= c_B B^{-1}b - \sum_{j \in J_1}(z_j - c_j)x_j - \sum_{j \in J_2}(z_j - c_j)x_j && (9.37) \end{aligned}$$

J_1 is the set of variables at lower bound, J_2 is the set of the variables at upper bound.
Notice that the right-hand-side no longer provide $c_B B^{-1}b$ and $B^{-1}b$. For the variable entering the basis, find the variable with

$$\max\{\max_{j \in J_1}\{z_j - c_j\}, \max_{j \in J_2}\{c_j - z_j\}\} \tag{9.38}$$

to enter the basis
Tip: "Most violated rule"
The minimum ratio rule is revised for bounded simplex

$$\Delta = \min\{\gamma_1, \gamma_2, u_k - l_k\} \tag{9.39}$$

$$\gamma_1 = \begin{cases} \min_{r \in J_1}\{\frac{\bar{b}_r - l_{B_r}}{y_{rk}} : y_{rk} > 0\} \\ \min_{r \in J_2}\{\frac{\bar{b}_r - l_{B_r}}{-y_{rk}} : y_{rk} < 0\} \\ \infty \end{cases} \tag{9.40}$$

$$\gamma_2 = \begin{cases} \min_{r \in J_1}\{\frac{u_{B_r} - \bar{b}_r}{-y_{rk}} : y_{rk} < 0\} \\ \min_{r \in J_2}\{\frac{u_{B_r} - \bar{b}_r}{y_{rk}} : y_{rk} > 0\} \\ \infty \end{cases} \tag{9.41}$$

Tip:
Use $l \leq x + \Delta \leq u$ to test the range of δ, if it hits lower bound, it is called γ_1, if it hits upper bound, it is called γ_2.

9.5 Simplex with Unrestricted Variables

Chapter 10

Duality and Sensitivity

10.1 Duality

10.1.1 Dual Formulation

For any prime problem

$$\begin{aligned} \min \quad & cx && (10.1)\\ \text{s.t.} \quad & Ax \geq b && (10.2)\\ & x \geq 0 && (10.3) \end{aligned}$$

we can have a dual problem

$$\begin{aligned} \max \quad & wb && (10.4)\\ \text{s.t.} \quad & wA \leq c && (10.5)\\ & w \geq 0 && (10.6) \end{aligned}$$

10.1.2 Mixed Forms of Duality

For the following prime problem

$$\begin{aligned} \text{P(or D)} \quad \min \quad & c_1x_1 + c_2x_2 + c_3x_3 && (10.7)\\ \text{s.t.} \quad & A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \geq b_1 && (10.8)\\ & A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \leq b_2 && (10.9)\\ & A_{31}x_1 + A_{32}x_2 + A_{33}x_3 = b_3 && (10.10)\\ & x_1 \geq 0 && (10.11)\\ & x_2 \leq 0 && (10.12)\\ & x_3 \quad \text{unrestricted} && (10.13) \end{aligned}$$

The dual of the problem

$$\begin{aligned} \text{D(or P)} \quad \max \quad & w_1b_1 + w_2b_2 + w_3b3 && (10.14)\\ \text{s.t.} \quad & w_1A_{11} + w_2A_{21} + w_3A_{31} \leq c_1 && (10.15)\\ & w_1A_{12} + w_2A_{22} + w_3A_{32} \geq c_2 && (10.16)\\ & w_1A_{13} + w_2A_{23} + w_3A_{33} = c_3 && (10.17)\\ & w_1 \geq 0 && (10.18)\\ & w_2 \leq 0 && (10.19)\\ & w_3 \quad \text{unrestricted} && (10.20) \end{aligned}$$

In sum, the relation between primal and dual problems are listed as following

	Minimization		Maximization	
Var	≥ 0	$\longleftrightarrow$	≤ 0	Cons
	≤ 0	$\longleftrightarrow$	≥ 0	
	Unrestricted	$\longleftrightarrow$	$=$	
Cons	≥ 0	$\longleftrightarrow$	≥ 0	Var
	≤ 0	$\longleftrightarrow$	≤ 0	
	$=$	$\longleftrightarrow$	Unrestricted	

10.1.3 Dual of the Dual is the Primal

For a primal problem (P)

$$\begin{aligned} (P) \quad \min \quad & cx && (10.21)\\ \text{s.t.} \quad & Ax \geq b && (10.22)\\ & x \geq 0 && (10.23) \end{aligned}$$

The dual problem (D) is

$$\begin{aligned} (D) \quad \max \quad & wb && (10.24)\\ \text{s.t.} \quad & wA \leq c && (10.25)\\ & w \geq 0 && (10.26) \end{aligned}$$

Rewrite the dual

$$\begin{aligned} \min \quad & -b^\top w^\top && (10.27)\\ \text{s.t.} \quad & -A^\top w^\top \geq -c^\top && (10.28)\\ & w^\top \geq 0 && (10.29) \end{aligned}$$

Find the dual of this problem

$$\begin{aligned} \max \quad & x^\top(-c^\top) && (10.30)\\ \text{s.t.} \quad & x^\top(-A^\top) \leq (-b^\top) && (10.31)\\ & x^\top \geq 0 && (10.32)\\ & && (10.33) \end{aligned}$$

Rewrite the dual of the dual

$$\begin{aligned} (P) \quad \min \quad & cx && (10.34)\\ \text{s.t.} \quad & Ax \geq b && (10.35)\\ & x \geq 0 && (10.36) \end{aligned}$$

10.1.4 Primal-Dual Relationships

Weak Duality Property

Let x_0 be any feasible solution of a primal minimization problem,

$$Ax_0 \geq b, \quad x_0 \geq 0 \tag{10.37}$$

Let x_0 be any feasible solution of a dual maximization problem,

$$w_0 A \leq c, \quad w_0 \geq 0 \tag{10.38}$$

Therefore, we have

$$cx_0 \geq w_0 Ax_0 \geq w_0 b \tag{10.39}$$

which is called the weak duality property. This property is for any feasible solution in the primal and dual problem. Therefore, any feasible solution in the maximization problem gives the lower bound of its dual problem, which is a minimization problem, vice versa. We use this to give the bounds in using linear relaxation to solve IP problem.

Fundamental Theorem of Duality

With regard to the primal and dual LP problems, one and only one of the following can be true.

- Both primal and dual has optimal solution x^* and w^*, where $cx^* = w^*b$
- One problem has an unbounded optimal objective value, the other problem must be infeasible
- Both problems are infeasible.

Strong Duality Property

From KKT condition, we know that in order to make x^* the optimal solution, the following condition should be met.

- Primal Optimal: $Ax^* \geq b$, $x^* \geq 0$
- Dual Optimal: $w^*A \leq c$, $w^* \geq 0$
- Complementary Slackness

$$\begin{cases} w^*(Ax^* - b) = 0 \\ (c - w^*A)x^* = 0 \end{cases} \tag{10.40}$$

The first condition means the primal has an optimal solution, the second condition means the dual has an optimal solution. The third condition means $cx^* = w^*b$, which is also called **strong duality property**

Notice: w in the dual problem is the same as the $w = c_B B^{-1}$ in primal problem.

Complementary Slackness Theorem

Let $\boldsymbol{x^*}$ and $\boldsymbol{w^*}$ be any feasible solutions, they are optimal iff

$$(c_j - \boldsymbol{w^* a_j})x_j^* = 0, \quad j = 1, ..., n \tag{10.41}$$

$$w_i^*(\boldsymbol{a^i x^*} - b_i) = 0, \quad i = 1, ..., m \tag{10.42}$$

In particular

$$x_j^* > 0 \Rightarrow \boldsymbol{w^* a_j} = c_j \tag{10.43}$$

$$\boldsymbol{w^* a_j} < c_j \Rightarrow x_j^* = 0 \tag{10.44}$$

$$w_i^* > 0 \Rightarrow \boldsymbol{a^i x^*} = b_i \tag{10.45}$$

$$\boldsymbol{a^i x^*} > b_i \Rightarrow w_i^* = 0 \tag{10.46}$$

It means, if in optimal solution a variable is positive (has to be in the basis), the correspond constraint in the other problem is tight. If the constraint in one problem is not tight, the correspond variable in the other problem is zero. In the dual problem, we solved some w which is positive, we can know that the correspond constraint in primal is tight, furthermore we can solve the basic variables from those tight constraints, which becomes equality and we can solve it using Gaussian-Elimination.

10.1.5 Shadow Price

Shadow Price under Non-degeneracy

Let B be an optimal basis for primal problem and the optimal solution x^* is non-degenerated.

$$z = c_B B^{-1} b - \sum_{j \in N} (z_j - c_j)x_j = w^* b - \sum_{j \in N} (z_j - c_j)x_j \tag{10.47}$$

therefore

$$\frac{\partial z^*}{\partial b_i} = c_B B_i^{-1} = w_i^* \tag{10.48}$$

w^* is the shadow prices for the right-hand-side vectors. We can also regard it as the **incremental cost** of producing one more unit of the ith product. Or w^* is the **fair price** we would pay to have an extra unit of the ith product.

Shadow Price under Degeneracy

For shadow price under degeneracy, the w^* may not be the true shadow price, for it may not be the right basis. In this case, the partial differentiation may not be valid, for component b_i, if $x_i = 0$ and x_i is a basic variable, we can't find the differentiation.

10.2 Sensitivity

- Change in the Cost Vector
  - Nonbasic Variable: c_B is not affected, $z_j = c_B B^{-1} a_j$ is not changed, say nonbasic variable cost coefficient c_k changed into c'_k. For now $z_k - c_k \leq 0$, if $z_k - c'_k$ is positive, x_k must into the basis, the optimal value changed. Otherwise stays at the same.
  - Basic Variable: If c_{B_t} is replaced by c'_{B_t}, then $z'_j - c_j$ is

$$z'_j - c_j = c'_B B^{-1} a_j - c_j = (z_j - c_j) - (c'_{B_t} - c_{B_t}) B^{-1} a_{B_t} \tag{10.49}$$

    for $j = k$, it is a basic variable, therefore original $z_k - c_k = 0$, $B^{-1}a_k = 1$. Hence $z'_k - c_k = c'_k - c_k \Rightarrow z'_k - c'_k = 0$. The basis stays the same. The optimal solution updated as $c'_B B^{-1} b = c_B B^{-1} b + (c'_{B_t} - c_{B_t}) B^{-1} b_{B_t}$.

- Change in the Right-Hand-Side: If b is replaced by b', then $B^{-1}b$ is replaced by $B^{-1}b'$. If $B^{-1}b' \geq 0$, the basis remains optimal. Otherwise, we perform dual simplex method to continue.

- Change in the Matrix
  - Changes in Activity(Variable) Vectors for Nonbasic Columns: If a nonbasic column a_j is replaced by a'_j, then $z_j = c_B B^{-1} a_j$ is replaced by $z'_j = c_B B^{-1} a'_j$, if new $z'_j - c_j \leq 0$, the basis stays optimal basis, the optimal value is the same because c_B stays the same.
  - Changes in Activity(Variable) Vectors for Basic Columns: If a basic columns changed, it means B and B^{-1} changed, and every column changed. We can do this in two steps:
    - step I: add a new column with a'_j
    - step II: remove the original column a_j

    If in step I the new variable can enter basis, i.e. $z'_j - c_j \leq 0$, let it enter the basis and eliminate the original column directly (because at this time the original column leave the basis the nonbasic variable is 0); otherwise, if the new column can not form a new basis, treat x_j, the original variable as an artificial variable.
  - Add a New Activity(Variable): Suppose we add a new variable x_{n+1} and c_{n+1} and a_{n+1} respectively. Calculate $z_{n+1} - c_{n+1}$ to determine if the new variable enters the basis, if not, remains the same optimal solution, otherwise, continue on to find a new optimal solution.
  - Add a New Constraint: This is the basic of Branch-and-Cut/Bound, also, we can perform dual simplex method after we add a new constraint(cut).

Chapter 11

Ellipsoid Methods

Chapter 12

Karmarkar's Algorithm

Part III

Integer Programming

Chapter 13

Branch and Bound

13.1 LP Relaxation Based Branch and Bound

Chapter 14

Cutting Plane Methods

14.1 Gomory Cuts

14.2 Kianfar's methods

14.3 Lift-and-project

Chapter 15

Decompositions

15.1 Dantzig-Wolfe Decomposition

15.2 Benders Decomposition

15.3 Lagrangian Relaxation

Chapter 16

Lagrangian Relaxation

16.1 Basic Constructions

Consider an integer program

$$\begin{aligned} z_{IP} = \max \quad & \mathbf{cx} \\ \text{s.t.} \quad & \mathbf{Ax} \leq \mathbf{b} \\ & \mathbf{x} \in \mathbb{Z}_+^n \end{aligned}$$

We can partial $\mathbf{A}$ and $\mathbf{b}$ into two sets of constraints, where $\mathbf{A} = \begin{bmatrix} \mathbf{A}^1 \\ \mathbf{A}^2 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} \mathbf{b}^1 \\ \mathbf{b}^2 \end{bmatrix}$.
So the IP model can be rewritten as

$$\begin{aligned} z_{IP} = \max \quad & \mathbf{cx} \\ \text{s.t.} \quad & \mathbf{A}^1\mathbf{x} \leq \mathbf{b}^1 \\ & \mathbf{A}^2\mathbf{x} \leq \mathbf{b}^2 \\ & \mathbf{x} \in \mathbb{Z}_+^n \end{aligned}$$

Usually, $\mathbf{A}^1\mathbf{x} \leq \mathbf{b}^1$ are the "bad" constraints and $\mathbf{A}^2\mathbf{x} \leq \mathbf{b}^2$ are the "nice" constraints.
We can derive an relaxation of the IP model by dropping $\mathbf{A}^1\mathbf{x} \leq \mathbf{b}^1$, and leave with the constraints that are easy, or relatively easier, to solve. This idea can be embedded into a more general framework called Lagrangian relaxation.
Define and rewrite the IP model as

$$\begin{aligned} \text{IP(Q)} \quad z_{IP} = \max \quad & \mathbf{cx} \\ \text{s.t.} \quad & \mathbf{A}^1\mathbf{x} \leq \mathbf{b}^1 \\ & \mathbf{x} \in Q, \quad Q = \{\mathbf{A}^2\mathbf{x} \leq \mathbf{b}^2 | \mathbf{Z}_+^n\} \end{aligned}$$

Notice that we have assumed the

Chapter 17

Total Unimodularity

Chapter 18

Semidefinite Programming

Chapter 19

Examples: Classical Problems

19.1 Packing and Matching

19.2 Knapsack Problem

Part IV

Graph and Network Theory

Chapter 20

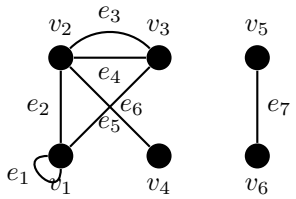
Basic Structures

20.1 Graph

Definition 20.1.1 (Graph)**.** A **graph** G consists of a finite set $V(G)$ on vertices, a finite set $E(G)$ on edges and an **incident relation** than associates with any edge $e \in E(G)$ an unordered pair of vertices not necessarily distinct called **ends**.

Example. The following graph can be represented as

$$V = V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6\} \tag{20.1}$$
$$E = E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\} \tag{20.2}$$
$$e_1 = v_1v_2, \quad e_2 = v_2v_4, \quad \dots \tag{20.3}$$



Definition 20.1.2 (Loop, Parallel, Simple Graph)**.** An edge with identical ends is called a **loop**, Two edges having the same ends are said to be **parallel**, A graph without loops or parallel edges is called **simple graph**

Definition 20.1.3 (Adjacent)**.** Two edges of a graph are **adjacent** if they have a common end, two vertices are **adjacent** if they are jointed by an edge.

Saving a graph in computer program can be implemented in the following ways:

- Adjacency matrix: $m \times n$ matrix, for $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
- Linked list: For every vertex v, there is a linked list containing all neighbors of v.

Assuming we are dealing with undirected graphs, n is the number of vertices, m is the number of edges, $n - 1 \leq m \leq n(n-1)/2$, d_v is the number of neighbors of v then

	Matrix	Linked lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u, v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbors of v	$O(n)$	$O(d_v)$

20.2 Subgraph

Definition 20.2.1 (Subgraph)**.** Given two graphs G and H, H is a **subgraph** of G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ and an edge has the same ends in H as it does in G. Furthermore, if $E(H) \neq E(G)$ then H is a proper subgraph.

Definition 20.2.2 (Spanning)**.** A subgraph H on G is **spanning** if $V(H) = V(G)$

Definition 20.2.3 (Vertex-induced, Edge-induced)**.** For a subset $V' \subseteq V(G)$ we define an **vertex-induced** subgraph $G[V']$ to be the subgraph with vertices V' and those edges of G having both ends in V'. The **edge-induced** subgraph $G[E']$ has edges E' and those vertices of G that are ends to edges in E'.

Notice: If we combine node-induced or edge-induced subgraphs $G(V')$ and $G(V - V')$, we cannot always get the entire graph.

20.3 Degree

Definition 20.3.1 (Degree)**.** Let $v \in V(G)$, then the **degree** of $v \in V(G)$ denote by $d_G(v)$ is defines to be the number of edges incident of v. Loops counted twice.

Theorem 20.1. *For any graph $G = (V, E)$*

$$\sum_{v \in V} d(v) = 2|E| \tag{20.4}$$

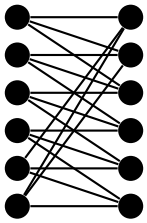
Proof. $\forall$ edge $e = uv$ with $u \neq v$, e is counted once for u and once for v, a total of two altogether. If $e = uu$, a loop, then it is counted twice for u. □

Problem 20.1. Explain clearly, what is the largest possible number of vertices in a graph with 19 edges and all vertices of degree at least 3. Explain why this is the maximum value.

Solution. The maximum number is 12.

Proof. First we prove 12 vertices is possible, then we prove 13 vertices is not possible

- The following graph contains 12 vertices and 18 edges, each vertex has a degree of 3.



- For 13 vertices and each vertex has a degree of at least 3 will require at least

$$2|E| = \sum_{v \in V} d(v) \geq 3 \times |N| = 3 \times 13 \Rightarrow |E| \geq 19.5 > 19 \tag{20.5}$$

  edges, i.e., 13 vertices is not possible.

□

Corollary 20.1.1. *Every graph has an even number of odd degree vertices.*

Proof.

$$V = V_E \cup V_O \Rightarrow \sum_{v \in V} d(v) = \sum_{v \in V_E} d(v) + \sum_{v \in V_O} d(v) = 2|E| \tag{20.6}$$

□

20.4 Special Graphs

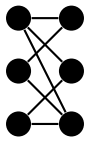
Definition 20.4.1 (Complete Graph)**.** A **complete** graph $K_n (n \geq 1)$ is a simple graph with n vertices and with exactly one edge between each pair of distinct vertices.

Definition 20.4.2 (Cycle)**.** A **cycle** graph $C_n (n \geq 3)$ consists of n vertices $v_1, ...v_n$ and n edges $\{v_1, v_2\}, \{v_2, v_3\}, ...\{v_{n-1}, v_n\}$

Definition 20.4.3 (Wheel)**.** A **wheel** graph $W_n (n \geq 3)$ is a simple graph obtains by adding one vertex to the cycle graph C_n, and connecting this new vertex to all vertices of C_n

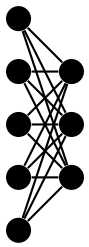
Definition 20.4.4 (Bipartite Graph)**.** A simple graph is said to be **bipartite** if the vertex set can be expressed as the union of two disjoint non-empty subsets V_1 and V_2 such that every edges has one end in V_1 and another end in V_2

Example. Here is an example for bipartite graph



Definition 20.4.5 (Complete Bipartite Graph)**.** The **complete bipartite graph** K_{mn} is the bipartite graph V_1 containing m vertices and V_2 containing n vertices such that each vertiex in V_1 is adjacent to every vertex in V_2

Example. Here is an example for K_{53}

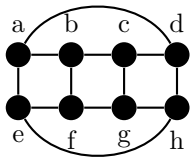


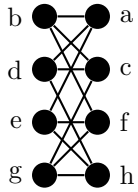
Theorem 20.2. *A graph G is bipartite iff every cycle is even.*

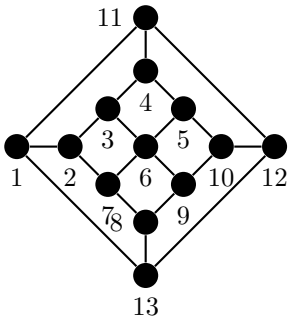
Proof. ($\Rightarrow$) If the graph G is bipartite, by definition, the vertices of graph can be partition into two groups, that within the group there is no connection between vertices. Therefore, for each cycle, the odd index of vertices and even index of vertices has to be choose alternatively from each groups. Therefore the cycle has to be even.
($\Leftarrow$) Prove by contradiction. A graph can be connected or not connected.
If G is connected and has at least two vertices, for an arbitrary vertex $v \in V(G)$, we can calculate the minimum number of edges between the other vertices v' and v (i.e., length, denoted by $l(v', v)$), for all the vertices that has odd length to v, assign them to set V_1, for the rest of vertices (and v), assign to set V_2. Assume that G is not bipartite, which means there are at least one edge between distinct vertices in set V_1 or set V_2, without lost of generality, assume that edge is uw, $u, w \in V_1$. For all vertices in V_1 there is an odd length of path between the vertex and v, therefore, there exists an odd $l(u, v)$, and an odd $l(w - v)$. The length of cycle $l(u, w, v) = 1 + l(u, v) + l(w, v)$, which is an odd number, it contradict with the prerequisite that all cycles are even, which means the assumption that G is not bipartite is incorrect, G should be bipartite.
If G is not connected. Then G can be partition into a set of disjointed subgraphs which are connected with at least two vertices or contains only one vertex. For the component that has more that one vertices, we already proved that it has to be bipartite. For the subgraph $G_i \subset G, i = 1, 2, ..., n$, the vertices can be partition into $V_{i1} \in V(G_i)$ and $V_{i2} \in V(G_i)$, where $V_{i1} \cap V_{i2} = \emptyset$, the union of those subgraphs are bipartite too because $V_1 = \cup_{i=1}^{n} V_{i1} \in V(G)$ and $V_2 = \cup_{i=1}^{n} V_{i2} \in V(G)$ satisfied the condition of bipartite. For the subgraph that has one one vertices, those vertices can be assigned into either V_1 or V_2. □

Example. The following graph is bipartite, it only contains even cycles.
We can rearrange the graph to be more clear as following
The vertices of graph G can be partition into two sets, $\{a, c, f, h\}$ and $\{b, d, e, g\}$

Example. The following graph is not bipartite
The cycle $c = v_1 v_1 1 v_4 v_3 v_2$ have odd number of vertices.







20.5 Directed Graph

Definition 20.5.1. A graph $G = (V, E)$ is called directed if for each edge $e \in E$, there is a **head** $h(e) \in V$ and a **tail** $t(e) \in V$ and the edges of e are precisely $h(e)$ and $t(e)$, denoted $e = (t(e), h(e))$

Definition 20.5.2. We call directed graphs **digraphs**, we call edges in a digraph are called **arcs**, and vertices in a digraph **nodes**

Definition 20.5.3. Similar as in the undirected case we have walks, traces, paths and cycles in digraphs.

Definition 20.5.4. A vertex $v \in V$ is **reachable** from a vertex $u \in V$ if there is a (u, v)-dipath. If at the same time u is reachable from v, they are **strongly connected**

Definition 20.5.5. A digraph is strongly connected if every pair of vertices are strongly connected.

Definition 20.5.6. A digraph is **strict** if it has no loops and whenever e and f are parallel, $h(e) = t(f)$

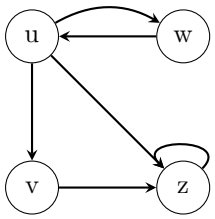
Definition 20.5.7. For a vertex v in a digraph D, the **indegree** of v in D, denoted by $d^+(v)$ is the number of arcs of D having head V. The **outdegree** of v is denoted by $d^-(v)$ is the number of arcs of D having tail v.

Let $D = (V, A)$ be a digraph with no loops a vertex-arc **incident matrix** for D is a $(0, 1, -1)$ matrix N with rows indexed by $V = \{v_1, ..., v_n\}$ and column indexed by $A = \{e_1, ..., e_m\}$ and where entry (i, j) in the matrix n_{ij} is

$$n_{ij} = \begin{cases} 1, & \text{if } v_i = h(e_j) \\ -1, & \text{if } v_i = t(e_j) \\ 0, & \text{otherwise} \end{cases} \tag{20.7}$$

$$\begin{bmatrix} -1 & 0 & -1 & -1 & 1 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{20.8}$$

20.6 Sperner's Lemma


u
w
v
z

Chapter 21

Paths, Trees, and Cycles

21.1 Walk

Definition 21.1.1 (walk)**.** A **walk** in a graph G is a finite sequence $w = v_0e_1v_1e_2...e_kv_k$, where for each $e_i = v_{i-1}v_i$ the edge and its ends exists in G. We say that walk v_0 to v_k on (v_0, v_k)-walk.

Example.

$$w = v_2e_4v_3e_4v_2e_5v_3 \tag{21.1}$$

is a walk, or (v_2, v_3)-walk

Definition 21.1.2 (origin, terminal, internal, length)**.** For (v_0, v_k)-walk, The vertices v_0 and v_k are called the **origin** and the **terminal** of the walk w, $v_1..v_{k-1}$ are called **internal** vertices. The integer k is the **length** of the walk. Length of w equals to the number of edges.

We can create a reverse walk w^{-1} by reversing w.

$$w^{-1} = v_ke_kv_{k-1}e_{k-1}...e_2v_1 \tag{21.2}$$

(The reverse walk is guaranteed to exist because it is an undirected graph)
Given two walks w and w' we can create a third walk denoted by ww' by concating w and w'. The new walk's origin is the same as terminal.

21.2 Path and Cycle

Definition 21.2.1 (trail)**.** A **trail** is a walk with no repeating edges. e.g., $v_3e_4v_2e_5v_3$

Definition 21.2.2 (path)**.** A **path** is a trail with no repeating vertices. e.g., $v_3e_4v_2$

Notice: Paths $\subseteq$ Trails $\subseteq$ Walks

Definition 21.2.3 (closed, cycle)**.** A path is **closed** if it has positive length and its origin and terminal are the same. e.g., $v_1e_2v_2e_4v_3e_3v_1$. A closed trail where origin and internal vertices are distinct is called a **cycle** (The only time a vertex is repeated is the origin and terminal)

Definition 21.2.4 (even/odd cycle)**.** A cycle is **even** if it has a even number of edges otherwise it is **odd**.

Problem 21.1. Prove that if C_1 and C_2 are cycles of a graph, then there exists cycles $K_1, K_2, ..., K_m$ such that $E(C_1)\Delta E(C_2) = E(K_1) \cup E(K_2) \cup ... \cup E(K_m)$ and $E(K_i) \cap E(K_j) = \emptyset, \forall i \neq j$. (For set X and Y, $X\Delta Y = (X - Y) \cup (Y - X)$, and is called the symmetric difference of X and Y)

Proof. Proof by constructing $K_1, K_2, ...K_m$. Denote

$$C_1 = v_{11}e_{11}v_{12}e_{12}v_{13}e_{13}...v_{1n}e_{1n}v_{11} \tag{21.3}$$

$$C_2 = v_{21}e_{21}v_{22}e_{22}v_{23}e_{23}...v_{2k}e_{2k}v_{21} \tag{21.4}$$

Assume both cycle start at the same vertice, $v_{11} = v_{12}$. (If there is no intersected vertex for C_1 and C_2, just simply set $K_1 = C_1$ and $K_2 = C_2$)
The following algorithm can give us all $K_j, j = 1, 2, ..., m$ by constructing $E(C_1)\Delta E(C_2)$. Also, the complexity is $O(mn)$, which makes the proof doable.

Now we prove that $K_i \cap K_j = \emptyset, \forall i \neq j$. For each K_j, it is defined by two (v_o, v_t)-paths in the algorithm. From

Algorithm 4 Find $K_1, K_2, ...K_m$ by constructing $E(C_1)\Delta E(C_2)$

Require: Graph G, cycle C_1 and C_2
Ensure: $K_1, K_2, ...K_m$
1: Initial, $K \leftarrow \emptyset$, $j = 1$
2: Set temporary storage units, $v_o \leftarrow v_{11}$, $v_t \leftarrow \emptyset$
3: **for** $i = 1, 2, ..., n$ **do**
4: **if** $e_{1i} \in C_2$ **then**
5: **if** $v_o \neq v_{1i}$ **then**
6: $v_t \leftarrow v_{1i}$
7: concate (v_o, v_t)-path $\subset C_1$ and (v_o, v_t)-path $\subset C_2$ to create a new K_j
8: Append K with K_j, $K \leftarrow K \cup K_j$
9: Reset temporary storage unit. $v_o \leftarrow v_{1(i+1)}$ (or v_{11} if $i = n$), $v_t \leftarrow \emptyset$
10: **else**
11: $v_o \leftarrow v_{1(i+1)}$ (or v_{11} if $i = n$)

the algorithm we know that all the edges in (v_o, v_t)-path in C_1 are not intersecting with C_2, because if the edge in C_1 is intersected with C_2, either we closed the cycle K_j before the edge, or we updated v_o after the edge (start a new K_j after that edge). By definition of cycle, all the (v_o, v_t)-path that are subset of C_1 are not intersecting with each other, as well as all the (v_o, v_t)-path that are subset of C_2. Therefore, $K_i \cap K_j = \emptyset, \forall i \neq j$. □

Definition 21.2.5 (connected vertices)**.** Two vertices u and v in a graph are said to be **connected** if there is a path between u and v.

Definition 21.2.6 (component)**.** Connectivity between vertices is an equivalence relation on $V(G)$, if $V_1, ...V_k$ are the corresponding equivalent classes then $G[V_1]...G[V_k]$ are **components** of G. If graph has only one component, then we say the graph is connected. A graph is connected iff every pair of vertices in G are connected, i.e., there exists a path between every pair of vertices.

Problem 21.2. If G is a simple graph with at least two vertices, prove that G has two vertices with the same degree.

Proof. A simple graph can only be connected or not connected.

- If G is connected, i.e., for all vertices, the degree is greater than 0. Also the graph is simple, for a graph with $|N|$ vertices, the degree of each vertex is less or equal to $|N| - 1$ (cannot have loop or parallel edge). For $|N|$ vertices, to make sure there is no two vertices that has same degree, it will need $|N|$ options for degrees, however, we only have $|N| - 1$ option. According to pigeon in holes principle, there has to be at least two vertices with the same degree.

- If G is not connected, i.e., the graph has more than one component. One of the following situation will happen:

  – For all components, each component contains only one vertex. Since we have at least two vertices, which means there are at least two component that has only one vertex. For those vertices, at least two vertices has the same degree as 0.

  – At least one component has more than one vertices. In this situation, we can find a component that has more than one vertices as a subgraph G' of the graph G. That G' is a connected simple graph by definition. We have already proved that a connected simple graph has two vertices with the same degree, which means G has two vertices with the same degree.

□

21.3 Tree and forest

Definition 21.3.1 (acyclic graph)**.** A graph is called **acyclic** if it has no cycles

Definition 21.3.2 (forest, tree)**.** A acyclic graph is called a **forest**. A connected forest is called a **tree**.

Theorem 21.1. *Prove that T is a tree, if T has exactly one more vertex than it has edges.*

Proof. 1. First we prove for any tree T that has at least two vertices, there has to be at least one leaf, i.e., now we prove that we can find u with degree of 1. Proof by constructing algorithm. (In fact we can prove that there are at least two leaves.)
The above algorithm is guaranteed to have an end because a tree is acyclic by definition

Algorithm 5 Find one leaf in a tree

Require: $d(u) = 1$
Ensure: A tree T has at least one vertex
1: Let u and v be any distinct vertex in a tree T
2: Let p be the path between u and v
3: **while** $d(u) \neq 1$ **do**
4: **if** $d(u) > 1$ **then**
5: Let $n(u)$ be the set of neighboring vertices of u
6: In $n(u)$, find a u' that the edge between u and u', denoted by e, $e \notin p$
7: $u \leftarrow u'$
8: $p \leftarrow p \cup e$

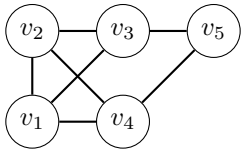
2. Then, if we remove one leaf in the tree, i.e., we remove an edge and a vertex, where that vertex only connects to the edge we removed. One of the following situations will happen:
   (a) Situation 1: The remaining of T is one vertex. In this case, T has two vertices an one edge. (Exactly one more vertex than it has edges)
   (b) Situation 2: The remaining of T is another tree T' (removal of edges will not change acyclic and connectivity), where $|V(T)| = |V(T')| + 1$ and $|E(T)| = |E(V'| + 1$. (one edge and one vertex has been removed)
3. Do the leaf removal process recursively to T' if Situation 2 happens until Situation 1 happens.

□

21.4 Spanning tree

Definition 21.4.1 (spanning tree)**.** A subgraph T of G is a **spanning tree** if it is spanning ($V(T) = V(G)$) and it is a tree.

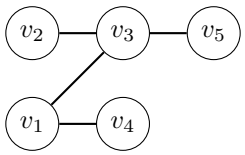
Example. In the following graph
This is a spanning tree



Problem 21.3. Prove that if T_1 and T_2 are spanning trees of G and $e \in E(T_1)$, then there exists a $f \in E(T_2)$, such that $T_1 - e + f$ and $T_2 + e - f$ are both spanning trees of G.

Proof. One of the following situation has to happen:

1. If for given $e \in E(T_1)$, $\exists f = e \in E(T_2)$, then $T_1 - e + f = T_1$, $T_2 + e - f = T_2$ are both spanning trees of G



2. If for given $e \in E(T_1)$, $e \notin E(T_2)$, the following will find an edge f that $T_1 - e + f$ and $T_2 + e - f$ are both spanning trees of G.
   (a) T_1 is a spanning tree, removal of $e \in E(T_1)$ will disconnect the spanning tree into two components (by definition of spanning tree), denoted by $G_1 \subset G$ and $G_2 \subset G$, by definition, $V(G_1)$ and $V(G_2)$ is a partition of $V(G)$.
   (b) Add e into T_2. We can proof that by adding an edge into a tree will create exactly one cycle, denoted by C, $e \in E(C)$.
   (c) For C, since it is a cycle and one end of e is in $V(G_1)$, the other end of e is in $V(G_2)$, there has to be at least two edges (can be more) that has one end in $V(G_1)$ and the other end in $V(G_2)$, denote the set of those edges as $E \subset E(C)$, one of those edges is $e \in E$
   (d) Choose any $f \in E$ and $f \neq e$, for that f, $T_1 - e + f$ and $T_2 + e - f$ are both spanning trees of G.
   (e) Prove that $T_1 - e + f$ is a spanning tree
      i. $T_1 - e + f$ have the same set of vertices as T_1, therefore it is spanning.
      ii. It is connected both within G_1 and G_2, for f, one end is in $V(G_1)$, the other end is in $V(G_2)$ therefore $T_1 - e + f$ is connected.
      iii. $T_1 - e + f$ have the same number of edges as T_1, which is $|T_1| - 1$, therefore $T_1 - e + f$ is a tree. (We have proven the connectivity in the previous step.)
      iv. $T_1 - e + f$ is spanning, connected, a tree, therefore it is a spanning tree.
   (f) Prove that $T_2 + e - f$ is a spanning tree
      i. $T_2 + e - f$ have the same set of vertices as T_2, therefore it is spanning.
      ii. T_2 is connected, adding an edge will not break connectivity, therefore $T_2 + e$ is connected, removing an edge in a cycle will not break connectivity, therefore $T_2 + e - f$ is connected.
      iii. $T_2 - e + f$ have the same number of edges as T_2, which is $|T_2| - 1$, therefore $T_2 + e - f$ is a tree. (We have proven the connectivity in the previous step.)
      iv. $T_2 - e + f$ is spanning, connected, a tree, therefore it is a spanning tree.

□

Theorem 21.2. *Every connected graph has a spanning tree.*

Proof. Prove by constructing algorithm: □

Algorithm 6 Find a spanning tree for connected graph (Prim's Algorithm in unweighted graph)

Require: a connected graph G and an enumeration $e_1, ...e_m$ of the edges of G
Ensure: a spanning tree T of G
1: Let T be the spanning subgraph of G with $V(T) = V(G)$ and $E(T) = \emptyset$
2: $i \leftarrow 1$
3: **while** $i \leq |E|$ **do**
4: **if** $T + e_i$ is acyclic **then**
5: $T \leftarrow T + e_i$
6: $i \leftarrow i + 1$

Notice: This algorithm can be improved, one idea is to make summation of edges in spanning subgraph less or equation to $|V| - 1$

For the complexity of spanning tree algorithm:

1. Space complexity, $2|E|$, which is $O(|E|)$
2. Time complexity
   (a) How to check for acyclic?
      i. At every stage T has certain components $V_1, ...V_t$, (every time we add an edge, the number of components minus 1)
      ii. So at the beginning $t = |V|$ with $|V_i| = 1 \forall i$ and at the end, $t = 1$.
   (b) Count the amount of work for the algorithm.
      i. Need to check for acyclic for each edge, which costs $O(|E|)$
      ii. Need to flip the pointer for each vertex, for each vertex, at most will be flipped $\log |V|$ times, altogether $|V| \log |V|$ times.
      iii. The time complexity is $O(|E| + |V| \log |V|)$
3. First we need to input the data, create an array such that the first and the second entries are the ends of e_1, third and fourth are the ends of e_2, and so on.
4. The amount of storage needs in $2|E|$, which is $O(|E|)$
5. The main work involved in the algorithm is for each edges e_i and the current T, to determine if $T + e_i$ creates a cycle.
6. suppose we keep each component V_i by keeping for each vertex a pointer from the vertex to the name of the component containing it. Thus if $\mu \in V_3$, there will be a pointer from μ to integer 3.
7. Then when edge $e_i = \mu v$ is encountered in Step 2, we see that $T + e_i$ contains a cycle if and only if μ and v point to same integer which means they are in the same component
8. If they are not in the same component, we want to add the edge which means then I have to update the pointers.

To prove algorithm we need to show the output is a spanning tree, which means three properties must hold:

- spanning (Step I)
- acyclic (We never add an edge that create a cycle)
- connected (Proof by contradiction)

So it is sufficient to show that the output will be connected.

Proof. (Proof by Contradiction) Suppose the output graph T of the algorithm is NOT connected. Let T_1 be a component of T, let $x \in T_1$ and $y \notin T_1$. But G is a connected graph (given from the beginning), so there must be a path in G that connects x and y. Let such a path in G be $p = xe_1v_1e_2, ..v_{k-1}e_ky$. Clearly, $p \notin T_1$. So there must be a first vertex in P that not in T_1. So $e_i \notin E(T)$, the only way this can happen when applying the algorithm is if $T + e_i$ creates a cycle C, i.e., $e_i \in C$, so $C - e_i$ is a path connecting v_{i-1} and v_i. So $c - e_i \in T$, so v_{i-1} is connected to $v_i \in T$. Contradiction. □

21.5 Cayley's Formula

21.6 Connectivity, DFS, BFS

21.6.1 Connectivity Problem

For connectivity problem, the input is a graph $G = (V, E)$, with linked list representation, and two vertices $s, t \in V$. The problem is to find whether there is a path connecting s to t in G
There are two commonly use methods to solve connectivity problem: depth-first search and breath-first search.

21.6.2 Depth-First Search (DFS)

The idea of DFS is enumerating children before siblings when search in the graph/tree. It is a recursive algorithm. First, start with s, then travel through the fist edge leading out of the current vertex, when reached a "visited" vertex, go back and travel through next edge. If tried all edges leading out of the current vertex, go back.
The algorithm is as following

Algorithm 7 Depth-First Search

```
1: Initialize, make all vertices as "unvisited"
2: RecursiveDFS(s)
```

Algorithm 8 RecursiveDFS(v)

```
1: Mark v as "visited"
2: for ∀u ∈ d_s do:
3:     if u is "unvisited" then
4:         RecursiveDFS(u)
```

The running time of DFS is $O(n + m)$.
If we only want to know if s and t are connected, the algorithm can be terminated if $u = t$.

21.6.3 Breadth-First Search (BFS)

The idea of BFS is enumerating siblings before children when search in the graph/tree. The general steps of BFS is as following: First, build layers $L_0, L_1, \cdots$; Next, set $L_0 = \{s\}$, where s is the starting vertex; Then, L_{j+1} contains all nodes that are not in $\cup_{i=0}^{j} L_i$ and have an edge to a vertex in L_j
The algorithm is as following

Algorithm 9 Breadth-First Search

```
1: Initialize, head ← 1, tail ← 1, queue[1] ← s, mark all vertices as "unvisited"
2: Mark s as "visited"
3: while head ≥ tail do
4:     v ← queue[tail], tail ← tail + 1
5:     for ∀u ∈ d_v do
6:         if u is "unvisited" then
7:             head ← head + 1, queue[head] = u
8:             Mark u as "visited"
```

The running time of BFS is $O(n + m)$
If we only want to know if s and t are connected, the algorithm can be terminated if $u = t$.

21.6.4 Cycle detection

The following algorithm is for connected graph, if the graph is not connected, run the algorithm for each component until cycle is detected or all the components have been calculated. Since the complexity for running in connected graph is $O(n + m)$, n as the number of vertices/nodes, and m as the number of edges/arcs, the running time of disconnected graph is the **summation** of running time in each component, where each component is connected. Therefore the complexity is the same in disconnected graph as in connected graph.
The main idea is starting with arbitrary vertex/node, using DFS or BFS to search on the graph try to revisit the vertex/node we start with. If succeed, a cycle is detected, otherwise if all the vertices/nodes has been visited, then no cycle exists. And in linked-list representation, the complexity is $O(|V| + |E|)$, i.e. $O(n + m)$. However, there is slightly different in undirected graph and directed graph, for undirected graph needs at least three vertices to form a cycle while directed graph needs at least two.
Here is the detail algorithm (DFS) for undirected graph:

Algorithm 10 Main algorithm

```
1: For all nodes, labeled as "unvisited"
2: Arbitrary choose a vertex v, add a dummy vertex w, add a dummy edge (w, v), label w as "visited"
3: run DFS(w, v)
4: Remove dummy vertex w and dummy edge (w, v)
5: if DFS(w, v) returns "Cycle is found" then
6:     return "Cycle is found"
7: else
8:     return "No cycle detected"
```

Algorithm 11 DFS(w, v)

```
1: Label v as "visited"
2: if number of v 's neighbor is 1 then
3:     return null
4: else
5:     for all neighbor u in linked-list of v excepts w do
6:         if u is labeled as "visited" then
7:             return "Cycle is found"
8:         else
9:             run DFS(v, u)
```

Now check the complexity. Denote $v \in N$ as a node in graph G, total number of nodes denoted by n, denote d_v as number of neighbors of node v. The complexity of $DFS(w, v)$ is $O(d_v)$ for each node v it visited (it should be $O(v)$ because we need $O(1)$ to check if a node is w), each node can only be visited, by "visited" it means $DFS(w, v)$ is executed, at most once, which is controlled by the "visited" label. The total complexity is $O(n + \sum_{v \in N} d_v) = O(n + m)$

For directed graph, we assume in the linked-list of each node, 1) it only contains the vertices where the node leading out to, or 2) we can distinguish the vertices it leading out to in $O(1)$. Here is the detail algorithm (DFS) for directed graph:

Algorithm 12 Main algorithm

```
1: For all nodes, labeled as "unvisited"
2: Arbitrary choose a node v, run DFS(v)
3: if DFS(v) returns "Cycle is found" then
4:     return "Cycle is found"
5: else
6:     return "No cycle detected"
```

Algorithm 13 DFS(v)

```
1: Label v as "visited"
2: if number of vertices v leading out to is 0 then
3:     return null
4: else
5:     for all leading out vertices u in linked-list of v do
6:         if u is labeled as "visited" then
7:             return "Cycle is found"
8:         else
9:             run DFS(u)
```

Now check the complexity. Similar to undirected case, denote $v \in N$ as a node in graph G, total number of nodes denoted by n, denote d_v as number of edges leading out from node v. The complexity of $DFS(v)$ is $O(d_v)$ for each

node v it visited, each node can only be visited, by "visited" it means $DFS(v)$ is executed, at most once, which is controlled by the "visited" label. The total complexity is $O(n + \sum_{v \in N} d_v) = O(n + m)$
Also, for undirected (connected) graph, if we don't need to find the cycle and only need to decide if there is a cycle or not, we can just check the number of vertices, n, and number of edges, m. If $n \leq m$ then G contains a cycle, otherwise no cycle, the complexity is $O(1)$.

21.6.5 Test Bipartiteness

We have proved that a bipartite graph only has even cycles, and the graph with only even cycles are bipartite graph, however, that is not very continent to test if a graph is bipartite because it needs to enumerate all cycles.
The other idea to test bipartiteness is try to color the vertices of the graph, if it can be 2-colored, then the graph is bipartite, otherwise it is not.
The following is the algorithm (using BFS)

Algorithm 14 Test Bipartiteness

```
1: Initialize, head ← 1, tail ← 1, queue[1] ← s, mark all vertices as "unvisited"
2: Mark s as "visited"
3: color[s] ← 0
4: while head ≥ tail do
5:     v ← queue[tail], tail ← tail + 1
6:     for ∀u ∈ d_v do
7:         if u is "unvisited" then
8:             head ← head + 1, queue[head] = u
9:             Mark u as "visited"
10:            color[u] ← 1 − color[v]
11:        else
12:            if color[u] == color[v] then return False
   return True
```

21.6.6 Topological Ordering

The topological ordering problem is given a directed acyclic graph $G = (V, E)$, output a 1-to-1 function $\pi : V \to \{1, 2, \cdots, n\}$ so that if $(u, v) \in E$, $\pi(u) < \pi(v)$
The idea is each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges. To make the algorithm efficient, we can 1) use linked lists of outgoing edges; 2) maintain the in degree d_v of vertices; 3) maintain a queue (or stack) of vertices v with $d_v = 0$
The following is the algorithm

Algorithm 15 topological-sort(G)

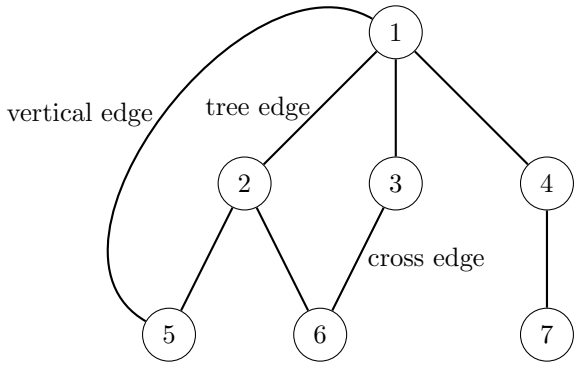
```
1: Initialize, let ∀v ∈ V, d_v = 0
2: for v ∈ V do
3:     for u, (u, v) ∈ E do
4:         d_u ← d_u + 1
5: S ← {v : d_v = 0}, i ← 0
6: while S ≠ ∅ do
7:     v ← v ∈ S, S ← S \ {v}
8:     i ← i + 1, π(v) ← i
9:     for u, (u, v) ∈ E do
10:        d_u ← d_u − 1
11:        if d_u = 0 then
12:            S ← S ∪ {u}
13: if i < n then return Not directed acyclic graph
```

Running time $O(n + m)$

21.6.7 Bridge

Definition 21.6.1 (Tree Edge, Cross Edge, Vertical Edge)**.** Given a graph $G = (V, E)$ and a rooted tree T in G, edge in G can be classified by three types:

- Tree edges: edge in T
- Cross edges: (u, v), u and v do not have an ancestor-descendant relation
- Vertical edges: (u, v), u is an ancestor of v, or v is an ancestor of u



In a BFS tree T of a graph G, there can not be vertical edges, there cannot be cross edges (u, v) with u and v 2 levels apart. (Cross edge at most 1 level apart)
In a DFS tree T of a graph G, there can not be cross edges, there can only be tree edges and vertical edges.

Definition 21.6.2 (bridge)**.** Given a connected graph $G = (V, E)$, an edge $e \in E$ is called a **bridge** if the graph $G = (V, E \setminus \{e\})$ is disconnected.

The idea to find bridge is through a DFS tree. Notice that there are only tree edges and vertical edges in DFS tree. Vertical edges are not bridges, a tree edge (u, v) is not a bridge if some vertical edge jumping from below u to above v. Other tree edges are bridges.
Define $level(v)$ as the level of vertex v in DFS tree. T_v as the sub tree rooted at v, $h(v)$ as the smallest level that can be reached using a vertical edge from vertices in T_v. $(parent(u), u)$ is a bridge if $h(u) \geq level(u)$. The algorithm is as following:

Algorithm 16 FindBridge(G)

```
1: Mark all vertices as "unvisited"
2: for v ∈ V do
3:     if v is "unvisited" then
4:         level(v) ← 0
5:         RecursiveDFS(v)
```

21.7 Blocks

Algorithm 17 RecursiveDFS(v)

```
1: mark v as "visited"
2: h(v) ← ∞
3: for u ∈ d_v do
4:     if u is "unvisited" then
5:         level(u) ← level(v) + 1
6:         RecursiveDFS(u)
7:         if h(u) ≥ level(u) then
8:             (u, v) is a bridge
9:         if h(u) < h(v) then
10:            h(v) ← h(u)
11:    else
12:        if level(u) < level(v) − 1 and level(u) < h(v) then
13:            h(v) ← level(u)
```

Chapter 22

Euler Tours and Hamilton Cycles

Chapter 23

Matriod, Planarity

23.1 Plane and Planar Graphs

23.2 Dual Graphs

23.3 Matroids

Definition 23.3.1 (Matroids)**.** Let S be a finite set of **elements** and let d be a collection of subsets of S satisfying the property

$$\text{If } x \leq y, y \in d, \Rightarrow x \in d \tag{23.1}$$

The pair (S, d) is called an **independent system** and the members of d are called **independent sets**.

Example. Let G be a graph and let $S \in E(G)$ define $M \subseteq S$ to be independent if M is a matching

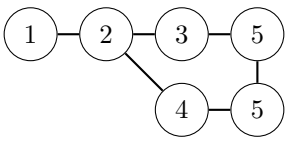
$$S = \{(1,2),(2,3),(2,4),(3,5),(4,6),(5,6)\} \tag{23.2}$$
$$d = \{\emptyset, \{(1,2)\}, \{(2,3)\}, ..., \{\text{other matching...}\}\} \tag{23.3}$$

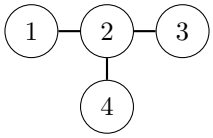
Example. Let G be a graph and let $S = V(G)$ define $X \subseteq S$ to be independent if no two member of x are adjacent.

$$S = \{1,2,3,4\} \tag{23.4}$$
$$d = \{\emptyset, 1, 2, 3, 4, (1,3), (1,4), (3,4), (1,3,4)\} \tag{23.5}$$

Example. Let G be a connected graph and let $S = E(G)$, define $X \subseteq S$ to be independent if $G[X]$ contains cycles.

Given any independent system, there is a natural combinatorial optimization problem of finding the maximum cardinality independent set.
Let's try the following: **Greedy algorithm**
Step 1: Set $I = \emptyset$
Step 2: If there exists $e \in S \setminus I$ such that $I + e$ is independent, set $I \leftarrow I + e$ and go to Step 1, otherwise stop.
Those Independent systems for which the greedy algorithm guarantee to find a maximum cardinality independent set are very special called **matroids**





23.4 Independent Sets

23.5 Ramsey's Theorem

23.6 Turán's Theorem

23.7 Schur's Theorem

23.8 Euler's Formula

23.9 Bridges

23.10 Kuratowski's Theorem

23.11 Four-Color Theorem

23.12 Graphs on other surfaces

Chapter 24

Matchings

24.1 Matching

Definition 24.1.1 (matching, M-saturated, M-unsaturated)**.** Let $G = (V, E)$ be a graph, a **matching** is a subset of edges $M \subseteq E$ such that no two elements of M are adjacent. The two ends of an edge in M are said to be **matched** under M. A matching M saturates a vertex v, and v is said to be **M-saturated**, if some edge of M is incident with v. Otherwise, v is **M-unsaturated**.

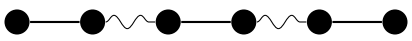
Definition 24.1.2 (perfect matching, maximum matching)**.** If every vertex of G is M-saturated, then the matching is said to be **perfect matching**, for perfect matching, we have $|M| = \frac{|V(G)|}{2}$. M is a **maximum matching** if G has no matching M' with $|M'| > |M|$. Every perfect matching is maximum. The maximum matching does not necessarily to be perfect. Perfect matching and maximum matching may not be unique.

Definition 24.1.3 (M-alternating)**.** An **M-alternating** path in G is a path whose edges are alternately in $E \setminus M$ and M.

Definition 24.1.4 (M-augmenting)**.** An **M-augmenting** path in G is an M-alternating path whose origin and terminus are M-unsaturated.

Lemma 24.1. *Every augmenting path P has property that let $M' = P\Delta M = (M \cup P) \setminus (M \cap P)$ then M' contains one more edge then M*

The following path is an M-augmenting path



The following path is $M' = P\Delta M(M \cup P) \setminus (M \cap P)$ and all the vertices are M-saturated.

Theorem 24.2 (Berge, 1957)**.** *A matching M in a graph G is maximum iff G has no M-augmenting path.*

Proof. ($\Rightarrow$) It is clear that if M is maximum, it has no augmenting paths since otherwise by problem claim we can increase by one.
($\Leftarrow$) Suppose M is not maximum and let M' be a bigger matching. Let $A = M\Delta M'$ now no vertex of G is incident to more than two members of A. For otherwise either two members of M or two members of M' would be adjacent. Contradict the definition of matching. It follows that every component of the edges incident subgraph $G[A]$ is either an even cycle with edge augmenting in $M\Delta M'$ or else A path with edges alternating between M and M'.
Since $|M'| \geq |M|$ then the even cycle cannot help because exchanging M and M' will have same cardinality.
The path case implies that p is alternating in M and since $|M'| > |M|$ the end arc exposed so that p is augmenting. □

Definition 24.1.5 (Vertex-cover)**.** The **vertex-cover** is a subset of vertices X such that every edge of G is incident to some member of X.

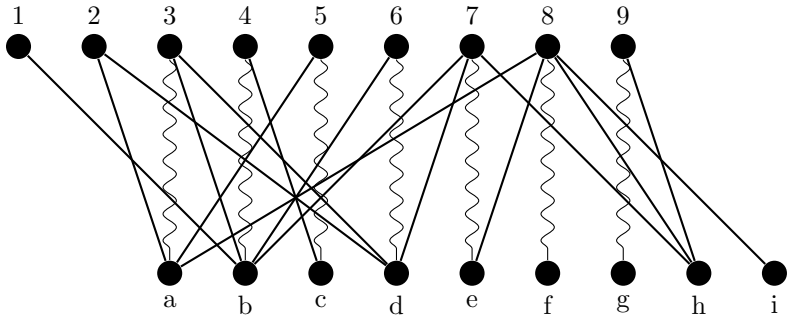
Lemma 24.3. *The cardinality of any matching is less than or equal to the cardinality of any vertex cover.*

Proof. Consider any matching. Any vertex cover must have nodes that at least incident to the edges in the matching. Since all the edges in the matching are disjointed, so for a single node can at most cover one edge in the matching. If the matching is not perfect, for the edges that not in the tree, they may or may not be possible to be covered by the nodes incident to the edges in the matching, with an easy triangle graph example, we can prove this lemma. □

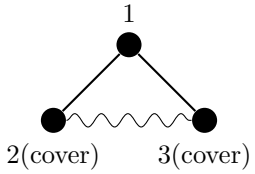
Theorem 24.4 (König Theorem)**.** *If G is bipartite, the cardinality of the maximum matching is equal to the cardinality of the minimum vertex cover.*

Proof. Let G be a bipartite graph, $G = (V, E)$ where $V = X \cup Y$ as X and Y are two disjointed sets of vertices. Let M be a maximum matching on G. For each edge in M, denoted by $e_i = a_i b_i$ where $e_i \in M$, $a_i \in A$ and $b_i \in B$ and $A = \{a_i : e_i \in M\} \subseteq X$, and $B = \{b_i : e_i \in M\} \subseteq Y$. Therefore, we can partition X by A and $U = X \setminus A$, partition Y by B and $W = Y \setminus B$.
We can further partition the matching M into M_1 and M_2. For all the edges in M_1 can be included into an M-alternating path starts from a vertex in U (which includes the edges directly linked to vertices in U), and $M_2 = M \setminus M_1$. For edges in M_1, we take the ends of edges in B in the vertex cover, denoted by B_1, take the ends of edges in A as a subset denoted by $A_1 \subseteq A$. For the edges in M_2, we take the ends of edges in A in the vertex cover, denoted by A_2, and the ends of edges in B as a subset denoted by $B_2 \subseteq B$.
We claim that all the vertices in U can only be connected to vertices in B_1 and vertices in W can only be connected to vertices in A_2.
$U \subset X$ connects to vertices in B_1 by definition. If vertices in $W \subset Y$ is connected to vertices in A_1, then we will have M-augmenting path which is contradicted to the assumption that M is maximum matching. □

The following is an example. Where the edge in the matching that accessible from members of $U = \{1, 2\}$ in an M-alternating path is edge $3a, 4b, 5c, 6d$.



In which $U = \{1, 2\}$, $M_1 = \{3a, 4b, 5c, 6d\}$. $U = \{1, 2, \}, A_1 = \{3, 4, 5, 6\}, A_2 = \{7, 8, 9\}, W = \{h, i\}, B_1 = \{a, b, c, d\}, B_2 = \{e, f, g\}$. The vertex cover is $\{a, b, c, d, 7, 8, 9\}$.
The above theorem does not apply to non-bipartite graph. The following is an example



The maximum matching has one edge, where the minimum cover has two vertices.

24.2 Maximum Matching Algorithm

Definition 24.2.1 (M-alternating tree)**.** An **M-alternating tree** T is a rooted tree satisfied the following condition:

- The root r is M-unsaturated
- The unique path from r to any vertex T is M-alternating
- Every vertex in T, except r is incident to a matching edge of T

A vertex x of T is called **inner** if (r, s)-path in T has an odd number of edges. Otherwise x is called **outer**.

Lemma 24.5. *Let M be a matching in G and let T be an M-alternating tree with root r, then the following conclusion hold*

- *If $v \neq r$ is an outer vertex and p is the unique (r, v)-path in T then the edge of p incident to v is in M*
- *The number of inner vertices in T equals the number of matching edges in T.*

Definition 24.2.2 (Alternating forest)**.** An **alternating forest** is a forest of G where every components is an alternating tree. An alternating forest (F, e) is an alternating forest to be then with an edge $e = M, V$ where M and V are outer vertices contained in two distinct components of F.

Example (Hungarian forest)**.** A **Hungarian forest** F is an alternating forest containing all exposed vertices of G and such that the outer vertices of F are adjacent in G only to inner vertices of F

Definition 24.2.3 (Augmenting forest)**.** An **augmenting forest** (F, e) where F is an augmenting forest and $e = uv$ connects two outer vertices in distinct components of F.

The plan is to grow an alternating forest that eventually become augmenting or Hungarian. Augmenting forest will increase the cardinality of the match, Hungarian implies that you have found optimal maximum cardinality matching.

Theorem 24.6. *Let (F, e) be an augmenting forest. And let T_1 and T_2 be the two components of F containing an end of e. Let p_i be the unique path in T_i from the root to the end of e, then $p_1 e p_2^{-1}$ is an augmenting path.*

Theorem 24.7. *If F is a Hungarian forest for some matching M then M is a maximum match.*

The above theorems suggest a method for computing maximum matching. Let M be a matching of G and let F be an alternating forest in G made up of all M-exposing vertices. If F happens to be Hungarian, stop with the max matching M. If (F, e) for some e is augmenting then we increase our matching by 1 and start process.
Suppose F is neither Hungarian nor augmenting, by definition, there must be an edge e incident to an outer vertex of F to no inner vertex of F. But e cannot be incident to two outer vertices of F in distinct components, since (F, e) is not augmenting. Hence there are only two cases:
Case 1: $e = uv$ where u is outer in F and v is not outer in F. The only way its possible if v is covered. Augmenting F to M will increase the matching.
Case 2: $e = uv$ where u and v are outer vertices in F. Let r be the root of the component of F containing u and v, let p_u and p_v be (r, u)-path and (r, v)-path in F. Let b be the last vertex these two paths have in common and let p be the (u, v)-path in F, let p'_u be $(b - u)$-path and p'_v be the (b, v)-path respectively. Let n be the length of p_u, let m be the length of p_v, let k be the length of (r, b)-path. Let c be the cycle $(p'_u e {p'_v}^{-1})$. Number of vertices in c is $n + m - 2k + 1$, n, m are even, so c is always an odd length cycle. If G has no odd cycles, we call those graph bipartite. To this case can not happen in bipartite graph so algorithm without case 2 will solve bipartite matching. If a graph has no odd cycles, i.e., bipartite, then we have an algorithm using augmenting forest and Hungarian forest and case 1.
We now have to deal with odd cycles. The idea is to "shrink" add cycles to a super node
Let $S \subseteq E(G)$, denote by $G : S$ the subgraph with edge set S

$$G : S = G \setminus (E(G) - S) \tag{24.1}$$

The contraction of S to be the $G \setminus S$ with $E(G/S) = E(G) - S$. $V(G/S)$ to be the components of $G : S$ and if $e \in E(G/S)$ then the ends of e in G/S are in components of $G : S$ containing both ends in G
Let network to general case 2. b is outer vertex. Let B be the set of edges of cycle C, we call B a **blossom**. We propose to replace M by $M - B$, G by G/B and F by F/B

24.3 Edmonds's Blossom Algorithm

$O(|V|^4)$ - Non-bipartite matching is one of very few problems in P, for which LP relaxation will not provide optimal solution.

24.4 Hall's Marriage Theorem

24.5 Transversal Theory

24.6 Menger's Theorem

24.7 The Hungarian Algorithm

Chapter 25

Colorings

Chapter 26

Minimum Spanning Tree Problem

26.1 Basic Concepts

Example. A company wants to build a communication network for their offices. For a link between office v and office w, there is a cost c_{vw}. If an office is connected to another office, then they are connected to with all its neighbors. Company wants to minimize the cost of communication networks.

Definition 26.1.1 (Cut vertex)**.** A vertex v of a connected graph G is a **cut vertex** if $G \setminus v$ is not connected.

Definition 26.1.2 (Connection problem)**.** Given a connected graph G and a positive cost C_e for each $e \in E$, find a minimum-cost spanning connnected subgraph of G. (Cycles all allowed)

Lemma 26.1. *An edge $e = uv \in G$ is an edge of a cycle of G iff there is a path $G \setminus e$ from u to v.*

Definition 26.1.3 (Minumum spanning tree problem)**.** Given a connected graph graph G, and a cost $C_e, \forall e \in E$, find a minimum cost spanning tree of G

The only way a connection problem will be different than MSP is if we relax the restriction on $C_e > 0$ in the connection problem.

26.2 Kroskal's Algorithm

Algorithm 18 Kroskal's Algorithm, $O(m \log m)$

Require: A connected graph
Ensure: A MST
  Keep a spanning forest $H = (V, F)$ of G, with $F = \emptyset$
  while $|F| < |V| - 1$ **do**
    add to F a least-cost edge $e \notin F$ such that H remains a forest.

26.3 Prim's Algorithm

Algorithm 19 Prim's Algorithm, $O(nm)$

Require: A connected graph
Ensure: A MST
  Keep $H = (V(H), T)$ with $V(H) = \{v\}$, where $r \in V(G)$ and $T = \emptyset$
  while $|V(T)| < |V|$ **do**
    Add to T a least-cost edge $e \notin T$ such that H remains a tree.

26.4 Extensible MST

Definition 26.4.1 (cut)**.** For a graph $G = (V, E)$ and $A \subseteq V$ we denote $\delta(A) = \{e \in E : e \text{ has an end in } A \text{ and an end in } V \setminus A\}$. A set of the form $\delta(A)$ for some A is called a **cut** of G.

Definition 26.4.2. We also define $\gamma(A) = \{e \in E : \text{both ends of } e \text{ are in } A\}$

Theorem 26.2. *A graph $G = (V, E)$ is connected iff there is no $A \subseteq V$ such that $\emptyset \neq A \neq V$ with $\delta(A) = \emptyset$*

Definition 26.4.3. Let us call a subset $A \in E$ **extensible** to a minimum spanning tree problem if A is contained in the edge set of some MST of G

Theorem 26.3. *Suppose $B \subseteq E$, that B is extensible to an MST and that e is a minimum cost edge of some cut D satisfying $D \cap B = \emptyset$, then $B \cup \{e\}$ is extensible to an MST.*

26.5 Solve MST in LP

Given a connected graph $G = (V, E)$ and a cost on the edges C_e for all $e \in E$, Then we can formulate the following LP

$$X_e = \begin{cases} 1, \text{if edge } e \text{ is in the optimal solution} \\ 0, \text{otherwise} \end{cases} \tag{26.1}$$

The formulation is as following

$$\min \quad \sum_{e \in E} c_e x_e \tag{26.2}$$

$$\text{s.t.} \quad \sum_{e \in E} x_e = |V| - 1 \tag{26.3}$$

$$x_e \geq 0 \tag{26.4}$$

$$e \in E \tag{26.5}$$

$$\sum_{e \in E(S)} x_e = |S| - 1, \forall S \subseteq V, S \neq \emptyset \tag{26.6}$$

$$\tag{26.7}$$

Chapter 27

Shortest-Path Problem

27.1 Basic Concepts

All Shortest-Path methods are based on the same concept, suppose we know there exists a dipath from r to v of a cost y_v. For each vertex $v \in V$ and we find an arc $(v, w) \in E$ satisfying $y_v + v_{vw} < y_w$. Since appending (v, w) to the dipath to v takes a cheaper dipath to w then we can update y_w to a lower cost dipath.

Definition 27.1.1 (feasible potential)**.** We call $y = (y_v : v \in V)$ a **feasible potential** if it satisfies

$$y_v + c_{vw} \geq y_w \quad \forall (v, w) \in E \tag{27.1}$$

and $y_r = 0$

Proposition 5. *Feasible potential provides lower bound for the shortest path cost.*

Proof. Suppose that you have a dipath $P = v_0 e_1 v_1, ..., e_k v_k$ where $v_0 = r$ and $v_k = v$, then

$$C(P) = \sum_{i=1}^{k} C_{e_i} \geq \sum_{i=1}^{k} (y_{v_i} - y_{v_{i-1}}) = y_{v_k} - y_{v_0} \tag{27.2}$$

□

27.2 Breadth-First Search Algorithm

27.3 Ford's Method

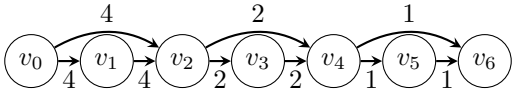
Define a predecessor function $P(w), \forall w \in V$ and set $P(w)$ to v whenever y_w is set to $y_v + c_{vw}$

Algorithm 20 Ford's Method

Ensure: Shortest Paths from r to all other nodes in V
Require: A digraph with arc costs,starting node r
  Initialize, $y_r = 0$ and $y_v = \infty, v \in V \setminus r$
  Initialize, $P(r) = 0, P(v) = -1, \forall v \in V \setminus r$
  while y is not a feasible potential **do**
    Let $e = (v, w) \in E$ (this could be problematic)
    if $y_v + c_{vw} < y_w$ (incorrect) **then**
      $y_w \leftarrow y_v + c_{vw}$ (correct it)
      $P(w) = v$ (set v as predecessor)

Notice: Technically speaking, this is not an algorithm, for the following reasons: 1) We did not specify how to pick e, 2) This procedure might not stop given some situations, e.g., if there is a cycle with minus total weight

Notice: This method can be really bad. Here is another example that could take $O(2^n)$ to solve.



27.4 Ford-Bellman Algorithm

Algorithm 21 Ford-Bellman Algorithm

Ensure: Shortest Paths from r to all other nodes in V
Require: A digraph with arc costs,starting node r
  Initialize y and p
  for $i = 0; i < N; i++$ **do**
    for $\forall e = (v, w) \in E$ **do**
      if $y_v + c_{vw} < y_w$ (incorrect) **then**
        $y_w \leftarrow y_v + c_{vw}$ (correct it)
        $P(w) = v$ (set v as predecessor)
  for $\forall e = (v, w) \in E$ **do**
    if $y_v + c_{vw} < y_w$ (incorrect) **then**
      Return error, negative cycle

Notice: Only correct the node that comes from a node that has been corrected.

A usual representation of a digraph is to store all the arcs having tail v in a list L_v to **scan** v means the following:

- For $(v, w) \in L_v$, if (v, w) is incorrect, then correct (v, w)

For Bellman, will either terminate with shortest path from r to all $v \in V \setminus r$ or it will terminate stating that there is a negative cycle. In $O(mn)$
In the algorithm if $i = n$ and there exists a feasible potential, the problem has a negative cycle.
Suppose that the nodes of G can be ordered from left to right so that all arcs go from left to right. That is suppose there is an ordering $v_1, v_2, ..., v_n \in V$ so that $(v_i, v_j) \in V$ implies $i < j$. We call such an ordering **topological** sort. If we order E in the sequence that $v_i v_j$ precedes $v_k v_i$ if $i < k$ based on topological order then ford algorithm will terminate in one pass.

27.5 SPFA Algorithm

27.6 Dijkstra Algorithm

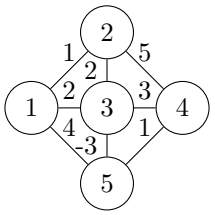
27.7 A* Algorithm

27.8 Floyd-Warshall Algorithm

If all weights/distances in the graph are nonnegative then we could use Dijkstra within starting nodes being any one of the vertices of the graph. This method will take $O(n^3)$
If weight/distances are arbitrary and we would like to find shortest path between all pairs of vertices or detect a negative cycle we could use Bellman-Ford Algorithm with $O(n^4)$
We would like an algorithm to find shortest path between any two pairs in a graph for arbitrary weights (determined, negative, cycles) in $O(n^3)$

Algorithm 22 Dijkstra Algorithm

Ensure: Shortest Paths from r to all other nodes in V
Require: A digraph with arc costs,starting node r
  Initialize y and p
  $S \leftarrow V$
  while $S \neq \emptyset$ **do**
    Choose $v \in S$ with minimum y_v
    $S \leftarrow S \setminus v$
    for $\forall w, (v, w) \in E$ **do**
      if $y_v + c_{vw} < y_w$ (incorrect) **then**
        $y_w \leftarrow y_v + c_{vw}$ (correct it)
        $P(w) = v$ (set v as predecessor)

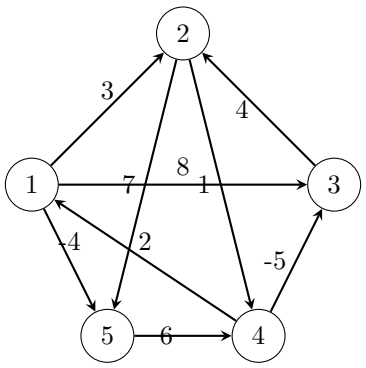


Let d_{ij}^k denote the length of the shortest path from i to j such that all intermediate vertices are contained in the set $\{1, ..., k\}$
In this case $d_{14}^5 = 5$
If the vertex k is not an intermediate vertex on p, then $d_{ij}^k = d_{ij}^{k-1}$, notice that $d_{15}^4 = -1$, node 4 is not intermediate, so $d_{15}^3 = -1$
If the vertex k is an intermediate on p, then $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$, $d_{14}^5 = 0$ ($p = 1 \to 3 \to 5 \to 4$), i.e., $d_{14}^5 = d_{15}^4 + d_{54}^4 = 0$
Therefore $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$
Input: graph $G = (V, E)$ with weight on edges Output: Shortest path between all pairs of vertices on existence of a negative cycle Step 1: Initialize

$$d_{ij}^0 = \begin{cases} c_{ij} & \text{distance from } i \text{ to } j \text{ if } (i,j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \notin E \end{cases} \tag{27.3}$$

Step: For k = 1 to n For i = 1 to n For j = 1 to n $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$ Next j Next i Next k Between optimal matrix D^n

$$D^0 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{27.4}$$



$$\Pi^0 = \begin{bmatrix} & 1 & 1 & & 1 \\ & & & 2 & 2 \\ & 3 & & & \\ 4 & & 4 & & \\ & & & 5 & \end{bmatrix} \tag{27.5}$$

$$D^1 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \mathbf{5} & -5 & 0 & \mathbf{-2} \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{27.6}$$

$$\Pi^1 = \begin{bmatrix} & 1 & 1 & & 1 \\ & & & 2 & 2 \\ & 3 & & & \\ 4 & \mathbf{1} & 4 & & \mathbf{1} \\ & & & 5 & \end{bmatrix} \tag{27.7}$$

$$D^2 = \begin{bmatrix} 0 & 3 & 8 & \mathbf{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \mathbf{5} & \mathbf{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{27.8}$$

$$\Pi^2 = \begin{bmatrix} & 1 & 1 & \mathbf{2} & 1 \\ & & & 2 & 2 \\ & 3 & & \mathbf{2} & \mathbf{2} \\ 4 & 1 & 4 & & 1 \\ & & & 5 & \end{bmatrix} \tag{27.9}$$

$$D^3 = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & \mathbf{-1} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{27.10}$$

$$\Pi^3 = \begin{bmatrix} & 1 & 1 & 2 & 1 \\ & & & 2 & 2 \\ & 3 & & 2 & 2 \\ 4 & \mathbf{3} & 4 & & 1 \\ & & & 5 & \end{bmatrix} \tag{27.11}$$

$$D^4 = \begin{bmatrix} 0 & 3 & \mathbf{-1} & 4 & -4 \\ \mathbf{3} & 0 & \mathbf{-4} & 1 & \mathbf{-1} \\ \mathbf{7} & 4 & 0 & 5 & \mathbf{3} \\ 2 & -1 & -5 & 0 & -2 \\ \mathbf{8} & \mathbf{5} & \mathbf{1} & 6 & 0 \end{bmatrix} \tag{27.12}$$

$$\Pi^4 = \begin{bmatrix} & 1 & \mathbf{4} & 2 & 1 \\ \mathbf{4} & & \mathbf{4} & 2 & \mathbf{1} \\ \mathbf{4} & 3 & & 2 & \mathbf{1} \\ 4 & 3 & 4 & & 1 \\ \mathbf{4} & \mathbf{3} & \mathbf{4} & 5 & \end{bmatrix} \tag{27.13}$$

$$D^5 = \begin{bmatrix} 0 & \mathbf{1} & \mathbf{-3} & \mathbf{2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \tag{27.14}$$

$$\Pi^5 = \begin{bmatrix} & \mathbf{3} & 4 & \mathbf{5} & 1 \\ 4 & & 4 & 2 & 1 \\ 4 & 3 & & 2 & 1 \\ 4 & 3 & 4 & & 1 \\ 4 & 3 & 4 & 5 & \end{bmatrix} \tag{27.15}$$

Time complexity $O(n^3)$
If during the previous processes, there exist an element of negative value in the diagonal, it means there exists negative cycle.

27.9 Johnson's Algorithm

Chapter 28

Maximum Flow Problem

28.1 Basic Concept

Let $D = (V, A)$ be a strict diagraph with distinguished vertices s and t. We call s the source and t the sink, let $u = \{u_e : e \in A\}$ be a nonnegative integer-valued capacity function defined on the arcs of D. The maximum flow problem on (D, s, t, u) is the following Linear program.

$$\max \quad v \tag{28.1}$$

$$\text{s.t.} \quad \sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = \begin{cases} -v, & \text{if } i = s \\ v, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \tag{28.2}$$

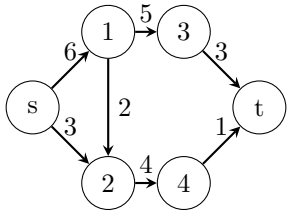
$$0 \le x_e \le u_e, \quad \forall e \in A \tag{28.3}$$

We think of x_e as being the flow on arc e. Constraint says that for $i \neq s, t$ the flow into a vertex has to be equal to the flow out of vertex. That is, flow is conceded at vertex i for $i = s$ and for $i = t$ the net flow in the entire digraph must be equal to v. A $\mathbf{x_e}$ that satisfied the above constraints is an (s, t)-flow of value v. If in addition it satisfies the bounding constraints, then it is a feasible (s, t)-flow. A feasible (s, t)-flow that has maximum v is optimal on maximum.

28.2 Solving Maximum Flow Problem in LP

Theorem 28.1. *For $S \subseteq V$ we define $(S, \bar{S})$ to be a (s, t)-cut if $s \in S$ and $t \in \bar{S} = V - S$, the capacity of the cut, denoted $u(S, \bar{S})$ as $\sum\{u_e : e \in \delta^-(S)\}$ where $\delta^-(S) = \{e \in A : t(e) \in S$ and $h(e) \in \bar{S}\}$*

Example. For the following graph:
Let $S = \{1, 2, 3, s\}$, $\bar{S} = \{4, t\}$

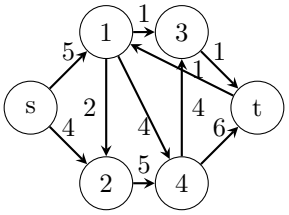


then $\delta^-(S) = \{(2, 4), (3, t)\} \Rightarrow u(S, \bar{S}) = 7$

Definition 28.2.1. If $(S, \bar{S})$ has minimum capacity of all (s, t)-cuts, then it is called **minimum cut**.

Definition 28.2.2. Let $\delta^+(S) = \delta^-(V - S)$

Example. Let $S = \{s, 1, 2, 3\}$, $\bar{S} = \{4, t\}$, $u(S, \bar{S}) = u_{14} + u_{24} + u_{3t} = 10$, $\delta^-(S) = \{(1, 4), (2, 4), (3, t)\}$, $\delta^+ S = \{(4, 3), (t, 1)\}$



Lemma 28.2. *If x is a (s,t) flow of value v and $(S,\bar{S})$ is a (s,t)-cut, then*

$$v = \sum_{e\in\delta^-(S)} x_e - \sum_{e\in\delta^+(S)} x_e \tag{28.4}$$

Proof. Summing the first set of constraints over the vertices of S,

$$\sum_{i\in S}(\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e) = -v \tag{28.5}$$

Now for an arc e with both ends in S, x_e will occur twice once with a positive and once with negative so they cancel and the above sum is reduced to

$$\sum_{e\in\delta^+(S)} x_e - \sum_{e\in\delta^-(S)} x_e = -v \tag{28.6}$$

□

Notice: Flow is the prime variable, capacity is the dual variable.

Corollary 28.2.1. *If x is a feasible flow of value v, and $(S,\bar{S})$ is an (s,t)-cut, then*

$$v \le u(S,\bar{S}) \quad \textit{(Weak duality)} \tag{28.7}$$

Definition 28.2.3. Define an arc e to be **saturated** if $x_e = u_e$, and to be **flowless** if $x_e = 0$

Corollary 28.2.2. *Let x be a feasible flow and $(S,\bar{S})$ be a (s,t)-cut, if $\forall e \in \delta^-(S)$ is saturated, and $\forall e \in \delta^+(S)$ is flowless, then x is a maximum flow and $(S,\bar{S})$ is a minimum cut. (Strong duality)*

Proof. If every arc of $\delta^-(S)$ is saturated then

$$\sum_{e\in\delta^-(S)} x_e = \sum_{e\in\delta^-(S)} u_e \tag{28.8}$$

If every arc of $\delta^+(S)$ is flowless then

$$\sum_{e\in\delta^+(S)} x_e = 0 \tag{28.9}$$

$\Rightarrow x$ is as large as it can get when as $u(S,\bar{S})$ is as small as it can get. □

28.3 Prime and Dual of Maximum Network Flow Problem

The LP of maximum flow can be modeled as following, WLOG, we let $s = v_1 \in V, t = v_{|V|} \in V$.

$$\max \quad f = \begin{bmatrix}0 & 0 & \cdots & 0 & 1\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ f\end{bmatrix} \tag{28.10}$$

$$\text{s.t.} \quad \begin{bmatrix}\mathbf{A} & \mathbf{F}\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ f\end{bmatrix} = \mathbf{0} \tag{28.11}$$

$$\mathbf{Ix} \le \mathbf{u} \tag{28.12}$$

$$\begin{bmatrix}\mathbf{x}\\ f\end{bmatrix} \ge 0 \tag{28.13}$$

In which $\mathbf{A}$ is the vertex-arc incident matrix and $\mathbf{F}$ is a column vector where the first row is -1, last row is 1 and all other rows are 0s, which is because we denote the first vertex as source s and the last vertex as the sink t. $\mathbf{u}$ is the column vector of upper bound of each arcs.

$$\mathbf{A} = \mathbf{A}_{|E|\times|V|} = [a_{ij}], \text{ where } a_{ij} = \begin{cases} 1, & \text{if } v_i = h(e_j) \\ -1, & \text{if } v_i = t(e_j) \\ 0, & \text{otherwise} \end{cases} \tag{28.14}$$

$$\mathbf{F} = \begin{bmatrix} -1 & \cdots & 0 & \cdots & 1 \end{bmatrix}^\top \tag{28.15}$$

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & \cdots & u_{|E|} \end{bmatrix}^\top \tag{28.16}$$

Then, we take the dual of LP

$$\min \quad \mathbf{u}\mathbf{w_E} \tag{28.17}$$

$$\text{s.t.} \quad \begin{bmatrix} \mathbf{w_V} & \mathbf{w_E} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \end{bmatrix} \geq 0 \tag{28.18}$$

$$\begin{bmatrix} \mathbf{w_V} & \mathbf{w_E} \end{bmatrix} \begin{bmatrix} \mathbf{F} \\ \mathbf{0} \end{bmatrix} = 1 \tag{28.19}$$

$$\mathbf{w_V} \quad \text{unrestricted} \tag{28.20}$$

$$\mathbf{w_E} \geq \mathbf{0} \tag{28.21}$$

In which $\mathbf{w_V}$ is "whether or not" vertex v is in S where $(S, \bar{S})$ represents a cut, $\mathbf{w_E}$ is "whether or not" an arc in in $\delta^+(S)$. $\mathbf{u}, \mathbf{E}, \mathbf{F}$ have the same meaning as in prime.

$$\mathbf{w_V} = \begin{bmatrix} w_1 & w_2 & \cdots & w_{|V|} \end{bmatrix}^\top \tag{28.22}$$

$$\mathbf{w_E} = \begin{bmatrix} w_{|V|+1} & w_{|V|+2} & \cdots & w_{|V|+|E|} \end{bmatrix}^\top \tag{28.23}$$

To make it more clear, it can be rewritten as following

$$\min \quad \sum_{e\in E} u_e w_e \tag{28.24}$$

$$\text{s.t.} \quad w_i - w_j + w_{|V|+e} \geq 0, \forall e = (i,j) \in E \tag{28.25}$$

$$-w_1 + w_{|V|} = 1 \tag{28.26}$$

$$\mathbf{w_V} \quad \text{unrestricted} \tag{28.27}$$

$$\mathbf{w_E} \geq \mathbf{0} \tag{28.28}$$

The meaning for the first set of constraint is to decide whether or not an arc is in $\delta^+(S)$ of a $(S, \bar{S})$, which is decided by w_V. The $w_1 - w_{|V|} = 1$, which is the second set of constraint means the source $s = v_1$ and the sink $t = v_{|V|}$ has to be in S and $\bar{S}$ respectively.

28.4 Maximum Flow Minimum Cut Theorem

Definition 28.4.1. Let P be a path, (not necessarily a dipath), P is called **unsaturated** if every **forward** arc is unsaturated ($x_e < u_e$) and ever **reverse** arc has positive flow ($x_e > 0$). If in addition P is an (s,t)-path, then P is called an **x-augmenting path**

Theorem 28.3. *A feasible flow x in a digraph D is maximum iff D has no augmenting paths.*

Proof. (Prove by contradiction)
($\Rightarrow$) Let x be a maximum flow of value v and suppose D has an augmenting path. Define in P (augmenting path):

$$D_1 = \min\{u_e - x_e : e \text{ forward in } P\} \tag{28.29}$$

$$D_2 = \min\{x_e : e \text{ backward in } P\} \tag{28.30}$$

$$D = \min\{D_1, D_2\} \tag{28.31}$$

Since P is augmenting, then $D > 0$, let

$$\hat{x_e} = \begin{cases} x_e + D & \text{If } e \text{ is forward in } P \\ x_e - D & \text{If } e \text{ is backward in } P \\ x_e & otherwise \end{cases} \tag{28.32}$$

It is easy to see that $\hat{x}$ is feasible flow and that the value is $V + D$, a contradiction.
($\Leftarrow$) Suppose D admits no x-augmenting path, Let S be the set of vertices reachable from s by x-unsaturated path clearly $s \in S$ and $t \notin S$ (because otherwise there would be an augmenting path). Thus, $(S, \bar{S})$ is a (s,t)-cut.
Let $e \in \delta^-(S)$ then e must be saturated. For otherwise we could add the $h(e)$ to S
Let $e \in \delta^+(S)$ then e must be flow less. For otherwise we could add the $t(e)$ to S.
According to previous corollary, that x is maximum. □

Theorem 28.4. *(Max-flow = Minimum-cut) For any digraph, the value of a maximum (s,t)-flow is equal to the capacity of a minimum (s,t)-cut*

28.5 Ford-Fulkerson Method

Finding augmenting paths is the key of max-flow algorithm, we need to describe two functions, labeling and scanning a vertex.
A vertex is first labeled if we can find x-unsaturated path from s, i.e., (s,v)-unsaturated path.
The vertex v is scanned after we attempted to extend the x-unsaturated path.
This algorithm is incomplete/incorrect, needs to be fixed

Algorithm 23 Labeling algorithm

Ensure: Max-flow x with value v
Require: Digraph with source s and sink t, a capacity function u and a feasible flow (could be $x_e = 0$)
  Initialize, $v \leftarrow x$
  Designate all vertices as unlabeled and unscanned
  Label s
  while There exists vertex unlabeled or unscanned **do**
    Let i be such a vertex, for each arc e with $t(e) = i, x_e < u_e$ and $h(e)$ unlabeled, label $h(e)$
    For each arc e with $h(e) = i, x_e > 0$ and $t(e)$ unlabeled, label $t(e)$, designate i as scanned.
    If t is not label
  x is the maximum.

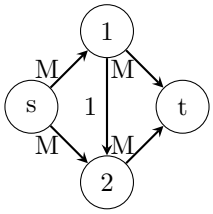
Algorithm 24 Ford-Fulkerson algorithm

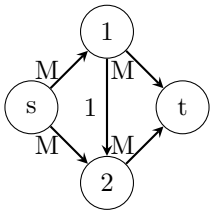
Ensure: Max-flow x with value v
Require: Digraph with source s and sink t, a capacity function u and a feasible flow (could be $x_e = 0$)
  Initialize, $v \leftarrow x$
  Designate all vertices as unlabeled and unscanned
  Label s
  while There exists vertex unlabeled or unscanned **do**
    Let i be such a vertex, for each arc e with $t(e) = i, x_e < u_e$ and $h(e)$ unlabeled, label $h(e)$
    For each arc e with $h(e) = i, x_e > 0$ and $t(e)$ unlabeled, label $t(e)$, designate i as scanned.
    If t is not label
  x is the maximum.

Labeling algorithm can be exponential, the following is an example

28.6 Polynomial Algorithm for max flow

Let (D, s, t, u) be a max flow problem and let x be a feasible flow for D, the **x-layers** of D are define be the following algorithm





Layer algorithm (Dinic 1977) Input: A network (D, s, t, u) and a feasible flow x Output: The **x-layers** $V_0, V_1, ..., V_l$ where $V_i \cap V_j = \emptyset \forall i \neq j$
Step 1: Set $V_0 = \{s\}, i \leftarrow 0 and l(x) = 0$ Step 2: Let R be the set of vertices w such that there is an arc e with either:

- $t(e) \in V_i, h(e) = w, x_e < u_e$ or
- $h(e) \in V_j, t(e) = w, x_e > 0$

Step 3: If $t \in R$, set $V_{i+1} = \{t\}$, $l(t) = i + 1$ and stop. Set $V_{i+1} \leftarrow R \setminus \cup_{0 \leq j \leq i} V_j$, $l \leftarrow i + 1, l(x) = i$, goto Step 2. If $V_{i+1} = \emptyset$, set $l(x) = i$ and Stop.

Example. For the following graph

$$\text{Second iteration} \tag{28.33}$$
$$V_0 = \{s\}, i = 0, l(x) = 0 \tag{28.34}$$
$$R = \{1, 2\} \tag{28.35}$$
$$V_1 \leftarrow \{1, 2\}, i = 1, l(x) = 1 \tag{28.36}$$
$$R = \{3, 4, 5\} \tag{28.37}$$
$$V_2 \leftarrow \{3, 4\}, i = 2, l(x) = 2 \tag{28.38}$$
$$R = \{1, 5, 6, 3\} \tag{28.39}$$
$$V_3 \leftarrow \{5, 6\}, i = 3, l(x) = 3 \tag{28.40}$$
$$R = \{4, t\} \tag{28.41}$$
$$V_4 = \{t\} \tag{28.42}$$
$$A_1 = \{(s, 1), (s, 2)\} \tag{28.43}$$
$$A_2 = \{(1, 3), (2, 4)\} \tag{28.44}$$
$$A_3 = \{(3, 5), (4, 6)\} \tag{28.45}$$
$$A_4 = \{(5, t), (6, t)\} \tag{28.46}$$

The layer network D_x is defined by $V(D_x) = V_0 \cup V_1 \cup V_2 \cdots \cup V_{l(x)}$
Suppose we have computed the layers of D and $t \in V_l(x)$, the last layer (last layer I am goin to V_e)
For each $i, 1 \leq i \leq l$, define a set of arcs A_i and a function $\hat{u}$ on A_i as following. For each $e \in A(D)$

- If $t(e) \in V_{i-1}, h(e) \in V_i$ and $x_e < u_e$ then add arc e to A_i and define $\hat{u}_e = u_e - x_e$
- If $h(e) \leftarrow V_{i-1}, t(e) \in V_i$ and $x_e > 0$ then add arc $e' = (h(e), t(e))$ to A_i with $\hat{u}_e - x_e$

Let $\hat{u}$ be the capacity function on D_x and let the source and sink of D_x be s and t
We can think of D_x as being make of arc shortest (in terms of numbers of arcs) x-augmenting paths.
A feasible flow in a network is said to be maximal (does not means maximum) if every (s, t)-directed path contains at least one saturated arc.
For layered algorithm $V_0, V_1, ..., V_L$
Arcs:

- If $t(e) \in V_{i-1}$, $h(e) \in V_i$ and $x_e < u_e$, then add e to A_i with $\hat{u_e} = u_e - x_e$
- If $h(e) \in V_{i-1}$, $t(e) \in V_i$ and $x_e > 0$, then add arc $e' = (h(e), t(e))$ to A_i and define $\hat{u_e} = x_e$

Maximal Flow: If every directed (s,t)-path has at least one saturated arc.
Computing maximal flow is easier than computing maximum flow, since we never need to consider canceling flows on reverse arcs,
Let $\hat{x}$ be a maximal flow on the layered network D_x, we can define new flows in $D(x')$ by

$$x'_e = x_e + \hat{x_e}, \quad \text{If } t(e) \in V_{i-1}, h(e) \in V_i \tag{28.47}$$
$$x'_e = x_e - \hat{x_e}, \quad \text{If } h(e) \in V_{i-1}, t(e) \in V_i \tag{28.48}$$

28.7 Dinic Algorithm

Input: A layered network $(D_x, s, t, \hat{u})$ and a feasible flow x Output: A maximal flow $\hat{x}$ from D_x
Step 1: Set $H \leftarrow D_x$ and $i \leftarrow S$ Step 2: If there is no arc e with $t(e) = i$, goto Step 4, otherwise let e be such an arc Step 3: Set $T(h(e)) \leftarrow i$ and $i \leftarrow h(e)$, if $i = t$ goto Step 5, otherwise goto Step 2. Step 4: If $i = s$, Stop, Otherwise delete i and all incident arcs with H, set $i \leftarrow T(i)$ and goto Step 2 Step 5: Construct the directed path, $s = i_0 e_1 i_1 e_2 ... e_k i_k = t$ where $i_{j-1} = T(i_j), 1 \leq j \leq k$. Set $D = \min\{\hat{u_{e_j}} - \hat{x_{e_j}} : i \leq j \leq k\}$, set $\hat{x_{e_j}} \leftarrow \hat{x_{e_j}} + D, i \leq j \leq k$. Delete from H all saturated arcs on this path, set $i \leftarrow 1$ and goto Step 2.

Algorithm 25 Dinic Algorithm

Ensure: A maximal flow $\hat{x}$ from D_x
Require: A layered network $(D_x, s, t, \hat{u})$ and a feasible flow x
Initialize $H \leftarrow D_x$ and $i \leftarrow S$

Theorem 28.5. *Dinic algorithm runs in $O(|E||V|^2)$*

Proof. Step 1 is $O(|E||V|)$ Step 2 runs Step 1 for $O(|V|)$ times □

Chapter 29

Minimum Cost Flow Problem

29.1 Transshipment Problem

Transshipment Problem (D, b, w) is a linear program of the form

$$\begin{aligned} \min \quad & wx && (29.1)\\ \text{s.t.} \quad & Nx = b && (29.2)\\ & x \geq 0 && (29.3) \end{aligned}$$

Where N is a vertex-arc incident matrix. For a feasible solution to LP to exist, the sum of all bs must be zero. Since the summation of rows of N is zero. The interpretation of the LP is as follows.
The variables are defined on the edges of the digraph and that x_e denote the amount of flow of some commodity from the tail of e to the head of e
Each constraints

$$\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = b_i \tag{29.4}$$

represents consequential of flow of all edges into k vertex that have a demand of $b_i > 0$, or a supply of $b_i < 0$. If $b_i = 0$ we call that vertex a transshipment vertex.

29.2 Network Simplex Method

Lemma 29.1. *Let C_1 and C_2 be distinct cycles in a graph G and let $e \in C_1 \cup C_2$. Then $(C_1 \cup C_2) \setminus e$ contains a cycle.*

Proof. Case 1: $C_1 \cap C_2 = \emptyset$. Trivia.
Case 2: $C_1 \cap C_2 \neq \emptyset$. Let $e \in C_2$ and $f = uv \in C_1 \setminus C_2$. Starting at v traverse C_1 in the direction away from u until the first vertex of C_2, say x. Denote the (v, x)-path as P. Starting at u traverse C_1 in the direction away from v until the first vertex of C_2, say y. Denote the (u, y)-path as Q. C_2 is a cycle, there are two (x, y)-path in C_2. Denote the (x, y)-path without e as R. Then $vPxRyQ^{-1}uf$ is a cycle. □

Theorem 29.2. *Let T be a spanning tree of G. And let $e \in E \setminus T$ then $T + e$ contains a unique cycle C and for any edge $f \in C$, $T + e - f$ is a spanning tree of G*

Let (D, b, w) be a transshipment problem. A feasible solutions x is a **feasible tree solution** if there is a spanning tree T such that $||x|| = \{e \in A, x_e \neq 0\} \subseteq T$.
The strategy of network simplex algorithm is to generate negative cycles, if negative cycle exists, it means the solution can be improved.
For any tree T of D and for $e \in A \setminus T$, it follows from above theorem that $T + e$ contains a unique cycle. Denote that cycle $C(T, e)$ and orient it in the direction of e, define

$$\begin{aligned} w(T, e) = & \sum \{w_e : e \text{ forward in } C(T, e)\} \\ & - \sum \{w_e : e \text{ reverse in } C(T, e)\} \end{aligned} \tag{29.5}$$

We think of $w(T,e)$ as the weight of $C(T,e)$.

29.2.1 Network Simplex Method

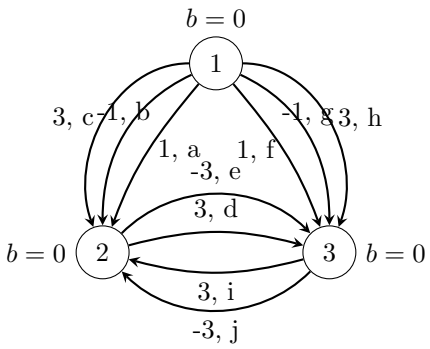
Algorithm 26 Network Simplex Method Algorithm

Ensure: An optimal solution or the conclusion that (D,b,w) is unbounded
Require: A transshipment problem (D,b,w) and a feasible tree solution x containing to a spanning tree T
  while $\exists e \in A \setminus T, w(T,e) < 0$ **do**
    let $e \in A \setminus T$ be such that $w(T,e) < 0$.
    if $C(T,e)$ has no reverse arcs **then**
      Return unboundedness
    else
      Set $\theta = \min\{x_f : f \text{ reverse in } C(T,e)\}$ and set $f = \{f \in C(T,e) : f \text{ reverse in } C(T,e), x_f = \theta\}$
      if f forward in $C(T,e)$ **then**
        $x_f \leftarrow x_f + \theta$
      else
        $x_f \leftarrow x_f - \theta$
      Let $f \in F$ and $T \leftarrow T + e - f$
  Return x as optimal

29.2.2 Example for cycling

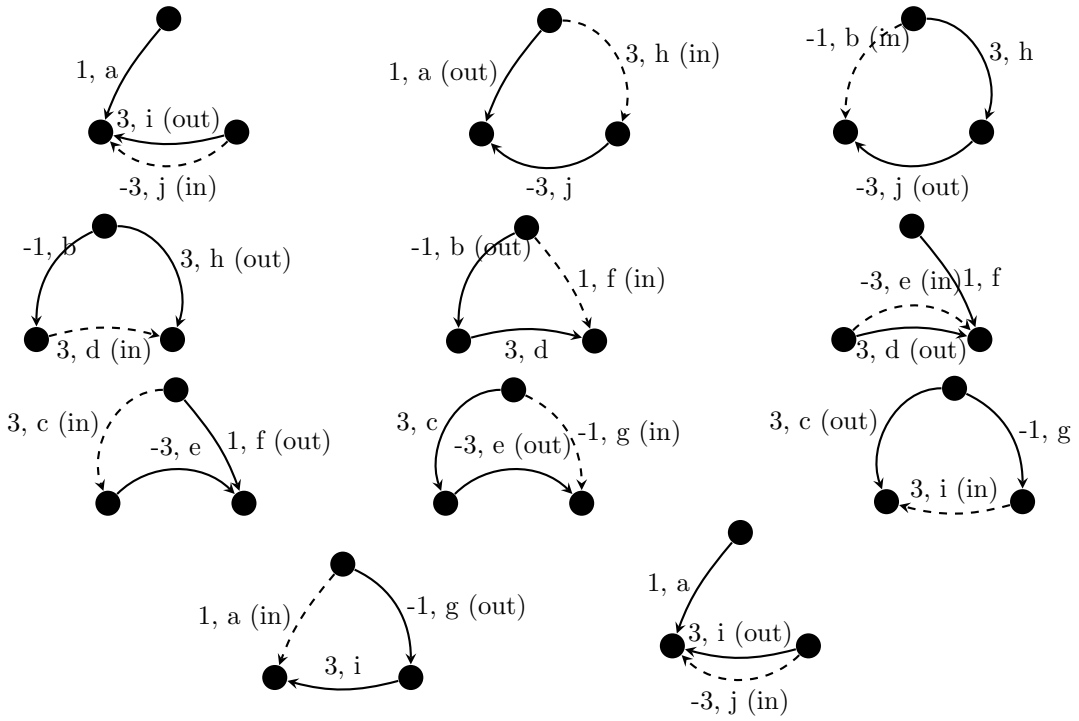
Notice: Similar to Simplex Method in LP, even though in worst case may be inefficient. In most cases it is simple and empirically efficient. Also, similarily, there will be cycling problems.

The following is an example of cycling



Then for the following steps we can detect cycling:
$w(T,j) = w_j - w_i = -3 - 3 = -6$, therefore j in entering basis, i is leaving basis.
$w(T,h) = w_h + w_j - w_a = 3 - 3 - 1 = -1$, therefore h is entering basis, a is leaving basis.
$w(T,b) = w_b - w_j - w_h = -1 + 3 - 3 = -1$, therefore b is entering basis, j is leaving basis.
$w(T,d) = w_d - w_h + w_b = 3 - 3 - 1 = -1$, therefore d is entering basis, h is leaving basis.
$w(T,f) = w_f - w_d - w_b = 1 - 3 + 1 = -1$, therefore f is entering basis, b is leaving basis.
$w(T,e) = w_e - w_d = -3 - 3 = -6$, therefore e is entering basis, d is leaving basis.
$w(T,c) = w_c + w_e - w_f = 3 - 3 - 1 = -1$, therefore c is entering basis, f is leaving basis.
$w(T,g) = w_g - w_e - w_c = -1 + 3 - 3 = -1$, therefore g is entering basis, e is leaving basis.
$w(T,i) = w_i - w_c + w_g = 3 - 3 - 1 = -1$, therefore i is entering basis, c is leaving basis.
$w(T,a) = w_a - w_i - w_g = 1 - 3 + 1 = -1$, therefore a is entering basis, g is leaving basis.

The last graph is the same as the first graph, i.e., cycling detected.



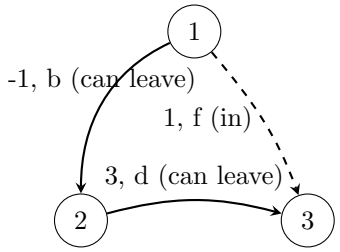
29.2.3 Cycling prevention

To Avoid cycling we will introduce the Modified Network Simplex Method. Let T be a **rooted** spanning tree. Let f be an arc in T, we say f is **away** from the root r if $t(f)$ is the component of $T - f$. Otherwise we say f is **towards** r.
Let x be a feasible tree solution associated with T, then we say T is a **strong feasible tree** if for every arc $f \in T$ with $x_f = 0$ then f is away from $r \in T$.
Modification to NSM:

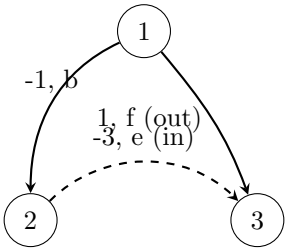
- The algorithm is initialed with a strong feasible tree.
- f in pivot phase is chosen to be the first reverse arc of $C(T, e)$ having $x_f = \theta$. By "first", we mean the first arc encountered in traversing $C(T, e)$ in the direction of e, starting at the vertex i of $C(T, e)$ that minimizes the number of arcs in the unique (r, i)-path in T.

Notice: In the second rule above, r could also be in the cycle, in that case, i is r.

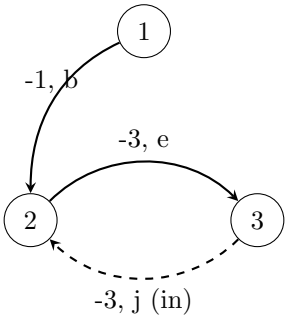
Continue the previous example. Now should how we can avoid cycling:
The first few (four) steps are the same as previous example, starting from
$w(T, f) = w_f - w_d - w_b = 1 - 3 + 1 = -1$. f is entering basis, both b and d can leave the basis, according to



the modified pivot rule, we choose the "first" arc encountered in traversing $C(T, e)$, which is d to leave the basis, instead of b.
$w(T, e) = w_e - w_f + w_b = -5$, e is entering basis, f is leaving basis. Now the only arc to enter basis and maintain

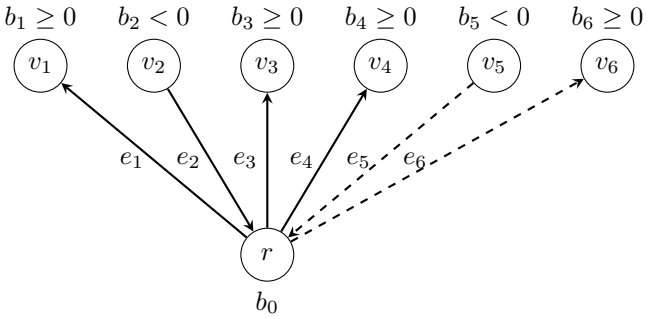


negative w is j.
$w(T, j) = w_j + w_e = -6$, but in $C(T, j)$ there is no reversing arc, therefore we detect unboundedness.



29.2.4 Finding Initial Strong Feasible Tree

Pick a vertex in D to be root r. The tree T has an arc e with the $t(e) = r$ and $h(e) = v$. For each $v \in V \setminus r$ with $b_v \geq 0$ and has an arc e with $h(e) = r$ and $t(e) = v$ for each $v \in V \setminus r$ for which $b_v < 0$. Wherever possible the arcs of T are chosen from A, where an appropriate arc doesn't exist. We create an **artificial arc** and give its weight $|V|(\max\{w_e : e \in A\}) + 1$. This is similar to Big-M method and if optimal solution contains artificial arcs ongoing arc problem is infeasible.
Here is an example after adding artificial arcs:
Where e_5 and e_6 are artificial arcs, the weight of those arcs are $|V|(\max\{w_e : e \in \mathcal{A}\}) + 1$. And the above tree is



a basic feasible solution.
We need to prove that such artificial arc has sufficiently large weight to guarantee

- It will leave the basis, and
- It will not enter the basis again (for this, just delete the artificial arc after it leaves the basis, then it will never enter the basis again)

Proof. Now prove that such arcs will always leave the basis. Before the prove we give some notation.

- Define set E as the set of arcs which is not artificial arc, in the above example, $E = \{e_1, e_2, e_3, e_4\}$.
- Define set A as the set of arcs which are artificial arcs, in the above example, $A = \{e_5, e_6\}$. Noticed that $E \cap A = \emptyset$.

- Define set M as the vertices in the spanning tree that is reachable from r by E, in the above example, $M = \{v_1, v_2, v_3, v_4\}$.
- Define $M' = (V \setminus r) \setminus M$ in the tree that can only be reached from r by A, i.e., artificial arcs, in the above example, $M' = \{v_5, v_6\}$.

Then the initial basic feasible solution is a graph

$$G_0 =< M \cup M' \cup \{r\}, E \cup A > \tag{29.6}$$

Denote the origin graph

$$G =< V, \mathcal{A} > \tag{29.7}$$

Notice that with the artificial arcs, G_0 is not a subgraph of G.
Let $(M \cup \{r\}, M')$ be a cut in the origin graph G. For the vertices in M', one of the following cases will happen:

- case 1: $\sum_{v \in M'} b_v \geq 0$
- case 2: $\sum_{v \in M'} b_v < 0$

For case 1, we claim that at least one of the vertices $v_{M'} \in M'$ with $b_{v_{M'}} \geq 0$ linked by an arc, say f, such that $h(f) = v_{M'}$ and $t(f) = v_M \in M$. Otherwise the balance of flow cannot hold in the origin graph G. Furthermore, denote the artificial arc from r to $v_{M'}$ by $e_{rv_{M'}}$.
Notice that for v_M there is not necessarily be an arc between r and v_M, but there must exists an (r, v_M)-path denoted by P, for M is the set of vertices that reachable from r by arcs in E.
Take that arc f as entering arc to the basis. Then

$$C(T, f) = re_{rv_{M'}} v_{M'} f v_M P r \tag{29.8}$$

For

$$w(T, f) = w_f - w_{e_{rv_{M'}}} + \sum_{e \in P} d_e w_e \tag{29.9}$$

where $d_e = 1$ if w_e is forward in P and $d_e = -1$ otherwise.
Now that $w_{e_{rv_{M'}}} = |V|(\max\{w_e : e \in \mathcal{A}\}) + 1$, it guarantees that

$$\begin{aligned} w(T, f) &= w_f + \sum_{e \in P} d_e w_e - w_{e_{rv_{M'}}} && (29.10)\\ &\leq w_f + \sum_{e \in P} w_e - w_{e_{rv_{M'}}} && (29.11)\\ &\leq \sum_{e \in \mathcal{A}} w_e - w_{e_{rv_{M'}}} && (29.12)\\ &\leq |V|(\max\{w_e : e \in \mathcal{A}\}) - w_{e_{rv_{M'}}} && (29.13)\\ &\leq -1 < 0 && (29.14) \end{aligned}$$

So f can enter the basis, and the artificial variable $e_{rv_{M'}}$ will leave the basis, for it is the most violated reverse arc in the $C(T, f)$. When we put f into the basis, update G_0, such that $M \leftarrow M \cup \{v_{M'}\}$ and $M' \leftarrow M' \setminus \{v_{M'}\}$.
For case 2, it is similar. At least one of the vertices $v_{M'} \in M'$ with $b_{v_{M'}} < 0$ linked by an arc, say $f\prime$, such that $t(f') = v_{M'}$ and $h(f') = v_M \in M$. Otherwise the balance of flow cannot hold in the origin graph G. Furthermore, denote the artificial arc from $v_{M'}$ to r by $e_{v_{M'}r}$.
Similarly we can find a cycle $C(T, f') = rP'v_M f' v_{M'} e_{v_{M'}r} r$. $w(T, f') = w_{f'} - w_{e_{rv_{M'}}} + \sum_{e \in P'} d_e w_e$, where $d_e = 1$ if w_e is forward in $P\prime$ and $d_e = -1$. We can prove $w(T, f') \leq -1 < 0$. That that f' as entering arc to the basis, similarly move $v_{M'}$ form set M' to M.
The above case can be dealt with iteratively until set M' become $\emptyset$, at which stage there is no artificial arc in the basic feasible solution. Which means all the artificial variable can leave the basis. □

Notice: This algorithm can be really bad, its mimic of Simplex Method of LP, which means we can run into exponential operations

29.3 Transshipment Problem and Circulation Problem

Definition 29.3.1. The minimum weight circulation problem is defined as follows:

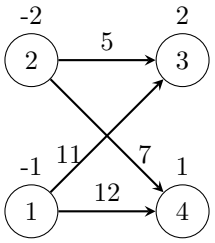
$$\min \quad wx \tag{29.15}$$
$$\text{s.t.} \quad Nx = 0 \tag{29.16}$$
$$l \leq x \leq u \tag{29.17}$$

It turns out that the circulation problem is equivalent with transshipment problem.

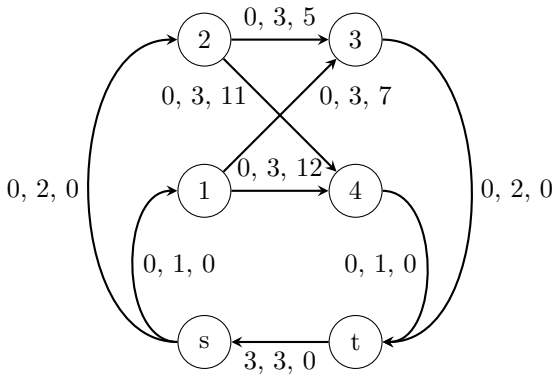
We will show how to transform any transshipment into circulation.
Let (D, b, w) be a transshipment problem and define two new vertices s and t.

- For each supply vertex x add the arc (s, x) to D with $l_x = 0, u_x = -b_x, w_x = 0$.
- Similarly, for each demand vertex x, add the arc (x, t) to D with $l_x = 0, u_x = b_x, w_x = 0$.
- Finally, add an arc (t, s) having $w_{ts} = 0, l_{ts} = u_{ts} = \sum\{b_x : \forall x, x \text{ is demand vertex}\}$.
- Each original arc is given a $l_x = 0, u_x = \sum\{b_x : \forall x, x \text{ is a demand vertex}\}$, w_x remains unchanged.

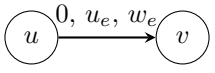
The following is a graph for transshipment problem.



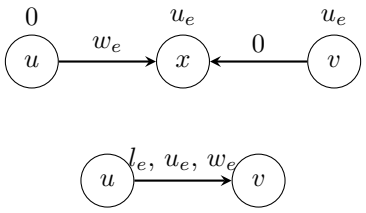
After the above procedures, it is now transformed into a circulation problem.



Then, we will show how to transform any circulation problem into transshipment problem.
If the lower bound of the arc is zero, i.e., $l_e = 0$, then for each such arc (u, v), introduce a vertex in between u and v, replace the arc $e = (u, v)$ by $e_1 = (u, x)$ and $e_2 = (x, v)$. Both arcs are uncapacitated. Let $w_1(u, x) = w(u, v)$ and $w_2(x, v) = 0$ be the new weights for the arcs. Let u_e be the demands of newly added vertex x and add u_e to the supplies of vertex v (in v the supplies is the summation from all arcs that go to v).

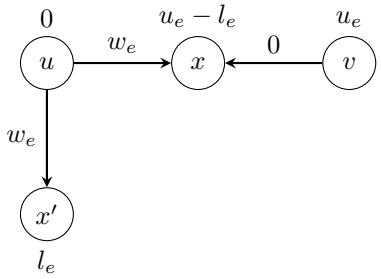


Will be transform into
If the lower bound of the arc is not zero, i.e., $l_e \neq 0$, then for each such arc (u, v), introduce two new vertices, one vertex is in between u and v, similar to the previous case, the difference is the demands of this new vertex is



$u_e - l_e$, the others stays the same. Then add the other vertex, this vertex, denoted as x' is added along with an arc between u and x', the weight of this arc is $w(u, v)$, the demands on this new vertex is l_e.

Will be transform into



Perform the above procedures to all the arcs in a circulation problem. In $O(E)$ (polynomially times) transformation, such problem can be transformed into transshipment problem.

29.4 Out-of-Kilter algorithm

This algorithm is a Primal-dual method and is applied to the minimum weight circulation problem.

For LP optimality conditions we need primal feasibility, dual feasibility and complementary slackness, i.e., KKT conditions. Primal and dual feasibility are obvious so we need to show complementary slackness through following theorem.

Theorem 29.3. *Let x be a feasible circulation flow for (D, l, u, w). And suppose there exists a real value vector $\{y_i : i \in V\}$ which we called **vertex-numbers**. For all edges $e \in A$*

$$y_{h(e)} - y_{t(e)} > w_e \text{ implies } x_e = u_e \tag{29.18}$$
$$y_{h(e)} - y_{t(e)} < w_e \text{ implies } x_e = l_e \tag{29.19}$$

Then x is optimal to the circulation problem.

Proof. For each $e \in A$ define

$$\gamma_e = \max\{y_{h(e)} - y_{t(e)} - w_e, 0\} \tag{29.20}$$
$$\mu_e = \max\{w_e - y_{h(e)} + y_{t(e)}, 0\} \tag{29.21}$$

Then

$$\gamma_e - \mu_e = y_{h(e)} - y_{t(e)} - w_e \tag{29.22}$$

Furthermore

$$\sum_{e\in A}(\mu_e l_e - \gamma_e u_e) \tag{29.23}$$

$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e) + \sum_{i\in V} y_i(\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e) \tag{29.24}$$

$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e + x_e(y_{h(e)} - y_{t(e)})) \tag{29.25}$$

$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e + x_e(\gamma_e - \mu_e + w_e)) \tag{29.26}$$

$$= \sum_{e\in A}(\gamma_e(x_e - u_e) + \mu_e(l_e - x_e) + x_e w_e) \tag{29.27}$$

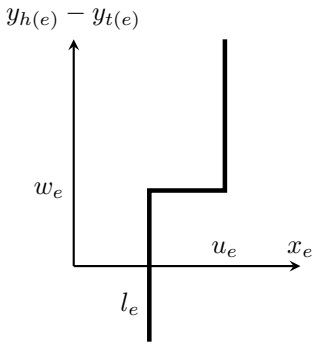
$$\leq \sum_{e\in A} x_e w_e \tag{29.28}$$

The last inequality will be satisfied as equality iff the first two hold. □

The following is the formulation of circulation problem

$$\begin{aligned}
&(\text{P}) \quad \min \quad && wx && (29.29)\\
&\qquad \text{s.t.} \quad && Nx = 0 \quad y && (29.30)\\
&\qquad && x \geq l \quad z^l && (29.31)\\
&\qquad && -x \leq -u \quad z^u && (29.32)\\
&(\text{D}) \quad \max \quad && lz^l - uz^u && (29.33)\\
&\qquad (\text{s.t.}) \quad && yN^{-1} + z^l - z^u \leq w && (29.34)\\
&\qquad && y \quad free && (29.35)\\
&\qquad && z^l, z^u \geq 0 && (29.36)\\
&\qquad (\text{CS}) \quad && y_{h(e)} - y_{t(e)} > w_e \Rightarrow x_e = u_e && (29.37)\\
&\qquad && y_{h(e)} - y_{t(e)} < w_e \Rightarrow x_e = l_e && (29.38)
\end{aligned}$$

There is an alternative way of circulation optimality for a circulation problem. We define a **kilter-diagram** as follows.
For every edge construct the following:

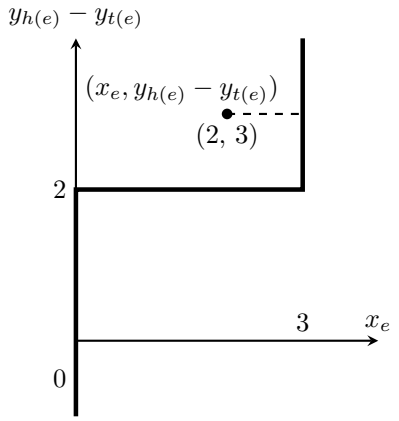


For each point $(x_e, y_{h(e)} - y_{t(e)})$ we define a **kilter-number** k_e, be the minimum positive distance change in x_e required to put in on the kilter line.

Example. For edge $e : w_e = 2, l_e = 0, u_e = 3$, assume $x_e = 2, y_{h(e)} - y_{t(e)} = 3$, then $k_e = 1$

Lemma 29.4. *If for every circulation x and vertex number y we have $\sum_{e\in A} k_e = 0$, then x is optimal.*

Proof. Since k_e is a nonnegative number, then the only way that $\sum_{e\in A} k_e = 0$ is $k_e = 0, \forall e \in A$, which means $\forall e \in A, l_e \leq x_e \leq u_e$. Furthermore, the complementary slackness are satisfied. □



General idea of algorithm follows. Suppose we are given a circulation x and vertex-numbers y (we do not require feasibility). Usually we pick $x = 0, y = 0$. If every edge is in kilter-line then we are optimal.
Otherwise there is at least one edge e^* that is out-of kilter. The algorithm consist of two phases, one called **flow-change** phase (horizontally), then other **number-change** phase (vertically).
In the flow-change phase, we want to find a new circulation for an out-of-kilter edge e^* say $\hat{e}$ such that we reduce the kilter number k_{e^*}, without increasing any other kilter number for other edges.
To do this, denote the edges of e^* to be s and t, where such that k_{e^*} will be decreased by increasing the flow from s to t on e^*.
If $e^* = (s, t)$ this will accomplished by increasing x_{e^*} and if $e = (t, s)$ it is accomplished by decreasing x_{e^*}.
To do this we look for an (s, t)-path p of the following edges.

- If e is forward in p, then increasing x_e does not increase k_e and
- If e is reversed in p, then decreasing x_e dose not increase k_e

In terms of kilter diagram, an arc satisfies "forward" if it is forward and in left side of kilter line, and it satisfies "reversed" if it is reverse and in right side of kilter line.
Suppose we can not find such a path. From s to t, let x be the vertices that can decrease by an augmenting path. Then either we can change the vertex numbers y so that $\sum_{e \in A} k_e$ does not increase but x does, or we can show that problem is infeasible.
INPUT a minimum circulation problem (D, l, u, w) a circulation x and vertex-numbers y
OUTPUT conclusion that (D, l, u, w) is infeasible or an minimum weighted flow.
Step 1: If every arc is in kilter ($k_e = 0, \forall e \in A$). Stop with x is optimal. Otherwise let e^* be an out-of-kilter arc. If increasing x_{e^*} decreases k_{e^*} set $s = h(e^*)$ and $t(e^*)$ otherwise set $s = t(e^*)$ and $t = h(e^*)$
Step 2: If there exists an (s, t) augmenting path p then goto Step 3, otherwise goto Step 4.
STEP 3: Set $y_e = y_{h(e)} - y_{t(e)}, e \in A$ Set $\Delta_1 = \min\{u_e - x_e : e \text{ is forward and } y_e \geq w_e\}$ Set $\Delta_2 = \min\{l_e - x_e : e \text{ is forward and } y_e < w_e\}$ Set $\Delta_3 = \min\{x_e - l_e : e \text{ is reverse and } y_e \leq w_e\}$ Set $\Delta_4 = \min\{x_e - u_e : e \text{ is reverse and } y_e > w_e\}$ $\Delta = \min\{\Delta_i, i = 1, 2, 3, 4\}$
Increase x_e by Δ on each forward arc in p, decrease x_e by Δ on each reverse arc in p.
If $e^* = (s, t)$ decrease x_{e^*} by Δ, otherwise increase x_{e^*} by Δ
If $k_{e^*} > 0$ goto Step 2. otherwise goto Step 1.
Step 4: Let X be the set of vertices reachable from s by augmenting paths, then $t \notin X$, if every arc e with $h(e) \in X$ has $x_e \leq l_e$ and every arc e with $t(e) \in X$ has $x_e \geq u_e$, and at least one of the above inequality is strict, then Stop with problem infeasible
Otherwise set $\delta_1 = \min\{w_e - y_e : t(e) \in X, y_e < w_e, x_e \leq u_e \neq l_e\}$ $\delta_2 = \min\{y_e - w_e : h(e) \in X, y_e > w_e, x_e \geq u_e \neq l_e\}$ $\delta = \min\{\delta_1, \delta_2\}$
Set $y_i = y_i + \delta$ for $i \notin X$
If $k_{e^*} > 0$, goto Step 2, otherwise goto Step 1.
Out-of-kilter takes $O(|E||V|K)$ where $K = \sum_{e \in A} k_e$. However, there is an algorithm called **scaling algorithm** that uses out-of-kilter as subroutine that runs in $O(R|E|^2|V|)$ where $R = \lceil \max\{\log_2 u_e : e \in A\} \rceil$

29.5 Complexity of Different Minimum Weighted Flow Algorithms

Let arc capacities between 1 and U, costs between $-C$ and C

Year	Discoverer	Method	Big O
1951	Dantzig	Network Simplex	$O(E^2V^2U)$
1960	Minty, Fulkerson	Out-of-Kilter	$O(EVU)$
1958	Jewell	Successive Shortest Path	$O(EVU)$
1962	Ford-Fulkerson	Primal Dual	$O(EV^2U)$
1967	Klein	Cycle Canceling	$O(E^2CU)$
1972	Edmonds-Karp, Dinitz	Capacity Scaling	$O(E^2 \log U)$
1973	Dinitz-Gabow	Improved Capacity Scaling	$O(EV \log U)$
1980	Rock, Bland-Jensen	Cost Scaling	$O(EV^2 \log C)$
1985	Tardos	ϵ-optimality	poly(E, V)
1988	Orlin	Enhanced Capacity Scaling	$O(E^2)$

Chapter 30

Social Network Analysis

Part V

Heuristic and Metaheuristic Optimization

Chapter 31

Search Algorithm

Special Topic: Vehicle Routing Problem

Chapter 32

The Traveling Salesman Problem

32.1 Formulations

In this section, we are going to compare between different formulations of TSP. Generally speaking, let $G = (V, A)$ be a graph where V is a set of n vertices, and A is a set of arcs (or edges). Let $C = c_{ij}$ be a cost (distance) matrix associated with A. The TSP consists of determining a minimum cost (distance) Hamiltonian circle (or cycle) that visits each vertex once and only once. If for all $i, j \in V, c_{ij} = c_{ji}$, then the TSP is symmetrical, otherwise is asymmetrical.
Define the decision variable x_{ij} as the following

$$x_{ij} = \begin{cases} 1, & \text{if goes from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}, \quad (i,j) \in A \tag{32.1}$$

The objective function will be

$$\min \quad \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{32.2}$$

32.1.1 Dantzig-Fulkerson-Johnson (DFJ) Formulation

The first famous formulations for TSP is the **Dantzig-Fulkerson-Johnson (DFJ) formulation**:

$$\sum_{j\in V,(i,j)\in A} x_{ij} = 1, \quad \forall i \in V \tag{32.3}$$

$$\sum_{i\in V,(i,j)\in A} x_{ij} = 1, \quad \forall j \in V \tag{32.4}$$

$$\sum_{j\notin S, i\in S,(i,j)\in A} x_{ij} \geq 1, \quad \forall S \subset V, 2 \leq |S| \leq n-1 \tag{32.5}$$

In the formulation, constraints (32.3) and constraints (32.4) are degree constraints, which specify that every vertex is entered exactly once. Constraints (32.5) is the sub-tour constraints, they prohibit the formation of sub-tours. S is a non-empty subset of V, and has at least 2 vertices. 32.5 can be replaced by

$$\sum_{i,j\in S,(i,j)\in A} x_{ij} \leq |S| - 1, \quad \forall S \subset V, 2 \leq |S| \leq n-1 \tag{32.6}$$

If we list all sub-tour constraints in DFJ, there will be $O(2^n)$ constraints and $O(n^2)$ binary variables. The exponential number of constraints makes it impractical to solve directly. Instead, lazy constraints are usually implemented for the sub-tour elimination constraints (32.5 or 32.6).

32.1.2 Miller-Tucker-Zemlin (MTZ) Formulation

We can also formulate TSP using sequential formulations, namely, **Miller-Tucker-Zemlin (MTZ) formulation**. In the MTZ formulation, the degree constraints (32.3 and 32.4) are the same as in DFJ formulation.

Define a new set of integer decision variables u_i, u_i defined as the sequence in which node i is visited, $u_1 = 1$.
The sub-tour constraints (32.5 or 32.6) are replaced by the following:

$$u_i - u_j + (n-1)x_{ij} \leq n-2, \quad i,j = 2,\cdots,n \in V, (i,j) \in A \tag{32.7}$$

$$1 \leq u_i \leq n-1, \quad i \in 2,\cdots,n \in V \tag{32.8}$$

In MTZ formulation, there are $O(n^2)$ constraints, $O(n^2)$ binary variables, and $O(n)$ continuous variables.

32.1.3 Quadratic Formulation (QAP)

In this section, we are going to go over a TSP formulation are super bad. However, it still has some value for further study.
The idea is to transform TSP into an assignment problem. Assuming we have n boxes, which represents n steps in the path. Define x_{ij} as

$$x_{ij} = \begin{cases} 1, & \text{Vertex } i \text{ is assigned to box } j \\ 0, & \text{Otherwise} \end{cases} \tag{32.9}$$

The constraints are simple as an assignment problem as following

$$\sum_{j=1}^{n} x_{ij} = 1, \quad \forall i \in V \tag{32.10}$$

$$\sum_{i \in V}^{n} x_{ij} = 1, \quad j = 1,\ldots,n \tag{32.11}$$

However, the tricky part is in the objective function

$$\min \quad \sum_{i \in V} \sum_{j \in V \setminus \{i\}} \sum_{k=1}^{n-1} c_{ij} x_{ik} x_{j,k+1} + \sum_{i \in V} \sum_{j \in V \setminus \{i\}} c_{ij} x_{in} x_{j1} \tag{32.12}$$

Notice that the objective function is not linear function, with the multiplications of decision variables. Now we are going to linearize them. The linearized version is as following

$$\min \quad \sum_{i \in V} \sum_{j \in V \setminus \{i\}} \sum_{k=1}^{n-1} c_{ij} w_{ij}^{k} + \sum_{i \in V} \sum_{j \in V \setminus \{i\}} c_{ij} w_{ij}^{n} \tag{32.13}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} x_{ij} = 1, \quad \forall i \in V \tag{32.14}$$

$$\sum_{i \in V}^{n} x_{ij} = 1, \quad j = 1,\ldots,n \tag{32.15}$$

$$w_{ij}^{k} \geq x_{ik} + x_{j,k+1} - 1, \quad i \in V, j \in V \setminus \{i\}, k = 1,\cdots,n-1 \tag{32.16}$$

$$w_{ij}^{k} \geq x_{ik} + x_{j1} - 1, \quad i \in V, j \in V \setminus \{i\}, k = n \tag{32.17}$$

$$w_{ij}^{k} \in \{0,1\}, \quad i \in V, j \in V \setminus \{i\}, k = 1,\ldots,n \tag{32.18}$$

$$x_{ij} \in \{0,1\}, \quad i \in V, j \in V \setminus \{i\} \tag{32.19}$$

We can prove that this is very very bad.

32.1.4 Flow Based Formulations

In this section, flow based formulations are discussed, which includes **Single Commodity Flow**, **Two Commodity Flow** and **Multi-Commodity Flow**. In these formulations, continuous variables are introduced to represent the flow on the arcs.
In Single Commodity Flow formulation, define y_{ij} as the flow in an arc $(i,j) \in A$.

Degree constraints 32.3 and 32.4 are retained. The following constraints are introduced:

$$y_{ij} \leq (n-1)x_{ij}, \quad \forall i,j \in V, (i,j) \in A \tag{32.20}$$

$$\sum_{j \in V,(1,j) \in A} y_{1j} = n-1 \tag{32.21}$$

$$\sum_{i \in V,(i,j) \in A} y_{ij} - \sum_{k \in V,(j,k) \in A} y_{jk} = 1, \quad \forall j \in V \setminus \{1\} \tag{32.22}$$

$$\tag{32.23}$$

Constraints (32.20) can be tighten by the following:

$$y_{ij} \leq (n-1)x_{ij}, \quad i = 1, j \in V \setminus \{1\}, (i,j) \in A \tag{32.24}$$

$$y_{ij} \leq (n-2)x_{ij}, \quad i,j \in V \setminus \{1\}, (i,j) \in A \tag{32.25}$$

$$\tag{32.26}$$

In SCM formulation, there are $O(n^2)$ constraints, $O(n^2)$ binary variables and $O(n^2)$ continuous variables.
In Two Commodity Flow formulation, define y_{ij} as the flow in an arc $(i,j) \in A$, for commodity type 1, and define z_{ij} as the flow in an arc $(i,j) \in A$, for commodity type 2.
Besides degree constraints, the other constraints are as following

$$y_{ij} + z_{ij} = (n-1)x_{ij}, \quad \forall i,j \in V, (i,j) \in A \tag{32.27}$$

$$\sum_{j \in V \setminus \{1\}} (y_{1j} - y_{j1}) = n-1, \quad (1,j) \in A \tag{32.28}$$

$$\sum_{j \in V} (y_{ij} - y_{ji}) = 1, \quad \forall i \in V \setminus \{1\}, (i,j) \in A \tag{32.29}$$

$$\sum_{j \in V \setminus \{1\}} (z_{1j} - z_{j1}) = 1-n, \quad (1,j) \in A \tag{32.30}$$

$$\sum_{j \in V} (z_{ij} - z_{ji}) = -1, \quad \forall i \in V \setminus \{1\}, (i,j) \in A \tag{32.31}$$

$$\sum_{j \in V} (y_{ij} + z_{ij}) = n-1, \quad \forall i \in V \tag{32.32}$$

$$\tag{32.33}$$

In TCM formulation, constraints (32.27) only allow flow in an arc if present. Constraints (32.28) and (32.29) forces $(n-1)$ units of commodity type 1 to flow in at node 1 and 1 unit to flow out at every other nodes. Constraints (32.30) and (32.31) are similar, those forces $(n-1)$ units of commodity type 2 to flow out at node 1 and 1 unit to flow in at every other nodes. Constraints (32.32) forces exactly $(n-1)$ units of combined commodity in each arc.
In TCM formulation, there are $O(n^2)$ constraints, $O(n^2)$ binary variables and $O(n^2)$ continuous variables.
The SCM and the TCM can be generalized into **Multi-Commodity Flow formulation**. As usual, degree constraints are retained. The following continuous variables are introduced. Define y_{ij}^k as the flow of commodity type k in arc $(i,j) \in A$.
The other constraints are

$$y_{ij}^k \leq x_{ij}, \quad \forall i,j,k \in N, k \neq 1 \tag{32.34}$$

$$\sum_{i \in V} y_{1i}^k = 1, \quad \forall k \in V \setminus \{1\} \tag{32.35}$$

$$\sum_{i \in V} y_{i1}^k = 0, \quad \forall k \in V \setminus \{1\} \tag{32.36}$$

$$\sum_{i \in V} y_{ik}^k = 1, \quad \forall k \in V \setminus \{1\} \tag{32.37}$$

$$\sum_{j \in V} y_{kj}^k = 0, \quad \forall k \in V \setminus \{1\} \tag{32.38}$$

$$\sum_{i \in V} y_{ij}^k - \sum_{i \in V} y_{ji}^k = 0, \quad \forall j,k \in V \setminus \{1\}, j \neq k \tag{32.39}$$

Constraints (32.34) only allow flow in an arc which is present. Constraints (32.35) forces exactly one unit of each type of commodity to flow in at node 1. Constraints (32.36) prevent any commodity flow out at node 1.Constraints (32.37), and Constraints (32.38), forces exactly one unit of type k commodity to flow out, and in, at every node except node 1. Constraints (32.39) forces balance of all types of commodities at every node except node 1.

This formulation has $O(n^3)$ constraints, $O(n^2)$ binary variables, and $O(n^3)$ continuous variables.

32.1.5 Shortest Path Formulation

A graph for timed staged shortest path

In this section, we are going to introduce another form of formulation with different definition of decision variable and objective function.

Assuming for a completed graph $G = (V, A)$. Define x_{ij}^t as the following

$$x_{ij}^t = \begin{cases} 1, & \text{If path crosses arc } (i,t) \text{ and } (j,t+1) \\ 0, & \text{Otherwise} \end{cases} \quad , \quad i \in V, j \in V \setminus \{i\}, t = 1, \cdots, n \tag{32.40}$$

The objective function will be

$$\min \quad \sum_{i \in V} \sum_{j \in V \setminus \{i\}} c_{ij} \sum_{t=1}^{n} x_{ij}^t \tag{32.41}$$

The constraints are as following

$$\sum_{j \in V \setminus \{1\}} x_{1j}^1 = 1 \tag{32.42}$$

$$\sum_{j \in V \setminus \{1,i\}} x_{ij}^2 - x_{1i}^1 = 0, \quad \forall i \in V \setminus \{1\} \tag{32.43}$$

$$\sum_{j \in V \setminus \{1,i\}} x_{ij}^t - \sum_{j \in V \setminus \{1,i\}} x_{ji}^{t-1} = 0, \quad \forall i \in V \setminus \{1\}, t \in \{2, \ldots, n-1\} \tag{32.44}$$

$$x_{i1}^n - \sum_{j \in V \setminus \{1,i\}} x_{ji}^{n-1} = 0, \quad \forall i \in V \setminus \{1\} \tag{32.45}$$

$$\sum_{i \in V \setminus \{1\}} x_{i1}^n = 1 \tag{32.46}$$

$$\sum_{t=2}^{n-1} \sum_{j \in V \setminus \{1,i\}} x_{ij}^t + x_{i1}^n \leq 1, \quad \forall i \in V \setminus \{1\} \tag{32.47}$$

Notice that constraint (32.47) can be replaced by

$$x_{1i}^1 + \sum_{t=2}^{n-1} \sum_{j \in V \setminus \{1,i\}} x_{ji}^t \leq 1, \quad \forall i \in V \setminus \{1\} \tag{32.48}$$

32.2 NP Completeness of TSP

32.2.1 Proof of TSP $\in$ NPC

32.2.2 Polynomially Solvable Special Cases of TSP

32.3 Lower Bounds of TSP

32.3.1 The Assignment Lower Bound

32.3.2 The Minimum Spanning Tree (Arborescence) Bound

32.3.3 The 2-match Problem

32.3.4 Held & Karp Bound (Lagrangian Relaxation)

In this section, we will solve the Dantzig-Fulkerson-Johnson formulation using Lagrangian Relaxation. The bound found by this method is also known as Held & Karp Bound. In the Held & Karp relaxation, the degree constraints are relaxed, as a result, we require our solution to be connected and to contain n edges, but it might not have exactly two edges incident to each vertex.
The feasible solution to the Lagrangian relaxation is called **1-tree**, which

- Have a spanning tree on nodes $\{2, 3, \cdots, n\}$
- Two edges incident to node 1

32.4 Constructive Heuristic

32.4.1 Nearest Neighborhood Algorithm

32.4.2 Insertion Algorithm

32.4.3 Sweep Algorithm

32.4.4 Christofides Algorithm

32.5 Local Search Heuristic

32.5.1 Lin-Kernighan Algorithm

The general idea of Lin-Kernighan Algorithm is based on a substantial generalization of the interchange transformation.

32.6 Metaheuristic

32.6.1 Simulated Annealing

32.6.2 Genetic Algorithm

Chapter 33

The Vehicle Routing Problem

33.1 The Family of VRP

33.2 Heuristic for VRP

33.2.1 CW Saving

Chapter 34

The Capacitate Vehicle Routing Problem

34.1 Formulations

In this section, we present four important formulations for the CVRP, the problem statement is as follows. The transportation requests consist of the distribution of goods from a single depot, denote as node 0, to a given set of n other locations, or customers, $N = \{1, 2, \cdots, n\}$. The amount that has to be delivered to customer $i \in N$ is called demand, which is given by a scalar $q_i \geq 0$. A fleet of vehicle $K = \{1, 2, \cdots, |K|\}$ is assumed to be homogeneous with the same capacity as $Q > 0$ and identical operation cost. A vehicle that service a subset $S \subseteq N$ starts at the depot, move once to each of the customers in S, and finally returns to the depot. A vehicle moving from i to j incurs the travel cost c_{ij}. Let $V = \{0\} \cup N$ be the set of nodes, for convenience, we usually define node 0 as depot and an additional node $|N| + 1$ as a copy of depot for the returning of vehicle. Set E is the set of all available links between nodes. The CVRP is then defined on graph $G = (V, E)$. We further denote $\delta^+(i) \in V \setminus \{i\}$ as the set of nodes that can be visited from node i, and denote $\delta^-(i) \in V \setminus \{i\}$ as the set of nodes that can visit node i.

34.1.1 Golden et al. 1977

Golden et al. 1997 proposed a 3-indexed formulation for solving CVRP. The resulting formulation involves $O(n^2T)$ variables and requires an exponential number of subtour elimination constraints.

Decision variables are

$$x_{ij}^k = \begin{cases} 1, & \text{If vehicle } k \text{ travels through arc } (i,j) \\ 0, & \text{Otherwise} \end{cases} \qquad (i,j) \in E, k \in \{1, 2, \cdots, T\}$$

where T is the maximum number of vehicle.

The objective function is

$$\min \sum_{k=1}^{T} \sum_{(i,j) \in E} c_{ij} x_{ij}^k$$

The constraints are as following

$$\sum_{k=1}^{T} \sum_{j \in \delta^+(i)} x_{ij}^k = 1, \quad i \in V \setminus \{0\} \tag{34.1}$$

$$\sum_{j \in \delta^+(i)} x_{ij}^k = \sum_{j \in \delta^-(i)} x_{ji}^k, \quad \forall i \in V \setminus \{0\}, k = \{1, 2, \cdots, T\} \tag{34.2}$$

$$\sum_{i \in \delta^+(0)} x_{0i}^k \leq 1, \quad k \in \{1, 2, \ldots, T\} \tag{34.3}$$

$$\sum_{i \in \delta^-(0)} x_{i0}^k \leq 1, \quad k \in \{1, 2, \ldots, T\} \tag{34.4}$$

$$\sum_{i \in V} d_i \sum_{j \in \delta^+(i)} x_{ij}^k \leq Q, \quad k \in \{1, 2, \ldots, T\} \tag{34.5}$$

$$\sum_{(i,j) \in E(S)} x_{ij}^k \leq |S| - 1, \quad \forall S \subset V \setminus \{0\}, k \in \{1, 2, \ldots, T\} \tag{34.6}$$

34.1.2 Two-index Flow Formulation,

Chapter 35

The Vehicle Routing Problem with Time Windows

Chapter 36

Pickup-and-Delivery Problem

36.1 Problem Formulation

Let N be the set of transportation requests. For each transportation request $i \in N$, a load of size $\bar{q_i} \in \mathbb{N}$ has to be transported from a set of origins N_i^+ to a set of destinations N_i^-. Each load is subdivided as follows

$$\bar{q_i} = \sum_{j \in N_i^+} q_j = - \sum_{j \in N_i^-} q_j \tag{36.1}$$

Define $N^+ = \cup_{i \in N} N_i^+$ as the set of all origins and $N^- = \cup_{i \in N} N_i^-$ as the set of all destinations. Let $V = N^+ \cup N^-$. Furthermore, the M be the set of vehicles. Each vehicle $k \in M$ has a capacity $Q_k \in \mathbb{N}$, a start location k^+, and an end location k^-. Define $M^+ = \{k^+ | k \in M\}$ as the set of start locations and $M^- = \{k^- | k \in M\}$ as the set of end locations. Let $W = M^+ \cup M^-$.
For all $i, j \in V \cup W$ let d_{ij} denote the travel distance, t_{ij} denote the travel time, and c_{ij} denote the travel cost. Note that the dwell time at origins and destinations can be easily incorporated in the travel time and therefore will not be considered explicitly.

Definition 36.1.1. A pickup and delivery route R_k for vehicle k is a directed route through a subset $V_k \in V$ such that:

- R_k starts in k^+
- $\forall i \in N, (N_i^+ \cup N_i^-) \cap V_k = \emptyset$ or $N_i^+ \cup N_i^-$
- If $N_i^+ \cup N_i^- \subseteq V_k$, then all locations in N_i^+ are visited before locations in N_i^-
- Vehicle k visits each location in V_k exactly once
- The vehicle load never exceeds Q_k
- R_k ends in k^-

Definition 36.1.2. A pickup and delivery plan is a set of routes $\mathcal{R} = \{R_k | k \in M\}$ such that:

- R_k is the pickup and delivery route for vehicle k, for each $k \in M$
- $\{V_k | k \in M\}$ is a partition of V,

Here are the special cases of the General Pickup and Delivery Problem

Example. The pickup and delivery problem, where $|W| = 1$, $|N_i^+| = |N_i^-| = 1, \forall i \in N$. In this case we define i^+ as the unique element of $|N_i^+|$ and i^- as the unique element of $|N_i^-|$.

Example. The dial-a-ride problem, where $|W| = 1$ and $|N_i^+| = |N_i^-| = 1, \bar{q_i} = 1, \forall i \in N$

Example. The vehicle routing problem, where $|W| = 1$, $|N_i^+| = |N_i^-| = 1 \forall i \in N$ and $N^+ = W$ or $N^- = W$.

Notice: Generally speaking, we usually have problem with $|N_i^+| = |N_i^-| = 1$. In the cases where $|N_i^+| > 1$ or $|N_i^-| > 1$, the transportation requests can be decomposed into several independent requests with $|N_i^+| = |N_i^-| = 1$, unless it has to be served by the same vehicle.

Notice: We are not aware of any real-life applications where both $|N_i^+| > 1$ and $|N_i^-| > 1$ at the same time yet.

The following is the formulation for General Pickup and Delivery Problem.

Table 36.1: Decision variable notation

z_i^k	For $i \in N, k \in M$ Equals to 1 if transportation request i is assigned to vehicle k and 0 otherwise.
x_{ij}^k	For $(i,j) \in (V \times V) \cup \{(k^+, j) \vert j \in V\} \cup \{(j, k^-) \vert j \in V\}, k \in$ Equals to 1 if vehicle k travels from location i to location j and 0 otherwise.
D_i	For $i \in V \cup W$, specifying the departure time at vertex i.
y_i	For $i \in V \cup W$, specifying the load of vehicle arriving at vertex i. Define $q_{k^+} = 0, \forall k \in M$.

$$\min \quad f(x) \tag{36.2}$$

$$\text{s.t.} \quad \sum_{k \in M} z_i^k = 1 \qquad \forall i \in N \tag{36.3}$$

$$\sum_{j \in V \cup W} x_{lj}^k = z_i^k \qquad \forall i \in N, l \in N_i^+ \cup N_i^-, k \in M \tag{36.4}$$

$$\sum_{j \in V \cup W} x_{jl}^k = z_i^k \qquad \forall i \in N, l \in N_i^+ \cup N_i^-, k \in M \tag{36.5}$$

$$\sum_{j \in V \cup \{k^-\}} x_{k^+ j}^k = 1 \qquad \forall k \in M \tag{36.6}$$

$$\sum_{j \in V \cup \{k^+\}} x_{ik^-}^k = 1 \qquad \forall k \in M \tag{36.7}$$

$$D_{k^+} = 0 \qquad \forall k \in M \tag{36.8}$$

$$D_p \leq D_q \qquad \forall i \in N, p \in N_i^+, q \in N_i^- \tag{36.9}$$

$$D_i + t_{ij} \leq D_j + M(1 - x_{ij}^k) \qquad \forall i, j \in V \cup W, k \in M \tag{36.10}$$

$$y_{k^+} = 0 \qquad \forall k \in M \tag{36.11}$$

$$y_l \leq \sum_{k \in M} Q_k z_i^k \qquad \forall i \in N, l \in N_i^+ \cup N_i^- \tag{36.12}$$

$$y_j \geq y_l + q_i x_{lj}^k - M(1 - x_{lj}^x) \qquad \forall i \in N, l \in N_i^+, j \in V \cup \{k^+\}, l \neq j \tag{36.13}$$

$$y_j \leq y_l + q_i x_{lj}^k + M(1 - x_{lj}^x) \qquad \forall i \in N, l \in N_i^+, j \in V \cup \{k^+\}, l \neq j \tag{36.14}$$

$$y_j \geq y_l - q_i x_{lj}^k - M(1 - x_{lj}^x) \qquad \forall i \in N, l \in N_i^-, j \in V \cup \{k^-\}, l \neq j \tag{36.15}$$

$$y_j \leq y_l - q_i x_{lj}^k + M(1 - x_{lj}^x) \qquad \forall i \in N, l \in N_i^-, j \in V \cup \{k^-\}, l \neq j \tag{36.16}$$

$$x_{ij}^k \in \{0, 1\} \qquad \forall i, j \in V \cup W, k \in M \tag{36.17}$$

$$z_i^k \in \{0, 1\} \qquad \forall i \in N, k \in M \tag{36.18}$$

$$D_i \geq 0 \qquad \forall i \in V \cup W \tag{36.19}$$

$$y_i \geq 0 \qquad \forall i \in V \cup W \tag{36.20}$$

Explanation of constraints:

- Constraint (36.3) - Each transportation request is assigned to exactly one vehicle.

- Constraints (36.4), (36.5) - A vehicle only enters or leaves a location l if it is an origin or a destination of a transportation request assigned to that vehicle.
- Constraints (36.6), (36.7) - Make sure the vehicle is leaving from /arriving at the correct place.
- Constraint (36.8) - Initial starting time for all vehicles is 0.
- Constraint (36.9) - For each item to be delivered, it should be picked up before delivery
- Constraint (36.10) - Traveling distance from location i to j
- Constraint (36.11) - All vehicles leaving the initial location with no loading.
- Constraint (36.12) - Load capacity limit for vehicles.
- Constraints (36.13), (36.14), (36.15), (36.16) - Load / unload item when arriving pickup / delivery location
- Constraints (36.17), (36.18), (36.19), (36.20) - Binary variable definition and non-negativity constraints.

36.2 Heuristic Method

Chapter 37

Stochastic Vehicle Routing Problem

Chapter 38

Dynamic Vehicle Routing Problem

Part VI

Nonlinear Programming

Chapter 39

Optimality Conditions and Duality

Chapter 40

Unconstrained Optimization

Chapter 41

Quadratic Programming

Chapter 42

Penalty and Barrier Functions

Part VII

Algorithms

Chapter 43

General Paradigms

This chapter we are going to discussion three types of strategies for algorithm design, greedy algorithm, divide-and-conquer, and dynamic programming.

- Greedy algorithm
  - Make a greedy choice
  - Prove that the greedy choice is safe
  - Reduce the problem to a sub-problem and solve it iteratively
  - Usually for optimization problems.
- Divide-and-Conquer
  - Break a problem into many independent sub-problems
  - Solve each sub-problem separately
  - Combine solutions for sub-problems to form a solution for the origin one
  - Usually used to design more efficient algorithm
- Dynamic Programming
  - Break up a problem into many overlapping sub-problems
  - Build solutions for larger and larger sub-problems
  - Use a table to store solutions for sub-problems for reuse

43.1 Greedy Algorithms

43.1.1 Introduction

Greedy algorithm solve the problem incrementally. Its often for optimization problems. Solving optimization problem typically requires a sequences of steps, at each step, an irrevocable decision will be made, and makes the choice looks the best at the moment. Based on that, a small instance will be added to the problem and we solve it again.
Greedy algorithm do not always yield optimal solution, but it usually can provide a relatively acceptable computational complexity. They often run in polynomial time due to the incrementally of instances. Sometimes even if we cannot guarantee the solution is optimal, we still use it in optimization, because of its cheap computation burden. One of the examples will be the constructive heuristic of VRP problems.
Greedy algorithm usually gives polynomial time complexity, but that is not all the cases. In simplex method, each pivot is greedily searching through for the extreme point. Although simplex method usually gives us traceable computation running time, however, it is not a polynomial algorithm.
The following is a very rough sketch of generic greedy algorithm
If we can prove the following, we can claim the greedy algorithm. First, it need to prove that for "current moment", the strategy is safe, i.e., there is always an optimum solution that agrees with the decision made according to the

Algorithm 27 Generic Greedy Algorithm

```
1: while The instance is non-trivial do
2:     Make the choice using the greedy strategy
3:     Reduce the instance
```

strategy, this is usually difficult. Then, it need to show that the remaining task after applying the strategy is to solve a/many smaller instance(s) of the same problem.
Although greedy algorithm is intuitive and usually leads to satisfactory complexity, however, for most of the problems there is **no** natural greedy algorithm that works.

43.1.2 Examples

Interval Scheduling

For n jobs, job i starts at time s_i, and finishes at time f_i. i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint. Find the maximum-size of mutually compatible jobs.
The following is the greedy algorithm to solve this problem

Algorithm 28 Interval Scheduling, $S(s, f, n)$

```
1: Sort jobs by f
2: t ← 0, S ← ∅
3: for every j ∈ [n] according to non-decreasing order of f_j do
4:     if s_j ≥ t then
5:         S ← S ∪ {j}
6:         t ← f_j
   return S
```

Now we proof the that it is safe to schedule the job j with the earliest finish time, i.e., there is an optimum solution where the job j with the earliest finish time is scheduled.

Proof. For arbitary optimum solution S, one of the following cases will happen:
Case 1: S contains j, done
Case 2: S does not contain j. Then the first job in S can be replaced by j to obtain another optimum schedule S'. □

Unit-length interval covering

Given a set of n points $X = \{x_1, x_2, \cdots, x_n\}$ on the real line, WLOG, assuming the points have already been sorted. We want to use the smallest number of unit-length closed intervals to cover all the points in X.
The following is a greedy algorithm to find the set of unit-length intervals that cover all the points in real line.

Algorithm 29 Cover points with unit-length intervals

```
1: Initial, S ← ∅
2: for i = 1, 2, ..., n do
3:     if x_i is not covered by any unit-length intervals then
4:         Add one unit-length interval starting from x_i, i.e., S ← S ∪ {x_i}
```

This strategy (algorithm) is a greedy algorithm because it build up the solution in steps, in each iteration it considers one more point to be covered, i.e., it is optimal for each step in the iteration. Now we prove this algorithm is "safe".

Proof. First we consider the case where there is only one point on the real line. Then the optimal number of unit-length interval will be 1, according to the algorithm, that interval will be started at that point.
Then assuming for the case where there are k points from left to right, i.e., $X = \{x_1, x_2, \cdots, x_k\}$, and p unit-intervals is already the minimal number and placed by the algorithm, then the $(k+1)^{th}$ point can only be one of the following cases:

Case 1: The $(k+1)^{th}$ point is covered by the p^{th} unit-length interval. According to the strategy, no new unit-length interval will be needed, the number of unit-length interval for $k+1$ points will be the same as when there are k points. Therefore in this case for $k+1$ points, p is the minimal (optimal) number. So the strategy is "safe" in this case.
Case 2: The $(k+1)^{th}$ point is not covered by the p^{th} unit-length interval. According to the strategy, there will be one new unit-length interval added. Notice that p unit-length intervals will not be feasible to cover $k+1$ points in this case, because if we move the p^{th} unit-length interval to the right, it will not be able to cover at least one point which overlapped with the starting point of that unit-length interval. Since p is infeasible and $p+1$ unit-length interval is feasible, then $p+1$ is the minimal (optimal) number. So the strategy is "safe" in this case.
Notice that for the k we have mentioned above, k can start from 1 to infinite number of integer. So this strategy is "safe" in every cases. □

43.2 Divide and Conquer

43.2.1 Introduction

The divide-and-conquer algorithm contains three steps: divide, conquer and combine. Step one, divide instances into many smaller instances. Step two, conquer small instance by solving each of smaller instances recursively and separately. Step three, combine solutions to small instances to obtain a solution for the origin large instance.
Divide and conquer can sometimes solve the problems that greedy algorithm cannot solve, but they often not strong enough to reduce exponential brute-force search down to polynomial time. What usually happen is that they reduce a running time that unnecessarily large, but already polynomial, down to a faster running time.

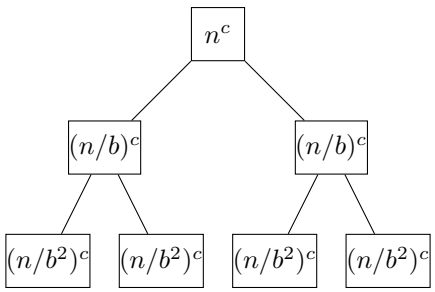
43.2.2 Master Theorem

The Master Theorem is useful in analyzing the running time of divide and conquer algorithm. Assume at each step we divide the origin problem of size n into subproblems of size n/b, run for a times of "itself" to conquer those subproblems with a combine operation of $O(n^c)$, then the total running time can be derived by the Master Theorem as following:

Theorem 43.1 (Master Theorem)**.** *For running time in forms of* $T(n) = aT(n/b) + O(n^c)$*, where* $a \geq 1$*,* $b > 1$*,* $c \geq 0$ *are constants. Then*

$$T(n) = \begin{cases} O(n^{\lg_b a}), & c < \lg_b a \\ O(n^c \lg n), & c = \lg_b a \\ O(n^c), & c > \lg_b a \end{cases} \tag{43.1}$$

Proof of Master theorem using recursion tree



Proof. The i^{th} level has a^{i-1} nodes. For the following cases, we can derive the time complexity:
Case 1: $c < \lg_b a$ bottom-level dominates $(\frac{a}{b^c}^{\lg_b n})n^c = O(n^{\lg_b a})$
Case 2: $c = \lg_b a$ all levels have same time $n^c \lg_b n = O(n^c \lg n)$
Case 3: $c > \lg_b a$ top-level dominates $O(n^c)$ □

43.2.3 Examples

Counting Inversions

43.3 Dynamic Programming

43.3.1 Introduction

The principal of dynamic programming is essentially opposite of the greedy algorithm. Dynamic programming implicitly explores the space of all possible solutions, by carefully decomposing the origin problem into subproblems, and store the solution of those subproblems. Base on those subproblems, then build up larger and larger problem until the origin problem is solved.

43.3.2 Examples

Weighted Interval Scheduling

Consider n jobs, job i starts at time s_i and finishes at time f_i, each job has a weight of $v_i > 0$. Job i and job j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint. Find the maximum-size subset of mutually compatible jobs. A special case of this problem is when the values of all jobs are equal, which will be the interval scheduling problem as discussed in the greedy algorithm examples.
Define $p(j)$ as for a job j, the largest index $i < j$ such that job j is disjoint with job i. Define $opt(i)$ as the optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$.
Before the algorithm for weighted interval scheduling, sort the jobs by non-decreasing order of finishing time first, in $O(n \lg n)$ The dynamic programming algorithm is as following:

Algorithm 30 ComputeOpt(i)

```
1: if i == 0 then
2:     return 0
3: else
4:     return max{v_i + ComputeOpt(p(i)), ComputeOpt(i − 1)}
```

For finding $p(i)$ for one job, it takes $O(\lg n)$ by binary search. For n jobs the complexity will be $O(n \lg n)$.
The running time of this algorithm can be exponential in n, if each time $ComputeOpt(i)$ is computed repeatedly. However, if we store the value of each $ComputeOpt(i)$ and reuse it, we can reduce the running time to $O(n)$.
We can recover the set of jobs for given (valid) $opt(i)$ by the following algorithm, assuming the jobs has been sorted by non-decreasing order of finishing time.

Algorithm 31 RecoverJobs()

```
1: Compute p_1, p_2, ··· , p_n
2: opt(0) ← 0
3: for i ← 1 to n do
4:     if opt(i − 1) ≥ v_i + opt(p_i) then
5:         opt(i) ← opt(i − 1)
6:         b[i] ← N
7:     else
8:         opt(i) ← v_i + opt(p_i)
9:         b[i] ← Y
10: i ← n, S ← ∅
11: while i ≠ 0 do
12:     if b[i] == N then
13:         i ← i − 1
14:     else
15:         S ← S ∪ {i}
16:         i ← p_i
17: return S
```

The above algorithm is using memorized recursion to solve the problem, there is a second efficient algorithm to solve the Weighted Interval Scheduling Problem.

Subset Sum Problem

Given an integer bound $W > 0$ and a set of n items, each with an integer weight $w_i > 0$, find a subset S of items that

$$\max \quad \sum_{i \in S} w_i \tag{43.2}$$

$$\text{s.t.} \quad \sum_{i \in S} w_i \leq W \tag{43.3}$$

Consider this instance, i items, $(w_1, w_2, \cdots, w_i)$, budget is W'. For $opt[i, W']$ there can only be one of the following cases:
Case 1: The value of optimum solution does not contain w_i, then $opt[i, W'] = opt[i-1, W']$, else
Case 2: The value of optimum solution contains w_i, then $opt[i, W'] = opt[i-1, W'-w_i] + w_i$
The algorithm is as following

Algorithm 32 Optimum Subset

```
1: for W' ← 0 to W do
2:     opt[0, W'] ← 0
3: for i ← 1 to n do
4:     for W' ← 0 to W do
5:         opt[i, W'] ← opt[i − 1, W']
6:         b[i, W'] ← N
7:         if w_i ≤ W' and opt[i − 1, W' − w_i] + w_i ≥ opt[i, W'] then
8:             opt[i, W'] ← opt[i − 1, W' − w_i] + w_i
9:             b[i, W'] ← Y
10: return opt[n, W]
```

Algorithm 33 Recover the Optimum Set

```
1: i ← n, W' ← W, S ← 0
2: while i > 0 do
3:     if b[i, W'] == Y then
4:         W' ← W' − w_i
5:         S ← ∪{i}
6:     i ← i − 1
7: return S
```

Optimum Binary Search Tree

Given n elements $e_1 < e_2 < \cdots < e_n$, e_i has frequency f_i, the goal is to build a binary search tree for $\{e_1, e_2, \cdots, e_n\}$ with the minimum accessing cost

$$\sum_{i=1}^{n} f_i d_i \tag{43.4}$$

Where d_i is the depth of e_i in the tree.
Suppose we choose e_k to be the root, than $e_1, e_2, \cdots, e_{k-1}$ are in left-sub tree, and $e_{k+1}, \cdots, e_n$ will be in right-sub

tree, then, denote the cost for the tree and subtrees to be C, C_L, C_R respectively

$$C = \sum_{j=1}^{n} f_j d_j \tag{43.5}$$

$$= \sum_{j=1}^{n} f_j + \sum_{j=1}^{n} f_j (d_j - 1) \tag{43.6}$$

$$= \sum_{j=1}^{n} f_j + \sum_{j=1}^{k-1} f_j (d_j - 1) + \sum_{j=k+1}^{n} f_j (d_j - 1) \tag{43.7}$$

$$= \sum_{j=1}^{n} f_j + C_L + C_R \tag{43.8}$$

Denote $opt(i, j)$ to be the optimal value for the instance of $(e_i, e_{i+1}, \cdots, e_j)$, then, for every i, j such that $1 \leq i \leq j \leq n$

$$opt(i, j) = \sum_{k=i}^{j} f_k + \min_{k: i \leq k \leq j} \{opt(i, k-1) + opt(k+1, j)\} \tag{43.9}$$

Here is an example

Example. Consider the following optimum binary search tree instance. We have 5 elements e_1, e_2, e_3, e_4 and e_5 with $e_1 < e_2 < e_3 < e_4 < e_5$ and their frequencies are $f_1 = 5, f_2 = 25, f_3 = 15, f_4 = 10$ and $f_5 = 30$. Recall that the goal is to find a binary search tree for the 5 elements so as to minimize $\sum_{i=1}^{5} \text{depth}(e_i) f_i$, where $\text{depth}(e_i)$ is the depth of the element e_i in the tree. You need to output the best tree as well as its cost. You can try to complete the following tables and show the steps. In the two tables, $opt(i, j)$ is the cost of the best tree for the instance containing $e_i, e_{i+1}, ..., e_j$ and $\pi(i, j)$ is the root of the best tree.

$opt(i,j)$ i \ j	1	2	3	4	5
1	5	35	65	95	170
2		25	55	85	155
3			15	35	90
4				10	50
5					30

$\pi(i,j)$ i \ j	1	2	3	4	5
1	1	2	2	2	3
2		2	2	2 (or 3)	3
3			3	3	5
4				4	5
5					5

Table 43.1: *opt* and π tables for the optimum binary search tree instance. For cleanness of the table, we assume $opt(i, j) = 0$ if $j < i$ and there are not shown in the left table.

$$
\begin{aligned}
opt(1,2) &= \min\{0 + opt(2,2), opt(1,1) + 0\} + (f_1 + f_2) = \min\{25, 5\} + 5 + 25 = 35 \\
opt(2,3) &= \min\{0 + opt(3,3), opt(2,2) + 0\} + (f_2 + f_3) = \min\{15, 25\} + 25 + 15 = 55 \\
opt(3,4) &= \min\{0 + opt(4,4), opt(3,3) + 0\} + (f_3 + f_4) = \min\{10, 15\} + 15 + 10 = 35 \\
opt(4,5) &= \min\{0 + opt(5,5), opt(4,4) + 0\} + (f_4 + f_5) = \min\{30, 10\} + 10 + 30 = 50 \\
opt(1,3) &= \min\{0 + f(2,3), f(1,1) + f(3,3), f(1,2) + 0\} + (f_1 + f_2 + f_3) \\
&= \min\{55, 20, 35\} + 5 + 25 + 15 = 65 \\
opt(2,4) &= \min\{0 + f(3,4), f(2,2) + f(4,4), f(2,3) + 0\} + (f_2 + f_3 + f_4) \\
&= \min\{35, 35, 55\} + 25 + 15 + 10 = 85 \\
opt(3,5) &= \min\{0 + f(4,5), f(3,3) + f(5,5), f(3,4) + 0\} + (f_3 + f_4 + f_5) \\
&= \min\{50, 45, 35\} + 15 + 10 + 30 = 90 \\
opt(1,4) &= \min\{0 + f(2,4), f(1,1) + f(3,4), f(1,2) + f(4,4), f(1,3) + 0\} \\
&\quad + (f_1 + f_2 + f_3 + f_4) = 95 \\
&= \min\{85, 40, 45, 65\} + 5 + 25 + 15 + 10 \\
opt(2,5) &= \min\{0 + f(3,5), f(2,2) + f(4,5), f(2,3) + f(5,5), f(2,4) + 0\} \\
&\quad + (f_2 + f_3 + f_4 + f_5) = 155 \\
&= \min\{90, 75, 85, 85\} + 25 + 15 + 10 + 30 \\
opt(1,5) &= \min\{0 + f(2,5), f(1,1) + f(3,5), f(1,2) + f(4,5), f(1,3) + f(5,5) + f(1,4) + 0\} \\
&\quad + (f_1 + f_2 + f_3 + f_4 + f_5) \\
&= \min\{155, 95, 85, 95, 95\} + 5 + 25 + 15 + 10 + 30 = 170
\end{aligned}
$$

Matrix Chain Multiplication

Given n matrices $A_1, A_2, \cdots, A_n$ of sizes $r_1 \times c_1, r_2 \times c_2, \cdots, r_n \times c_n$ where $c_i = r_{i+1}$ for every $i = 1, 2, \cdots, n-1$. The Matrix Chain Multiplication finds the order of computing $A_1 A_1 \cdots A_n$ with the minimum number of multiplications. The idea is as following. Assume the last step in the multiplication is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$. The cost of this step will be $r_i \times c_i \times c_n$. So we need to optimally solve two sub-instances, i.e., $(A_1 A_2 \cdots A_i)$ and $(A_{i+1} A_{i+2} \cdots A_n)$.
Denote $opt[i, j]$ as the minimum cost of computing $A_i A_{i+1} \cdots A_j$, then

$$
opt[i,j] = \begin{cases} 0, & i = j \\ \min_{k:i \le k < j}(opt[i,k] + opt[k+1,j] + r_i \times c_k \times c_j), & j < j \end{cases} \tag{43.10}
$$

The algorithm is as following

Algorithm 34 MatrixChainMultiplication

```
1: opt[i, i] ← 0   ∀ i = 1, 2, ··· , n
2: for l ← 2 to n do
3:     for i ← 1 to n − l + 1 do
4:         j ← i + l − 1
5:         opt[i, j] ← ∞
6:         for k ← i to j − 1 do
7:             if opt[i, k] + opt[k + 1, j + r_i c_k c_j < opt[i, j] then
8:                 opt[i, j] ← opt[i, k] + opt[k + 1, j] + r_i c_k c_j
9:                 π[i, j] ← k
10: return opt[1, n]
```

With above algorithm, to construct the optimal solution, the follow algorithm is needed

43.4 Compare between three paradigms

Algorithm 35 PrintOptimalOrder(i, j)

```
1: if i == j then
2:     Print(A_i)
3: else
4:     Print("(")
5:     PrintOptimalOrder(i, π[i, j])
6:     PrintOptimalOrder(π[i, j] + 1, j)
7:     Print(")")
```

Chapter 44

Sorting

44.1 Exchange Sorts

44.1.1 Bubble Sort

44.1.2 Cocktail Shaker Sort

44.1.3 Odd-even Sort

44.1.4 Comb Sort

44.1.5 Gnome Sort

44.1.6 Quicksort

The idea of quicksort is to recursively divide an array into two subarrays, one with smaller number and one with large number, and concatenate the subarrays after all subarrays are singletons.
The most ideal way is to divide the subarrays equally, which requires an algorithm to find the median of an array of size n in $O(n)$ time.
The quicksort algorithm is as following

Algorithm 36 Quicksort(A, n)

```
1: if n = 1 then return A
2: Initial, A_L ← ∅, A_H ← ∅
3: x ← Median(A)
4: for element ∈ A do
5:     if element ≤ x then
6:         A_L ← A_L ∪ {element}
7:     else
8:         A_H ← A_H ∪ {element}
9: B_L ← Quicksort(A_L, A_L.size)
10: B_H ← Quicksort(A_H, A_H.size)
11: t ← number of times element appear in A return B_L + element^t + B_H
```

Running time $T(n) = 2T(n/2) + O(n)$, $T(n) = O(n \lg n)$
For the median finding algorithm

Algorithm 37 Median(A)

```
1: (To be finished)
```

If we don't use the median finding algorithm, we can modify the `Quicksort(A, n)` to be a random algorithm by replacing line 3 by $x \leftarrow$ `RandomElement(A)`. This modified algorithm has an expected running time of $O(n \lg n)$. The worse case running time is $O(n^2)$.

Based on Quicksort algorithm, we can define an $O(n)$ algorithm to find the ith smallest number in A, given that we have an $O(n)$ algorithm to find median of array.

The selection algorithm is as follows

Algorithm 38 Selection(A, n, i)

```
1:  if n = 1 then
2:      return A
3:  else
4:      x ← Median(A)
5:      for element ∈ A do
6:          if element ≤ x then
7:              A_L ← A_L ∪ {element}
8:          else
9:              A_H ← A_H ∪ {element}
10:     if i ≤ A_L.size then
11:         return Selection(A_L, A_L.size, i)
12:     else if i > n − A_R.size then
13:         return Selection(A_R, A_R.size, i − (n − A_R.size))
14:     else
15:         return x
```

Similarly, without `Median(A)`, we can replace line 4 by $x \leftarrow$ `RandomElement(A)`. Then the expected running time will be $O(n)$

44.2 Selection Sorts

44.2.1 Selection Sort

44.2.2 Heapsort

44.2.3 Smoothsort

44.2.4 Cartesian Tree Sort

44.2.5 Tournament Sort

44.2.6 Cycle Sort

44.2.7 Weak-heap Sort

44.3 Insertion Sorts

44.3.1 Insertion Sort

44.3.2 Shell Sort

44.3.3 Splaysort

44.3.4 Tree Sort

44.3.5 Library Sort

44.3.6 Patience Sorting

44.4 Merge Sorts

44.4.1 Merge Sort

Merge sort is a typical divide and conquer algorithm. It recursively separate an array into two subarrays, and sort while merging them. The algorithm is as following

Algorithm 39 MergeSort(A, n)

1: **if** $n = 1$ **then return** A
2: **else**
3: $B \leftarrow$ `MergeSort(`$A[0..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor$`)`
4: $C \leftarrow$ `MergeSort(`$A[\lceil n/2 \rceil..n], \lceil n/2 \rceil$`)`
return `Merge(`$B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil$

Chapter 45

Mathematical Algorithm

45.1 Polynomial Multiplication

For given two polynomials of degree $n-1$, the algorithm outputs the product of two polynomials.

Example.

$$(3x^3 + 2x^2 - 6x + 9) \times (-2x^3 + 7x^2 - 8x + 4) \tag{45.1}$$
$$= -6x^6 + 17x^5 + 24x^4 - 60x^3 + 119x^2 - 96x + 36 \tag{45.2}$$

Then for input as (3, 2, -6, 9) and (-2, 7, -8, 4), the output will be (-6, 17, 2, -60, 119, -96, 36)

A naive algorithm to solve this problem will be $O(n^2)$

Algorithm 40 PolyMultNaive(A, B, n)

```
1: Let C[k] = 0 for every k = 0, 1, ··· , 2n − 2
2: for i ← 0 to n − 1 do
3:     for j ← 0 to n − 1 do
4:         C[i + j] ← C[i + j] + A[i] × B[j]
   return C
```

Use divide and conquer can reduce the running time. The idea is to divide the polynomial with degree of $n-1$ (WLOG, let n be even number) by two polynomials, i.e.

$$p(x) = p_H(x)x^{\frac{n}{2}} + p_L(x) \tag{45.3}$$

Both p_H and p_L are polynomials with degree of $\frac{n}{2} - 1$, then

$$p(x)q(x) = (p_H(x)x^{\frac{n}{2}} + p_L(x)) \times (q_H(x)x^{\frac{n}{2}} + q_L(x)) \tag{45.4}$$
$$= p_H q_H x^n + (p_H q_L + p_L q_H)x^{\frac{n}{2}} + p_L q_L \tag{45.5}$$

Therefore

$$multiply(p, q) = multiply(p_H, q_H)x^n \tag{45.6}$$
$$+ (multiply(p_H, q_L) + multiply(p_L, q_H))x^{\frac{n}{2}} + multiply(p_L, q_L) \tag{45.7}$$
$$= multiply(p_H, q_H)x^n \tag{45.8}$$
$$+ (multiply(p_H + p_L, q_H + q_L) - multiply(p_H, q_H) - multiply(p_L, q_L))x^{\frac{n}{2}} \tag{45.9}$$
$$+ multiply(p_L, q_L) \tag{45.10}$$
$$\tag{45.11}$$

The algorithm is as following
The running time $T(n) = 3T(n/2) + O(n)$, $T(n) = O(n^{\lg_2 3})$

Algorithm 41 PolyMultiDC(A, B, n)

1: **if** $n = 1$ **then return** $A[0]B[0]$
2: $A_L \leftarrow A[0..n/2-1]$, $A_H \leftarrow A[n/2..n-1]$
3: $B_L \leftarrow B[0..n/2-1]$, $B_H \leftarrow B[n/2..n-1]$
4: $C_L \leftarrow$ `PolyMultiDC(`A_L`, `B_L`, n/2)`
5: $C_H \leftarrow$ `PolyMultiDC(`A_H`, `B_H`, n/2)`
6: $C_M \leftarrow$ `PolyMultiDC(`A_H `+` A_L`, `B_H `+` B_L`, n/2)`
7: $C \leftarrow 0$ array of length $2n-1$
8: **for** $i \leftarrow 0$ to $n-2$ **do**
9: $\quad C[i] \leftarrow C[i] + C_L[i]$
10: $\quad C[i+n] \leftarrow C[i+n] + C_H[i]$
11: $\quad C[i+n/2] \leftarrow C[i+n/2] + C_M[i] - C_L[i] - C_H[i]$
return C

45.2 Matrices Multiplication

45.3 Gaussian Elimination

45.4 Curve Fitting

45.5 Integration

Chapter 46

Searching

Chapter 47

String

Chapter 48

Data Structures

Special Topic: Computational Geometry

Chapter 49

Convex Hull

Chapter 50

Intersections

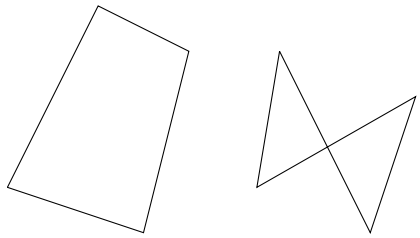
Chapter 51

Triangulation and Partitioning

51.1 Polygon Triangulation

51.1.1 Types of Polygons

Definition 51.1.1 (simple polygon)**.** A **simple polygon** is a closed polygonal curve without self-intersection.

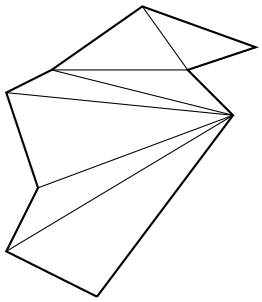


Simple Polygon Non-simple Polygon

Polygons are basic building blocks in most geometric applications. It can model arbitrarily complex shapes, and apply simple algorithms and algebraic representation/manipulation.

51.1.2 Triangulation

Definition 51.1.2 (Triangulation)**. Triangulation** is to partition polygon P into non-overlapping triangles using diagonals only. It reduces complex shapes to collection of simpler shapes. Every simple n-gon admits a triangulation which has $n-2$ triangles.



Triangulation

Theorem 51.1. *Every polygon has a triangulation*

Lemma 51.2. *Every polygon with more than three vertices has a diagonal.*

Proof. (by Meisters, 1975) Let P be a polygon with more than three vertices. Every vertex of a P is either *convex* or *concave*. W.L.O.G.(any polygon must has convex corner) Assume p is a convex vertex. Denote the neighbors of p as q and r. If $\bar{qr}$ is a diagonal, done, and we call $\triangle pqr$ is an *ear*. If $\triangle pqr$ is not an ear, it means at least one vertex is inside $\triangle pqr$, assume among those vertexes inside $\triangle pqr$, s is a vertex closest to p, then $\bar{ps}$ is a diagonal. □

51.1.3 Art Gallery Theorem

Theorem 51.3. *Every n-gon can be guarded with $\lfloor \frac{n}{3} \rfloor$ vertex guards*

Lemma 51.4. *Triangulation graph can be 3-colored.*

Problem 51.1. The floor plan of an art gallery modeled as a simple polygon with n vertices, there are guards which is stationed at fixed positions with 360 degree vision but cannot see through the walls. How many guards does the art gallery need for the security? (Fun fact: This problem was posted to Vasek Chvatal by Victor Klee in 1973).

Proof. - P plus triangulation is a planar graph
- 3-coloring means there exist a 3-partition for vertices that no edge or diagonal has both endpoints within the same set of vertices.
- Proof by Induction:
- Remove an ear (there will always exist ear)
- Inductively 3-color the rest
- Put ear back, coloring new vertex with the label not used by the boundary diagonal. □

51.1.4 Triangulation Algorithms

Chapter 52

Voronoi Diagrams

Chapter 53

Arrangement and Duality

Chapter 54

Delaunay Triangulations

Chapter 55

Search

Chapter 56

Motion Planning

Chapter 57

Quadtrees

Chapter 58

Visibility Graphs

Part VIII

Stochastic Methods

Chapter 59

Markov Chain

Chapter 60

Queueing Theory

Part IX

Inventory Theory

Part X

Game Theory

Part XI

Simulation

Chapter 61

Random Numbers

Chapter 62

Monte Carlo Simulation

Chapter 63

Discrete Event Simulation