Notes for Operations Research & More

Lan Peng, PhD Candidate

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

August 6, 2020

August 6, 2020

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:

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

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 2) Infinite number of solution 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 Problem Manipulation

1.2.1 Inequalities and Equalities

An inequality can be transformed into an equation by adding or subtracting the nonnegative slack or surplus variable

$$\sum_{j=1}^{n} a_{ij}x_j \geq b_i \Rightarrow \sum_{j=1}^{n} a_{ij}x_j - x_{n+1} = b_i \tag{1.5}$$

or

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \Rightarrow \sum_{j=1}^{n} a_{ij}x_j + x_{n+1} = b_i \tag{1.6}$$

Although it is not the practice, equality can be transformed into inequality too

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \Rightarrow \begin{cases} \sum_{j=1}^{n} a_{ij}x_j \leq b_i \\ \sum_{j=1}^{n} a_{ij}x_j \geq b_i \end{cases} \tag{1.7}$$

Also, in linear programming, we only care about close set, so we will not have $<$, $>$ in the formulation, we can use the following

$$\sum_{j=1}^{n} a_{ij}x_j > b_i \Rightarrow \sum_{j=1}^{n} a_{ij}x_j \geq b_i + \epsilon \tag{1.8}$$

where ϵ is a small number.

1.2.2 Minimization and Maximization

To convert a minimization problem into a maximization problem, we can use the following to define a new objective function

$$\min \sum_{j=1}^{n} c_jx_j = -\max \sum_{j=1}^{n} c_jx_j \tag{1.9}$$

1.2.3 Standard Form and Canonical Form

Standard Form

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_jx_j && (1.10) \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} = \mathbf{b} && (1.11) \\ & \mathbf{x} \geq \mathbf{0} && (1.12) \end{aligned}$$

Canonical Form

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} c_jx_j && (1.13) \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} && (1.14) \\ & \mathbf{x} \geq \mathbf{0} && (1.15) \end{aligned}$$

1.3 Typical Linear Programming Problems

1.4 Linear Programming Formulation Skills

1.4.1 Absolute Value

Consider the following model statement:

$$\begin{aligned} \min \quad & \sum_{j\in J} c_j|x_j|, \quad c_j > 0 && (1.16)\\ \text{s.t.} \quad & \sum_{j\in J} a_{ij}x_j \gtreqless b_i, \quad \forall i\in I && (1.17)\\ & x_j \quad \text{unrestricted}, \quad \forall j \in J && (1.18)\end{aligned}$$

Modeling:

$$\begin{aligned} \min \quad & \sum_{j\in J} c_j(x_j^+ + x_j^-), \quad c_j > 0 && (1.19)\\ \text{s.t.} \quad & \sum_{j\in J} a_{ij}(x_j^+ - x_j^-) \gtreqless b_i, \quad \forall i\in I && (1.20)\\ & x_j^+, x_j^- \geq 0, \quad \forall j \in J && (1.21)\end{aligned}$$

1.4.2 A Minimax Objective

Consider the following model statement:

$$\begin{aligned} \min \quad & \max_{k\in K} \sum_{j\in J} c_{kj}x_j && (1.22)\\ \text{s.t.} \quad & \sum_{j\in J} a_{ij}x_j \gtreqless b_i, \quad \forall i\in I && (1.23)\\ & x_j \geq 0, \quad \forall j \in J && (1.24)\end{aligned}$$

Modeling:

$$\begin{aligned} \min \quad & z && (1.25)\\ \text{s.t.} \quad & \sum_{j\in J} a_{ij}x_j \gtreqless b_i, \quad \forall i\in I && (1.26)\\ & \sum_{j\in J} c_{kj}x_j \leq z, \quad \forall k\in K && (1.27)\\ & x_j \geq 0, \quad \forall j \in J && (1.28)\end{aligned}$$

1.4.3 A Fractional Objective

Consider the following model statement:

$$\begin{aligned} \min \quad & \frac{\sum_{j\in J} c_jx_j + \alpha}{\sum_{j\in J} d_jx_j + \beta} && (1.29)\\ \text{s.t.} \quad & \sum_{j\in J} a_{ij}x_j \gtreqless b_i, \quad \forall i\in I && (1.30)\\ & x_j \geq 0, \quad \forall j \in J && (1.31)\end{aligned}$$

Modeling:

$$\min \quad \sum_{j\in J} c_j x_j t + \alpha t \tag{1.32}$$
$$\text{s.t.} \quad \sum_{j\in J} a_{ij} x_j \gtreqless b_i, \quad \forall i \in J \tag{1.33}$$
$$\sum_{j\in J} d_j x_j t + \beta t = 1 \tag{1.34}$$
$$t > 0 \tag{1.35}$$
$$x_j \geq 0, \quad \forall j \in J \tag{1.36}$$
$$(t = \frac{1}{\sum_{j\in J} d_j x_j + \beta}) \tag{1.37}$$

For the following statement:

$$\min \quad z^P = \frac{\mathbf{c}^\top \mathbf{x} + d}{\mathbf{e}^\top \mathbf{x} + f} \tag{1.38}$$
$$\text{s.t.} \quad \mathbf{G}\mathbf{x} \leq \mathbf{h} \tag{1.39}$$
$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1.40}$$

Modeling

$$\min \quad z^R = \mathbf{c}^\top \mathbf{y} + dz \tag{1.41}$$
$$\text{s.t.} \quad \mathbf{G}\mathbf{y} - \mathbf{h}z \leq 0 \tag{1.42}$$
$$\mathbf{A}\mathbf{y} - \mathbf{b}z = 0 \tag{1.43}$$
$$\mathbf{e}^\top \mathbf{y} + fz = 1 \tag{1.44}$$
$$z \geq 0 \tag{1.45}$$

1.4.4 A Range Constraint

Consider the following model statement:

$$\min \quad \sum_{j\in J} c_j x_j \tag{1.46}$$
$$\text{s.t.} \quad d_i \leq \sum_{j\in J} a_{ij} x_j \leq e_i, \quad \forall i \in I \tag{1.47}$$
$$x_j \geq 0, \quad \forall j \in J \tag{1.48}$$

Modeling:

$$\min \quad \sum_{j\in J} c_j x_j, \quad c_j > 0 \tag{1.49}$$
$$\text{s.t.} \quad u_i + \sum_{j\in J} a_{ij} x_j = e_i, \quad \forall i \in I \tag{1.50}$$
$$x_j \geq 0, \quad \forall j \in J \tag{1.51}$$
$$0 \leq u_i \leq e_i - d_i, \quad \forall i \in I \tag{1.52}$$

1.5 Typical Integer Programming Problems

1.6 Integer Programming Formulation Skills

1.6.1 A Variable Taking Discontinuous Values

In algebraic notation:

$$x = 0, \quad \text{or} \quad l \leq x \leq u \tag{1.53}$$

Modeling:

$$x \le uy \tag{1.54}$$
$$x \ge ly \tag{1.55}$$
$$y \in \{0,1\} \tag{1.56}$$

where

$$y = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } l \le x \le u \end{cases} \tag{1.57}$$

1.6.2 Fixed Costs

In algebraic notation:

$$C(x) = \begin{cases} 0 & \text{for } x = 0 \\ k + cx & \text{for } x > 0 \end{cases} \tag{1.58}$$

Modeling:

$$C^*(x,y) = ky + cx \tag{1.59}$$
$$x \le My \tag{1.60}$$
$$x \ge 0 \tag{1.61}$$
$$y \in \{0,1\} \tag{1.62}$$

where

$$y = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } x \ge 0 \end{cases} \tag{1.63}$$

1.6.3 Either-or Constraints

In algebraic notation:

$$\sum_{j \in J} a_{1j}x_j \le b_1 \text{ or } \sum_{j \in J} a_{2j}x_j \le b_2 \tag{1.64}$$

Modeling:

$$\sum_{j \in J} a_{1j}x_j \le b_1 + M_1 y \tag{1.65}$$
$$\sum_{j \in J} a_{2j}x_j \le b_2 + M_1(1-y) \tag{1.66}$$
$$y \in \{0,1\} \tag{1.67}$$

where

$$y = \begin{cases} 0, & \text{if } \sum_{j \in J} a_{1j}x_j \le b_1 \\ 1, & \text{if } \sum_{j \in J} a_{2j}x_j \le b_2 \end{cases} \tag{1.68}$$

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

1.6.4 Conditional Constraints

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

$$\text{If } \sum_{j \in J} a_{1j}x_j \le b_1 \text{ then } \sum_{j \in J} a_{2j}x_j \le b_2 \tag{1.69}$$

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

$$\sum_{j \in J} a_{1j}x_j > b_1 \text{ or } \sum_{j \in J} a_{2j}x_j \le b_2 \tag{1.70}$$

Furthermore, it is equivalent to

$$\sum_{j\in J} a_{1j}x_j \geq b_1 + \epsilon \text{ or } \sum_{j\in J} a_{2j}x_j \leq b_2 \tag{1.71}$$

Where ϵ is a very small positive number.
Modeling:

$$\sum_{j\in J} a_{1j}x_j \geq b_1 + \epsilon - M_2 y \tag{1.72}$$

$$\sum_{j\in J} a_{2j}x_j \leq b_2 + M_2(1-y) \tag{1.73}$$

$$y \in \{0,1\} \tag{1.74}$$

1.6.5 Special Ordered Sets

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

$$x_1 = 1, x_2 = x_3 = \cdots = x_n = 0 \tag{1.75}$$

$$\text{or} \tag{1.76}$$

$$x_2 = 1, x_1 = x_3 = \cdots = x_n = 0 \tag{1.77}$$

$$\text{or ...} \tag{1.78}$$

Modeling:

$$\sum_i x_i = 1, \quad i \in N \tag{1.79}$$

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

$$\sum_{i=1}^{n} x_i \leq 2 \tag{1.80}$$

$$x_i + x_j \leq 1, \forall i \in \{1,2,...,n\}, j \in \{i+2, i+3, ..., n\} \tag{1.81}$$

$$x_i \in \{0,1\} \tag{1.82}$$

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.

1.6.6 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

$$d_i = \begin{cases} 1, & x \in (x_i, x_{i+1}) \\ 0, & \text{otherwise} \end{cases} \tag{1.83}$$

Then the objective function is

$$\sum_{i\in\{1,2,...,k-1\}} y = d_i f_i(x) \tag{1.84}$$

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

$$\sum_{i\in\{1,2,...,k-1\}} d_i = 1 \tag{1.85}$$

$$d_i \in \{0,1\}, i \in \{1,2,...,k-1\} \tag{1.86}$$

$$x = \sum_{i\in\{1,2,...,k\}} w_i x_i \tag{1.87}$$

$$y = \sum_{i\in\{1,2,...,k\}} w_i y_i \tag{1.88}$$

$$w_1 \leq d_1 \tag{1.89}$$

$$w_i \leq d_{i-1} + di, i \in \{2,3,...,k-1\} \tag{1.90}$$

$$w_k \leq d_{k-1} \tag{1.91}$$

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

1.6.7 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:

$$\sum_{k\in\{1,2,...,m\}} x_k \leq n \tag{1.92}$$

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

$$\sum_{k\in\{1,2,...,m\}} x_k = n \tag{1.93}$$

Choose x_j only if $x_k = 1$
Modeling:

$$x_j = x_k \tag{1.94}$$

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

$$y \leq x_i, i \in \{1,2,...,m\} \tag{1.95}$$

$$y \geq \sum_{i\in\{1,2,...,m\}} x_i - (m-1) \tag{1.96}$$

1.6.8 Elimination of Products of Variables

For variables x_1 and x_2,

$$y = x_1 x_2 \tag{1.97}$$

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

$$y \leq u x_1 \tag{1.98}$$

$$y \leq x_2 \tag{1.99}$$

$$y \geq x_2 - u(1 - x_1) \tag{1.100}$$

$$y \geq 0 \tag{1.101}$$

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

1.7 Typical Nonlinear Programming Problems

1.8 Nonlinear Programming Formulation Skills

Chapter 2

Review 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.

$$< \alpha|\beta >= x_1y_1 - x_2y_1 - x_1y_2 + 4x_2y_2 \tag{2.3}$$

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 $\mathbf{A}\mathbf{x}$ is a scalar multiple of $\mathbf{x}$, i.e.

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

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

Review of Real Analysis

3.1 The Real Number System

3.2 Open Sets and Closed Sets

Definition 3.2.1 (neighborhood)**.** $N_\epsilon = \{y \in \mathbb{R}^n | \|y - x\| < \epsilon\}$ as the **neighborhood** of $x \in \mathbb{R}^n$

Definition 3.2.2 (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 3.2.3 (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 3.2.4 (closure)**.** $x \in S$ belongs to the **closure** of S, denoted $cl(s)$ if $\forall \epsilon > 0$, $N_\epsilon(x) \cap S = \emptyset$

Definition 3.2.5 (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 3.2.6 (ball)**.** Let X be a metric space. A **ball** B of radius r around a point $x \in X$ is

$$B = \{y \in X | d(x, y) < r\} \tag{3.1}$$

Definition 3.2.7 (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 3.1. *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 3.2. *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

$$S_i = (A_i, B_i) \subset \mathbb{R} \tag{3.2}$$

Then

$$S = \cap_{i=1}^{\infty} S_i = [a, b] \subset \mathbb{R} \tag{3.3}$$

and S is a closed set.

Definition 3.2.8 (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 3.2.9 (closed set)**.** A set C is **closed** iff it contains all of its limit points.

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

Theorem 3.3. *$S \in \mathbb{R}^n$ is closed* $\iff \forall \{x_k\}_{k=1}^{\infty} \in S, \lim_{k\to\infty}\{x_k\}_{k=1}^{\infty} \in S$

Theorem 3.4. *Every intersection of closed sets is closed.*

Theorem 3.5. *Every finite union of closed sets is closed.*

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

Theorem 3.6. *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.

3.3 Functions, Sequences, Limits and Continuity

3.4 Differentiation

3.5 Integration

3.6 Infinite Series of Constants

3.7 Power Series

3.8 Uniform Convergence

3.9 Arcs and Curves

3.10 Partial Differentiation

3.11 Multiple Integrals

3.12 Improper Integrals

3.13 Fourier Series

Chapter 4

Review of Probability Theory

4.1 Relationship between Some Random Variables

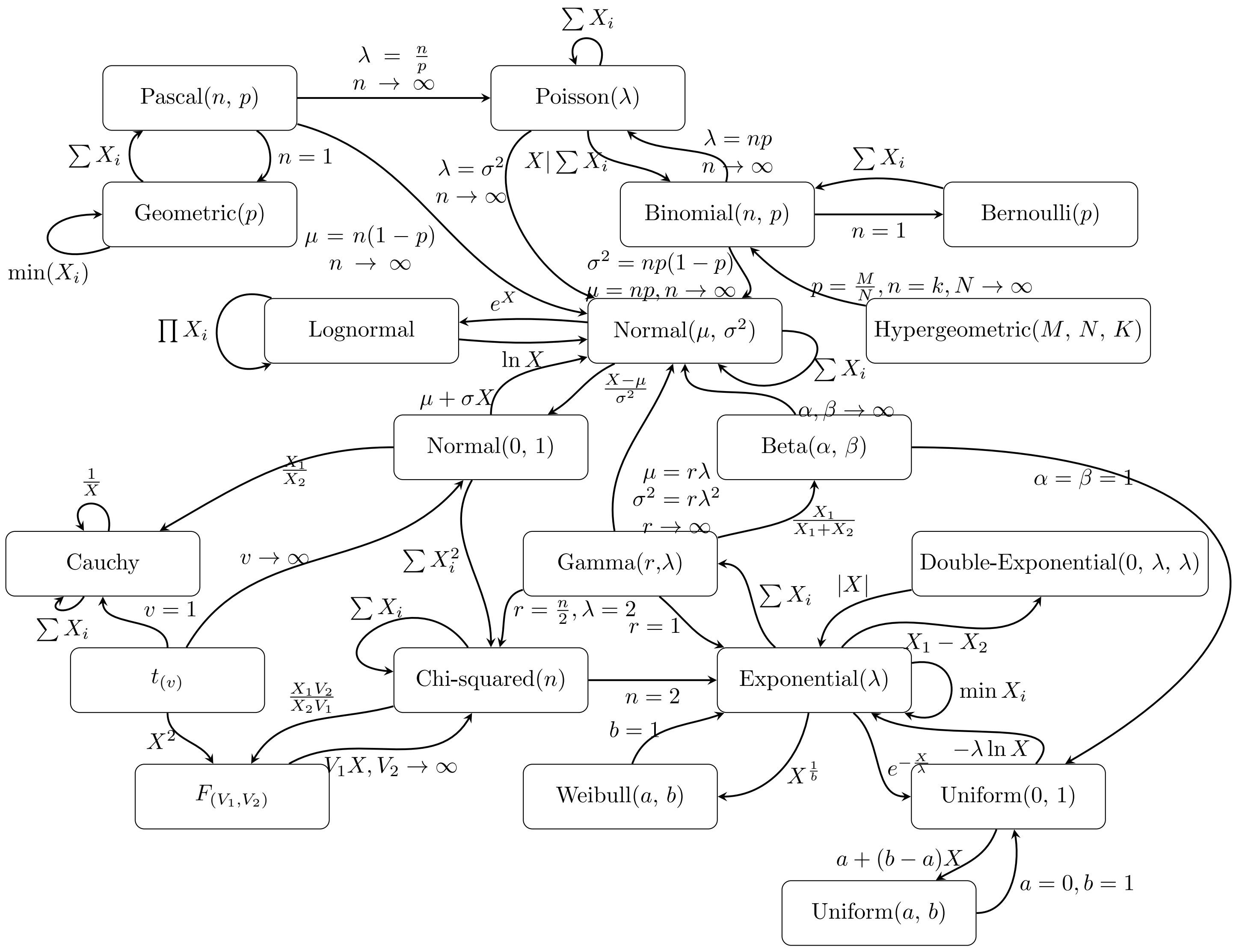


Figure 4.1: Relationship between Some Random Variables

4.2 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)	$\binom{n-1+x}{x}p^n(1-p)^x$ $x=0,1,2,...,n,...$	$1-I_p(k+1,n)$ $x=0,1,2,...,n,...$	$\frac{n(1-p)}{p}$	$\frac{n(1-p)}{p^2}$	$(\frac{p}{1-(1-p)e^t})^n$ $t<-\ln(1-p)$

(4.1)

4.3 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)	$\frac{\lambda^n x^{n-1}e^{-\lambda x}}{(n-1)!}$ $x>0$	$1-\sum_{i=0}^{n-1}\frac{\lambda^n x^n e^{-\lambda x}}{n!}$ $x>0$	$\frac{n}{\lambda}$	$\frac{n}{\lambda^2}$	$\frac{1}{(1-\frac{t}{\lambda})^n}$ $t<\lambda$

(4.2)

Part II

Linear Programming

Chapter 5

The Simplex Method - Basic

5.1 Basic Feasible Solutions and Extreme Points

Definition 5.1.1 (Basic Feasible Solutions)**.** Consider the system $\{\mathbf{A_{m\times n}x = b_m, b_m \geq 0}\}$, 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 $\mathbf{x_B} = \mathbf{B}^{-1}\mathbf{b}$ and $\mathbf{x_N} = \mathbf{0}$ is called **basic solution** of system. If $\mathbf{x_B} \geq \mathbf{0}$, it is called **basic feasible solution** or **B.F.S.**. If $\mathbf{x_B} > \mathbf{0}$ it is called **non-degenerate basic feasible solution**. For $\mathbf{x_B} \geq \mathbf{0}$, 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 5.1. $\mathbf{x}$ *is an extreme point* $\iff$ $\mathbf{x}$ *is a basic feasible solution.*

Proof. This proof is lack of details. Denote $\mathcal{S}$ as feasible region. FIXME
($\Rightarrow$) 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 $u_i = v_i = 0$. $\mathbf{u}, \mathbf{v} \in \mathcal{S} \Rightarrow \mathbf{Au = Av = b} \Rightarrow \mathbf{A(u - v) = 0} \Rightarrow \sum_{i=1}^{n}(u_i - v_i)a_i = 0$.

•

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

5.2 The Simplex Method

5.2.1 Key to Simplex Method

Cost Coefficient

The cost coefficient can be derived from the following

$$\begin{aligned} z &= cx && (5.1)\\ &= c_Bx_B + c_Nx_N && (5.2)\\ &= c_B(B^{-1}b - B^{-1}Nx_N) + c_Nx_N && (5.3)\\ &= c_BB^{-1}b - \sum_{j\in N}(c_BB^{-1}a_j - c_j)x_j && (5.4)\\ &= c_BB^{-1}b - \sum_{j\in N}(z_j - c_j)x_j && (5.5)\end{aligned}$$

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

$$\min \quad z = z_0 - \sum_{j \in N} (z_j - c_j) x_j \tag{5.6}$$

$$\text{s.t.} \quad \sum_{j \in N} y_j x_j + x_B = \bar{b} \tag{5.7}$$

$$x_j \geq 0, j \in N \tag{5.8}$$

$$x_B \geq 0 \tag{5.9}$$

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

$$x_{B_i} = \bar{b}_i - y_{ik} x_k \geq 0 \tag{5.10}$$

Therefore we have the minimum ratio rule

$$x_k = \min_{i \in B} \{ \frac{\bar{b}_i}{y_{ik}}, y_{ik} > 0 \} \tag{5.11}$$

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

5.2.2 Simplex Method Algorithm

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

5.3 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

(5.12)

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	$-\frac{3}{11}$	$\frac{1}{11}$	$\frac{3}{11}$

(5.13)

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

$$B = \begin{pmatrix} \frac{13}{11} & \frac{3}{11} & \frac{97}{11} \\ \frac{5}{11} & \frac{2}{11} & \frac{50}{11} \\ -\frac{3}{11} & \frac{1}{11} & \frac{3}{11} \end{pmatrix} \tag{5.14}$$

Algorithm 1 Simplex Method

```
Require: Given a basic feasible solution with basis $B$
Ensure: Optimal objective value $\min z = cx$
 1: Set $\mathbf{B}$ for basic variables, $\mathbf{N}$ for nonbasic variables
 2: $\mathbf{B} \leftarrow$ all slack variables
 3: $\mathbf{N} \leftarrow$ all variables excepts slack variables
 4: for $\forall j$ do
 5:     $z_j = c_B B^{-1} a_j = 0$
 6: while $\exists z_j - c_j > 0$ do
 7:     $z_j = wa_j - c_j = c_B B^{-1} a_j - c_j$
 8:     $z_k - c_k = \max_{j \in \mathbf{N}} \{z_j - c_j\}$
 9:     $y_k = B^{-1} a_k$
10:     if $\exists y_{ik} > 0$ then
11:         $\theta_r = \min_{i \in \mathbf{B}} \{\theta_i = \frac{\bar{b}_i}{y_{ik}} : y_{ik} > 0\}$
12:         $\mathbf{B} \leftarrow \mathbf{B} \backslash \{k\}$
13:         $\mathbf{N} \leftarrow \mathbf{N} \cup \{k\}$
14:         $\mathbf{B} \leftarrow \mathbf{B} \cup \{r\}$
15:         $\mathbf{N} \leftarrow \mathbf{N} \backslash \{r\}$
16:     else
17:         Unbounded
18: $x_B^* = B^{-1} b = \bar{b}$
19: $x_N = 0$
20: $z^* = c_B B^{-1} b = c_B \bar{b} \mathbf{a_{B_k}}$
```

- 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.

$$B = \begin{pmatrix} 1 & -2 & 1 \\ 0 & 1 & -2 \\ 0 & 3 & 5 \end{pmatrix} \tag{5.15}$$

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

$$c_B = \begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix} \tag{5.16}$$

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

$$w = c_B B^{-1} = \begin{pmatrix} 0 \\ -\frac{12}{11} \\ -\frac{7}{11} \end{pmatrix} \tag{5.17}$$

5.4 Artificial Variable

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.

$$\sum_{i=1}^{n} a_i x_i \geq (or =) 0 \Rightarrow \sum_{j=1}^{n} a_i x_i + x_a = 0 \tag{5.18}$$

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.

5.4.1 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.

$$\min \quad 1x_a \tag{5.19}$$
$$\text{s.t.} \quad Ax + x_a = b \tag{5.20}$$
$$x \geq 0 \tag{5.21}$$
$$x_a \geq 0 \tag{5.22}$$

If the optimal $1x_a \neq 0$, 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}	$B^{-1}b$

(5.23)

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	$B^{-1}N$	$B^{-1}b$

(5.24)

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.

5.4.2 Big M Method

5.4.3 Single Artificial Variable

5.5 Degeneracy and Cycling

5.5.1 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.

5.5.2 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.

5.5.3 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

5.6 As a Search Algorithm

5.6.1 Improving Search Algorithm

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

$$x^t = x^{t-1} + \lambda_{t-1} d^{t-1} \tag{5.25}$$

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

5.6.2 Optimality Test

$$\begin{aligned}
z &= cx && (5.26)\\
&= \begin{bmatrix} c_B & c_N \end{bmatrix} \begin{bmatrix} x_B \\ x_N \end{bmatrix} && (5.27)\\
&= c_B x_B + c_N x_N && (5.28)\\
\text{and } \because & Ax = b && (5.29)\\
\therefore & Bx_B + Nx_N = b, x_B \geq 0, x_N \geq 0 && (5.30)\\
\therefore & x_B = B^{-1}b - B^{-1}Nx_N && (5.31)\\
z &= c_B B^{-1} b - c_B B^{-1} N x_N + c_N x_N && (5.32)
\end{aligned}$$

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

$$z - \hat{z} = \begin{bmatrix} 0 & c_N - c_B B^{-1} N \end{bmatrix} \begin{bmatrix} x_B \\ x_N \end{bmatrix} \tag{5.33}$$

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 $\hat{z}$.

5.6.3 Find Direction

Suppose we choose x_k as a candidate to pivot into Basis

$$x = \begin{bmatrix} B^{-1}b - B^{-1}a_k x_k \\ 0 + e_k x_k \end{bmatrix} = \begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix} + \begin{bmatrix} -B^{-1}a_k \\ e_k \end{bmatrix} x_k \tag{5.34}$$

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.)

5.6.4 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

$$B^{-1}b - B^{-1}a_k x_k \geq 0 \tag{5.35}$$

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.

5.6.5 Simplex Method Algorithm

5.6.6 Simplex Method Tableau

5.6.7 Simplex Method as a Search Algorithm

Chapter 6

The Simplex Method - Improved

6.1 Revised Simplex Method

6.1.1 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	$B^{-1} N$	$B^{-1} b$

(6.1)

The revised matrix:

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

(6.2)

For each pivot iteration, calculate $z_j - c_j = w a_j - c_j = c_B B^{-1} a_j - c_j, \forall j \in N$, 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

(6.3)

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

B^{-1}	RHS
w	$c_B \bar{b}$
	$\bar{b}_1$
	$\bar{b}_2$
	...
B^{-1}	$\bar{b}_r$
	...
	$\bar{b}_m$

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

(6.4)

6.1.2 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	$\times$	$(m+1)^2 + m(n-m)$
	$+$	$m(m+1) + m(n-m)$

(6.5)

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.

6.1.3 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

$$\begin{aligned} \bar{B} &= (a_{B_1}, a_{B_2}, ..., a_{B_{r-1}}, a_k, a_{B_{r+1}}, a_m) && (6.6)\\ &= (Be_1, Be_2, ..., Be_{r-1}, By_k, Be_{r+1}, ..., Be_m) && (6.7)\\ &= BT && (6.8) \end{aligned}$$

where T is

$$T = \begin{bmatrix} 1 & 0 & ... & 0 & y_{1k} & 0 & ... & 0 \\ 0 & 1 & ... & 0 & y_{2k} & 0 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 1 & y_{r-1,k} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & y_{rk} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & y_{r+1,k} & 1 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 0 & y_{mk} & 0 & ... & 1 \end{bmatrix} \tag{6.9}$$

and

$$E = T^{-1} = \begin{bmatrix} 1 & 0 & ... & 0 & \frac{-y_{1k}}{y_{rk}} & 0 & ... & 0 \\ 0 & 1 & ... & 0 & \frac{-y_{2k}}{y_{rk}} & 0 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 1 & \frac{-y_{r-1,k}}{y_{rk}} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & \frac{1}{y_{rk}} & 0 & ... & 0 \\ 0 & 0 & ... & 0 & \frac{-y_{r+1,k}}{y_{rk}} & 1 & ... & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & ... & 0 & \frac{-y_{mk}}{y_{rk}} & 0 & ... & 1 \end{bmatrix} \tag{6.10}$$

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

$$B_t^{-1} = E_{t-1}E_{t-2} \cdots E_2 E_1 I \tag{6.11}$$

Using E in calculation can simplify the product of matrix where

$$cE = c_1, c_2, ..., c_m \begin{bmatrix} 1 & 0 & ... & g_1 & ... & 0 \\ 0 & 1 & ... & g_2 & ... & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & ... & g_m & ... & 1 \end{bmatrix} \tag{6.12}$$

$$= (c_1, c_2, ..., c_{r-1}, cg, c_{r+1}, ..., c_m) \tag{6.13}$$

and

$$Ea = \begin{bmatrix} 1 & 0 & ... & g_1 & ... & 0 \\ 0 & 1 & ... & g_2 & ... & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & ... & g_m & ... & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \tag{6.14}$$

$$= \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{r-1} \\ 0 \\ a_{r+1} \\ \vdots \\ a_m \end{bmatrix} + a_r \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_{r-1} \\ g_r \\ g_{r+1} \\ \vdots \\ g_m \end{bmatrix} \tag{6.15}$$

$$= \bar{a} + a_r g \tag{6.16}$$

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

$$w = c_B B^{-1} = c_B E_{t-1} E_{t-2} ... E_2 E_1 \tag{6.17}$$

$$y_k = B^{-1} a_k = E_{t-1} E_{t-2} ... E_2 E_1 a_k \tag{6.18}$$

$$\bar{b} = B_{t+1}^{-1} b = E_t E_{t-1} E_{t-2} ... E_2 E_1 b \tag{6.19}$$

6.2 Simplex for Bounded Variables

6.2.1 Bounded Variable Formulation

$$\min \quad cx \tag{6.20}$$

$$\text{s.t.} \quad Ax = b \tag{6.21}$$

$$l \leq x \leq b \tag{6.22}$$

Reason why we don't the following formulation

$$\min \quad cx \tag{6.23}$$

$$\text{s.t.} \quad Ax = b \tag{6.24}$$

$$x - Ix_l = l \tag{6.25}$$

$$x + Ix_u = u \tag{6.26}$$

$$x \geq 0 \tag{6.27}$$

$$x_l \geq 0 \tag{6.28}$$

$$x_u \geq 0 \tag{6.29}$$

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.

6.2.2 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

$$\bar{x}_B = B^{-1}b - B^{-1}N_l x_{N_l} - B^{-1}N_u x_{N_u} \tag{6.30}$$

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.

6.2.3 Improving Basic Feasible Solution

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

$$\begin{aligned}
x_B &= B^{-1}b - B^{-1}N_l x_{N_l} - B^{-1}N_u x_{N_u} && (6.31)\\
z &= c_B x_B + c_{N_l} x_{N_l} + c_{N_u} x_{N_u} && (6.32)\\
&= c_B(B^{-1}b - B^{-1}N_l x_{N_l} - B^{-1}N_u x_{N_u}) && (6.33)\\
&\quad + c_B x_B + c_{N_l} x_{N_l} + c_{N_u} x_{N_u} && (6.34)\\
&= c_B B^{-1}b + (c_{N_l} - c_B B^{-1}N_l)x_{N_l} && (6.35)\\
&\quad + (c_{N_u} - c_B B^{-1}N_u)x_{N_u} && (6.36)\\
&= c_B B^{-1}b - \sum_{j \in J_1}(z_j - c_j)x_j - \sum_{j \in J_2}(z_j - c_j)x_j && (6.37)
\end{aligned}$$

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

$$\max\{\max_{j \in J_1}\{z_j - c_j\}, \max_{j \in J_2}\{c_j - z_j\}\} \tag{6.38}$$

to enter the basis
$\boxed{\textbf{Tip:}}$ "Most violated rule"
The minimum ratio rule is revised for bounded simplex

$$\Delta = \min\{\gamma_1, \gamma_2, u_k - l_k\} \tag{6.39}$$

$$\gamma_1 = \begin{cases} \min_{r \in J_1}\{\frac{\bar{b}_r - l_{B_r}}{y_{rk}} : y_{rk} > 0\} \\ \min_{r \in J_2}\{\frac{\bar{b}_r - l_{B_r}}{-y_{rk}} : y_{rk} < 0\} \\ \infty \end{cases} \tag{6.40}$$

$$\gamma_2 = \begin{cases} \min_{r \in J_1}\{\frac{u_{B_r} - \bar{b}_r}{-y_{rk}} : y_{rk} < 0\} \\ \min_{r \in J_2}\{\frac{u_{B_r} - \bar{b}_r}{y_{rk}} : y_{rk} > 0\} \\ \infty \end{cases} \tag{6.41}$$

$\boxed{\textbf{Tip:}}$
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.

6.3 Dual Simplex Method

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

Chapter 7

Duality, Sensitivity and Relaxation

7.1 Duality

7.1.1 Dual Formulation

For any prime problem

$$\begin{aligned} \min \quad & cx && (7.1)\\ \text{s.t.} \quad & Ax \geq b && (7.2)\\ & x \geq 0 && (7.3) \end{aligned}$$

we can have a dual problem

$$\begin{aligned} \max \quad & wb && (7.4)\\ \text{s.t.} \quad & wA \leq c && (7.5)\\ & w \geq 0 && (7.6) \end{aligned}$$

7.1.2 Mixed Forms of Duality

For the following prime problem

$$\begin{aligned} \text{P(or D)} \quad \min \quad & c_1x_1 + c_2x_2 + c_3x_3 && (7.7)\\ \text{s.t.} \quad & A_{11}x_1 + A_{12}x_2 + A_{13}x_3 \geq b_1 && (7.8)\\ & A_{21}x_1 + A_{22}x_2 + A_{23}x_3 \leq b_2 && (7.9)\\ & A_{31}x_1 + A_{32}x_2 + A_{33}x_3 = b_3 && (7.10)\\ & x_1 \geq 0 && (7.11)\\ & x_2 \leq 0 && (7.12)\\ & x_3 \quad \text{unrestricted} && (7.13) \end{aligned}$$

The dual of the problem

$$\begin{aligned} \text{D(or P)} \quad \max \quad & w_1b_1 + w_2b_2 + w_3b3 && (7.14)\\ \text{s.t.} \quad & w_1A_{11} + w_2A_{21} + w_3A_{31} \leq c_1 && (7.15)\\ & w_1A_{12} + w_2A_{22} + w_3A_{32} \geq c_2 && (7.16)\\ & w_1A_{13} + w_2A_{23} + w_3A_{33} = c_3 && (7.17)\\ & w_1 \geq 0 && (7.18)\\ & w_2 \leq 0 && (7.19)\\ & w_3 \quad \text{unrestricted} && (7.20) \end{aligned}$$

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	

7.1.3 Dual of the Dual is the Primal

For a primal problem (P)

$$(P) \quad \min \quad cx \tag{7.21}$$
$$\text{s.t.} \quad Ax \geq b \tag{7.22}$$
$$x \geq 0 \tag{7.23}$$

The dual problem (D) is

$$(D) \quad \max \quad wb \tag{7.24}$$
$$\text{s.t.} \quad wA \leq c \tag{7.25}$$
$$w \geq 0 \tag{7.26}$$

Rewrite the dual

$$\min \quad -b^T w^T \tag{7.27}$$
$$\text{s.t.} \quad -A^T w^T \geq -c^T \tag{7.28}$$
$$w^T \geq 0 \tag{7.29}$$

Find the dual of this problem

$$\max \quad x^T(-c^T) \tag{7.30}$$
$$\text{s.t.} \quad x^T(-A^T) \leq (-b^T) \tag{7.31}$$
$$x^T \geq 0 \tag{7.32}$$
$$\tag{7.33}$$

Rewrite the dual of the dual

$$(P) \quad \min \quad cx \tag{7.34}$$
$$\text{s.t.} \quad Ax \geq b \tag{7.35}$$
$$x \geq 0 \tag{7.36}$$

7.1.4 Primal-Dual Relationships

Weak Duality Property

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

$$Ax_0 \geq b, \quad x_0 \geq 0 \tag{7.37}$$

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

$$w_0 A \leq c, \quad w_0 \geq 0 \tag{7.38}$$

Therefore, we have

$$cx_0 \geq w_0 A x_0 \geq w_0 b \tag{7.39}$$

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:

$$\begin{cases} w^*(Ax^* - b) = 0 \\ (c - w^*A)x^* = 0 \end{cases} \tag{7.40}$$

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**
$\boxed{\textbf{Tip:}}$ 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

$$(c_j - \boldsymbol{w^* a_j})x_j^* = 0, \quad j = 1, ..., n \tag{7.41}$$

$$w_i^*(\boldsymbol{a^i x^*} - b_i) = 0, \quad i = 1, ..., m \tag{7.42}$$

In particular

$$x_j^* > 0 \Rightarrow \boldsymbol{w^* a_j} = c_j \tag{7.43}$$

$$\boldsymbol{w^* a_j} < c_j \Rightarrow x_j^* = 0 \tag{7.44}$$

$$w_i^* > 0 \Rightarrow \boldsymbol{a^i x^*} = b_i \tag{7.45}$$

$$\boldsymbol{a^i x^*} > b_i \Rightarrow w_i^* = 0 \tag{7.46}$$

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.

Use Dual to Solve the Primal

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.

7.1.5 Shadow Price

Shadow Price under Non-degeneracy

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

$$z = c_B B^{-1} b - \sum_{j \in N}(z_j - c_j)x_j = w^* b - \sum_{j \in N}(z_j - c_j)x_j \tag{7.47}$$

therefore

$$\frac{\partial z^*}{\partial b_i} = c_B B_i^{-1} = w_i^* \tag{7.48}$$

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.

7.2 Sensitivity

7.2.1 Change in the Cost Vector

Case 1: 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.

Case 2: Basic Variable

If c_{B_t} is replaced by $c_{B_t}^{'}$, then $z_j^{'} - c_j$ is

$$z_j^{'} - c_j = c_B^{'} B^{-1} a_j - c_j = (z_j - c_j) - (c_{B_t}^{'} - c_{B_t}) B^{-1} a_{B_t} \tag{7.49}$$

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 $c_B^{'} B^{-1} b = c_B B^{-1} b + (c_{B_t}^{'} - c_{B_t}) B^{-1} b_{B_t}$.

7.2.2 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.

7.2.3 Change in the Matrix

Case 1: 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.

Case 2: 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)

7.3 Relaxation

7.3.1 Why Rounding Can be Bad - IP Example

Rounding can be bad because the optimal of IP can be far away from optimal of LP. For example,

$$\begin{aligned}
\max \quad & z = x_1 + 0.64x_2 && (7.50)\\
\text{s.t.} \quad & 50x_1 + 31x_2 \leq 250 && (7.51)\\
& 3x_1 - 2x_2 \geq -4 && (7.52)\\
& x_1, x_2 \geq 0 \quad \text{(for LP)} && (7.53)\\
& x_1, x_2 \in Z^+ \quad \text{(for IP)} && (7.54)
\end{aligned}$$

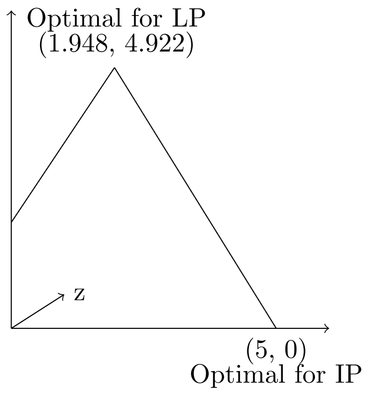


Figure 7.1: Optimal solution for LP / IP

7.3.2 Why Rounding Can be Bad - QAP example

Rounding can make the LP useless. For example, for QAP problem, the IP model is

$$\begin{aligned}
\min \quad & z = \sum_{i\in D}\sum_{s\in O} c_i s x_i s + \sum_{i\in D}\sum_{j\in D}\sum_{s\in O}\sum_{t\in O} w_{ij}^{st} y_{ij}^{st} && (7.55)\\
\text{s.t.} \quad & \sum_{i\in D} x_{is} = 1, \quad s \in D && (7.56)\\
& \sum_{s\in O} x_{is} = 1, \quad i \in D && (7.57)\\
& x_{is} \in \{0,1\}, \quad i \in D, s \in O && (7.58)\\
& y_{ij}^{st} \geq x_{is} + x_{jt} - 1, \quad i \in D, j \in D, s \in O, t \in O && (7.59)\\
& y_{ij}^{st} \geq 0, \quad i \in D, j \in D, s \in O, t \in O && (7.60)\\
& y_{ij}^{st} \leq x_{is}, \quad i \in D, j \in D, s \in O, t \in O && (7.61)\\
& y_{ij}^{st} \leq x_{jt}, \quad i \in D, j \in D, s \in O, t \in O && (7.62)
\end{aligned}$$

We can get the optimal solution for LP supposing $\forall i, s \quad x_{is} \in [0,1]$

$$\begin{aligned}
& x_{is} = \frac{1}{|D|}, \quad i \in D, s \in O && (7.63)\\
& y_{ij}^{st} = 0, \quad i \in D, j \in D, s \in O, t \in O && (7.64)
\end{aligned}$$

7.3.3 IP and Convex Hull

For IP problem

$$Z_{IP} \quad \max \quad z = cx \tag{7.65}$$
$$\text{s.t.} Ax \le b \tag{7.66}$$
$$x \in Z^n \tag{7.67}$$

In feasible region $S = \{x \in Z^n, Ax \le b\}$, the optimal solution $Z_{IP} = \max\{cx : x \in S\}$.
Denote $conv(S)$ as the convex hull of S then

$$Z_{IP}(S) = Z_{IP}(conv(S)) \tag{7.68}$$

7.3.4 Local Optimal and Global Optimal

Let

$$Z_s = \min\{f(x) : x \in S\} \tag{7.69}$$
$$Z_t = \min\{f(x) : x \in T\} \tag{7.70}$$
$$S \subset T \tag{7.71}$$

then

$$Z_t \le Z_s \tag{7.72}$$

Notice that if $x_T^* \in S$ then $x_S^* = x_T^*$, to generalized it,
We have

$$\begin{cases} x_T^* \in \arg\min\{f(x) : x \in T\} \\ x_T^* \in S \end{cases} \tag{7.73}$$
$$\Rightarrow x_T^* \in \arg\min\{f(x) : x \in S\} \tag{7.74}$$

Especially for IP, we can take the LP relaxation as T and the original feasible region of IP as S, therefore, if we find an optimal solution from LP relaxation T which is also a feasible solution of S, then it is the optimal solution for IP (S)

7.3.5 LP Relaxation

To perform the LP relaxation, we expand the feasible region

$$x \in \{0,1\} \to x \in [0,1] \tag{7.75}$$
$$y \in Z^+ \to y \ge 0 \tag{7.76}$$

If we have $Z_LP(s) = conv(s)$ then

$$LP(s) : x \in R_+^n : Ax \le b \tag{7.77}$$

The closer $LP(s)$ is to $conv(s)$ the better. Interestingly, we only need to know the convex in the direction of c.
There are several formulation problem have the property of $Z_{LP}(s) = conv(s)$, such as:
- Assignment Problem
- Spawning Tree Problem
- Max Flow Problem
- Matching Problem

Chapter 8

Polynomial-Time Algorithm for Linear Programming

Part III

Integer and Combinatorial Programming

Chapter 9

Polyhedral Theory

9.1 Polyhedral and Dimension

9.1.1 Polyhedral

Definition 9.1.1 (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 9.1.2 (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 9.1.3 (cone)**.** $C \subseteq \mathbb{R}^n$ is a **cone** if $x \in C$ implies $\lambda x \in C, \forall \lambda \in \mathbf{R}_+$

9.1.2 Dimension of Polyhedral

Definition 9.1.4 (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 9.1.5 (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 9.1.6 (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 9.0.1. *Every nonempty polyhedron has at least one inner point.*

Corollary 9.0.2. *A polyhedron has an interior point iff P is full-dimensional, i.e., there is no equality constraint.*

9.2 Face and Facet

9.2.1 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) **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 $\max\{\pi x | x \in P\} = \pi_0\}$

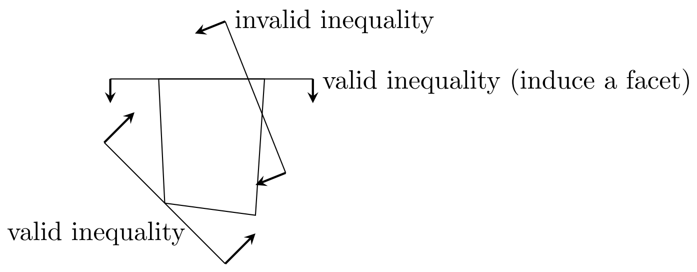


Figure 9.1: Example of valid/invalid inequality

- If the face F is nonempty, we say it **supports** P
- Let P be a polyhedron with equality set $M^{=}$. If

$$F = \{x \in P | \pi^T x = \pi_0\} \tag{9.1}$$

is not empty, then F is a polyhedron. Let

$$M^{=} \subseteq M_F^{=}, M_F^{\leq} = M \setminus M_F^{=} \tag{9.2}$$

then

$$F = \{x | a_i^T x = b_i, \forall i \in M_F^{=}, a_i^T x \leq b_i, \forall i \in M_f^{\leq}\} \tag{9.3}$$

9.2.2 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

9.2.3 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 $\sum_i a_i x_i = b_i$

9.2.4 Domination

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

$$\begin{cases} \Pi \geq \mu M, \mu > 0 \\ \Pi_0 \leq \mu M_0, \mu > 0 \\ (\Pi, \Pi_0) \neq (M, M_0) \end{cases} \tag{9.4}$$

Chapter 10

Branch and Bound

10.1 LP based Branch and Bound

10.1.1 Idea of Divide and Conquer

For each iteration, divide the feasible region of LP into two feasible parts and an infeasible part, solve the LP in those parts.

In this iteration, the original feasible region have been partition into three parts, where S_2 is infeasible for IP

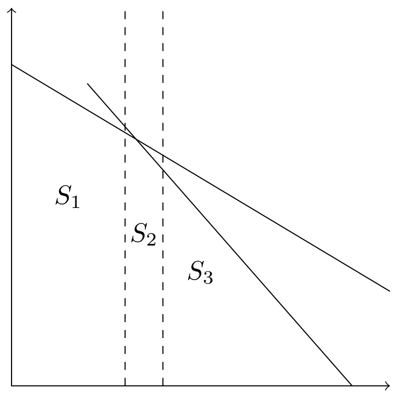


Figure 10.1: Divide and Conquer

because there is not integer point in it. We continue the iteration for S_1 and S_2. Each partition is suppose to give a new upper bound / lower bound and reduce the infeasible space.
If the temp optimal integer in S_1 is larger than the LP relaxation in S_3, we can cut S_3.
For each iteration, we use dual simple method, for the following two reasons:
- We can process new constraint very fast
- Always gives us a valid bound.

10.1.2 Relation Between LP Relaxation and IP

Let

$$Z_{IP} = \max_{x \in S} cx, \quad \text{where } s \text{ is a set of integer solutions} \tag{10.1}$$

$$Z_{LP} = \max cx, \quad \text{the LP relaxation of IP} \tag{10.2}$$

then

$$Z_{IP} = \max_{1\leq i\leq k} \{\max_{x\in S_i} cx\} \tag{10.3}$$

$$\text{iff} \quad S = \bigcup_{1\leq i\leq k} S_i \tag{10.4}$$

Notice that S_i don't need to be disjointed.
Important! (For maximization problem)
- Any feasible solution provides a lower bound L, which is also the *Prime Bound*

$$\hat{x} \in S \rightarrow Z_{IP} \geq c\hat{x} \tag{10.5}$$

- After branching, solving the LP relaxation over sub-feasible-region S_i produces an upper bound, which is also the *Dual Bound*, on each sub-problem
- If $u(S_i) \leq L$, remove S_i
- LP can produce the first upper bound, but there might be possible to find other upper bound with other method (e.g. Lagrangian relaxation)

10.1.3 LP feasibility and IP(or MIP) feasibility

Solve the LP relaxation, one of the following things can happen
- LP is infeasible $\rightarrow$ MIP is infeasible
- LP is unbounded $\rightarrow$ MIP is infeasible or unbounded
- LP has optimal solution $\hat{x}$ and $\hat{x}$ are integer ($\hat{x} \in S$), $\rightarrow Z_{IP} = Z_{LP}$
- LP has optimal solution $\hat{x}$ and $\hat{x}$ are not integer ($\hat{x} \notin S$), now defines a new upper bound, $Z_{LP} \geq Z_{IP}$

If the first three happens, stop, if the fourth happens, we branch and recursively solve the sub-problems.

10.2 Terminology in Branch and Bound

- If we picture the sub-problems, they will form a **search tree** (typically a binary tree)
- Each node in the search tree is a **sub-problem**
- Eliminating a node is called **pruning**, we also stop considering its children
- A sub-problem that has not being processed is called a **candidate**, we keep a list of candidates

10.3 Bounding

Notice! this section is for maximization, if it is for minimization, reverse upper bound and lower bound.

10.3.1 Upper Bound

- Upper bound it the Prime bound. which means it has to be a feasible solution
- Some methods to get an upper bound:
- Rounding
- Heuristic
- Meta-heuristic

10.3.2 Lower Bound

- Lower Bound is the Dual bound,we can use LP relaxation to get it
- The tighter the better, LP is better

Algorithm 2 Branch and Bound

```
1: find a feasible solution as the initial Lower bound $L$
2: put the original LP relaxation in candidate list $S$
3: while $S \neq \emptyset$ do
4:     select a problem $\hat{S}$ from $S$
5:     solve the LP relaxation of $\hat{S}$ to obtain $u(\hat{S})$
6:     if LP is infeasible then
7:         $\hat{S}$ pruned by infeasibility
8:     else if LP is unbounded then
9:         $\hat{S}$ pruned by unboundness or infeasibility
10:    else if LP $u(\hat{S}) \leq L$ then
11:        $\hat{S}$ pruned by bound
12:    else if LP $u(\hat{S}) > L$ then
13:        if $\hat{x} \in S$ then
14:            $u(\hat{S})$ becomes new $L$, $L = u(\hat{S})$
15:        else if $\hat{x} \notin S$ then
16:            branch and add the new sub-problems to $S$
17:            if LP $u(\hat{S})$ is at current best upper bound then
18:                set $U = u(\hat{S})$
19: if Lower bound exists then
20:     find the optimal at $L$
21: else
22:     Infeasible
```

10.4 Branch and Bound Algorithm

10.5 The goal of Branching

- Divide the problem into easier sub-problems
- We want to chose the branching variables that minimize the sum of the solution times of the sub-problems
- If after branching the $u(S_i)$ changes a lot,
- I can find a good L first
- The branch may get worse than the current bound first
- Instead of solving the potential two branches for all candidates to optimality, solve a few iterations of the dual simplex, each iteration of pivoting yields an upper bound.

10.6 Choose Branching Variables

10.6.1 The Most Violated Integrality constraint

Pick the j of which $x_j - \lfloor \hat{x_j} \rfloor$ is closer to 0.5

10.6.2 Strong Branching

Select a few candidates (K), create all sub-problems for each of these variables, run a few dual simplex iterations to see the improved bounds, select the variable with the best bounds.
for variable $x_j \in K$, we branch and do a few iterations to find two reductions of gaps, i.e. D_j^+ and D_j^-,

10.6.3 pseudo-cost Branching

Pseudo-cost is an estimate of per-unit change in the objective function, for each variable

$$\begin{cases} P_j^+, & \text{bound reduction if rounded up} \\ P_j^-, & \text{bound reduction if rounded down} \end{cases} \tag{10.6}$$

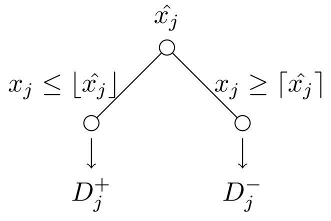

$\hat{x_j}$
$x_j \leq \lfloor \hat{x_j} \rfloor$
$x_j \geq \lceil \hat{x_j} \rceil$
D_j^+
D_j^-

Figure 10.2: Strong Branching

define $f_j = x_j - \lfloor x_j \rfloor$

$$\begin{cases} D_j^+ = P_j^+(1 - f_j) \\ D_j^- = P_j^- f_j \end{cases} \tag{10.7}$$

10.7 Choose the Node to Branch

10.7.1 Update After Branching

For those variables in K find the
- $\max\{\min\{D_j^+, D_j^-\}\}$, or
- $\max\{\max\{D_j^+, D_j^-\}\}$, or
- $\max\{\frac{D_j^+ + D_j^-}{2}\}$, or
- $\max\{\alpha_1 \min\{D_j^+, D_j^-\} + \alpha_2 \max\{D_j^+, D_j^-\}\}$

to branch.

10.7.2 Branch on Important Variables First

Branch on variables that affects many decisions.

10.7.3 Some Search Strategy

- Best Bound First: select the node with the largest bound (good for closing the gap)
- Deep First: Good for finding Lower bound and easier to do dual simplex
- Mix: Start with "Deep First" until we find a good bound and do "Best Bound First"

10.8 Types of Branching

10.8.1 Traditional Branching

For $\hat{x} \notin S$, $\exists j \in N$ such that

$$\hat{x_j} - \lfloor \hat{x_j} \rfloor > 0 \tag{10.8}$$

Create two sub-problems

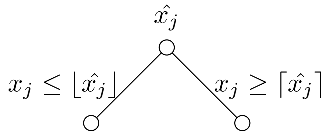

$\hat{x_j}$
$x_j \leq \lfloor \hat{x_j} \rfloor$
$x_j \geq \lceil \hat{x_j} \rceil$

Figure 10.3: Traditional Branching

10.8.2 Constraint Branching

Use parallel constraints to branch, e.g.

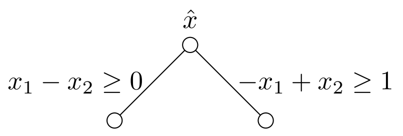


Figure 10.4: Traditional Branching

10.8.3 SOS

For SOS1, For SOS2 (using the first definition),

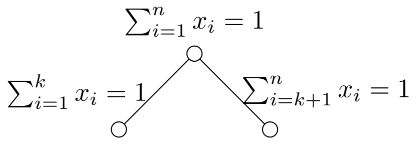


Figure 10.5: Traditional Branching

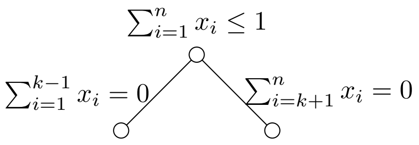

$\sum_{i=1}^{n} x_i \leq 1$
$\sum_{i=1}^{k-1} x_i = 0$
$\sum_{i=k+1}^{n} x_i = 0$

Figure 10.6: Traditional Branching

10.8.4 GUB

This is where $x_i \in \{0, 1\}$, at most one variable can be 1,

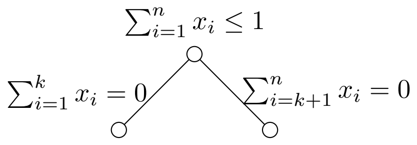


Figure 10.7: Traditional Branching

10.8.5 Ryan-Foster

Ryan-Foster is for Set covering problem. The typical model is

$$\min \sum_{i \in C} x_i \tag{10.9}$$

$$\text{s.t.} \sum_{i \in C} a_{ij} x_i \geq 1, \quad \forall j \in U \tag{10.10}$$

$$x_i \in \{0, 1\}, \quad \forall i \in C \tag{10.11}$$

Observation For any fractional solution, there are at least two elements (i, j) so that i and j are both partially covered by the same set S, but there is another set T that only covers i

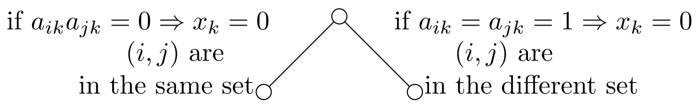

if $a_{ik}a_{jk} = 0 \Rightarrow x_k = 0$
(i, j) are
in the same set
if $a_{ik} = a_{jk} = 1 \Rightarrow x_k = 0$
(i, j) are
in the different set

Figure 10.8: Traditional Branching

Chapter 11

Branch and Cut

11.1 Separation Algorithm

Basic idea is to separate the feasible region so that the current ”solution” (which is an fractional solution) is not included in the feasible region.

11.1.1 Vertices Packing

The current solution is $\bar{x} \in [0,1]^n$, we have two options to do the separation:
Option 1 - find the maximum clique:
(This approach is as hard as the original problem)
denote

$$y_i = \begin{cases} 1, & \text{if } i \in C \\ 0, & \text{otherwise} \end{cases} \tag{11.1}$$

Find the maximum clique via:

$$\max \quad \sum \bar{x}_i y_i \tag{11.2}$$

$$\text{s.t.} \quad y_i + y_j \leq 1, \forall \{i,j\} \notin E \tag{11.3}$$

Option 2 - Heuristic:

Algorithm 3 Heuristic method to find a clique

```
1: find $v = \text{argmax}_{i \in V}\{\bar{x_i}\}, C = \{v\}$
2: **while** $u \in \text{argmax}_{i \in \cap_{i \notin C} N_{(i,j)} \notin C}\{\bar{x_1}\}$ exists **do**
3:     C.add(u)
4: return C
```

If $\sum_{i \in C} \bar{x}_i > 1$ then add cut $\sum_{i \in C} x_i \leq 1$

11.1.2 TSP

When we have a solution, i.e. $\bar{x}$, perform the sub-tour searching algorithm, if there exists any sub-tour, add the corresponded constraint. That is the separation.

11.2 Optional v.s. Essential Inequalities

11.2.1 Valid (Optional) Inequalities

See Figure 11.1

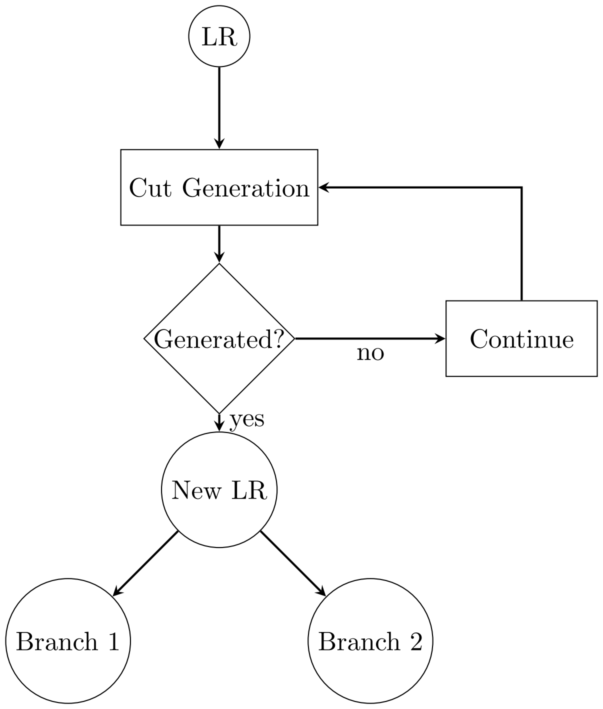


Figure 11.1: Branch and Cut for Optional Inequality

11.2.2 Essential Inequalities (Lazy Cuts)

See Figure 11.2

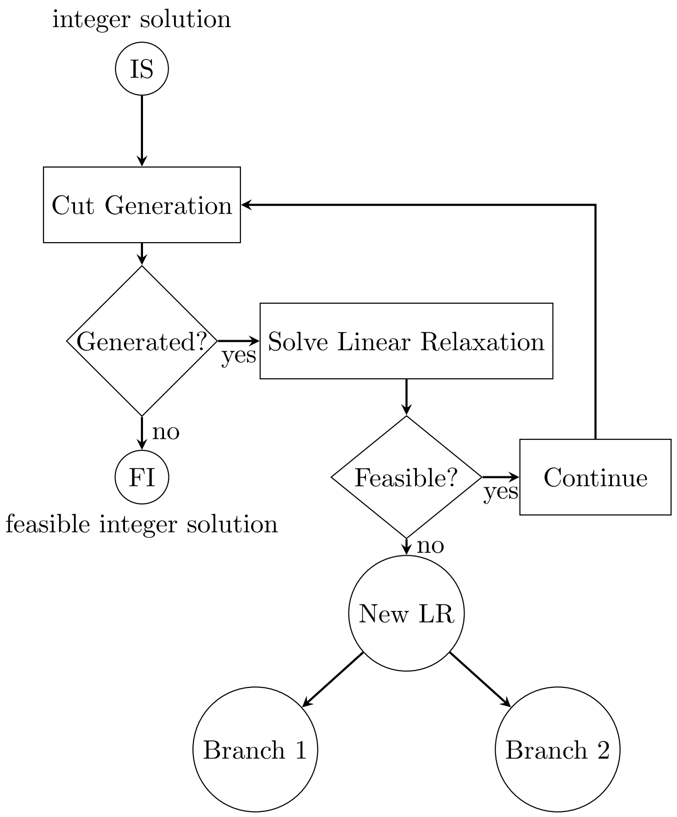


Figure 11.2: Branch and Cut for Essential Inequality

11.3 Chvatal-Gomory Cut

11.3.1 Chvatal-Gomory Rounding Procedure

For $x = P \cap \mathbb{Z}^n_+$, where $P = \{x \in \mathbb{R}^n_+ | Ax \leq b\}$, A is an mxn matrix with columns $\{a_1, ..., a_n\}$ and $u \in \mathbb{R}^n_+$
- The inequality

$$\sum_{j=1}^{n} u a_j x_j \leq ub \tag{11.4}$$

is valid
- Therefore the inequality

$$\sum_{j=1}^{n} \lfloor u a_j \rfloor x_j \leq ub \tag{11.5}$$

is valid
- Furthermore, the inequality

$$\sum_{j=1}^{n} \lfloor u a_j \rfloor x_j \leq \lfloor ub \rfloor \tag{11.6}$$

is valid.

11.3.2 Gomory Cutting Plane

For a IP problem

$$\begin{aligned} \max \quad & cx && (11.7) \\ \text{s.t.} \quad & Ax = b && (11.8) \\ & x \in \mathbb{B}^n && (11.9) \end{aligned}$$

let $\bar{x}$ be an optimal basic solution for the LR of P.

$$\bar{x} = \begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix} = \begin{bmatrix} x_B \\ x_N \end{bmatrix} \tag{11.10}$$

We have

$$\begin{aligned} & Bx_B + Nx_N = b && (11.11) \\ \Rightarrow \quad & x_B + B^{-1}Nx_N = B^{-1}b && (11.12) \\ \Rightarrow \quad & x_B + [\bar{a}_1, \bar{a}_2, ...]x_N = \bar{b} && (11.13) \\ \Rightarrow \quad & x_i + \sum_{j \in NB} \bar{a}_{ij} x_j = \bar{b}_i \quad \text{(for the } i\text{th row)} && (11.14) \end{aligned}$$

Assume that $x_i \in \{0, 1\}$, use CG-Procedure

$$x_i + \sum_{j \in NB} \lfloor \bar{a}_{ij} \rfloor x_j \leq \lfloor \bar{b}_i \rfloor \tag{11.15}$$

is a valid constraint for P, furthermore,

$$(\bar{b}_i - \sum_{j \in NB} \bar{a}_{ij} x_j) + \sum_{j \in NB} \lfloor \bar{a}_{ij} \rfloor x_j \leq \lfloor \bar{b}_i \rfloor \tag{11.16}$$

Move the item, we get a new Gomory Cutting Plane

$$\sum_{j \in NB} (\bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor) x_j \geq \bar{b}_i - \lfloor \bar{b}_i \rfloor \tag{11.17}$$

Add this inequality to the LR, use the dual simplex method to do one pivot, we get a new solution. Use Gomory cutting plane iteratively and we can find the optimal solution for IP.

Chapter 12

Packing and Matching

12.1 Vertices Packing and Matching Formulation

Given a graph $G = (V, E)$, with $|V| = n$. A vertices packing solution is that no two neighboring vertices can be chosen at the same time.

$$PACK(G) = \{x \in \mathbb{B}^n | x_i + x_j \leq 1, \forall (i,j) \in E\} \tag{12.1}$$

Example. The following is an example:
The PACK of this graph is

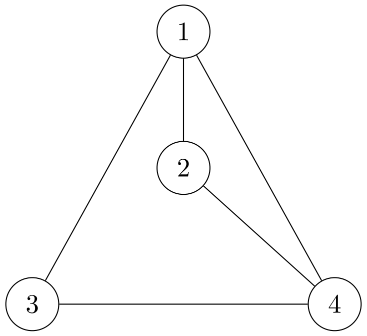


Figure 12.1: Example of vertices packing problem

$$PACK = conv\left(\begin{pmatrix}0\\0\\0\\0\end{pmatrix}, \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\end{pmatrix}, \begin{pmatrix}0\\1\\1\\0\end{pmatrix}\right) \tag{12.2}$$

Given a graph $G = (V, E)$, denote $\delta(i)$ as the set of all the edges introduced to vertice $i \in V$. A matching solution is that no two edges introduced to the same vertice can be chosen at the same time.

$$MATCH(G) = \{\sum_{e \in \delta(i)} x_e \leq 1 | i \in V\} \tag{12.3}$$

12.2 Dimension of PACK(G)

The dimension of PACK, i.e. $dim(PACK(G))$ is (full-dimensional)

$$dim(PACK(G)) = |V| \tag{12.4}$$

To prove that $dim(PACK(G)) = |V|$, we need to find $|V| + 1$ affinely independent vectors.

Proof.

$$rank\left(\begin{bmatrix}0 & I_{|V|}\\ 1 & 1\end{bmatrix}\right) = |V| + 1 \tag{12.5}$$

Therefore, in PACK, $rank(A^=, b^=) = 0$ □

12.3 Clique

- A **clique** is a subset of a graph that in the clique every two vertices linked with each other (complete sub-graph).
- A **maximum clique** is a clique that any other vertice can not form a clique with all the points in this clique.

12.4 Inequalities and Facets of conv(VP)

Example:

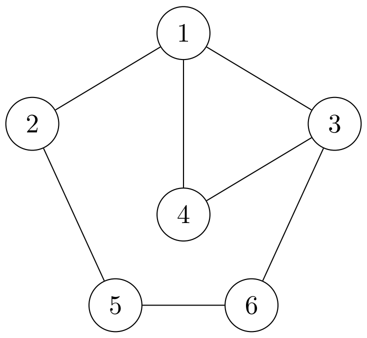


Figure 12.2: Example

12.4.1 Type 1 (Nonnegative Constraints)

$x_i \geq 0$ induce facets.

Proof:

$$rank\left(\begin{bmatrix}0 & 0\\ 0 & I_{|V|}\end{bmatrix}\right) = |V| + 1 \tag{12.6}$$

12.4.2 Type 2 (Neighborhood Constraints)

$x_i + x_j \leq 1$ is a valid constraint, but it **DOES NOT** always induce facet.

12.4.3 Type 3 (Odd Hole)

H is an odd hole if it contains circle of k nodes, such that k is odd and there is no cords. e.g. {1, 2, 5, 6, 3}. Then, the following inequality is valid,

$$\sum_{i \in H} x_i \leq \frac{|H| - 1}{2} \tag{12.7}$$

Odd Hole inequality **DOES NOT** always induce facets.
This inequality can be derived from Gomory cut.

12.4.4 Type 4 (Maximum Clique)

C is a maximum clique, then the following inequality is valid and induce a facet,

$$\sum_{i \in C} x_i \leq 1 \tag{12.8}$$

Proof:
First, if $C = V$

$$rank\,([I]) = |C| = |V| \tag{12.9}$$

Second, if C is a subset of V, for each vertice in $V \setminus C$, there should be at least one vertice in C that is not linked with it. Therefore for each vertice in C we can find a packing.

12.5 Gomory Cut in Set Covering

Consider a graph $G = (V, E)$, the covering problem is

$$\sum_{e \in \delta(i)} x_e \leq 1, i \in V, x_e \in \{0,1\}, e \in E \tag{12.10}$$

For $T \subset V$, denote $\delta(i)$ as all edges induce to $i \in V$, denote $E(T) \subset E$ as all the edges linked between $(i,j), i \in T, j \in T$, therefore we have

$$\sum_{i \in T} \sum_{e \in \delta(i)} x_e \leq |T| \tag{12.11}$$

For edges linking $i \in T, j \in T$, count them twice, for edges linking $i \in T, j \notin T$, count them once.We can have a new constraint

$$2 \sum_{e \in E(T)} x_e + \sum_{e \in \delta(V \setminus T, T)} x_e \leq |T| \tag{12.12}$$

Perform the Gomory Cut, the following constraint is a valid:

$$\sum_{e \in E(T)} x_e \leq \lfloor \frac{|T|}{2} \rfloor \tag{12.13}$$

Chapter 13

Knapsack Problem

13.1 Knapsack Problem Formulation

Consider the knapsack set KNAP

$$conv(KNAP) = conv(\{x \in \mathbb{B}^n | \sum_{j \in N} a_j x_j \leq b\}) \tag{13.1}$$

in where
- $N = \{1, 2, ..., n\}$
- With out lost of generality, assume that $a_j > 0, \forall j \in N$ and $a_j < b, \forall j \in N$

13.2 Valid Inequalities for a Relaxation

For $P = \{x \in \mathbb{B}^n | Ax \leq b\}$, each row can be regard as a Knapsack problem, i.e. for row i

$$P_i = \{x \in \mathbb{B}^n | a_i^T x \leq b_i\} \tag{13.2}$$

is a relaxation of P, therefore,

$$P \subseteq P_i, \forall i = 1, 2, ..., m \tag{13.3}$$

$$P \subseteq \cap_{i=1}^m P_i \tag{13.4}$$

So any inequality valid for a relaxation of an IP is also valid for IP itself.

13.3 Cover and Extended Cover

A set $C \subseteq N$ is a cover if $\sum_{j \in C} a_j > b$, a cover C is minimal cover if

$$C \subseteq N | \sum_{j \in C} a_j > b, \sum_{j \in C \setminus k} a_j < b, \forall k \in C \tag{13.5}$$

For a cover C, we can have the cover inequality

$$\sum_{j \in C} x_j \leq |C| - 1 \tag{13.6}$$

The inequality is trivial considering the pigeonhole principle.
$C \subseteq N$ is a minimal cover, then $E(C)$ is defined as following:

$$E(C) = C \cup \{j \in N | a_j \geq a_i, \forall i \in C\} \tag{13.7}$$

is called an extended cover. Then we have,

$$\sum_{i \in E(C)} x_i \leq |C| - 1 \text{ dominates } \sum_{i \in C} x_i \leq |C| - 1 \tag{13.8}$$

and

$$\sum_{i\in E(C)} x_i \le |C|-1 \text{ dominates } \sum_{i\in E(C)} x_i \le |E(C)|-1 \tag{13.9}$$

Hereby we need to prove that $\sum_{i\in E(C)} x_i \le |C|-1$ is valid, by contradiction.

$\boxed{\textbf{Proof:???}}$ Suppose $x^R \in KNAP$, R is a feasible solution, Where

$$x_j^R = \begin{cases} 1, & \text{if } j \in R \\ 0, & \text{otherwise} \end{cases} \tag{13.10}$$

Then

$$\sum_{j\in E(C)} x_j^R \ge |C| \Rightarrow |R \cap E(C)| \ge |C| \tag{13.11}$$

therefore

$$\sum_{j\in R} a_j \ge \sum_{j\in R\cap E(C)} a_j \ge \sum_{j\in C} a_j > b \tag{13.12}$$

which means R is a cover, it is contradict to $\sum_{j\in E(C)} x_j^R \ge |C|$ so $x^R \notin KNAP$

13.4 Dimension of KNAP

$conv(KNAP)$ is full dimension, i.e. $dim(conv(KNAP)) = n$.

$\boxed{\textbf{Proof:}}$ $0, e_j, \forall j \in N$ are $n+1$ affinely independent points in $conv(KNAP)$

13.5 Inequalities and Facets of conv(KNAP)

13.5.1 Type 1 (Lower Bound and Upper Bound Constraints):

- $x_k \ge 0$ is a facet of $conv(KNAP)$

$\boxed{\textbf{Proof:}}$ $0, e_j, \forall j \in N \setminus k$ are n affinely independent points that satisfied $x_k = 0$

- $x_k \le 1$ is a facet iff $a_j + a_k \le b, \forall j \in N \setminus k$

$\boxed{\textbf{Proof:}}$ $e_k, e_j + e_k, \forall j \in N \setminus k$ are n affinely independent points that satisfied $x_k = 1$

13.5.2 Type 2 (Extended Cover)

Order the variables so that $a_1 \ge a_2 \ge \cdots \ge a_n$, therefore $a_1 = a_{max}$
Let C be a cover with $\{j_1, j_2, \ldots, j_r\}$ where $j_1 < j_2 < \cdots < j_r$ so that $a_{j_1} \ge a_{j_2} \ge \cdots \ge a_{j_r}$
Let $p = \min\{j | j \in N \setminus E(C)\}$
Then

$$\sum_{j\in E(C)} x_j \le |C|-1 \tag{13.13}$$

is a facet of $conv(KNAP)$ if

- $C = N$

$\boxed{\textbf{Proof:}}$

$$R_k = C \setminus k, \forall k \in C = N \setminus k, \forall k \in N \tag{13.14}$$

have $|N|$ affinely independent vectors

- $E(C) = N$ and $\sum_{j\in C\setminus\{j_1,j_2\}} a_j + a_{max} \le b$

$\boxed{\textbf{Proof:}}$ (j_1, j_2 are two heaviest elements in C)

$$S_k = C \setminus \{j_1, j_2\} \cup \{k\}, \forall k \in E(C) \setminus C \tag{13.15}$$

$R_k \cup S_k$ have $|C| + |E(C) \setminus C| = |E(C)| = |N|$ affinely independent vectors

- $C = E(C)$ and $\sum_{j\in C\setminus j_1} a_j + a_p \le b$)

$\boxed{\textbf{Proof:}}$ (j_1 is the heaviest element in C, k is the lightest element outside extended cover)

$$T_k = C \setminus j_i \cup \{k\}, \forall k \in N \setminus E(C) \tag{13.16}$$

$R_k \cup T_k$ have $|N \setminus E(C)| + |E(C)| = |N \setminus C| + |C| = |N|$ affinely independent vectors
- $C \subset E(C) \subset N$ and $\sum_{j \in C \setminus \{j_1, j_2\}} a_j + a_{max} \leq b$ and $\sum_{j \in C \setminus j_1} a_j + a_p \leq b$

$\boxed{\textbf{Proof:}}$ $S_k \cup T_k$ have $|E(C) \setminus C| + |N \setminus E(C)| = |N|$ affinely independent vectors

13.6 Lifting

13.6.1 Up Lifting

For KNAP problem

$$KNAP = \{x \in \mathbb{B}^n | \sum_j a_j x_j \leq b\} \tag{13.17}$$

For $P = conv(KNAP)$ denote

$$P_{k_1,k_2,\ldots,k_m} \tag{13.18}$$
$$= conv(KNAP \cap \{x \in \mathbb{B}^n | x_{k_1} = x_{k_2} = \cdots = x_{k_m} = 0\}) \tag{13.19}$$

Therefore

$$P_{k_1,k_2,\ldots,k_m} \tag{13.20}$$
$$= conv(KNAP \cap \{x \in \mathbb{B}^n | \sum_{j \in N \setminus \{k_1,k_2,\ldots,k_m\}} a_j x_j \leq b\}) \tag{13.21}$$

The $C = N$ cover inequality for $P_{k_1,k_2,\ldots,k_m}$ implies

$$\sum_{j \in N \setminus \{k_1,k_2,\ldots,k_m\}} x_j \leq n - m - 1 \tag{13.22}$$

is a facet of $P_{k_1,k_2,\ldots,k_m}$
The lifting process is to find a facet for $P_{k_1,k_2,\ldots,k_{m-1}}$ using facet of $P_{k_1,k_2,\ldots,k_m}$, i.e. find α_m for the following constraint to be a facet.

$$\alpha_m x_m + \sum_{j \in N \setminus \{k_1,k_2,\ldots,k_m\}} x_j \leq n - m - 1 \tag{13.23}$$

If $x_m = 0$, $\alpha_m \geq 0$,
If $x_m = 1$, $\alpha_m \leq (n - m + 1) - \gamma$ where

$$\gamma = \max\{\sum_{j \in N \setminus \{k_1,k_2,\ldots,k_m\}} x_j | x \in P_{k_1,k_2,\ldots,k_{m-1}}, x_m = 1\} \tag{13.24}$$
$$= \max\{\sum_{j \in N \setminus \{k_1,k_2,\ldots,k_m\}} x_j | \sum_{j \in N \setminus \{k_1,k_2,\ldots,k_m\}} a_j x_j \leq b - a_m\} \tag{13.25}$$

Then let $\alpha_m = n - m + 1 - \gamma$, we uplifted a constraint. Repeat this procedure for $\{k_1, k_2, \ldots, k_m\}$ and finally we can find a family of facets for $conv(KNAP)$

13.6.2 Down Lifting

Similar to up lifting, we can perform the lifting in a different way.
Denote

$$P'_{k_1,k_2,\ldots,k_m} \tag{13.26}$$
$$= conv(KNAP \cap \{x \in \mathbb{B}^n | x_{k_1} = x_{k_2} = \cdots = x_{k_m} = 1\}) \tag{13.27}$$

13.7 Separation of a Cover Inequality

$C \in N$ is a cover if $\sum_{i\in C} a_i > b$, let C be a minimal cover

$$\sum_{i\in C} x_i \le |C| - 1 \tag{13.28}$$

$$\Rightarrow \quad |C| - \sum_{i\in C} x_i = \sum_{i\in C}(1 - x_i) \ge 1 \tag{13.29}$$

$$\tag{13.30}$$

let $\bar{x}$ be a fractional solution of $\{\sum_{i\in N} a_i x_i \le b, x_i \in [0,1], i \in N\}$, find a cover C of which $\sum_{i\in C}(1-\bar{x}_i) < 1$
Decision variable:

$$y_i = \begin{cases} 1, & \text{if } i \in C \\ 0, & \text{otherwise} \end{cases} \tag{13.31}$$

$$\min \quad \sum_{i\in N}(1-\bar{x}_i)y_i = z \tag{13.32}$$

$$\text{s.t.} \quad \sum_{i\in N} a_i y_i \ge b + 1 \tag{13.33}$$

$$y_i \in \{0,1\}, i \in N \tag{13.34}$$

if $z < 1$, then the cover cut associated with y is violation by $\bar{x}$

Chapter 14

Network Flow Problem

(Network Flow Problem is a special type of IP problem, its linear relaxation is the convex hull of the original problem.)

14.1 Shortest Path Problem

A graph $G = (A, N)$ is a directed graph
Denote:

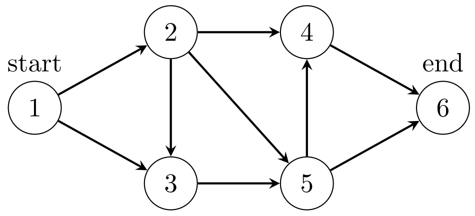


Figure 14.1: Example of directed graph

$$x_{ij} = \begin{cases} 1, & \text{if goes from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases} \tag{14.1}$$

The shortest path problem can be formulated as the following:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{14.2}$$

$$\sum_{i\in N\setminus(\{S\}\cup\{E\}),(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = 0 \tag{14.3}$$

$$\sum_{i=\{S\},(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = 1 \tag{14.4}$$

$$\sum_{i=\{E\},(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = -1 \tag{14.5}$$

$$x_{ij} \in [0,1], (i,j) \in A \tag{14.6}$$

Although initially $x_{ij} \in [0,1]$, in the optimized solution, $x \in \{0,1\}$.

14.2 Maximum Flow Problem

$$\min \sum_{(i,j)\in A} F \tag{14.7}$$

$$\sum_{i\in N\backslash(\{S\}\cup\{E\}),(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = 0 \tag{14.8}$$

$$\sum_{i=\{S\},(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = F \tag{14.9}$$

$$\sum_{i=\{E\},(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = -F \tag{14.10}$$

$$l_{ij} \leq x_{ij} \leq u_{ij}, (i,j) \in A \tag{14.11}$$

In where F means the flow from source to target.

14.3 Minimum Cost Flow

The shortest path problem is a special case of Minimum Cost Flow Problem, which can be formulated as the following:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{14.12}$$

$$\sum_{i\in N\backslash(\{S\}\cup\{E\}),(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = 0 \tag{14.13}$$

$$\sum_{i=\{S\},(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = 1 \tag{14.14}$$

$$\sum_{i=\{E\},(i,j)\in A} x_{ij} - \sum_{j\in N,(i,j)\in A} x_{ji} = -1 \tag{14.15}$$

$$x_{ij} \in [0,1], (i,j) \in A \tag{14.16}$$

14.4 Unimodularity

14.4.1 Unimodular Matrix and Total Unimodular Matrix

- A unimodular matrix M is a squared matrix, where $det(M) = 1$ or -1.
- Total unimodular matrix is a matrix where all its sub-matrices are unimodular matrix.

14.4.2 Importance of Unimodular Matrix

Let $M_{m\times m}$ be a unimodular matrix, if $b \in \mathbb{Z}^m$, the solution for $Mx = b$ is always integer.
$\boxed{\textbf{Proof:}}$ By Cramer's Rule

$$x_i = \frac{detM_i}{detM} \tag{14.17}$$

in which M_i is matrix M replace ith column with b. Therefore $det(M_i)$ is integer. Also, $det(M) \neq 0$, so $det(M) = 1$ or $det(M) = -1$. Proved.

14.4.3 Structures of Total Unimodular Matrix

Structure 1: Matrix M that has only 1, -1, 0 enters and each column has at most 2 non-zeros is a TU matrix if it satisfies the following conditions:
We can split the rows in to tops and bottoms, such that for all columns j having 2 non-zeros
- If the non-zeros have the same sign, then one value should be in top and the other should be in bottom

- If the non-zeros have the different sign, then both of them should be in top or both of them should be in bottom

Structure 2: If all the columns in matrix M has only 0 or consecutive 1s (or -1s), matrix M is a TU matrix

14.4.4 Construct a New Unimodular Matrix

Let F be a unimodular matrix, then

$$\begin{bmatrix} F \\ I \end{bmatrix} \tag{14.18}$$

is a unimodular matrix, also

$$\begin{bmatrix} F & 0 \\ I & I \end{bmatrix} \tag{14.19}$$

is a unimodular matrix.

Part IV

Graph and Network Theory

Chapter 15

Basic Structures

15.1 Graph

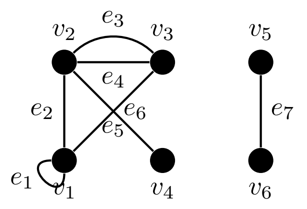
Definition 15.1.1 (Graph)**.** A **graph** G consists of a finite set $V(G)$ on vertices, a finite set $E(G)$ on edges and an **incident relation** than associates with any edge $e \in E(G)$ an unordered pair of vertices not necessarily distinct called **ends**.

Example. The following graph can be represented as

$$V = V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6\} \tag{15.1}$$

$$E = E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\} \tag{15.2}$$

$$e_1 = v_1v_2, \quad e_2 = v_2v_4, \quad \dots \tag{15.3}$$



Definition 15.1.2 (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 15.1.3 (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	$O(n)$	$O(d_v)$

15.2 Subgraph

Definition 15.2.1 (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 15.2.2 (Spanning)**.** A subgraph H on G is **spanning** if $V(H) = V(G)$

Definition 15.2.3 (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.

15.3 Degree

Definition 15.3.1 (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 15.1. *For any graph $G = (V, E)$*

$$\sum_{v \in V} d(v) = 2|E| \tag{15.4}$$

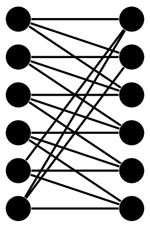
Proof. $\forall$ edge $e = uv$ with $u \neq v$, e is counted once for u and once for v, a total of two altogether. If $e = uu$, a loop, then it is counted twice for u. □

Problem 15.1. 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

$$2|E| = \sum_{v \in V} d(v) \geq 3 \times |N| = 3 \times 13 \Rightarrow |E| \geq 19.5 > 19 \tag{15.5}$$

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

□

Corollary 15.1.1. *Every graph has an even number of odd degree vertices.*

Proof.

$$V = V_E \cup V_O \Rightarrow \sum_{v \in V} d(v) = \sum_{v \in V_E} d(v) + \sum_{v \in V_O} d(v) = 2|E| \tag{15.6}$$

□

15.4 Special Graphs

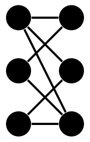
Definition 15.4.1 (Complete Graph)**.** A **complete** graph $K_n (n \geq 1)$ is a simple graph with n vertices and with exactly one edge between each pair of distinct vertices.

Definition 15.4.2 (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 15.4.3 (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

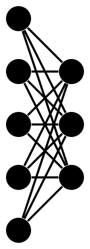
Definition 15.4.4 (Bipartite Graph)**.** A simple graph is said to be **bipartite** if the vertex set can be expressed as the union of two disjoint non-empty subsets V_1 and V_2 such that every edges has one end in V_1 and another end in V_2

Example. Here is an example for bipartite graph



Definition 15.4.5 (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}

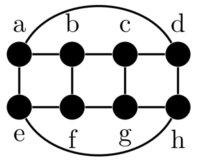


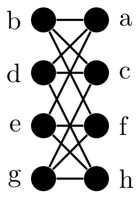
Theorem 15.2. *A graph G is bipartite iff every cycle is even.*

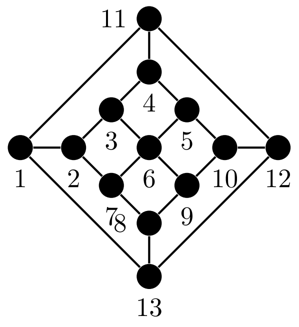
Proof. ($\Rightarrow$) If the graph G is bipartite, by definition, the vertices of graph can be partition into two groups, that within the group there is no connection between vertices. Therefore, for each cycle, the odd index of vertices and even index of vertices has to be choose alternatively from 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.







15.5 Directed Graph

Definition 15.5.1. 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 15.5.2. We call directed graphs **digraphs**, we call edges in a digraph are called **arcs**, and vertices in a digraph **nodes**

Definition 15.5.3. Similar as in the undirected case we have walks, traces, paths and cycles in digraphs.

Definition 15.5.4. 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 15.5.5. A digraph is strongly connected if every pair of vertices are strongly connected.

Definition 15.5.6. A digraph is **strict** if it has no loops and whenever e and f are parallel, $h(e) = t(f)$

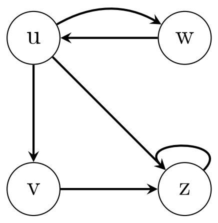
Definition 15.5.7. For a vertex v in a digraph D, the **indegree** of v in D, denoted by $d^+(v)$ is the number of arcs of D having head V. The **outdegree** of v is denoted by $d^-(v)$ is the number of arcs of D having tail v.

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

$$n_{ij} = \begin{cases} 1, & \text{if } v_i = h(e_j) \\ -1, & \text{if } v_i = t(e_j) \\ 0, & \text{otherwise} \end{cases} \tag{15.7}$$

$$\begin{bmatrix} -1 & 0 & -1 & -1 & 1 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{15.8}$$

15.6 Sperner's Lemma


u
w
v
z

Chapter 16

Paths, Trees, and Cycles

16.1 Walk

Definition 16.1.1 (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.

$$w = v_2e_4v_3e_4v_2e_5v_3 \tag{16.1}$$

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

Definition 16.1.2 (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.

$$w^{-1} = v_ke_kv_{k-1}e_{k-1}...e_2v_1 \tag{16.2}$$

(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.

16.2 Path and Cycle

Definition 16.2.1 (trail)**.** A **trail** is a walk with no repeating edges. e.g., $v_3e_4v_2e_5v_3$

Definition 16.2.2 (path)**.** A **path** is a trail with no repeating vertices. e.g., $v_3e_4v_2$

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

Definition 16.2.3 (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 16.2.4 (even/odd cycle)**.** A cycle is **even** if it has a even number of edges otherwise it is **odd**.

Problem 16.1. 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

$$C_1 = v_{11}e_{11}v_{12}e_{12}v_{13}e_{13}...v_{1n}e_{1n}v_{11} \tag{16.3}$$
$$C_2 = v_{21}e_{21}v_{22}e_{22}v_{23}e_{23}...v_{2k}e_{2k}v_{21} \tag{16.4}$$

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 16.2.5 (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 16.2.6 (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 16.2. 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.

□

16.3 Tree and forest

Definition 16.3.1 (acyclic graph)**.** A graph is called **acyclic** if it has no cycles

Definition 16.3.2 (forest, tree)**.** A acyclic graph is called a **forest**. A connected forest is called a **tree**.

Theorem 16.1. *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:         $p \leftarrow p \cup e$
```

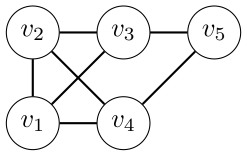
2. Then, if we remove one leaf in the tree, i.e., we remove an edge and a vertex, where that vertex only connects to the edge we removed. One of the following situations will happen:
   (a) Situation 1: The remaining of T is one vertex. In this case, T has two vertices an one edge. (Exactly one more 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.

□

16.4 Spanning tree

Definition 16.4.1 (spanning tree)**.** A subgraph T of G is a **spanning tree** if it is spanning ($V(T) = V(G)$) and it is a tree.

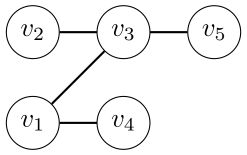
Example. In the following graph
This is a spanning tree



Problem 16.3. 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 16.2. *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. □

16.5 Cayley's Formula

16.6 Connectivity, DFS, BFS

16.6.1 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.

16.6.2 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 $\forall u \in d_s$ 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 $u = t$.

16.6.3 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, $head \leftarrow 1$, $tail \leftarrow 1$, $queue[1] \leftarrow s$, mark all vertices as "unvisited"
2: Mark $s$ as "visited"
3: while $head \geq tail$ do
4:     $v \leftarrow queue[tail]$, $tail \leftarrow tail + 1$
5:     for $\forall u \in d_v$ do
6:         if $u$ is "unvisited" then
7:             $head \leftarrow head + 1$, $queue[head] = u$
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 $u = t$.

16.6.4 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 $O(1)$.

16.6.5 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, $head \leftarrow 1$, $tail \leftarrow 1$, $queue[1] \leftarrow s$, mark all vertices as "unvisited"
 2: Mark $s$ as "visited"
 3: $color[s] \leftarrow 0$
 4: **while** $head \geq tail$ **do**
 5:     $v \leftarrow queue[tail]$, $tail \leftarrow tail + 1$
 6:     **for** $\forall u \in d_v$ **do**
 7:         **if** $u$ is "unvisited" **then**
 8:             $head \leftarrow head + 1$, $queue[head] = u$
 9:             Mark $u$ as "visited"
10:             $color[u] \leftarrow 1 - color[v]$
11:         **else**
12:             **if** $color[u] == color[v]$ **then return** False
    **return** True
```

16.6.6 Topological Ordering

The topological ordering problem is given a directed acyclic graph $G = (V, E)$, output a 1-to-1 function $\pi : V \rightarrow \{1, 2, \cdots, n\}$ 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)

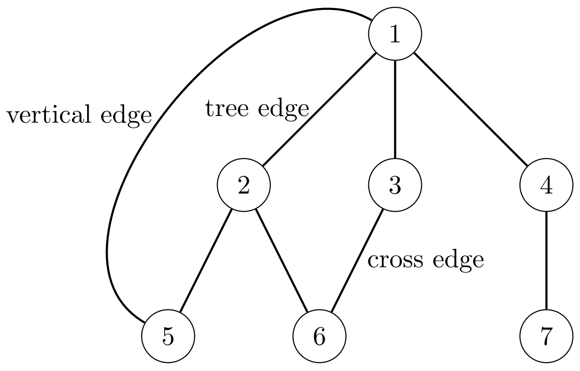
```
 1: Initialize, let $\forall v \in V, d_v = 0$
 2: **for** $v \in V$ **do**
 3:     **for** $u, (u, v) \in E$ **do**
 4:         $d_u \leftarrow d_u + 1$
 5: $S \leftarrow \{v : d_v = 0\}, i \leftarrow 0$
 6: **while** $S \neq \emptyset$ **do**
 7:     $v \leftarrow v \in S$, $S \leftarrow S \setminus \{v\}$
 8:     $i \leftarrow i + 1, \pi(v) \leftarrow i$
 9:     **for** $u, (u, v) \in E$ **do**
10:         $d_u \leftarrow d_u - 1$
11:         **if** $d_u = 0$ **then**
12:             $S \leftarrow S \cup \{u\}$
13: **if** $i < n$ **then return** Not directed acyclic graph
```

Running time $O(n + m)$

16.6.7 Bridge

Definition 16.6.1 (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 16.6.2 (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 $v \in V$ do
3:     if $v$ is "unvisited" then
4:         $level(v) \leftarrow 0$
5:         RecursiveDFS(v)
```

16.7 Blocks

Algorithm 17 RecursiveDFS(v)

```
1: mark $v$ as “visited”
2: $h(v) \leftarrow \infty$
3: for $u \in d_v$ do
4:     if $u$ is “unvisited” then
5:         $level(u) \leftarrow level(v) + 1$
6:         RecursiveDFS(u)
7:         if $h(u) \geq level(u)$ then
8:             $(u, v)$ is a bridge
9:         if $h(u) < h(v)$ then
10:            $h(v) \leftarrow h(u)$
11:    else
12:        if $level(u) < level(v) - 1$ and $level(u) < h(v)$ then
13:            $h(v) \leftarrow level(u)$
```

Chapter 17

Euler Tours and Hamilton Cycles

17.1 Euler Tours

17.2 Hamilton Cycles

17.3 The Chinese Postman Problem

17.4 The Travelling Salesman Problem

Chapter 18

Matriod, Planarity

18.1 Plane and Planar Graphs

18.2 Dual Graphs

18.3 Matroids

Definition 18.3.1 (Matroids)**.** Let S be a finite set of **elements** and let d be a collection of subsets of S satisfying the property

$$\text{If } x \leq y, y \in d, \Rightarrow x \in d \tag{18.1}$$

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

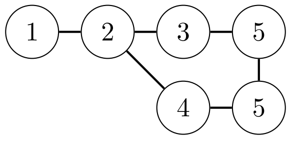
$$S = \{(1,2),(2,3),(2,4),(3,5),(4,6),(5,6)\} \tag{18.2}$$
$$d = \{\emptyset, \{(1,2)\}, \{(2,3)\}, ..., \{\text{other matching...}\}\} \tag{18.3}$$

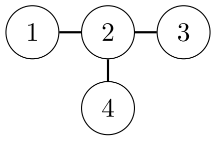
Example. Let G be a graph and let $S = V(G)$ define $X \subseteq S$ to be independent if no two member of x are adjacent.

$$S = \{1,2,3,4\} \tag{18.4}$$
$$d = \{\emptyset, 1, 2, 3, 4, (1,3), (1,4), (3,4), (1,3,4)\} \tag{18.5}$$

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**





18.4 Independent Sets

18.5 Ramsey's Theorem

18.6 Turán's Theorem

18.7 Schur's Theorem

18.8 Euler's Formula

18.9 Bridges

18.10 Kuratowski's Theorem

18.11 Four-Color Theorem

18.12 Graphs on other surfaces

Chapter 19

Matchings

19.1 Maximum Matching

Definition 19.1.1 (Matching)**.** 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** or **M-covered**, if some edge of M is incident with v. Otherwise, v is **M-unsaturated** or **M-exposed**.

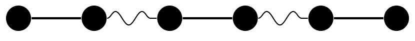
Definition 19.1.2 (Perfect matching, Maximum matching)**.** If every vertex of G is M-saturated, then the matching is said to be **perfect matching**. M is a **maximum matching** if G has no matching M' with $|M'| > |M|$. Every perfect matching is maximum. The maximum matching does not necessarily to be perfect. Perfect matching and maximum matching may not be unique.

Definition 19.1.3 (M-alternating)**.** An **M-alternating** path in G is a path whose edges are alternately in $E \setminus M$ and M.

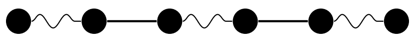
Definition 19.1.4 (M-augmenting)**.** An **M-augmenting** path in G is an M-alternating path whose origin and terminus are M-unsaturated.

Lemma 19.1. *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 19.2 (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 19.1.5 (Vertex-cover)**.** The **vertex-cover** is a subset of vertices X such that every edge of G is incident to some member of X.

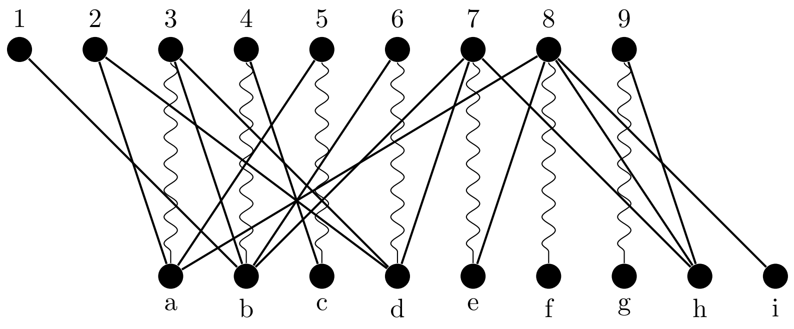
Lemma 19.3. *The cardinality of any matching is less than or equal to the cardinality of any vertex cover.*

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. ☐

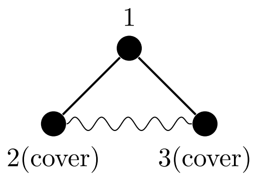
Theorem 19.4 (König Theorem)**.** *If G is bipartite, the cardinality of the maximum matching is equal to the cardinality of the minimum vertex cover.*

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.

19.2 Maximum Matching Algorithm

Definition 19.2.1 (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 19.5. *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 19.2.2 (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 19.2.3 (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 19.6. *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 19.7. *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

$$G : S = G \setminus (E(G) - S) \tag{19.1}$$

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

19.3 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.

19.4 Hall's Marriage Theorem

19.5 Transversal Theory

19.6 Menger's Theorem

19.7 The Hungarian Algorithm

Chapter 20

Colorings

Chapter 21

Minimum Spanning Tree Problem

21.1 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 21.1.1 (Cut vertex)**.** A vertex v of a connected graph G is a **cut vertex** if $G \setminus v$ is not connected.

Definition 21.1.2 (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 21.1. *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 21.1.3 (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.

21.2 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.
```

21.3 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.
```

21.4 Extensible MST

Definition 21.4.1 (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 21.4.2. We also define $\gamma(A) = \{e \in E : \text{both ends of } e \text{ are in } A\}$

Theorem 21.2. *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 21.4.3. 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 21.3. *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.*

21.5 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

$$X_e = \begin{cases} 1, \text{if edge } e \text{ is in the optimal solution} \\ 0, \text{otherwise} \end{cases} \tag{21.1}$$

The formulation is as following

$$\min \quad \sum_{e \in E} c_e x_e \tag{21.2}$$
$$\text{s.t.} \quad \sum_{e \in E} x_e = |V| - 1 \tag{21.3}$$
$$x_e \geq 0 \tag{21.4}$$
$$e \in E \tag{21.5}$$
$$\sum_{e \in E(S)} x_e = |S| - 1, \forall S \subseteq V, S \neq \emptyset \tag{21.6}$$
$$\tag{21.7}$$

Chapter 22

Shortest-Path Problem

22.1 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 22.1.1 (feasible potential)**.** We call $y = (y_v : v \in V)$ a **feasible potential** if it satisfies

$$y_v + c_{vw} \geq y_w \quad \forall (v, w) \in E \tag{22.1}$$

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

$$C(P) = \sum_{i=1}^{k} C_{e_i} \geq \sum_{i=1}^{k} (y_{v_i} - y_{v_{i-1}}) = y_{v_k} - y_{v_0} \tag{22.2}$$

$\square$

22.2 Breadth-First Search Algorithm

22.3 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.

$$\begin{tikzcd}
v_0 \arrow[r, "4"'] \arrow[rr, bend left, "4"] & v_1 \arrow[r, "4"'] & v_2 \arrow[r, "2"'] \arrow[rr, bend left, "2"] & v_3 \arrow[r, "2"'] & v_4 \arrow[r, "1"'] \arrow[rr, bend left, "1"] & v_5 \arrow[r, "1"'] & v_6
\end{tikzcd}$$

22.4 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.

22.5 SPFA Algorithm

22.6 Dijkstra Algorithm

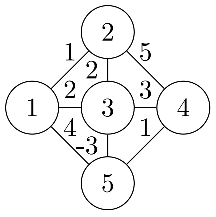
22.7 A* Algorithm

22.8 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., $d_{14}^5 = d_{15}^4 + d_{54}^4 = 0$

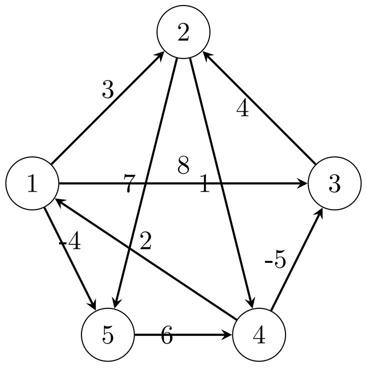
Therefore $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$

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

$$d_{ij}^0 = \begin{cases} c_{ij} & \text{distance from } i \text{ to } j \text{ if } (i,j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \notin E \end{cases} \tag{22.3}$$

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

$$D^0 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{22.4}$$



$$\Pi^0 = \begin{bmatrix} & 1 & 1 & & 1 \\ & & & 2 & 2 \\ & 3 & & & \\ 4 & & 4 & & \\ & & & 5 & \end{bmatrix} \tag{22.5}$$

$$D^1 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \mathbf{5} & -5 & 0 & -\mathbf{2} \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{22.6}$$

$$\Pi^1 = \begin{bmatrix} & 1 & 1 & & 1 \\ & & & 2 & 2 \\ & 3 & & & \\ 4 & \mathbf{1} & 4 & & \mathbf{1} \\ & & & 5 & \end{bmatrix} \tag{22.7}$$

$$D^2 = \begin{bmatrix} 0 & 3 & 8 & \mathbf{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \mathbf{5} & \mathbf{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{22.8}$$

$$\Pi^2 = \begin{bmatrix} & 1 & 1 & \mathbf{2} & 1 \\ & & & 2 & 2 \\ & 3 & & \mathbf{2} & \mathbf{2} \\ 4 & 1 & 4 & & 1 \\ & & & 5 & \end{bmatrix} \tag{22.9}$$

$$D^3 = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -\mathbf{1} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{22.10}$$

$$\Pi^3 = \begin{bmatrix} & 1 & 1 & 2 & 1 \\ & & & 2 & 2 \\ & 3 & & 2 & 2 \\ 4 & \mathbf{3} & 4 & & 1 \\ & & & 5 & \end{bmatrix} \tag{22.11}$$

$$D^4 = \begin{bmatrix} 0 & 3 & -\mathbf{1} & 4 & -4 \\ \mathbf{3} & 0 & -\mathbf{4} & 1 & -\mathbf{1} \\ \mathbf{7} & 4 & 0 & 5 & \mathbf{3} \\ 2 & -1 & -5 & 0 & -2 \\ \mathbf{8} & \mathbf{5} & \mathbf{1} & 6 & 0 \end{bmatrix} \tag{22.12}$$

$$\Pi^4 = \begin{bmatrix} & 1 & \mathbf{4} & 2 & 1 \\ \mathbf{4} & & \mathbf{4} & 2 & \mathbf{1} \\ \mathbf{4} & 3 & & 2 & \mathbf{1} \\ 4 & 3 & 4 & & 1 \\ \mathbf{4} & \mathbf{3} & \mathbf{4} & 5 & \end{bmatrix} \tag{22.13}$$

$$D^5 = \begin{bmatrix} 0 & \mathbf{1} & -\mathbf{3} & \mathbf{2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \tag{22.14}$$

$$\Pi^5 = \begin{bmatrix} & \mathbf{3} & 4 & \mathbf{5} & 1 \\ 4 & & 4 & 2 & 1 \\ 4 & 3 & & 2 & 1 \\ 4 & 3 & 4 & & 1 \\ 4 & 3 & 4 & 5 & \end{bmatrix} \tag{22.15}$$

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.

22.9 Johnson's Algorithm

Chapter 23

Maximum Flow Problem

23.1 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.

$$\max \quad v \tag{23.1}$$

$$\text{s.t.} \quad \sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = \begin{cases} -v, & \text{if } i = s \\ v, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \tag{23.2}$$

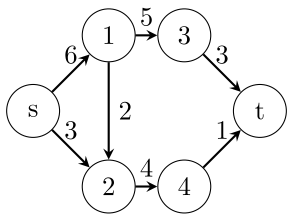
$$0 \le x_e \le u_e, \quad \forall e \in A \tag{23.3}$$

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.

23.2 Solving Maximum Flow Problem in LP

Theorem 23.1. *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 $S = \{1, 2, 3, s\}$, $\bar{S} = \{4, t\}$

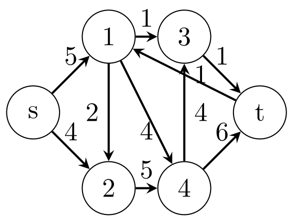


then $\delta^-(S) = \{(2, 4), (3, t)\} \Rightarrow u(S, \bar{S}) = 7$

Definition 23.2.1. If $(S, \bar{S})$ has minimum capacity of all (s, t)-cuts, then it is called **minimum cut**.

Definition 23.2.2. 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 23.2. *If x is a (s,t) flow of value v and $(S,\bar{S})$ is a (s,t)-cut, then*

$$v = \sum_{e\in\delta^-(S)} x_e - \sum_{e\in\delta^+(S)} x_e \tag{23.4}$$

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

$$\sum_{i\in S}\Big(\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e\Big) = -v \tag{23.5}$$

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

$$\sum_{e\in\delta^+(S)} x_e - \sum_{e\in\delta^-(S)} x_e = -v \tag{23.6}$$

□

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

Corollary 23.2.1. *If x is a feasible flow of value v, and $(S,\bar{S})$ is an (s,t)-cut, then*

$$v \leq u(S,\bar{S}) \quad \textit{(Weak duality)} \tag{23.7}$$

Definition 23.2.3. Define an arc e to be **saturated** if $x_e = u_e$, and to be **flowless** if $x_e = 0$

Corollary 23.2.2. *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

$$\sum_{e\in\delta^-(S)} x_e = \sum_{e\in\delta^-(S)} u_e \tag{23.8}$$

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

$$\sum_{e\in\delta^+(S)} x_e = 0 \tag{23.9}$$

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

23.3 Prime and Dual of Maximum Network Flow Problem

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

$$\max \quad f = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ f \end{bmatrix} \tag{23.10}$$

$$\text{s.t.} \quad \begin{bmatrix} \mathbf{A} & \mathbf{F} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ f \end{bmatrix} = \mathbf{0} \tag{23.11}$$

$$\mathbf{Ix} \leq \mathbf{u} \tag{23.12}$$

$$\begin{bmatrix} \mathbf{x} \\ f \end{bmatrix} \geq 0 \tag{23.13}$$

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.

$$\mathbf{A} = \mathbf{A}_{|E|\times|V|} = [a_{ij}], \text{ where } a_{ij} = \begin{cases} 1, & \text{if } v_i = h(e_j) \\ -1, & \text{if } v_i = t(e_j) \\ 0, & \text{otherwise} \end{cases} \tag{23.14}$$

$$\mathbf{F} = \begin{bmatrix} -1 & \cdots & 0 & \cdots & 1 \end{bmatrix}^\top \tag{23.15}$$

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & \cdots & u_{|E|} \end{bmatrix}^\top \tag{23.16}$$

Then, we take the dual of LP

$$\min \quad \mathbf{u}\mathbf{w_E} \tag{23.17}$$

$$\text{s.t.} \quad \begin{bmatrix} \mathbf{w_V} & \mathbf{w_E} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \end{bmatrix} \geq 0 \tag{23.18}$$

$$\begin{bmatrix} \mathbf{w_V} & \mathbf{w_E} \end{bmatrix} \begin{bmatrix} \mathbf{F} \\ \mathbf{0} \end{bmatrix} = 1 \tag{23.19}$$

$$\mathbf{w_V} \quad \text{unrestricted} \tag{23.20}$$

$$\mathbf{w_E} \geq \mathbf{0} \tag{23.21}$$

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.

$$\mathbf{w_V} = \begin{bmatrix} w_1 & w_2 & \cdots & w_{|V|} \end{bmatrix}^\top \tag{23.22}$$

$$\mathbf{w_E} = \begin{bmatrix} w_{|V|+1} & w_{|V|+2} & \cdots & w_{|V|+|E|} \end{bmatrix}^\top \tag{23.23}$$

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

$$\min \quad \sum_{e\in E} u_e w_e \tag{23.24}$$

$$\text{s.t.} \quad w_i - w_j + w_{|V|+e} \geq 0, \forall e = (i,j) \in E \tag{23.25}$$

$$-w_1 + w_{|V|} = 1 \tag{23.26}$$

$$\mathbf{w_V} \quad \text{unrestricted} \tag{23.27}$$

$$\mathbf{w_E} \geq \mathbf{0} \tag{23.28}$$

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.

23.4 Maximum Flow Minimum Cut Theorem

Definition 23.4.1. 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 23.3. *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):

$$D_1 = \min\{u_e - x_e : e \text{ forward in } P\} \tag{23.29}$$

$$D_2 = \min\{x_e : e \text{ backward in } P\} \tag{23.30}$$

$$D = \min\{D_1, D_2\} \tag{23.31}$$

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

$$\hat{x_e} = \begin{cases} x_e + D & \text{If } e \text{ is forward in } P \\ x_e - D & \text{If } e \text{ is backward in } P \\ x_e & \textit{otherwise} \end{cases} \tag{23.32}$$

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 23.4. *(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*

23.5 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.
```

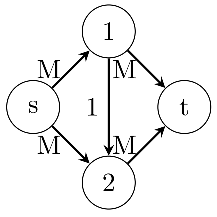
Algorithm 24 Ford-Fulkerson algorithm

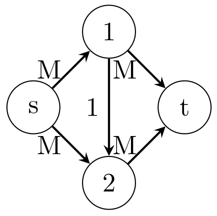
```
Ensure: Max-flow $x$ with value $v$
Require: Digraph with source $s$ and sink $t$, a capacity function $u$ and a feasible flow (could be $x_e = 0$)
  Initialize, $v \leftarrow x$
  Designate all vertices as unlabeled and unscanned
  Label $s$
  while There exists vertex unlabeled or unscanned do
      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

23.6 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

$$\text{Second iteration} \tag{23.33}$$
$$V_0 = \{s\}, i = 0, l(x) = 0 \tag{23.34}$$
$$R = \{1, 2\} \tag{23.35}$$
$$V_1 \leftarrow \{1, 2\}, i = 1, l(x) = 1 \tag{23.36}$$
$$R = \{3, 4, 5\} \tag{23.37}$$
$$V_2 \leftarrow \{3, 4\}, i = 2, l(x) = 2 \tag{23.38}$$
$$R = \{1, 5, 6, 3\} \tag{23.39}$$
$$V_3 \leftarrow \{5, 6\}, i = 3, l(x) = 3 \tag{23.40}$$
$$R = \{4, t\} \tag{23.41}$$
$$V_4 = \{t\} \tag{23.42}$$
$$A_1 = \{(s, 1), (s, 2)\} \tag{23.43}$$
$$A_2 = \{(1, 3), (2, 4)\} \tag{23.44}$$
$$A_3 = \{(3, 5), (4, 6)\} \tag{23.45}$$
$$A_4 = \{(5, t), (6, t)\} \tag{23.46}$$

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

$$x'_e = x_e + \hat{x_e}, \quad \text{If } t(e) \in V_{i-1}, h(e) \in V_i \tag{23.47}$$
$$x'_e = x_e - \hat{x_e}, \quad \text{If } h(e) \in V_{i-1}, t(e) \in V_i \tag{23.48}$$

23.7 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 23.5. *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 24

Minimum Cost Flow Problem

24.1 Transshipment Problem

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

$$\begin{aligned} \min \quad & wx && (24.1)\\ \text{s.t.} \quad & Nx = b && (24.2)\\ & x \geq 0 && (24.3) \end{aligned}$$

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

$$\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = b_i \tag{24.4}$$

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.

24.2 Network Simplex Method

Lemma 24.1. *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 24.2. *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

$$\begin{aligned} w(T, e) = & \sum \{w_e : e \text{ forward in } C(T, e)\} \\ & - \sum \{w_e : e \text{ reverse in } C(T, e)\} \end{aligned} \tag{24.5}$$

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

24.2.1 Network Simplex Method

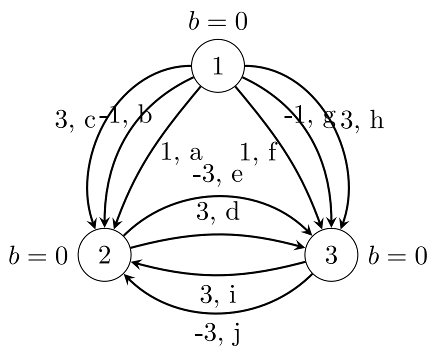
Algorithm 26 Network Simplex Method Algorithm

```
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
```

24.2.2 Example for cycling

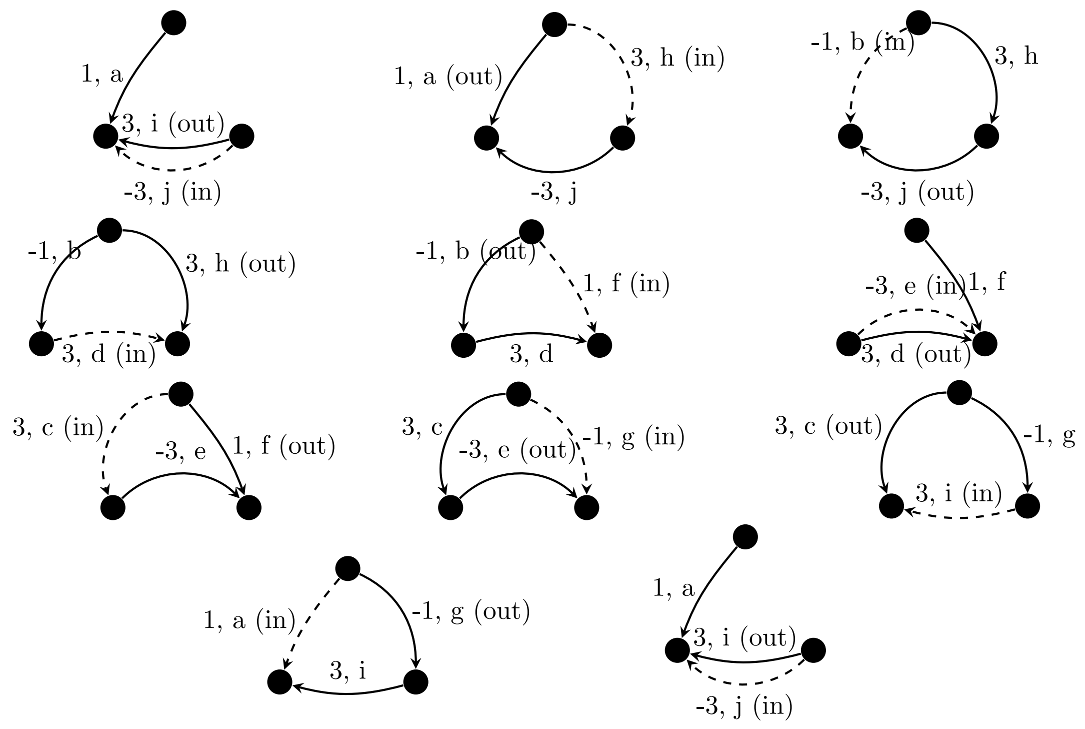
Notice: Similar to Simplex Method in LP, even though in worst case may be inefficient. In most cases it is simple and empirically efficient. Also, similarily, there will be cycling problems.

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.



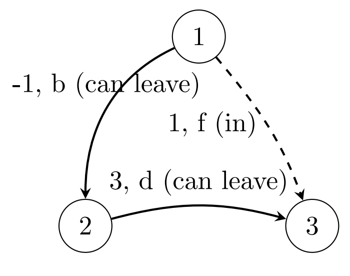
24.2.3 Cycling prevention

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:

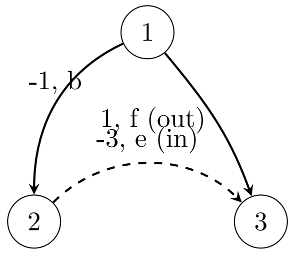
- The algorithm is initialed with a strong feasible tree.
- f in pivot phase is chosen to be the first reverse arc of $C(T, e)$ having $x_f = \theta$. By "first", we mean the first arc encountered in traversing $C(T, e)$ in the direction of e, starting at the vertex i of $C(T, e)$ that minimizes the 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.

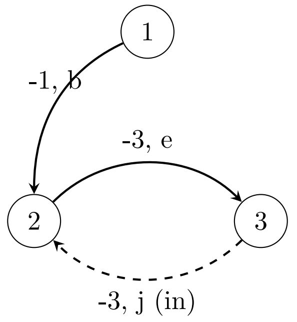
Continue the previous example. Now should how we can avoid cycling:
The first few (four) steps are the same as previous example, starting from
$w(T, f) = w_f - w_d - w_b = 1 - 3 + 1 = -1$. f is entering basis, both b and d can leave the basis, according to



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

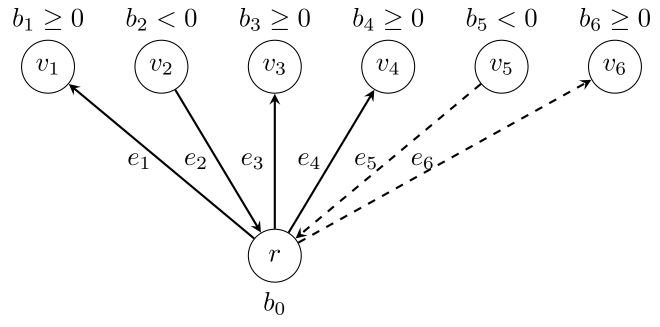


negative w is j.
$w(T, j) = w_j + w_e = -6$, but in $C(T, j)$ there is no reversing arc, therefore we detect unboundedness.



24.2.4 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

$$G_0 =< M \cup M' \cup \{r\}, E \cup A > \tag{24.6}$$

Denote the origin graph

$$G =< V, \mathcal{A} > \tag{24.7}$$

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

$$C(T, f) = re_{rv_{M'}}v_{M'}fv_MPr \tag{24.8}$$

For

$$w(T, f) = w_f - w_{e_{rv_{M'}}} + \sum_{e \in P} d_e w_e \tag{24.9}$$

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

$$\begin{aligned}
w(T, f) &= w_f + \sum_{e \in P} d_e w_e - w_{e_{rv_{M'}}} & (24.10)\\
&\leq w_f + \sum_{e \in P} w_e - w_{e_{rv_{M'}}} & (24.11)\\
&\leq \sum_{e \in \mathcal{A}} w_e - w_{e_{rv_{M'}}} & (24.12)\\
&\leq |V|(\max\{w_e : e \in \mathcal{A}\}) - w_{e_{rv_{M'}}} & (24.13)\\
&\leq -1 < 0 & (24.14)
\end{aligned}$$

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 $C(T, f') = rP'v_Mf'v_{M'}e_{v_{M'}r}r$. $w(T, f') = w_{f'} - w_{e_{rv_{M'}}} + \sum_{e \in P'} d_e w_e$, 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

24.3 Transshipment Problem and Circulation Problem

Definition 24.3.1. The minimum weight circulation problem is defined as follows:

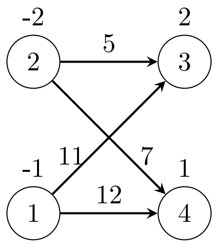
$$\begin{aligned} \min \quad & wx && (24.15)\\ \text{s.t.} \quad & Nx = 0 && (24.16)\\ & l \le x \le u && (24.17) \end{aligned}$$

It turns out that the circulation problem is equivalent with transshipment problem.

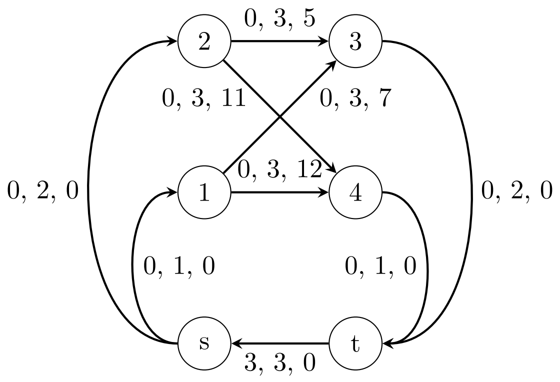
We will show how to transform any transshipment into circulation.
Let (D, b, w) be a transshipment problem and define two new vertices s and t.

- 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.

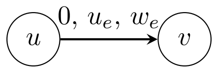
The following is a graph for transshipment problem.



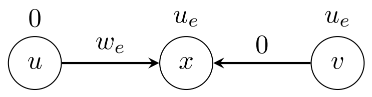
After the above procedures, it is now transformed into a circulation problem.

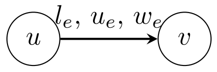


Then, we will show how to transform any circulation problem into transshipment problem.
If the lower bound of the arc is zero, i.e., $l_e = 0$, then for each such arc (u, v), introduce a vertex in between u and v, replace the arc $e = (u, v)$ by $e_1 = (u, x)$ and $e_2 = (x, v)$. Both arcs are uncapacitated. 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).



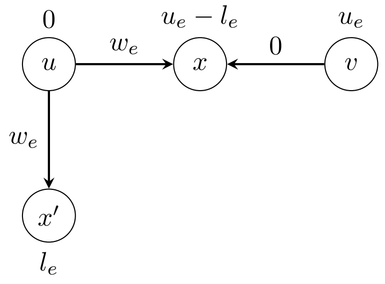
Will be transform into
If the lower bound of the arc is not zero, i.e., $l_e \neq 0$, then for each such arc (u, v), introduce two new vertices, one vertex is in between u and v, similar to the previous case, the difference is the demands of this new vertex is





$u_e - l_e$, 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.

24.4 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 24.3. *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 $e \in A$*

$$y_{h(e)} - y_{t(e)} > w_e \text{ implies } x_e = u_e \tag{24.18}$$
$$y_{h(e)} - y_{t(e)} < w_e \text{ implies } x_e = l_e \tag{24.19}$$

Then x is optimal to the circulation problem.

Proof. For each $e \in A$ define

$$\gamma_e = \max\{y_{h(e)} - y_{t(e)} - w_e, 0\} \tag{24.20}$$
$$\mu_e = \max\{w_e - y_{h(e)} + y_{t(e)}, 0\} \tag{24.21}$$

Then

$$\gamma_e - \mu_e = y_{h(e)} - y_{t(e)} - w_e \tag{24.22}$$

Furthermore

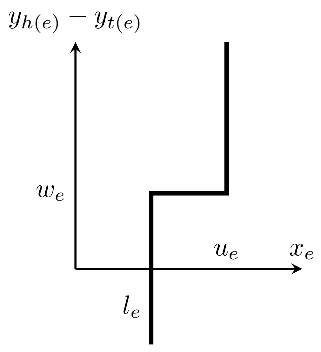
$$\sum_{e\in A}(\mu_e l_e - \gamma_e u_e) \tag{24.23}$$
$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e) + \sum_{i\in V} y_i(\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e) \tag{24.24}$$
$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e + x_e(y_{h(e)} - y_{t(e)})) \tag{24.25}$$
$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e + x_e(\gamma_e - \mu_e + w_e)) \tag{24.26}$$
$$= \sum_{e\in A}(\gamma_e(x_e - u_e) + \mu_e(l_e - x_e) + x_e w_e) \tag{24.27}$$
$$\le \sum_{e\in A} x_e w_e \tag{24.28}$$

The last inequality will be satisfied as equality iff the first two hold. □

The following is the formulation of circulation problem

$$\begin{aligned}
&(P) && \min && wx && (24.29)\\
& && \text{s.t.} && Nx = 0 \quad y && (24.30)\\
& && && x \ge l \quad z^l && (24.31)\\
& && && -x \le -u \quad z^u && (24.32)\\
&(D) && \max && lz^l - uz^u && (24.33)\\
& && (\text{s.t.}) && yN^{-1} + z^l - z^u \le w && (24.34)\\
& && && y \quad free && (24.35)\\
& && && z^l, z^u \ge 0 && (24.36)\\
& && (CS) && y_{h(e)} - y_{t(e)} > w_e \Rightarrow x_e = u_e && (24.37)\\
& && && y_{h(e)} - y_{t(e)} < w_e \Rightarrow x_e = l_e && (24.38)
\end{aligned}$$

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:

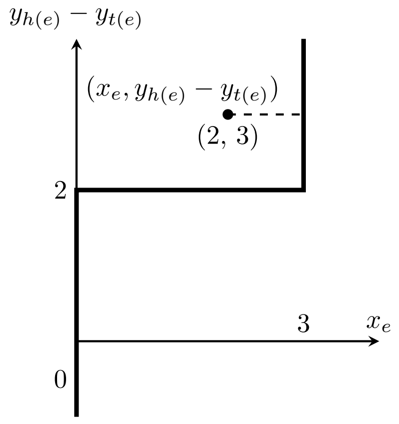


For each point $(x_e, y_{h(e)} - y_{t(e)})$ we define a **kilter-number** k_e, be the minimum positive distance change in x_e required to put in on the kilter line.

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 24.4. *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 $\forall e \in A, l_e \le x_e \le u_e$. 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 $R = \lceil \max\{\log_2 u_e : e \in A\} \rceil$

24.5 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 25

Social Network Analysis

Part V

Heuristic Optimization

Chapter 26

Special Topic: Vehicle Routing Problem

Chapter 27

Traveling Salesman Problem

27.1 Formulation

Consider a Graph $G = \{A, N\}$
Denote:

$$x_{ij} = \begin{cases} 1, & \text{if goes from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases} \tag{27.1}$$

Dantzig-Fulkerson-Johnson Formulation:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{27.2}$$

$$\sum_{j\in N,(i,j)\in A} x_{ij} = 1 \tag{27.3}$$

$$\sum_{i\in N,(i,j)\in A} x_{ij} = 1 \tag{27.4}$$

$$\sum_{j\notin S, i\in S,(i,j)\in A} x_{ij} = 1 \text{ or } \sum_{i,j\in S,(i,j)\in A} x_{ij} \leq |S| - 1 \tag{27.5}$$

$$\forall S \subset N, S \neq \emptyset, 2 \leq |S| \leq n-1 \tag{27.6}$$

Miller-Tucker-Zemlin Formulation:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{27.7}$$

$$\sum_{j\in N,(i,j)\in A} x_ij = 1 \tag{27.8}$$

$$\sum_{i\in N,(i,j)\in A} x_ij = 1 \tag{27.9}$$

$$u_i - u_j + nx_{ij} \leq n-1 \quad i,j \in 2,...,n,(i,j) \in A \tag{27.10}$$

$$u_1 = 1 \tag{27.11}$$

$$2 \leq u_i \leq n, i \in N, i > 1 \tag{27.12}$$

27.2 Sub-tour Searching Algorithm

In the graph $G = (N, A)$, let $\bar{G} = (N, \bar{A})$ be the connected components of graph, where

$$\bar{G} = (G, \bar{A}), \bar{A} = \{(i,j) \in A | \bar{x_{ij}} = 1\} \tag{27.13}$$

denote

$$\bar{FS}(i) = \{(i,j) \in \bar{A}\} \tag{27.14}$$

Then the algorithm to find all sub-tour is the following:

Algorithm 27 Sub-tour Searching Algorithm

```
1:  $K = \emptyset$
2:  $d_i = 0, \forall i \in N$
3:  for $i \in N$ do
4:      $C = \emptyset$
5:      $Q = \emptyset$
6:      if $d_i == 0$ then
7:          $d_i = 1$
8:          $C = C \cup \{i\}$
9:          Q.append(i)
10:         while $Q \neq \emptyset$ do
11:             v = Q.pop()
12:             for $u \in \bar{F}S(v)$ do
13:                 if $d_u == 0$ then
14:                     $d_u = 1$
15:                     $C = C \cup \{u\}$
16:                     Q.append(u)
17:     $K = K \cup C$
```

27.3 Up and Lower Bounds

Chapter 28

Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is defined on a graph $G = (V, E)$ where $V = \{0, \ldots, n\}$, is a vertex set. $E = \{(i, j)|i, j \in V\}$ represents an edge set. Usually, vertex 0 is a depot while the remaining vertices are customers, for simplicity, researchers sometimes add a dummy vertex N' as a replicate of depot, so that vehicles leave depot from vertex 0 and return to depot at vertex N', the distance or cost between vertex 0 and vertex N' is 0.

Chapter 29

Important Variants

29.1 VRPTW

29.2 VRPPD

29.3 VRPSD

29.4 DVRP

Chapter 30

Exact Algorithms

30.1

30.2 Valid Cuts

Chapter 31

Heuristic Algorithms

31.1 Clarke and Wright Saving Algorithm

31.2 Sweep Algorithm

31.3 Mole and Jameson Insertion Algorithm

31.4 Christofides Insertion Algorithm

31.5 Petal Algorithms

31.6 Cluster-first, Route-second Algorithm

31.7 Lin's λ-opt Local Improvement

Chapter 32

Metaheuristic

Chapter 33

Hybridizations

Part VI

Nonlinear Programming

Chapter 34

Convex Analysis

34.1 Convex Sets

Definition 34.1.1. A set $S \in \mathbb{R}^n$ is said to be convex if $\forall x_1, x_2 \in S, \lambda \in (0,1) \Rightarrow \lambda x_1 + (1-\lambda)x_2 \in S$

The following are some familes 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

$$S_1 \oplus S_2 = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{x} = \mathbf{x_1} + \mathbf{x_2}, \mathbf{x_1} \in S_1, \mathbf{x_2} \in S_2\} \tag{34.1}$$

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

$$S_1 \oplus S_2 = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{x} = \mathbf{x_1} - \mathbf{x_2}, \mathbf{x_1} \in S_1, \mathbf{x_2} \in S_2\} \tag{34.2}$$

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

Theorem 34.1 (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 34.2. *Let S be a convex set with nonempty interior. Let $\mathbf{x}_1 \in cl(S)$ and $\mathbf{x}_2 \in int(S)$, then $\mathbf{y} = \lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \in int(S), \forall\lambda \in (0,1)$*

34.2 Convex Functions

Definition 34.2.1. Let $C \in \mathbb{R}^n$ be a convex set. A function $f : C \to \mathbf{R}$ is (resp. strictly) convex if

$$f(\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1-\lambda)f(\mathbf{x}_2) \tag{34.3}$$
$$\forall \mathbf{x}_1, \mathbf{x}_2 \in C, \forall\lambda \in (0,1) \tag{34.4}$$

(resp.)

$$f(\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) < \lambda f(\mathbf{x}_1) + (1-\lambda)f(\mathbf{x}_2) \tag{34.5}$$
$$\forall \mathbf{x_1} \neq \mathbf{x}_2 \in C, \forall\lambda \in (0,1) \tag{34.6}$$

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 34.2.2. 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 34.3. *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 34.2.3 (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 34.4. *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 34.2.4 (Epigraphs, Hypographs)**.** Let $S \in \mathbb{R}^n$ be such that $S \neq \emptyset$. The **epigraph** of f, denoted by *epi*(f) is

$$epi(f) = \{(\mathbf{x}, y) \in S | \mathbf{x} \in S, y \in \mathbb{R}, y \geq f(x)\} \in \mathbb{R}^{n+1} \tag{34.7}$$

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

$$hypo(f) = \{(\mathbf{x}, y) \in S | \mathbf{x} \in S, y \in \mathbb{R}, y \leq f(x)\} \in \mathbb{R}^{n+1} \tag{34.8}$$

Theorem 34.5. *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 34.6. *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).*

34.3 Subgradients and Subdifferentials

Definition 34.3.1 (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

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \xi^\top (\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \in S \tag{34.9}$$

Definition 34.3.2 (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 34.7. *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*

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \xi^\top (\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \in S \tag{34.10}$$

In particular, the hyperplane

$$\mathcal{H} = \{(\mathbf{x}, y) | y = f(\bar{\mathbf{x}}) + \xi^\top (\mathbf{x} - \bar{\mathbf{x}})\} \tag{34.11}$$

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

Theorem 34.8. *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*

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \xi^\top (\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \in S \tag{34.12}$$

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.

34.4 Differentiable Functions

Definition 34.4.1 (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

$$f(\mathbf{x}) = f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})^\top(\mathbf{x} - \bar{\mathbf{x}}) + \alpha(\bar{\mathbf{x}}, \mathbf{x} - \bar{\mathbf{x}})\|\mathbf{x} - \bar{\mathbf{x}}\| \tag{34.13}$$

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 34.9. *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 34.10. *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 $\forall \bar{\mathbf{x}} \in S$*

$$f(\mathbf{x}) \geq f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \in S \tag{34.14}$$

(resp.)

$$f(\mathbf{x}) > f(\bar{\mathbf{x}}) + \nabla f(\bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}}), \forall \mathbf{x} \neq \bar{\mathbf{x}} \in S \tag{34.15}$$

Theorem 34.11 (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*

$$f(\mathbf{x}_2) = f(\mathbf{x}_2) + \nabla f(\hat{\mathbf{x}})(\mathbf{x}_2 - \mathbf{x}_1) \tag{34.16}$$

where

$$\hat{\mathbf{x}} = \lambda\mathbf{x}_1 + (1 - \lambda)\mathbf{x}_2 \tag{34.17}$$

Chapter 35

Optimality Conditions and Duality

35.1 The Fritz John Optimality Conditions

35.2 The Karush-Kuhn-Tucker Optimality Conditions

35.3 Constraint Qualification

35.4 Lagrangian Duality and Saddle Point Optimality Condition

Chapter 36

Unconstrained Optimization

Chapter 37

Quadratic Programming

Chapter 38

Penalty and Barrier Functions

Part VII

Algorithms

Chapter 39

Computational Complexity

39.1 Asymptotic Notation

39.1.1 Asymptotic Analysis

Definition 39.1.1 (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 $f(n) > 0$

39.1.2 O-Notation, Ω-Notation and Θ-Notation

Definition 39.1.2 (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 39.1.3 (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 39.1.4 (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 39.1.5 (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 39.1.6 (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

(39.1)

Theorem 39.1. *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,

$$\forall n > n_0, f(n) \le 2cg(n) \Rightarrow f(n) = O(g(n)) \tag{39.2}$$

$$\forall n > n_0, f(n) \ge \frac{1}{2}cg(n) \Rightarrow f(n) = \Omega(g(n)) \tag{39.3}$$

□

A set of properties:

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \tag{39.4}$$

$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } g(n) = O(f(n)) \tag{39.5}$$

Another set of properties:

$$f = O(g), g = O(h) \Rightarrow f = O(h) \tag{39.6}$$

$$f = \Omega(g), g = \Omega(h) \Rightarrow f = \Omega(h) \tag{39.7}$$

$$f = \Theta(g), g = \Theta(h) \Rightarrow f = \Theta(h) \tag{39.8}$$

Theorem 39.2. *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 $g \notin O(f)$

39.2 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: $\lg n < \sqrt{n} < n < n\lg n < n^2 < n^{\sqrt{n}} < 2^n < e^n < n! < n^n$

Chapter 40

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

40.1 Greedy Algorithms

40.1.1 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 28 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 algortihm that works.

40.1.2 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 29 Interval Scheduling, $S(s, f, n)$

```
1: Sort jobs by $f$
2: $t \leftarrow 0$, $S \leftarrow \emptyset$
3: for every $j \in [n]$ according to non-decreasing order of $f_j$ do
4:     if $s_j \geq t$ then
5:         $S \leftarrow S \cup \{j\}$
6:         $t \leftarrow f_j$
   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 30 Cover points with unit-length intervals

```
1: Initial, $S \leftarrow \emptyset$
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., $S \leftarrow S \cup \{x_i\}$
```

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. □

40.2 Divide and Conquer

40.2.1 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.

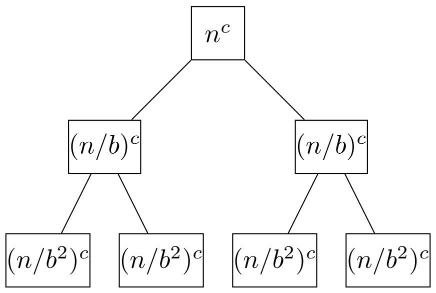
40.2.2 Master Theorem

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 40.1 (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*

$$T(n) = \begin{cases} O(n^{\lg_b a}), & c < \lg_b a \\ O(n^c \lg n), & c = \lg_b a \\ O(n^c), & c > \lg_b a \end{cases} \tag{40.1}$$

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 $O(n^c)$ □

40.2.3 Examples

Counting Inversions

40.3 Dynamic Programming

40.3.1 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.

40.3.2 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 31 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 32 RecoverJobs()

```
1: Compute $p_1, p_2, \cdots, p_n$
2: $opt(0) \leftarrow 0$
3: for $i \leftarrow 1$ to $n$ do
4:     if $opt(i-1) \geq v_i + opt(p_i)$ then
5:         $opt(i) \leftarrow opt(i-1)$
6:         $b[i] \leftarrow N$
7:     else
8:         $opt(i) \leftarrow v_i + opt(p_i)$
9:         $b[i] \leftarrow Y$
10: $i \leftarrow n, S \leftarrow \emptyset$
11: while $i \neq 0$ do
12:     if $b[i] == N$ then
13:         $i \leftarrow i - 1$
14:     else
15:         $S \leftarrow S \cup \{i\}$
16:         $i \leftarrow p_i$
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

$$\max \quad \sum_{i \in S} w_i \tag{40.2}$$

$$\text{s.t.} \quad \sum_{i \in S} w_i \leq W \tag{40.3}$$

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 33 Optimum Subset

```
1: for $W' \leftarrow 0$ to $W$ do
2:     $opt[0, W'] \leftarrow 0$
3: for $i \leftarrow 1$ to $n$ do
4:     for $W' \leftarrow 0$ to $W$ do
5:         $opt[i, W'] \leftarrow opt[i-1, W']$
6:         $b[i, W'] \leftarrow N$
7:         if $w_i \leq W'$ and $opt[i-1, W' - w_i] + w_i \geq opt[i, W']$ then
8:             $opt[i, W'] \leftarrow opt[i-1, W' - w_i] + w_i$
9:             $b[i, W'] \leftarrow Y$
10: return $opt[n, W]$
```

Algorithm 34 Recover the Optimum Set

```
1: $i \leftarrow n, W' \leftarrow W, S \leftarrow 0$
2: while $i > 0$ do
3:     if $b[i, W'] == Y$ then
4:         $W' \leftarrow W' - w_i$
5:         $S \leftarrow \cup\{i\}$
6:     $i \leftarrow i - 1$
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

$$\sum_{i=1}^{n} f_i d_i \tag{40.4}$$

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

$$C = \sum_{j=1}^{n} f_j d_j \tag{40.5}$$
$$= \sum_{j=1}^{n} f_j + \sum_{j=1}^{n} f_j(d_j - 1) \tag{40.6}$$
$$= \sum_{j=1}^{n} f_j + \sum_{j=1}^{k-1} f_j(d_j - 1) + \sum_{j=k+1}^{n} f_j(d_j - 1) \tag{40.7}$$
$$= \sum_{j=1}^{n} f_j + C_L + C_R \tag{40.8}$$

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 $1 \le i \le j \le n$

$$opt(i,j) = \sum_{k=i}^{j} f_k + \min_{k: i \le k \le j} \{opt(i, k-1) + opt(k+1, j)\} \tag{40.9}$$

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 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.

$opt(i,j)$ j i	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

$\pi(i,j)$ j i	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 40.1: *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.

$$\begin{aligned}
opt(1,2) &= \min\{0+opt(2,2), opt(1,1)+0\} + (f_1+f_2) = \min\{25,5\}+5+25 = 35\\
opt(2,3) &= \min\{0+opt(3,3), opt(2,2)+0\} + (f_2+f_3) = \min\{15,25\}+25+15 = 55\\
opt(3,4) &= \min\{0+opt(4,4), opt(3,3)+0\} + (f_3+f_4) = \min\{10,15\}+15+10 = 35\\
opt(4,5) &= \min\{0+opt(5,5), opt(4,4)+0\} + (f_4+f_5) = \min\{30,10\}+10+30 = 50\\
opt(1,3) &= \min\{0+f(2,3), f(1,1)+f(3,3), f(1,2)+0\} + (f_1+f_2+f_3)\\
&= \min\{55,20,35\}+5+25+15 = 65\\
opt(2,4) &= \min\{0+f(3,4), f(2,2)+f(4,4), f(2,3)+0\} + (f_2+f_3+f_4)\\
&= \min\{35,35,55\}+25+15+10 = 85\\
opt(3,5) &= \min\{0+f(4,5), f(3,3)+f(5,5), f(3,4)+0\} + (f_3+f_4+f_5)\\
&= \min\{50,45,35\}+15+10+30 = 90\\
opt(1,4) &= \min\{0+f(2,4), f(1,1)+f(3,4), f(1,2)+f(4,4), f(1,3)+0\}\\
&\qquad + (f_1+f_2+f_3+f_4) = 95\\
&= \min\{85,40,45,65\}+5+25+15+10\\
opt(2,5) &= \min\{0+f(3,5), f(2,2)+f(4,5), f(2,3)+f(5,5), f(2,4)+0\}\\
&\qquad + (f_2+f_3+f_4+f_5) = 155\\
&= \min\{90,75,85,85\}+25+15+10+30\\
opt(1,5) &= \min\{0+f(2,5), f(1,1)+f(3,5), f(1,2)+f(4,5), f(1,3)+f(5,5)+f(1,4)+0\}\\
&\qquad + (f_1+f_2+f_3+f_4+f_5)\\
&= \min\{155,95,85,95,95\}+5+25+15+10+30 = 170
\end{aligned}$$

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

$$opt[i,j] = \begin{cases} 0, & i = j \\ \min_{k: i \le k < j}(opt[i,k] + opt[k+1,j] + r_i \times c_k \times c_j), & j < j \end{cases} \tag{40.10}$$

The algorithm is as following

Algorithm 35 MatrixChainMultiplication

```
1: $opt[i,i] \leftarrow 0 \quad \forall\ i = 1, 2, \cdots, n$
2: for $l \leftarrow 2$ to $n$ do
3:     for $i \leftarrow 1$ to $n - l + 1$ do
4:         $j \leftarrow i + l - 1$
5:         $opt[i,j] \leftarrow \infty$
6:         for $k \leftarrow i$ to $j - 1$ do
7:             if $opt[i,k] + opt[k+1, j + r_i c_k c_j < opt[i,j]$ then
8:                 $opt[i,j] \leftarrow opt[i,k] + opt[k+1,j] + r_i c_k c_j$
9:                 $\pi[i,j] \leftarrow k$
10: return $opt[1,n]$
```

With above algorithm, to construct the optimal solution, the follow algorithm is needed

40.4 Compare between three paradigms

Algorithm 36 PrintOptimalOrder(i, j)

```
1: if $i == j$ then
2:     Print($A_i$)
3: else
4:     Print("(")
5:     PrintOptimalOrder($i, \pi[i, j]$)
6:     PrintOptimalOrder($\pi[i, j] + 1, j$)
7:     Print(")")
```

Chapter 41

Sorting

41.1 Exchange Sorts

41.1.1 Bubble Sort

41.1.2 Cocktail Shaker Sort

41.1.3 Odd-even Sort

41.1.4 Comb Sort

41.1.5 Gnome Sort

41.1.6 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 37 Quicksort(A, n)

```
1: if $n = 1$ then return A
2: Initial, $A_L \leftarrow \emptyset, A_H \leftarrow \emptyset$
3: $x \leftarrow$ Median(A)
4: for $element \in A$ do
5:     if $element \leq x$ then
6:         $A_L \leftarrow A_L \cup \{element\}$
7:     else
8:         $A_H \leftarrow A_H \cup \{element\}$
9: $B_L \leftarrow$ Quicksort($A_L$, $A_L$.size)
10: $B_H \leftarrow$ Quicksort($A_H$, $A_H$.size)
11: $t \leftarrow$ 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 38 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 39 Selection(A, n, i)

```
1: if $n = 1$ then
2:     return A
3: else
4:     $x \leftarrow$ Median(A)
5:     for $element \in A$ do
6:         if $element \leq x$ then
7:             $A_L \leftarrow A_L \cup \{element\}$
8:         else
9:             $A_H \leftarrow A_H \cup \{element\}$
10:    if $i \leq A_L.size$ 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 $O(n)$

41.2 Selection Sorts

41.2.1 Selection Sort

41.2.2 Heapsort

41.2.3 Smoothsort

41.2.4 Cartesian Tree Sort

41.2.5 Tournament Sort

41.2.6 Cycle Sort

41.2.7 Weak-heap Sort

41.3 Insertion Sorts

41.3.1 Insertion Sort

41.3.2 Shell Sort

41.3.3 Splaysort

41.3.4 Tree Sort

41.3.5 Library Sort

41.3.6 Patience Sorting

41.4 Merge Sorts

41.4.1 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 40 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($B, C$, $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$
```

41.4.2 Cascade Merge Sort

41.4.3 Oscillating Merge Sort

41.4.4 Polyphase Merge Sort

41.5 Distribution Sorts

41.5.1 American Flag Sort

41.5.2 Bead Sort

41.5.3 Bucket Sort

41.5.4 Burstsort

41.5.5 Counting Sort

41.5.6 interpolation Sort

41.5.7 Pigenhole Sort

41.5.8 Proxmap Sort

41.5.9 Radix Sort

41.5.10 Flashsort

41.6 Concurrent Sorts

41.6.1 Bitonic Sorter

41.6.2 Batcher Odd-even Mergesort

41.6.3 Pairwise Sorting Network

41.6.4 Samplesort

41.7 Hybird Sorts

41.7.1 Block Merge Sort

41.7.2 Timsort

41.7.3 Spreadsort

41.7.4 Merge-insertion Sort

41.8 Please Don't Do that Sorts

41.8.1 Slowsort

41.8.2 Bogosort

41.8.3 Stooge Sort

Chapter 42

Mathematical Algorithm

42.1 Polynomial Multiplication

For given two polynomials of degree $n-1$, the algorithm outputs the product of two polynomials.

Example.

$$(3x^3 + 2x^2 - 6x + 9) \times (-2x^3 + 7x^2 - 8x + 4) \tag{42.1}$$
$$= -6x^6 + 17x^5 + 24x^4 - 60x^3 + 119x^2 - 96x + 36 \tag{42.2}$$

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 41 PolyMultNaive(A, B, n)

```
1: Let $C[k] = 0$ for every $k = 0, 1, \cdots, 2n-2$
2: for $i \leftarrow 0$ to $n-1$ do
3:     for $j \leftarrow 0$ to $n-1$ do
4:         $C[i+j] \leftarrow C[i+j] + A[i] \times B[j]$
   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.

$$p(x) = p_H(x)x^{\frac{n}{2}} + p_L(x) \tag{42.3}$$

Both p_H and p_L are polynomials with degree of $\frac{n}{2} - 1$, then

$$p(x)q(x) = (p_H(x)x^{\frac{n}{2}} + p_L(x)) \times (q_H(x)x^{\frac{n}{2}} + q_L(x)) \tag{42.4}$$
$$= p_H q_H x^n + (p_H q_L + p_L q_H)x^{\frac{n}{2}} + p_L q_L \tag{42.5}$$

Therefore

$$\begin{aligned}
\mathit{multiply}(p, q) &= \mathit{multiply}(p_H, q_H)x^n & (42.6)\\
&+ (\mathit{multiply}(p_H, q_L) + \mathit{multiply}(p_L, q_H))x^{\frac{n}{2}} + \mathit{multiply}(p_L, q_L) & (42.7)\\
&= \mathit{multiply}(p_H, q_H)x^n & (42.8)\\
&+ (\mathit{multiply}(p_H + p_L, q_H + q_L) - \mathit{multiply}(p_H, q_H) - \mathit{multiply}(p_L, q_L))x^{\frac{n}{2}} & (42.9)\\
&+ \mathit{multiply}(p_L, q_L) & (42.10)\\
& & (42.11)
\end{aligned}$$

The algorithm is as following
The running time $T(n) = 3T(n/2) + O(n)$, $T(n) = O(n^{\lg_2 3})$

Algorithm 42 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:     $C[i] \leftarrow C[i] + C_L[i]$
10:    $C[i+n] \leftarrow C[i+n] + C_H[i]$
11:    $C[i+n/2] \leftarrow C[i+n/2] + C_M[i] - C_L[i] - C_H[i]$
   return $C$
```

42.2 Matrices Multiplication

42.3 Gaussian Elimination

42.4 Curve Fitting

42.5 Integration

Chapter 43

Searching

Chapter 44

String

44.1 String Searching

44.2 Pattern Matching

44.3 Longest Common Subsequence

44.4 Parse

44.5 Optimal Caching

44.6 File Compression

44.7 Cryptology

Chapter 45

Data Structures

45.1 Elementary Data Structures

45.2 Hash Tables

45.3 Binary Search Trees

45.4 Red-Black Trees

45.5 B-Trees

45.6 Fibonacci Heaps

45.7 van Emde Boas Trees

Chapter 46

NP and Computational Intractability

46.1 P, NP and Co-NP

Definition 46.1.1 (Decision Problem)**.** A problem X is a **decision problem** if the output is either 0 or 1 (yes/no). Further, a

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 46.1.2 (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 46.1.3 (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 46.1.4 (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 46.1.5 (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 46.1.6 (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 46.1.7 (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 wither 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 the algorithm it 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 $P = Co-NP$

46.2 Polynomial-Time Reductions

Definition 46.2.1 (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

$$X \leq_P Y \tag{46.1}$$

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 46.1. *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 46.2. *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 46.3. *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 46.4. *Vertex Cover $\leq_P$ Set Cover*

Proof. □

Lemma 46.5. *Set Cover $\leq_P$ Vertex Cover*

Proof. □

Lemma 46.6. *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 46.7. *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 46.8. *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 46.9. *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 46.10. *Independent Set $\leq_P$ Set Packing*

46.3 NP-Completeness

Definition 46.3.1 (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 46.11 (Cook's Theorem)**.** *Circuit Satisfiability is NP-Complete.*

Proof. □

Theorem 46.12. *If X is NP-Complete and $X \in P$, then $P = NP$*

Proof. Direct result from Cook's theorem. □

46.4 NP-Complete Problems

Special Topic: Computational Geometry

Chapter 47

Convex Hull

47.1 Computing Slope Statistics

47.2 Convexity

47.3 Graham's Scan

47.4 Turning and orientations

Chapter 48

Intersections

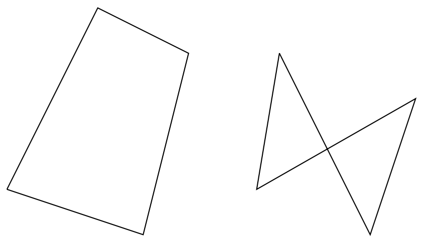
Chapter 49

Triangulation and Partitioning

49.1 Polygon Triangulation

49.1.1 Types of Polygons

Definition 49.1.1 (simple polygon)**.** A **simple polygon** is a closed polygonal curve without self-intersection.

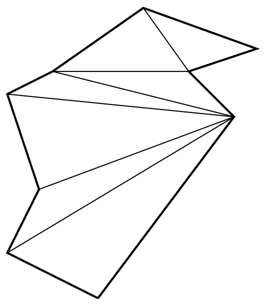


Simple Polygon   Non-simple Polygon

Polygons are basic building blocks in most geometric applications. It can model arbitrarily complex shapes, and apply simple algorithms and algebraic representation/manipulation.

49.1.2 Triangulation

Definition 49.1.2 (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 49.1. *Every polygon has a triangulation*

Lemma 49.2. *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 $\bar{q}r$ 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 $\bar{p}s$ is a diagonal. □

49.1.3 Art Gallery Theorem

Theorem 49.3. *Every n-gon can be guarded with* $\lfloor \frac{n}{3} \rfloor$ *vertex guards*

Lemma 49.4. *Triangulation graph can be 3-colored.*

Problem 49.1. 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. □

49.1.4 Triangulation Algorithms

Chapter 50

Voronoi Diagrams

Chapter 51

Arrangement and Duality

Chapter 52

Delaunay Triangulations

Chapter 53

Search

Chapter 54

Motion Planning

Chapter 55

Quadtrees

Chapter 56

Visibility Graphs

Part VIII

Stochastic Methods

Chapter 57

Markov Chain

Chapter 58

Queueing Theory

Chapter 59

Inventory Theory

Part IX

Game Theory

Part X

Simulation