Algorithm Design and Analysis

Vinsong

November 24, 2025

Abstract

The lecture note of 2025 Fall Algorithm Design and Analysis by professor 呂學一. 希望我可以活著度過這學期∼∼∼∼

Contents

Chapter 0

Introduction

Lecture 1

0.1 Design and Analysis

4 Sep. 14:20

0.1.1 Design

Remark. Find the point to cut into the problem.

Question (Coffee and Milk). 把 500 毫升的咖啡倒入 10 毫升，再從 510 毫升牛奶咖啡取 10 毫升倒入 490 毫升牛奶中，試問兩邊比例？

Answer. 兩邊都固定 500 毫升，一邊少的必定出現在另一邊，**切入點對了根本不用計算** ⊛

0.1.2 Analysis

Question (Card). 把牌洗亂（平均）需要幾次？

Note. **定義何為亂？**
排列出現機率皆為

$$\frac{1}{52!}$$

七次是充分必要條件（嚴謹分析 on paper）n card should shuffle $\frac{3}{2}\log_2 n + \theta$ times. ⊛

Definition 0.1.1 (亂). With n-cards, we have to let the probbability of every combination become

$$\frac{1}{n!}$$

Question (Top-in shuffle). Consider Top-in shuffle with the cards. How to get it "randomly" ?

Answer. Define the k-th section to be 初始底牌從底下數上來是 k-th card.

1. bottom $k-1$ cards must be 亂
2. 每次都可以用 n/k 次將他洗亂，因為出現機率皆為 k/n

We can shuffle $n \cdot H_n$ times. ⊛

Theorem 0.1.1. 底下 $k-1$ 張卡片永遠是亂的

Proof. 考慮 top-in shuffle，利用數學歸納法

- 第一輪要插入底牌下方，只有 1 個空隙，因此必須插入，因此插入的機率是
$$\frac{1}{1!}$$
- 底下如果有 k 張牌，假設下面 k 張是亂的，表示他的排列 $k!$ 種，每種順序機率都是
$$\frac{1}{k!}$$
- 再插入一張，共有 $k+1$ 個空隙，排起來每種順序出現的機率為
$$\frac{1}{(k+1)} \cdot \frac{1}{k!} = \frac{1}{(k+1)!}$$
符合亂的定義

■

第 k 階段插入到下面都是從 n 個空隙裡面找到 k 個空隙插入，因此出現機率必定為 $\frac{k}{n}$，因此需要 shuffle 次數為
$$\frac{n}{k}$$
接著考慮第 n 階段，底牌不是亂的，因此要再洗一次，因此最終的和為
$$\sum_{i=1}^{n} \frac{n}{i} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = n \cdot H_n$$

Note. choose another card to be "bottom", 可以減少第一次的 $1/n$ 就可以少 $n/1$ 次 shuffle. 因此可以把次數減少為：
$$n \cdot H_n - n$$

Remark. **簡單的分析點交換就可以造成巨大的影響**

0.2 Jargons

Definition 0.2.1 (Problems). 「問題」(Problem) 是一個對應關係，就是一個函數

- 演算法核心是在探討問題的解決難易度
- 有些問題確定很難，就不用妄想想出簡單演算法

Definition 0.2.2 (Instance). 「個例」(instance)，也就是問題的合法輸入

Definition 0.2.3 (Computation Model). 「計算模型」(Computation Model)，也就是遊戲規則，同個問題在不同的規則下可能難易度不同

- Comparision base & Computation base

Definition 0.2.4 (Algorithm). 「演算法」Algorithm is a detail step-by-step instruction

- 符合規則
- 詳細步驟

Definition 0.2.5 (Hardness). 「難度」(Hardness)，想知道一個「**問題**」有多難解，用最厲害的一個「**解法**」，對於每個「**個例**」，都至少要用多少「**工夫**」才能解完

- 魔方問題：對於所有解法，存在至少一個初始 instance 讓解法需要 20 次才能轉完，切入點是找到一個固定的初始狀態，這是一個已經最佳化的問題

Theorem 0.2.1 (Confirm Hardness). 用 **upper bound** 和 **lower bound** 去夾起來決定難度

- 當 **upper bound** = **lower bound** 的時候，我們才知道問題的確切難度
- 有些情況，就算夾起來也不一定可以確定難度

Proof.

$$\begin{array}{ccc} \boxed{\text{upper bound}} & & \\ \Downarrow & & \boxed{\text{upper bound}} \\ & \Rightarrow & \Downarrow \\ & & \Uparrow \\ \Uparrow & & \boxed{\text{lower bound}} \\ \boxed{\text{lower bound}} & & \end{array}$$

■

Note. 我們在這門課都討論 worst case

Chapter 1

Complexity for a Problem

Lecture 2

11 Sep. 13:20

1.1 函數成長率 (Rate of Growth)

Question (棋癡國王與文武大臣). 國王愛下棋，文武大臣要獎賞

- 武大臣每下一個棋子，獎賞多一袋米，起始為一袋米
- 文大臣每下一個棋子，獎賞雙倍，起始為一粒米

Answer. 棋盤 64 格

- 武大臣：n 袋米
- 文大臣：2^n 粒米

2^n 的成長率遠遠高於 n，單位的影響不及成長率 ⊛

1.2 成長率的比較

Note. 雖然 200 年前就有 Asymptotic Notation 的概念，但直到 1970 年代才被演算法分析之父 Donald Ervin Knuth 正式定義到 CS 領域內。

Question (Why Asymptotic Notation). 為什麼要用 Asymptotic Notation？

Answer. 問題難度通常單位不一致

- $n = 3$ 魔方問題要 20 轉
- n 個信封的老大問題要 $n - 1$ 次比較

兩者難度無法比較 ⊛

Definition 1.2.1 (Rate of Growth). 沒有人有明確定義，但是成長率很好比較，有很多東西也是無法定義但可以比較，e.g. 無限集合可以比大小。

1.3 Big Oh Notation

Definition 1.3.1 (Big Oh Notation)**.** For functions $f, g : \mathbb{N} \to \mathbb{R}$, we write

$$f(n) = O(g(n))$$

to satisfy the extistence of positive constants c and n_0 such that the inequality

$$0 \le f(n) \le c \cdot g(n)$$

holds for all integer $n \ge n_0$.

Note. $f(n), g(n)$ should be non-negative for sufficiently large n.

The definition of

$$f(n) = O(g(n))$$

says that there exist a positive constant c such that the value of $f(n)$ is upper-bounded by $c \cdot g(n)$ for all sufficiently large positive n.

Remark. 因此 $O(g(n))$ 可以理解成一個成長率不高過 g 的函數所成的集合

1.3.1 等號左邊也有 Big-Oh

Definition 1.3.2. The equality $O(g(n)) = O(h(n))$ signifies that

$$f(n) = O(h(n))$$

holds for all functions $f(n)$ with

$$f(n) = O(g(n))$$

i.e. $O(g(n)) = O(h(n))$ signifies that $f(n) = O(g(n))$ implies $f(n) = O(h(n))$.

The equality $=$ in $O(g(n)) = O(h(n))$ is more like $\subseteq$, i.e., $O(g(n)) \subseteq O(h(n))$.

Theorem 1.3.1. $O(g(n)) = O(h(n))$ if and only if $g(n) = O(h(n))$.

Proof. Consider the two directions separately.

- For the $(\Rightarrow)$ case: We can easily proof that

$$g(n) = O(g(n))$$

then we can deduce that

$$g(n) = O(g(n)) = O(h(n))$$

- For the $(\Leftarrow)$ case:

> **As previously seen** (Definition 1.3.1).
>
> $$g(n) = O(h(n)) \quad \Rightarrow \quad \exists\, c_1, n_1 > 0,\ \forall n \geq n_1,\ 0 \leq g(n) \leq c_1 \cdot h(n)$$

Let f be the function such that $f(n) = O(g(n))$. Then, by definition, we can deduce that

$$\exists\, c_2, n_2 > 0,\ \forall n \geq n_2,\ 0 \leq f(n) \leq c_2 \cdot g(n).$$

Assume $n \geq \max\{n_1, n_2\}$. Then, we have

$$0 \leq f(n) \leq c_2 \cdot g(n) \leq c_2 \cdot (c_1 \cdot h(n)) = (c_1 c_2) \cdot h(n).$$

Thus, we can conclude that

$$f(n) = O(g(n)) = O(h(n))$$

Hence,

$$O(g(n)) = O(h(n)) \Leftrightarrow g(n) = O(h(n)).$$

■

1.4 Big-Oh 的運算

Question. 所以，Big-Oh 相加的意思是什麼？

Definition 1.4.1 (Big-Oh Addition). The equality

$$O(g_1(n)) + O(g_2(n)) = O(h(n))$$

signifies that the equality

$$f_1(n) + f_2(n) = O(h(n))$$

holds for any functions $f_1(n)$ and $f_1(n)$ with

$$f_1(n) = O(g_1(n))$$

$$f_2(n) = O(g_2(n)).$$

That is, $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ together imply $f_1(n) + f_2(n) = O(h(n))$.

Remark. 雖然 $O(g_1(n)) + O(g_2(n))$ 看起來像是兩個集合的聯集，但相同集合想法**無法帶到減乘除**。

Definition 1.4.2 (Big-Oh $\circ$). The equality

$$O(g_1(n)) \circ O(g_2(n)) = O(h(n))$$

$$g_1(n) \circ O(g_2(n)) = O(h(n))$$

集合的復合操作

Notation.
$$\{f_1(n) \circ f_2(n) \mid f_1(n) \in S_1 \text{ and } f_2(n) \in S_2\}$$

可以被理解成

- 把 $=$ 解成 $\subseteq$
- 把 $g_1(n)$ 理解成 $\{g_1\}$
- $O(g1(n))$ 解為成長率不超過 g_1 的成長率的所有函數所組成的集合

Remark. 減乘除應被理解成與剛剛加法類似的模式，而無法被理解為集合的運算

Definition 1.4.3 (Big-Oh $-$, $\cdot$, $/$)**.** (Take $-$ as the example) The equality

$$O(g_1(n)) - O(g_2(n)) = O(h(n))$$

signifies the equality

$$f_1(n) - f_2(n) = O(h(n))$$

holds for any functions $f_1(n)$ and $f_2(n)$ with

$$f_1(n) = O(g_1(n))$$

$$f_2(n) = O(g_2(n))$$

Question. Proof or disproof:

$$O(n)^{O(\log_2 n)} = O(2^n)$$

Answer. First, we take log on both sides:

$$\text{LHS} = O(\log n) \cdot O(\log n) = (O(\log n))^2$$

$$\text{RHS} = O(n)$$

LHS grows slower than RHS, therefore the original statement is true. ⊛

Remark. log 的底數不影響成長率，因此可忽略。

Definition 1.4.4 (Big-Oh 套 Big-Oh)**.** The equality

$$O(O(g(n))) = O(h(n))$$

signifies that the equality

$$O(f(n)) = O(h(n))$$

holds for any function f with

$$f(n) = O(g(n))$$

i.e. $f(n) = O(g(n))$ implies $O(f(n)) = O(h(n))$.

Theorem 1.4.1. $g(n) = O(h(n))$ if and only if $O(O(g(n))) = O(h(n))$

Proof. Consider the two directions separately.

- For the ($\Rightarrow$) case:

  > **As previously seen** (Definition 1.3.1).
  >
  > $$g(n) = O(h(n)) \implies \exists\, c_0, n_0 > 0,\ \forall n \geq n_0,\ 0 \leq g(n) \leq c_0 \cdot h(n)$$

  $f(n) = O(O(g(n)))$ signifies that for $c_1, c_2, n_1, n_2 > 0$

  $$\forall n \geq n_1,\ 0 \leq f(n) \leq c_2 \cdot u(n); \quad \forall n \geq n_2,\ 0 \leq u(n) \leq c_1 \cdot g(n)$$

  Get all together, we have

  $$0 \leq f(n) \leq c_2 \cdot (c_1 \cdot g(n)) \leq c_2 c_1 c_0 \cdot h(n) \quad \implies \quad f(n) = O(h(n))$$

  Thus, we can conclude that

  $$O(O(g(n))) = O(h(n))$$

- We can easily proof that

  $$g(n) \subseteq O(g(n)) \subseteq O(O(g(n)))$$

  Then we can get

  $$g(n) = O(O(g(n))) = O(h(n))$$

Hence, $g(n) = O(h(n)) \Leftrightarrow O(O(g(n))) = O(h(n))$ ■

Theorem 1.4.2 (Rules of Computation in Big-Oh)**.** The following statements hold for functions $f, g : \mathbb{N} \to \mathbb{R}$ such that there is a constant n_0 such that $f(n)$ and $g(n)$ for any integer $n \geq n_0$:

- **Rule 1**: $f(n) = O(f(n))$.
- **Rule 2**: If c is a positive constant, then $c \cdot f(n) = O(f(n))$.
- **Rule 3**: $f(n) = O(g(n))$ if and only if $O(f(n)) = O(g(n))$.
- **Rule 4**: $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$.
- **Rule 5**: $O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$

Proof. For **Rule 5**: By the Definition 1.3.1, $u(n) = O(f(n) \cdot g(n))$ signifies that there exist positive constants c_1 and n_1 such that the inequality

$$\exists\, c_0, n_0 > 0,\ \forall n \geq n_0,\ 0 \leq u(n) \leq c_0 \cdot f(n) \cdot g(n)$$

the definition of $u(n) = f(n) \cdot O(g(n))$ is

$$\exists\, c_1, n_1 > 0,\ \forall n \geq n_1,\ 0 \leq u(n) \leq f(n) \cdot c_1 \cdot g(n)$$

which are equivilence to each other. ■

1.5 More Asymptotic Notation

Definition 1.5.1 (Little-oh). For any function $f, g : \mathbb{N} \to \mathbb{R}$, we write

$$f(n) = o(g(n))$$

to signify that for any constant $c > 0$, there is a positive constant $n_0(c)$ such that

$$0 \leq f(n) < c \cdot g(n)$$

holds for each integer $n \geq n_0(c)$

> **Note.** $n_0(c)$ is a function of c. When we $n_0(c)$ is a constant, we means that it does not depend on n.

白話來說 $f(n) = o(g(n))$ 的定義是說，不管是多小的常數 c，要 n 夠大（i.e., $n \geq n_0(c)$），

$$0 \leq f(n) < c \cdot g(n)$$

都還是成立。

Example.

$$n = o(n^2)$$

Observe that for any positive constant c, as long as $n > \frac{1}{c}$, we have

$$0 \leq n < c \cdot n^2$$

Therefore, we may let $n_0(c) = \frac{1}{c} + 1$ and have $n = o(n^2)$ proved.

Definition 1.5.2 (Other notation). The other notation can be defined via O and o notation:

- We write $f(n) = \Omega(g(n))$ if
$$g(n) = O(f(n)).$$
- We write $f(n) = \Theta(g(n))$ if
$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$
- We write $f(n) = \omega(g(n))$ if
$$g(n) = o(n)$$

Limit notation 可以幫我們判斷各種 Asymptotic Notation:

- If
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$
, the we can guess $f(n) = o(g(n))$.
- If
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$
, the we can guess $f(n) = \Theta(g(n))$.

- If
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$
, the we can guess $f(n) = \omega(g(n))$.

然而，極限不一定應可以推至 Asymptotic Notation:

- Let $f(n) = g(n) = (-1)^n$. We have
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1,$$
but $f(n) \neq O(g(n))$, $f(n) \neq \Omega(g(n))$, and $f(n) \neq \Theta(g(n))$.
- Let $f(n) = (-1)^n$ and $g(n) = n \cdot (-1)^n$. We have
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,$$
but $f(n) \neq o(g(n))$.
- Let $f(n) = 2 + (-1)^n$ and $g(n) = 2 - (-1)^n$. We have
$$f(n) = \Theta(g(n)),$$
but $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist.

Question. Can we just use $\leq$ instead of $<$ in the definition of o ?

Answer. In most part of it will be right. However there will be a special situation:
$$o(0) = 0$$
which is definetly wrong. ⊛

Question. 為何不都用 $\exists\, c_0, n_0$ 或都用 $\forall c, n_0(c)$?

Answer. 如果都用 $\exists\, c_0, n_0$，那 o 就會退化，變成 O 而已，並且 $<, \leq$ 是沒有太大差別的

Proof. Suppose that $\hat{n}_0(c)$ is the constant ensured by the $\leq$-version. We simply let
$$n_0(c) = \max(m_0, \hat{n}_0(c/2)).$$
As a result, for any positive constant c, if $n \geq n_0(c)$, we have $g(n) > 0$ and thus
$$\begin{aligned} 0 < f(n) &\leq \frac{c}{2} \cdot g(n) \\ &< c \cdot g(n). \end{aligned}$$
■

證畢，由此可知符號並無太大影響，不可讓 o 退化 ⊛

Lecture 3

1.6 問題的難度

18 Sep. 14:20

如果，

- P 不比 Q 難且
- Q 不比 P 簡單

那兩個問題的難度相同（兩者等價）

Definition 1.6.1. We say that the (worst-case) time complexity of Problem P is $\Theta(f(n))$ if

- the time complexity of Problem P is $O(f(n))$, i.e.

  there exists an $O(f(n))$-time algorithm that solves Problem P

- the time complexity of Problem P is $\Omega(f(n))$, i.e.

  any algorithm that solves Problem P requires $\Omega(f(n))$ time (in the worst case).

  對於任何演算法，只要存在一組 instance 可以達成，一組 $\Omega(f(n))$ 即可推出

Note. 若沒有特別說，n 代表的是 input(instance) size，儲存資料所需的容量

Note. 「正確的演算法」就是對於所有合法輸入都可以對應出正確的輸出，的解決問題方法

1.7 演算法複雜度比較

$$f(n) = O(g(n)) : \begin{cases} O(f(n)) = O(g(n)) \\ o(f(n)) = O(g(n)) \\ \Theta(f(n)) = O(g(n)) \end{cases}$$

$$f(n) = \Omega(g(n)) : \begin{cases} \Omega(f(n)) = \Omega(g(n)) \\ \omega(f(n)) = \Omega(g(n)) \\ \Theta(f(n)) = \Omega(g(n)) \end{cases}$$

$$f(n) = \Theta(g(n)) : \begin{cases} \Theta(f(n)) = \Theta(g(n)) \end{cases}$$

$$f(n) = o(g(n)) : \begin{cases} O(f(n)) = o(g(n)) \\ o(f(n)) = o(g(n)) \\ \Theta(f(n)) = o(g(n)) \end{cases}$$

$$f(n) = \omega(g(n)) : \begin{cases} \Omega(f(n)) = \omega(g(n)) \\ \omega(f(n)) = \omega(g(n)) \\ \Theta(f(n)) = \omega(g(n)) \end{cases}$$

Comparing Algorithm A and B, We say that Algorithm A is no worse than Algorithm B in terms of worst-case time complexity if there exists a function $f : \mathbb{N} \to \mathbb{R}$ such that

- Algorithm A runs in time $O(f(n))$
- Algorithm B runs in time $\Omega(f(n))$ (in the worst case)

> **Remark.** 第一句 Big-Oh 並沒有出現「in the worst case」是因為我們在此處分析的是「**worst case complexity**」，所以其實在 lower bound 分析的時和通常也不說。

Comparing Algorithm A and B, We say that Algorithm A is strictly better than Algorithm B in terms of worst-case time complexity if there exists a function $f : \mathbb{N} \to \mathbb{R}$ such that

- Algorithm A runs in time $O(f(n))$
- Algorithm B runs in time $\omega(f(n))$ (in the worst case)

or

- Algorithm A runs in time $o(f(n))$
- Algorithm B runs in time $\Omega(f(n))$ (in the worst case)

1.8 分析演算法複雜度下界

儘管有些 case 可以，但 Big-Omega 不可以跟 Big-Oh 一樣分析（多增加）

> **Remark.** **Ω-time 必須要一組一組 instance 分析**

1.9 問題上下界 vs 演算法上下界

- 一個問題 P 的任何正確演算法 A 的複雜度上界都是問題 $O(f(n))$ 都是問題 P 的複雜度上界
- 一個問題 P 的複雜度下界 $\Omega(f(n))$ 都是 P 的任何正確演算法 A 的複雜度下界

Chapter 2

演算法的設計與分析

2.1 Half Sorted

Definition 2.1.1 (Half Sorting Problem). An n-element array A is half-sorted if

$$A[i] \leq A\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$$

holds for each index i with $2 \leq i \leq n$.

Half-sorting Problem:

- Input:

  An array A of n distinct numbers.

- Output:

  A half-sorted array that is reordered from A.

Note. 正確的輸出未必唯一，因此輸入輸出就不是一個函數，而是一個「relation」

2.1.1 排序法 Sorting method

Theorem 2.1.1. 歸約 Reduction（問題重整），把問題的難度如果問題 P 可以「多項式時間歸約」成問題 Q，就寫作

$$P \leq_p Q$$

意思是：只要能解決問題 Q，就能透過快速轉換來解決問題 P，所以：

- 如果 Q 是容易的（有快速演算法），那麼 P 也會是容易的。
- 如果 P 已知很難，那麼 Q 至少也不會比較容易。

Note. 把問題的性質變強，便可以順便證明性質較弱的問題

因此，我們知道用排序法一定可以解決半排法，我們可以把半排問題「歸約」到「排序」問題，因此我們首先分析一下快速排序法：

Listing 2.1: Quicksort in Python

```python
def qsort(A, l, r):
    if l >= r:
        return
    (i, j, k) = (l, r, A[l])

    while i != j:
        while A[j] > k and i < j:
            j -= 1
        while A[i] <= k and i < j:
            i += 1
        if i < j:
            (A[i], A[j]) = (A[j], A[i])

    (A[l], A[i]) = (A[i], k)

    qsort(A, l, i-1)
    qsort(A, i+1, r)
```

我們必須分析他的正確性及複雜度

Theorem 2.1.2. The function `qsort()` is correct.

Proof. First, we know that every round of `qsort()` will let the array become:

$$A[l\ ..\ p-1] < A[p] < A[p+1\ ..\ r] \quad A[p] = \text{pivot}$$

(How to proof)

Let m be the number of elements in the array. By the induction, we can start with

- Case $m = 1$: The array is well sorted.
- Case $\forall t \leq m \to (m+1)$: Every round of iteration we can get a p such that

  $$\forall x \in A[l\ ..\ p-1],\ x \leq A[p], \quad \forall y \in A[p+1\ ..\ r],\ y \geq A[p]$$

  We assume that array with length equal to t, $\forall t \leq m$, has been sorted. Then we can know that that `qsort(A, l, p-1)`, `qsort(A, p+1, r)` is well sorted. Thus, we can combined $A[l\ ..\ p-1]$, $A[p]$, $A[p+1\ ..\ r]$ to get a well-sorted array $A[l\ ..\ r]$ with length m.

Hence, by induction, `qsort()` is correct. ∎

Then, we can stat to analyze the time complexity (worst case):

2.1.2 順調法

Definition 2.1.2 (順調法).

為了方便觀察我們可以將這個陣列化成樹的形式（不是真的改變資料結構）

- Each $A[i]$-to-root path is increasing

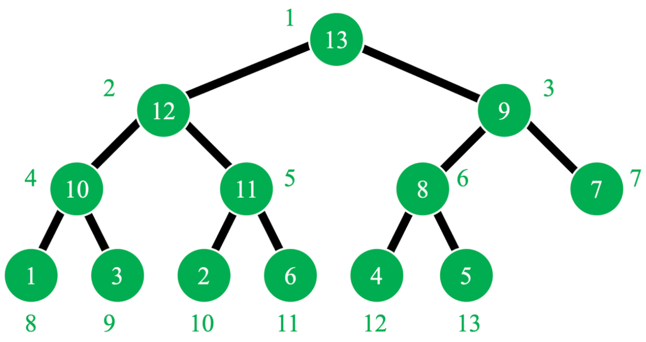


Figure 2.1: Display with Tree structure

2.1.3 逆調法

Lecture 4

25 Sep. 14:20

2.2 Sorting Problem

Note. quick sort

Note. half sort sort

2.2.1 排序問題下界

解決了排序問題的下界就可以一次解決

- The (worst-case) time complexity of the comparison-based sorting problem is $\Omega(n \log n)$.
- The $O(n \log n)$-time analysis for the Half-Sort-Sort algorithm is tight.
- Learning Reduction

Definition 2.2.1 (Permutation Problem). For the instance

- Input: An array A of n distinct integers.
- Output: Reorder the n-index array $B = [1, 2, \cdots, n]$ such that

$$A[B[1]] < A[B[2]] < \cdots < A[B[n]].$$

排列難度 $\leq$ 排序難度。If the comparison-based sorting problem can be solved in $O(f(n))$ time, then so can the comparison-based permutation problem.

2.3 Amortized Analysis

Chapter 3

Advanced Analysis Techniques

Lecture 5

9 Oct. 14:20

3.1 Greedy Algorithm

3.2 Devide and Conquer

3.3 Dynamic Programming

Chapter 4

Graph Theory: Path and Shortest Path Problems

Lecture 8

Definition 4.0.1 (path). Let G be an n-vertex m-weighted directed graph with weight w (which can be positive, negative or zero). The weight of a path P of G is defined as 30 Oct. 14:20

$$w(P) = \sum_{xy \in E(P)} w(xy)$$

For vertices u and v of G, we call a path of G from u to v a **uv-path** of G.

Definition 4.0.2 (distance). For vertices u and v of G, the **distance** from u to v in G, denoted by $d_G(u, v)$, is defined as

$$d_G(u,v) = \begin{cases} \infty & \text{if there is no uv-path in } G \\ w(P) & \forall Q \in \text{uv-paths},\ w(P) \leq w(Q) \\ -\infty & \text{otherwise} \end{cases}$$

Comment (1). 這個 P 就叫做 **shortest uv-path**

Comment (2). 真正的 path 是不允許重複經過點的。這裡定義的 path 其實在真正的 graph theory 裡面叫做 **walk**。$V(P)$, $E(P)$ 都是 multiset。

4.1 Single-Source Shortest Path Problem

Problem 4.1.1 (Single-Source Distance Problem). Given

- Input: a directed graph G with edge weights $w : E(G) \to \mathbb{R}$ and a **source** vertex $r \in V(G)$.
- Output: $d_G(r, v)$ for all vertices $v \in V(G)$.

Note. 我們可以用下面這個問題可以規約 (reduce) 到上面的問題

Problem 4.1.2 (Single-Source Shortest Path Problem). Given

- Input: a directed graph G with edge weights $w : E(G) \to \mathbb{R}$ and a **source** vertex $r \in V(G)$.
- Output: a (shortest-path) tree T of G rooted at r such that if G contains a shortest rv-path of G, then rv-path of T is a shortest rv-path of G.

所以我們應該要解決 Single-Source Distance Problem，先做兩個假設 $m = \Omega(n)$

Comment (1). 我們可以用 DFS 先處理掉 r 無法到達的點，所以可以假設

$$d_G(r, v) < \infty, \ \forall v \in V(G)$$

Comment (2). r 固定，簡寫 $d(v) := d_G(r, v)$

4.1.1 Bellman-Ford Algorithm

Algorithm. For each vertex $v \in V(G)$, we use $d[v]$ to estimate $d(v)$.

- Initialization
$$d[i] = \begin{cases} 0 & i = r \\ \infty & \text{otherwise} \end{cases}$$
- Repeat $n - 1$ times relaxation step: for each edge $uv \in E(G)$, 更新
$$d[v] = \min\{d[v], \ d[u] + w(uv)\}$$
- Relaxation 結束後，For each edge $uv \in E(G)$, if $d[v] > d[u] + w(uv)$, then
$$d[v] = -\infty$$
- For each vertex $v \in V(G)$, 如果他可以被任何 u which $d[u] = -\infty$ reach（DFS $O(m + n)$），則
$$d[v] = -\infty$$

Note. The running time is $O(mn)$.

Proof. 我們先做一些觀察

Observation (1). 在 $n - 1$ 次 relaxation 後，$\forall v \in V(G)$, $d[v] \geq d(v)$，永遠不會小於真正的 $d(v)$.

Observation (2). If P is a shortest rs-path of G for some $s \in V(G)$,

- 在這條 rs-path 上的每一個 v of P，這條 rv-path 也會是 shortest rv-path of G.
- 對於每一個 edge uv of P, if 在先前的 relaxation step 後，會有

$$d[u] = d(u)$$

在這次 relaxation step 後，會有

$$d[v] = d(v)$$

現在我們來證明 Bellman-Ford Algorithm 的正確性。我們分三種情況討論，Case 1 已經在之前就證明可以用 DFS 處理掉了。

- Case 2: $d(v) \neq -\infty$. Let P be a shortest rv-path of G. for each vertex u_j of P, where $j = 0, 1, \ldots, |V(P)| - 1$, $u_0 = r$ and $u_{|V(P)|-1} = v$. 根據我們的 Obs.2，我們知道在第 i 次 relaxation step 後

$$d[u_j] = d(u_j) \quad \forall j \in \{0, \cdots, \min(i, |V(P)| - 1)\}$$

- Case 3: $d(v) = -\infty$: 因為到達不了的點一經被處理掉了，因此必定存在 rv-path P of G, which contain a cycle C such that $w(C) < 0$. 所以我們可以

Claim. At the end of n-th round,

$$d[u] = -\infty \quad \forall u \in V(C)$$

By $u \in V(P)$, we have $d[v] = -\infty$ at the end.

To prove this claim, we assume for contradiction. The n-th of round 並沒有成功更新 $d[u]$ for all $u \in V(C)$. 我們嘗試對每一個邊做 relaxation 都應該失敗。Thus,

$$d[x] + w(xy) \geq d[y] \quad \forall xy \in E(C)$$

把 C 上所有這種 inequality 全部加起來，我們會得到

$$\sum_{xy \in E(C)} w(xy) \geq 0$$

contradiction to $w(C) < 0$.

Hence, Bellman-Ford Algorithm is correct. ■

4.1.2 Lawler's Algorithm

Remark. 針對 Acyclic Graph 的 Algorithm，Since the input graph has no cycle, it has no negative cycle.

Algorithm. 只需要 One Relaxation Step 就可以了

- 用 $O(m+n)$ 做一次 Topological Sort on the input directed acyclic graph G to get a topological order u_i, $\forall i \in \{1, \ldots, n\}$

- Initialization

$$d[u_i] = \begin{cases} 0 & i = 0(d[r] = 0) \\ \infty & \text{otherwise} \end{cases}$$

- For i from 1 to n, we do relaxation step for each $u_i v$

$$d[v] = \min\{d[v],\ d[u_i] + w(u_i v)\}$$

Note. The running time is $O(m + n)$.

Proof. 因為這是一個 DAG，所以做完一次 Topological Sort 後，我們就可以知道每個點的 outgoing edge 順序，也就是知道他們在 shortest path 裡面的順序，因此即便我們不知道這條 $u_i v$-path 在哪裡，但我們可以保證在我們處理到 u_i 的時候，$d[u_i]$ 已經是正確的 $d(u_i)$ 了。因此我們只需要做一次 relaxation step 就可以了。 ■

4.1.3 Dijkstra's algorithm

Remark. 本質上是針對 Non-Negative Weighted Graph 的 Greedy Algorithm，可以做更近一步的簡化，Since the input graph has no negative edge, it has no negative cycle.

Algorithm. One round of estimate improvement suffices, although we cannot rely on topological sort (since G may contain cycles).

- Initialization

$$d[v] = \begin{cases} 0 & v = r \\ \infty & \text{otherwise} \end{cases}$$

- 有 n 次 iteration，每次 iteration 從還沒被處理過的點中選出 $d[u]$ 最小的點 u，並對 u 的每一個 outgoing edge uv 做 relaxation

Note. The running time is $O(m + n \log n)$.

Proof. Let's prove the correctness by contradiction.

1° Let v be the first vertex selected in S with $d[v] \neq d(v)$. We have

$$d[v] > d(v)$$

接下來我們考慮 v 被加入 S 的情況

2° Let P be the shortest rv-path of G.

3° Let xy be an arbitrary edge of P such that $x \in S$ and $y \notin S$. (必定存在因為 $r \in S$ 而 $v \notin S$)

4° 因為我們正要處理 v，所以 xy is already processed, we have

$$d[y] = d(y),\ \ y \neq v$$

5° Since G are nonnegative and y precedes v in P, we have

$$d[y] \leq d(v)$$

6° By 1°, 5°, and 4° we have

$$d[v] > d(v) \geq d[y] = d(y)$$

which contradicts the selection of v. 我們就不該選到 v 因為他並不是最小的那個，還有一個 y 更小

Hence, Dijkstra's algorithm is correct. ■

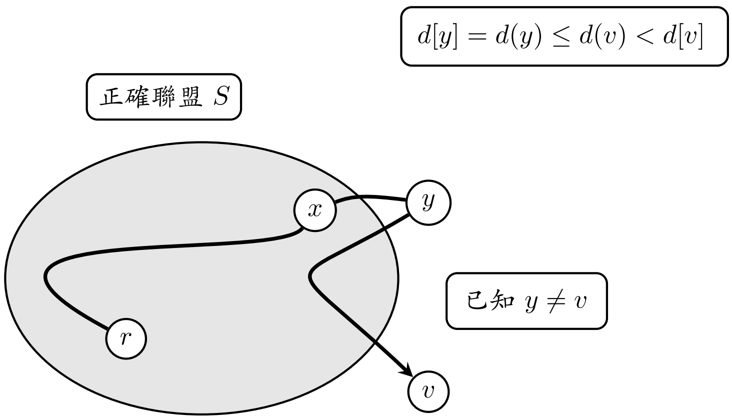


Figure 4.1: Dijkstra's Algorithm Correctness

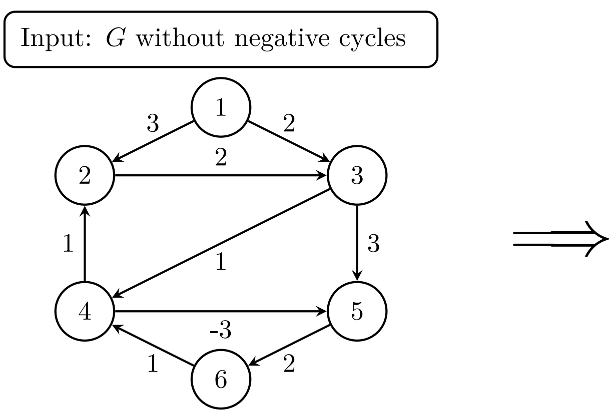
Lecture 9

4.2 All-Pairs Shortest Path Problem

13 Nov. 14:20

Problem 4.2.1 (All-Pairs Distance Problem). Given

- Input: an edge-weighted directed graph G with $V(G) = \{1, 2, \cdots, n\}$ edge weights $w : E(G) \to \mathbb{R}^+$, without negative cycles.
- Output: $d_G(i, j)$ for all $i, j \in V(G)$.



Output: $d_G(i, j)$ for all $i, j \in V(G)$

d	1	2	3	4	5	6
1	0	3	2	3	0	2
2	∞	0	2	1	-2	0
3	∞	∞	0	1	-2	0
4	∞	∞	∞	0	-3	-1
5	∞	∞	∞	∞	0	2
6	∞	∞	0	∞	∞	0

Figure 4.2: All-Pairs Distance Problem Example

Algorithm (Naive Solution). Sloving the single-source shortest path problem for each vertex using Dijkstra, Lawler, or Bellman-Ford algorithm.

4.2.1 A Naive DP Solution

Definition 4.2.1. Let $w_k(i,j)$ be the length of the shortest ij-path in G having at most k edges. It will be ∞ if no such path exists.

$$\begin{cases} w_1(i,j) &= w(ij) \\ w_{n-1}(i,j) &= d_G(i,j) \end{cases}$$

Algorithm. Use the recurrence relation for $w_k(i,j)$ is

$$\begin{cases} w_1(i,j) &= w(ij) \\ w_{2k}(i,j) &= \min_{1\le t\le n} (w_k(i,t) + w_k(t,j)) \end{cases}$$

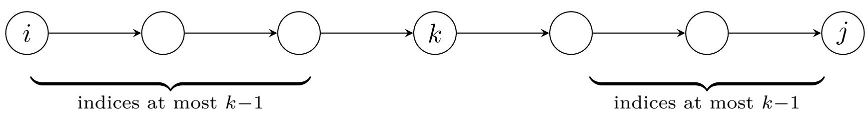
- For each (i,j,k), take $O(n)$ time to compute $w_{2k}(i,j)$ from $w_k(i,j)$.
- For each k, there are n^2 pairs of (i,j), using $O(n^3)$ time to compute all w_{2k} from all w_k.
- It take $O(\log n)$ iterations to compute $d_G = w_{n-1}$ from w_1.

Note. The running time is $O(n^3 \log n)$.

4.2.2 Floyd and Warshall's DP algorithm

Definition 4.2.2. Let $d_k(i,j)$ be the length of the shortest ij-path in G whose intermediate vertices are at most k. It will be ∞ if no such path exists.

$$\begin{cases} d_0(i,j) &= w(ij) \\ d_n(i,j) &= d_G(i,j) \end{cases}$$



Algorithm (Floyd and Warshall's DP Algorithm). Using the recurrence relation for $d_k(i,j)$ is

$$\begin{cases} d_0(i,j) &= w(ij) \\ d_k(i,j) &= \min\{d_{k-1}(i,j),\ d_{k-1}(i,k) + d_{k-1}(k,j)\} \end{cases}$$

- For each (i,j,k), take $O(1)$ time to compute $d_k(i,j)$ from $d_{k-1}(i,j)$.
- For each k, there are n^2 pairs of (i,j), using $O(n^2)$ time to compute all d_k from all d_{k-1}.
- It take n iterations to compute $d_G = d_n$ from d_0.

Note. The running time is $O(n^3)$.

4.2.3 Johnson's Reweighting Technique

Algorithm (Naive Solution with Dijkstra)**.** 如果我們可以拿到一個 nonnegative edge-weight 的 graph，我們就可以簡單地用 Dijkstra's algorithm 來解 All-Pairs Shortest Path Problem

- For each vertex i of G, run Dijkstra's algorithm + Quake heap in $O(m + n \log n)$ time to compute $d_G(i, j)$ for all $j \in V(G)$.

Note. The running time is $O(nm + n^2 \log n)$.

所以我們需要一個方法把有負邊權的 graph 轉換成 nonnegative edge-weight 的 graph，Reweighting w into $\hat{w}$ such that

- $\hat{w}$ is nonnegative
- If $\hat{w}$ is the reweighted shortest ij-path, then the original shortest ij-path is w.

Algorithm (Johnson's Reweighting Technique)**.** Following these steps:

- Assign a weight $h(i)$ to each vertex i of G.
- Let
$$\hat{w}(ij) = w(ij) + h(i) - h(j)$$
- Then for any ij-path P, we have
$$\hat{w}(P) = w(P) + h(i) - h(j)$$

Remark. P is a shortest ij-path in G with respect to $\hat{w}$ if and only if it is a shortest ij-path in G with respect to w.

$(4 + 5 + 3 - 1) + 3 - 2 = (8 + 2 + 1 + 1)$ $\qquad$ $(7 + 2 + 3) + 3 - 2 = (0 + 2 + 1)$

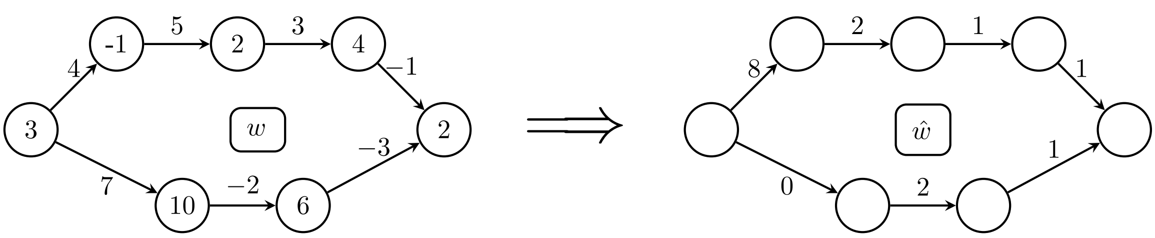


Figure 4.3: Reweighting Example

挑戰是在哪裡找 $h(i)$ 使得 $\hat{w}$ nonnegative？如果有了，我們就可以用 Dijkstra's algorithm 來解 All-Pairs Shortest Path Problem。

Algorithm (Johnson's Technique: Finding $h(i)$)**.** Following these steps:

- Let graph H be obtained by adding a new vertex s to G and adding an edge s_i of weight 0 for each vertex i of G.

  > **Note.** H has no negative cycle iff G has no negative cycle.

- Let $h(i)$ be the distance from s to i in H, i.e.

$$h(i) = d_H(s, i)$$

- The $d_H(s, i)$ can be computed using Bellman-Ford algorithm in $O(m + n)$ time.

Proof. To see that $\hat{w}$ is nonnegative, observe the Figure 4.4.

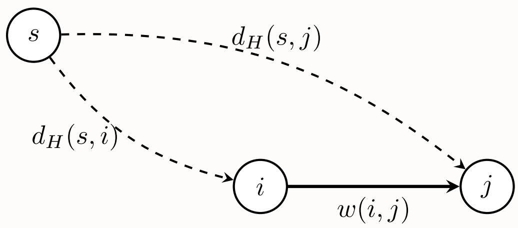


Figure 4.4: Proof of correctness of Johnson's Reweighting Technique

By observation, we have

$$\begin{aligned}\hat{w}(ij) &= w(ij) + h(i) - h(j) \\ &\geq \underbrace{(w(ij) + d_H(s,i))}_{\text{shortest } sj\text{-path which contain } i} - \underbrace{d_H(s,j)}_{\text{shortest } sj\text{-path}} \qquad \text{(Triangle Inequality)} \\ &= 0\end{aligned}$$

■

Recall. Using Naive Solution with

- General edge weights: Bellman-Ford algorithm in $O(mn^2)$ time, which can be $\Theta(n^4)$ when $m = \Theta(n^2)$.
- Acyclic edge weights: Lawler's algorithm in $O(mn + n^2)$ time.
- Nonnegative edge weights: Dijkstra's algorithm in $O(mn + n^2 \log n)$ time.

Using Floyd–Marshall ' s DP algorithm in for general edge weights in $O(n^3)$ time.

Algorithm (Johnson's algorithm). Using Johnson's Reweighting Technique + Dijkstra's algorithm

- Obtain $h(i)$ for all vertex i using Bellman-Ford algorithm in $O(mn)$ time, and get $\hat{w}$ from w in $O(m)$ time.
- For each vertex i of G, run Dijkstra's algorithm + Quake heap in

$$O(m + n \log n)$$

  time on G with edge weights $\hat{w}$ to compute $d_{\hat{G}}(i, j)$ for all $j \in V(G)$. Then obtain a shortest-paths tree of $G(\hat{G})$ rooted at i.
- Compute $d_G(i, j)$ for all $j \in V(G)$ using

$$d_G(i, j) = d_{\hat{G}}(i, j) + h(j) - h(i)$$

  in $O(n^2)$ time.

Note. The running time is $O(mn + n^2 \log n)$.

Chapter 5

Graph Theory: Maximum Flow Problem

Problem 5.0.1 (Maximum Flow Problem). Give

- Input: A directed graph G with edge capacities

$$c : E(G) \to \mathbb{R}^+$$

, and two distinct vertices $s, t \in V(G)$ called **source** and **sink** respectively.

- Output: A "*st*-flow" with maximum "(flow) value".

Comment. 在這個問題下我們允許 multiple/parallel edges 不需要合併成 simple network

Definition. Here are some definitions related to flows:

Definition 5.0.1 (*st*-flow). A *st*-flow is a function

$$f : E(G) \to \mathbb{R}^+ \cup \{0\}$$

that satisfies the following two conditions:

- Capacity constraint:

$$f(e) \leq c(e) \quad \forall e \in E(G)$$

- Conservation law:

$$\sum_{uv \in E(G)} f(uv) = \sum_{vu \in E(G)} f(vu) \quad \forall v \in V(G) \setminus \{s, t\}$$

Definition 5.0.2 (Flow value). The flow value of a flow f is defined as

$$|f| = \sum_{sv \in E(G)} f(sv) - \sum_{us \in E(G)} f(us)$$

5.1 Ford–Fulkerson's Algorithm

Intuition. We can reduce the Maximum Flow Problem into reachability problem for a sequence of **residual graphs** R.

Definition 5.1.1 (Residual Graph)**.** The residual graph $R(f)$ with respect to a flow f of G with $V(G) = V(R(f))$ is defined as follows for each $uv \in E(G)$:

- If $f(uv) < c(uv)$, then $R(f)$ contains an edge uv with capacity
$$c_{R(f)}(uv) = c(uv) - f(uv)$$
- If $f(uv) > 0$, then $R(f)$ contains a reverse edge vu with capacity
$$c_{R(f)}(vu) = f(uv)$$

Comment (1)**.** $R(f)$ 跟 G 一樣，所有 $c(uv)$ 都會是正的，不會是 0 或負的。

Comment (2)**.** G 最多讓 *flow* 增加 2 倍，因為每條邊 uv 在 $R(f)$ 裡面最多會有兩條邊：uv 和 vu，只要兩個條件都達成。

Lemma 5.1.1. For any st-flow f in G, we have the following properties:

- If $d_{R(f)} = \infty$, then f is a maximum st-flow in G.
- If $d_{R(f)} < \infty$, and g is an st-flow in $R(f)$, then $f + g$ remains an st-flow in G, where
$$(f+g)(uv) = f(uv) + g(uv) - g(vu), \quad \forall uv \in E(G)$$

Note. 這裡的 $g(uv), g(vu)$ 都是由原始的 uv-edge 產生的，因此原始圖若有 vu edge 必須分開處理，不能混在上面兩個式子裡面。

Proof. Let f' be the maximum st-flow in G, but not f. We defined h as follows:
$$h(uv) = f(uv) - f'(uv), \quad \forall uv \in E(G)$$
Since f and f' are both st-flows in G, we have conservation law for f and f', so h satisfies conservation law as well.
$$\sum_{uv \in E(G)} h(uv) = \sum_{vu \in E(G)} h(vu) \quad \forall v \in V(G) \setminus \{s, t\}$$
Now consider some vertex $x, y, z \in V(G) \setminus \{s, t\}$. If $h(xy) > 0$, because h satisfies conservation law, there must exist some $h(yz) > 0$. Continuing this process, we can find a path P,
$$P = s \to v_1 \cdots \to v_k \to t \quad \text{such that } h(v_i v_{i+1}) > 0 \quad \forall i = 0, 1, \cdots, k$$
If $h(uv) > 0$, we have
$$f(uv) > f'(uv) \geq 0 \tag{1}$$

we know $f'(uv)$ can not exceed $c(uv)$, so

$$f'(uv) \leq c(uv) \tag{2}$$

by (1) and (2), we have

$$f(uv) < c(uv)$$

which means

$$c_{R(f)}(uv) = c(uv) - f(uv) > 0$$

Therefore, all edges in $R(f)$ along path P have positive capacities. Which is a st-path in $R(f)$, contradicting the assumption that $d_{R(f)} = \infty$.

Comment. 第二個不會

■

Algorithm 5.1: Ford-Fulkerson Algorithm

Input: A flow network $G = (V, E)$ with capacity $c(u, v)$; source s; sink t.
Output: A maximum flow f.

Initialize $f(u, v) \leftarrow 0$ for all $(u, v) \in E$
Compute residual capacity $c_{R(f)}(u, v) = c(u, v) - f(u, v)$ for all (u, v)
while $\exists$ *st − augmenting path P in $R(f)$* **do**
    Obtain an st-path P of $R(f)$, let $q = \min_{uv \in P} c_{R(f)}(u, v)$
    Obtain a st-flow g of $R(f)$ by setting

$$g(uv) = \begin{cases} q & \text{if } uv \in P \\ 0 & \text{otherwise} \end{cases}$$

    Update flow $f \leftarrow f + g$
end
return f

correctness. We separately prove three things:

- Initialization: f is a valid flow in G with value 0.
- According to Lemma 5.1.1: In every round g is a valid flow in $R(f)$, so $f + g$ is a valid st-flow in G.
- Termination: When the algorithm terminates, $d_{R(f)}(s, t) = \infty$, so by Lemma 5.1.1, f is a maximum st-flow in G.

Proof complete. ■

Definition 5.1.2 (augmenting path)**.** In Ford-Fulkerson algorithm, obtain a st-path P of $R(f)$, let $q = \min_{uv \in P} c_{R(f)}(u, v)$. The path P is called an **augmenting path** with respect to flow f.

Definition 5.1.3 (saturating flow). In Ford-Fulkerson algorithm, obtain a st-flow g of $R(f)$ by setting

$$g(uv) = \begin{cases} q & \text{if } uv \in P \\ 0 & \text{otherwise} \end{cases}$$

. The flow g is called a **saturating flow** with corresponding to P.

Lecture 10

考慮 Ford-Fulkerson algorithm 的複雜度分析，如果所有容量都是整數，則每次增廣至少增加 1 單位的流量，而 m 條管線的容量總和是 $C = \sum_{e \in E} c(e)$，因此最多增廣 C 次，每次找增廣路徑花費 $O(m)$ 的時間，總複雜度是 20 Nov. 14:20

$$T(m, C) = O(C) \cdot O(m) = O(mC)$$

但，這個算是「多項式時間」(polynomial time) 演算法嗎？還是是「指數時間」(exponential time) 演算法？

Remark. Complexity is according to the input size of an instance.

Example. For an $n \times n$ matrix multiplication problem, the input size is $N = \Theta(n^2)$ (set all the number is of size $O(1)$).

The complexity is

- Linear-time if $T(N) = O(N) = O(n^2)$.
- Quadratic-time if $T(N) = O(N^2) = O(n^4)$.
- Polynomial-time if $T(N) = O(N)^{O(1)} = O(n)^{O(1)}$.
- Exponential-time if $T(N) = O(1)^N = O(1)^{n^2}$ or more.

Definition 5.1.4 (Complexity of Linear/Quadratic/Polynomial-time Algorithms). For any instance I，define its input size as a non-negative integer function

$$N = \text{size}(I),$$

其中 N 表示描述輸入實例所需的位元數或其他適當的度量方式。令 $T(N)$ 為某演算法在輸入大小為 N 時的最壞情況執行時間。我們對時間複雜度作如下分類：

- Linear-time algorithm：若 $T(N) = O(N)$。
- Quadratic-time algorithm：若 $T(N) = O(N^2)$。
- Polynomial-time algorithm：若存在常數 k 使得 $T(N) = O(N)^{O(1)}$。
- Exponential-time algorithm：若存在常數 $c > 1$ 使得 $T(N) = O(1)^N$ or more。

Example. For a prime testing problem, given an integer N as input.

We have to consider the input size.

- If input size is $N = O(1)$, then the method of checking all integers from 2 to $\sqrt{N}$ is

$$O(\sqrt{N}) = O(1)$$

  which is a linear-time algorithm.

- If the size of N is not constrainted, then the input size is $\Theta(\log N)$ (bits to represent N). The time of checking all integers from 2 to $\lfloor \sqrt{N} \rfloor$ is $\Omega(\sqrt{N})$
  - According to

$$(\log N)^{O(1)} = o(\sqrt{N}) = o(N^{1/2})$$

    this algorithm is not polynomial-time.
  - According to

$$O(N^{1/2}) = O(1)^{O(\log N)}$$

    this algorithm is singly exponential-time.

Note. 所以根據 maximum flow problem 的 input size 是

- 若 $C = O(1)$，input size 是 $O(m) \cdot O(1) = O(m)$，所需要花的時間是 $O(mC) = O(m)$，演算法是 linear-time。
- 若無大小限制，input size 是 $O(m) \cdot O(\log C) = O(m \log C)$，所需要花的時間是

$$O(mC) \neq O(m \log C)^{O(1)}$$

因此演算法不是 polynomial-time。

Remark. Ford-Fulkerson algorithm 還會出現無限迴圈的問題，例如下圖的圖，設計出非整數的容量，會導致無限迴圈。Ford-Fulkerson 的論文就有提出其他反例。

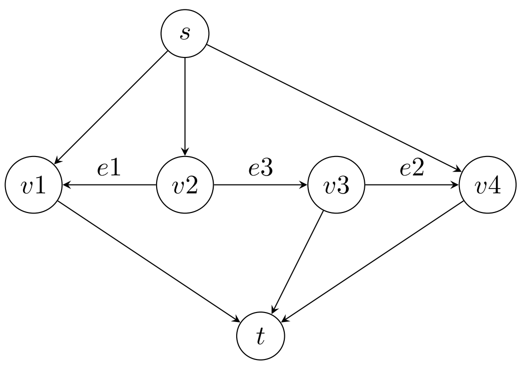


Figure 5.1: A graph that may cause infinite loop in Ford-Fulkerson algorithm

5.2 Edmonds-Karp Algorithm

這是史上第一個被證明是 polynomial-time 的 maximum flow algorithm。

Chapter 6

Computational Geometry

6.1 Nearest Point Pair