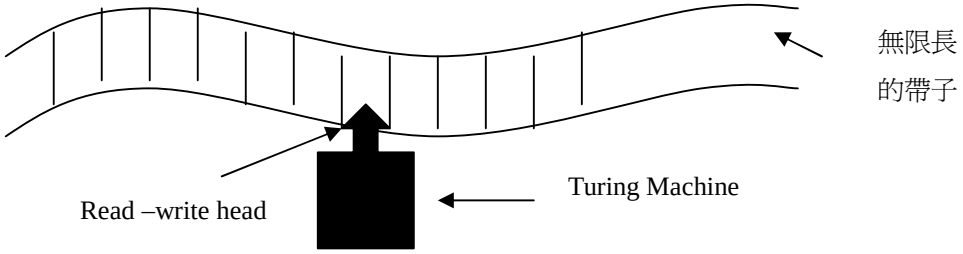
演算法的複雜度-時間與結果的考量

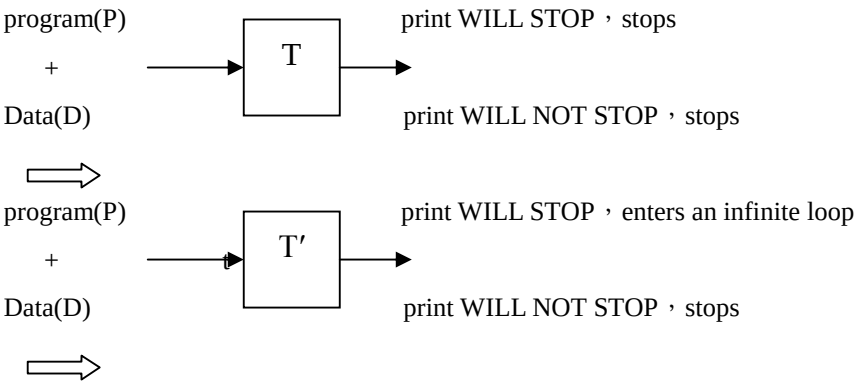
劉炯朗,謝禎鋐
國立清華大學資工系

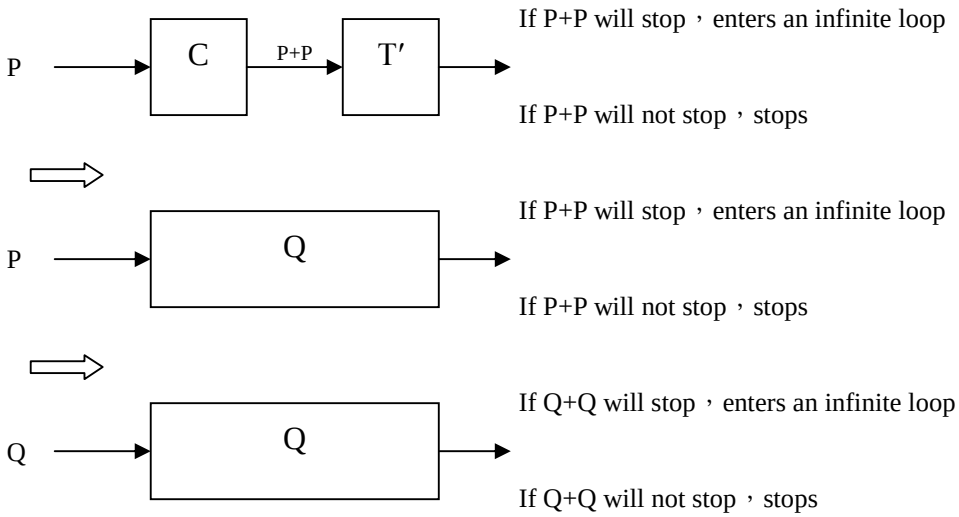
liucl@mx.nthu.edu.tw
pentel@nthucad.cs.nthu.edu.tw

爲了要以簡單，共通的方法算出問題的答案，人們發現問題中存在的規律性後就將其定爲所謂的公式，一旦找出公式後，只要將問題的資料代入對映的公式中就能得到正確的答案，經由此種方式，人們對於許多相同的問題就能在很短的時間算出，例如有名的牛頓定律 $F=ma$，狹義相對論中的愛因斯坦定律 $E=mc^2$，和工業界的摩爾定律 $D=c2^{m/18}$ 等，根據 Alan Turing (1912 -1954) 所提出的 Turing Machine，可知公式都是可算的函數(computable functions)，簡單的描述 Turing Machine 架構如下:將某個問題的計算規則都輸入 Turing Machine 後，在一條無限長的帶子(可前後移動)打上此問題的 input data， Turing Machine 可將之讀入並把 output data 打在帶子上，將正確的規則輸入 Turing Machine 後，Turing Machine 能算出的問題，都是 computable。



但不是所有的函數 Turing Machine 都算的出來，還是有不可算的函數(non-computable functions)，就舉個最基本的問題作爲例子，假設存在一個程式(T)會將任何程式(P)和其 input data(D)當成自己的 input data 然後判斷出那個程式是否最後能停止而不會跑進無窮迴圈，則





由上圖可知

1. 如果 program Q 用 program Q 當作 input data 時會停止時，program Q 將會進入無窮迴圈
2. 如果 program Q 用 program Q 當作 input data 時會進入無窮迴圈時，program Q 將會停止

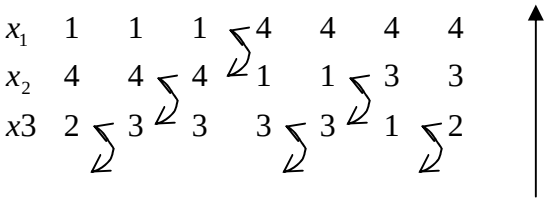
但此兩個描述都是自相矛盾的，因此推得並無 T 這個程式(可視作某個函數)存在，這就是一個無法算的函數了。

可算的問題(函數)是我們比較有興趣的，就目前的情形而言，只要能找出演算法(algorithm)的題目都是可計算的，相同的題目又可能有一種以上的演算法，例如我們要解 sorting problem 時就有許多的演算法如 Bubble sort，Quick sort，Insertion sort…..等等，而種種問題和其演算法的計算複雜度(time complexity)就是用來判斷計算所需時間的依據，我們先看看 sorting problem 和它的演算法的計算複雜度(time complexity)

Problem	Sorting	$nlogn$
Algorithm	Bubble sort	n^2
	Quick sort	$nlogn$
	Insertion sort	$nlogn$

n 代表 problem input size

一個問題的計算複雜度是其所有演算法中所需的最小計算複雜度，例如 sorting，它最小的計算複雜度的演算法是 $nlogn$，至於每種 algorithm 有用到不同的方法，所以會有不盡相同的計算複雜度，以下是 bubble sort 的 algorithm:



x_4 3 2 2 2 2 2 1

共需要 $n+(n-1)+(n-2)+..+1=n(n-1)/2$ 次 比較$\Rightarrow$ time complexity 爲 n^2

根據不同的 time complexity 我們又可以把 algorithm 分作快的和慢的兩大類，快的 algorithm 就是其 time complexity 對 problem size(n)是 polynomial function，如$\sqrt{n}$ ， n ， $nlogn$ ，n^3…..，慢的 algorithm 就是其 time complexity 對 problem size(n)是 exponential function，如$2^n, 3^n, n!$…，由此可知 bubble sort，quick sort 是快的 algorithm，而 partitioning 則是慢的 algorithm，當然問題也可以依此方法來分類，如果其 time complexity 是以 polynomial function 當作上限，則分在容易處理類(tractable)Ex: Sorting ， Linear Programming，若其 time complexity 是以 exponential function 當作下限的話就分在不易處理類(intractable) Ex: Boolean Equivalence;下表可明顯看出 polynomial function 和 exponential function 在 n 由小到大時的變化，在 n 變大時，exponential function 會遠比 polynomial function 的值來的大，也就是說在 n 不大時所需的計算時間並不相差很多，但隨著 n 的變大，所需的計算時間會以難以想像的速度成長。

	2	5	10	50	60	100
N	2	5	10	50	60	10^2
n^2	4	25	10^2	2.5×10^3	2.5×10^3	10^4
n^3	8	125	10^3	1.25×10^5	2.16×10^5	10^6
n^5	32	3125	10^5	3.12×10^8	7.78×10^8	10^{10}
2^n	4	32	10^3	1.13×10^{15}	1.15×10^{18}	1.27×10^{30}
3^n	9	243	5.9×10^4	7.18×10^{23}	4.24×10^{28}	5.15×10^{47}
$N!$	2	120	3.63×10^6	3.04×10^{64}	8.32×10^{81}	9.33×10^{177}

用計算複雜度把問題分作易處理和不易處理後，其實中間還是有些灰色地帶尚未能做出分類，而這個地帶就是 NP-Complete problems，爲何 NP-Complete problems 無法分類呢?因爲沒有人能找出 NP-Complete problems 的 polynomial-time algorithm，卻又不能證明他們有 exponential-time 的下限，也就是找不出問題的計算複雜度，可是很多重要的問題都是 NP-Complete，NP-Complete 有個很特別的性質，就是一旦有一個 NP-Complete 問題被找出 polynomial-time 的 algorithm 的話，那麼所有的 NP-Complete 問題都找的到 polynomial-time 的 algorithm，相反的，如果有一個 NP-Complete 中的問題被證明出有 exponential-time 的下限的話，那麼所有的 NP-Complete 問題都有 exponential-time 的下限，也就是這些問題已經無法算出最佳解了，可知 NP-Complete problems 全是容易處理的(tractable)，或著全是不易處理的(intractable)，但要將 NP-complete 歸類是和求 NP-complete 的 polynomial-time algorithm 一樣難的問題，所以目前沒人能回答，那退而求其次，要如何判斷一個問題是否爲 NP-Complete 呢?如果一個問題不是 NP-Complete，則可以試著找 polynomial-time 的 algorithm，如果是 NP-Complete，

最好想其他方法，才不會浪費時間，以下就是判斷一個問題是否爲 NP-Complete 的方法：

1.先判斷問題 L 是否是 NP(nondeterministic polynomial time)，若不是 NP 就不是 NP-complete。
2.選一個已知是 NP-complete 的問題 L'。
3.描述一個演算法可以算出一個函數 f，而 f 可將 L'轉變成 L。
4.證明函數 f 滿足此條件: $x \in L'$ 若且唯若 $f(x) \in L$ ，對所有的 $x \in \{0,1\}^*$ 。
5.證明可算出 f 的演算法會在 polynomial time 執行完成

在確定問題是 NP-complete 或是不易處理(intractable)後，面對這兩種沒有 polynomial-time algorithms 的問題該怎麼做呢?我們要作出選擇，若要算出最佳解可以用 exponential-time 的 algorithm 或窮舉法，若是近似值也能接受的話，就可找近似演算法(approximation algorithms)來作計算，當然你必須先考慮是時間重要還是最佳解重要了。

舉個有名的 NP-Complete 問題 Traveling Salesman 來比較這兩種方法:
求通過 a,b,c,d,e,且不重覆的最短路徑(此路徑爲一 Hamiltonian circuit)

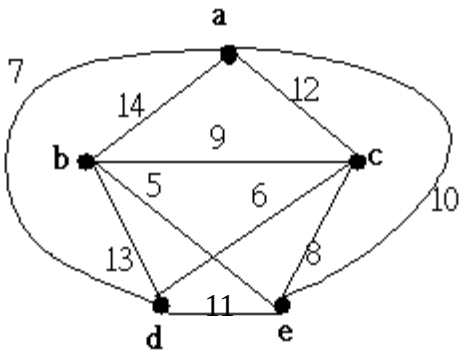


Fig 1.1

Exact algorithm(Brute force algorithm):

Try (n-1)! Possible solutions

4!=24　　9!=362，800　　69!= 1.71122×10^{98}

$\frac{1.71122 \times 10^{98}}{10^{10}} = 1.71122 \times 10^{88}$ **seconds**

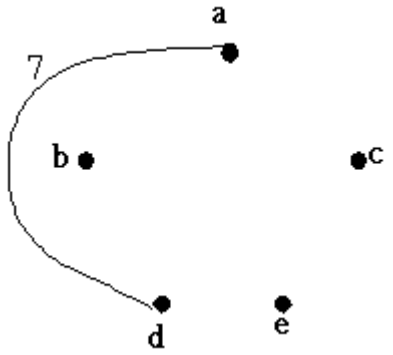
$\frac{1.71122 \times 10^{88}}{60 \times 60 \times 24 \times 365 \times 100} = 5.42626 \times 10^{78}$ **centuries**

可知要求最佳解是要算蠻久的

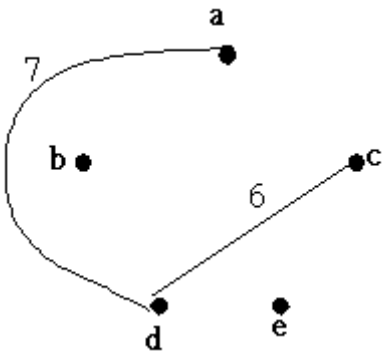
Approximation algorithm: *nearest-neighbor method*

1.先在 G 上任選一點，並找出離它最近的點來形成第一條路徑。
2.設 x 爲最近加入路徑的點，並找一個離 x 最近且尚未加入路徑的點，將此點和它與 x 的連接線加入路徑，重覆此步直到所有在 G 上的點都加入路徑爲止。
3.將第一點和最後一點的連線加入路徑後就形成一個環路。

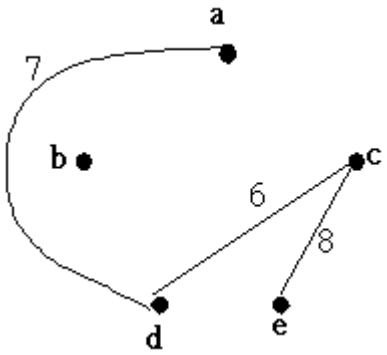
Step 1



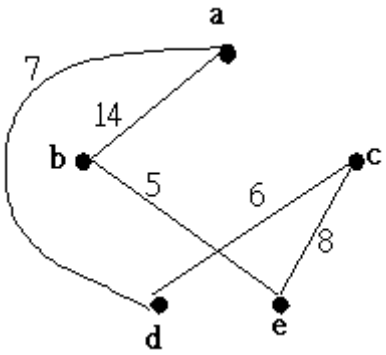
Step 2



Step 3

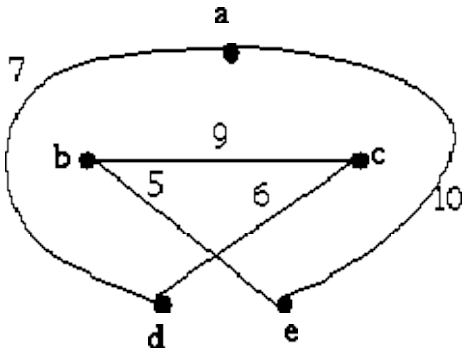


Step 4



Total length *d*=40

把所得結果和最佳解比較
the best result is



Total length d_0=37

Fig 1.2

可找出 d 和 d_0 的關係爲一定律 $\frac{d}{d_0} \le \frac{1}{2}\lceil \lg n \rceil + \frac{1}{2}$ ，n 表示要通過的點數

證明:

在進行證明之前，先讓我們藉由思考一特定情況來說明我們對證明的作法，設 D 是用 nearest-neighbor method 得到的 Hamiltonian circuit。設 l_1 是 D 中最長一

段線段的長度，l_2 是 D 中次長線段的長度，依此類推，l_i 是第 i 長線段的長度，所以可知

$$d = \sum_{i=1}^{n} l_i$$

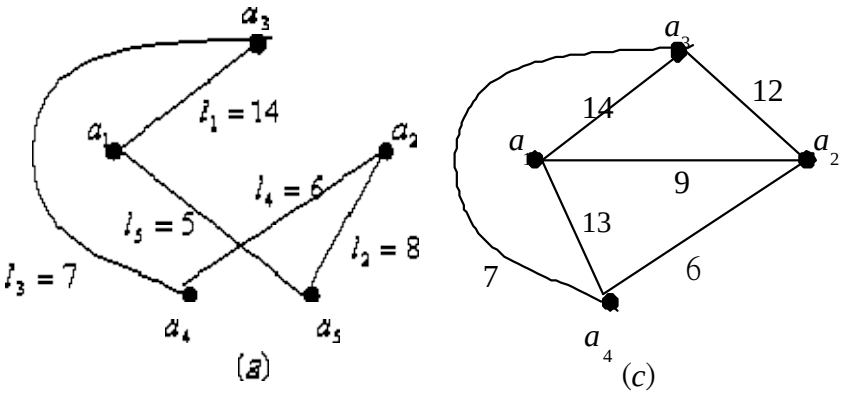
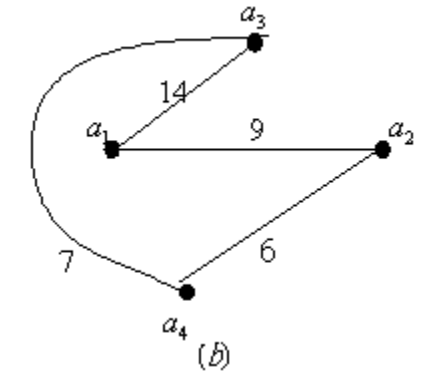


Fig 1.3



爲了要說明，先設 n=14，如果我們能證明

$$\begin{aligned} d_0 &\geq 2l_1 \\ d_0 &\geq 2l_2 \\ d_0 &\geq 2(l_3 + l_4) \\ d_0 &\geq 2(l_5 + l_6 + l_7 + l_8) \\ d_0 &\geq 2(l_9 + l_{10} + l_{11} + l_{12} + l_{13} + l_{14}) \end{aligned} \qquad (1.1)$$

則我們應該可得

$$5d_0 \geq 2\sum_{i=1}^{14} li = 2d$$

亦即

$$\frac{d}{d_0} \geq \frac{5}{2} = \frac{1}{2}\lceil \lg 14 \rceil + \frac{1}{2}$$

對一般的 n，我們可以建立一組類似於(1.1)的不等式組，亦即我們將證明

$$d_0 \geq 2l_1 \qquad (1.2)$$

$$d_0 \geq 2\sum_{i=k+1}^{2k} l_i \qquad 1 \leq k \leq \left\lfloor \frac{n}{2} \right\rfloor \qquad (1.3)$$

和

$$d_0 \geq 2\sum_{i=\lceil n/2 \rceil+1}^{n} l_i \qquad (1.4)$$

要注意如果 *n* 是偶數時(1.4)已被包含在(1.3)中。

利用三角不等式，(1.2)顯然成立。假設在 *D* 中最長的線段是以 *x* 和 *y* 爲端點，則依三角不等式可知任何從 x 到 y 的任何路徑都大於或等於 l_1。既然任何 *G* 中的 Hamiltonian circuit 都可以被拆成 x 和 y 之間的兩段路徑，因此(1.2)成立。

令 a_i 爲利用 nearest-neighbor method 找 Hamiltonian circuit D 時，D 中的第 i 長的線段所加進去的點(ex:根據這個命名法，在圖 Fig. 1.1 的點的名字即如圖 Fig. 1.3a 所示)。對一個固定的 k，$1 \leq k \leq \left\lfloor \frac{n}{2} \right\rfloor$，設 H 是包含了點 a_i，$1 \leq i \leq 2k$，的 *G* 的完全子圖。設 *T* 爲 *H* 中的 Hamiltonian circuit，而它所經過的點的順序是和 *G* 中最短(佳)的 Hamiltonian circuit 經過 G 中的點的順序一樣(所有在 *G* 中卻不在 *H* 中的點都將跳過)。設 *t* 是 *T* 的長度(在圖 Fig1.1，設 *k*=2。*H* 如 Fig1.3c 所示。在圖 Fig1.2 所示的最短 Hamiltonian circuit 中，經過點的順序是 a_1, a_2, a_3, a_4, a_5。所以 *T* 環路就如 Fig1.3b 所示，且 *t*=36)。藉由三角不等式，可得

$$d_0 \geq t \qquad (1.5)$$

設 $\{a_i, a_j\}$ 是 *T* 中的一條線段。假如在我們根據 nearest-neighbor method 形成 Hamiltonian circuit 時，a_i 比 a_j 先經過，則 $w(a_i, a_j) \geq l_i$。假如 a_j 比 a_i 先經過，則 $w(a_j, a_i) \geq l_j$。亦即 $w(a_i, a_j) \geq l_j$，因此，不管先經過 a_i 或 a_j，都可得

$$w(a_i, a_j) \geq \min\{l_i, l_j\} \qquad (1.6)$$

將(1.6)對 T 中的所有的線段依照取和，可以得到

$$t \geq \sum_{(a_i, a_j) \in T} \min(l_i, l_j) \qquad (1.7)$$

在不等式(1.7)中，$\min\{l_i, l_j\}$ 中最小的值是 l_{2k}，第二小的值是 l_{2k-1}，等等依此類推。同時，對任意 i，$1 \leq i \leq 2k$，l_i 在(1.7)右邊的和之中，最多出現兩次。因爲 *T* 中共有 2*k* 個線段，(1.7)右邊的和大於或等於 *T* 中比較小的 *k* 個線段的長的和的兩倍，因此可得

$$t \geq \sum_{(a_i, a_j) \in T} \min(l_i, l_j) \geq 2(l_{2k} + l_{2k-1} + \ldots + l_{k+1}) = 2\sum_{i=k+1}^{2k} l_i \qquad (1.8)$$

合併(1.5)和(1.8)，即可得(1.3)

不等式(1.4)的證明方法跟不等式(1.3)類似(只需考慮 n 爲奇數的情形)。令 D_0 代表最短的 Hamiltonian circuit。利用導出(1.7)的同樣討論，可得

$$d_0 \geq \sum_{(a_i,a_j)\in D_0} \min(l_i, l_j)$$

利用導出(1.8)的同樣討論法，可得

$$d_0 \geq \sum_{(a_i,a_j)\in D_0} \min(l_i, l_j)$$

$$\geq 2(l_n + l_{n-1} + ... + l_{\lceil n/2 \rceil + 1}) + l_{\lceil n/2 \rceil}$$

$$\geq 2(l_n + l_{n-1} + ... + l_{\lceil n/2 \rceil + 1}) = 2\sum_{i=\lceil n/2 \rceil + 1}^{n} l_i$$

當 k=1， 2^1 ， 2^2 ，….. ， $2^{\lceil \lg n \rceil - 2}$ 時，(1.3)可導出 $\lceil \lg n \rceil - 1$ 個不等式：

$d_0 \geq 2l_2$

$d_0 \geq 2(l_3 + l_4)$

$d_0 \geq 2(l_5 + l_6 + l_7 + l_8)$

……

$d_0 \geq 2(l_{2^{\lceil \lg n \rceil - 2}+1} + l_{2^{\lceil \lg n \rceil - 2}+2} + ... + l_{2^{\lceil \lg n \rceil - 2}+1})$

將上面這些不等式相加，可得：

$$(\lceil \lg n \rceil - 1) d_0 \geq 2(\sum_{i=2}^{2^{\lceil \lg n \rceil - 1}} l_i) \qquad (1.9)$$

因爲 $\left\lceil \frac{n}{2} \right\rceil + 1 \leq 2^{\lceil \lg n \rceil - 1} + 1$ ，如果 n>1

由(1.4)可以導出

$$d_0 \geq 2\sum_{i=2^{\lceil \lg n \rceil - 1}+1}^{n} l_i \qquad (1.10)$$

將(1.2),(1.9)和(1.10)相加，可得

$$(\lceil \lg n \rceil - 1) d_0 \geq 2\sum_{i=1}^{n} l_i = 2d$$

亦即

$$\frac{d}{d_0} \le \frac{1}{2}\lceil \lg n \rceil + \frac{1}{2} \qquad \#$$

當 n=14 $\quad \frac{d}{d_0} \le \frac{1}{2}\lceil \lg 14 \rceil + \frac{1}{2} = \frac{5}{2}$

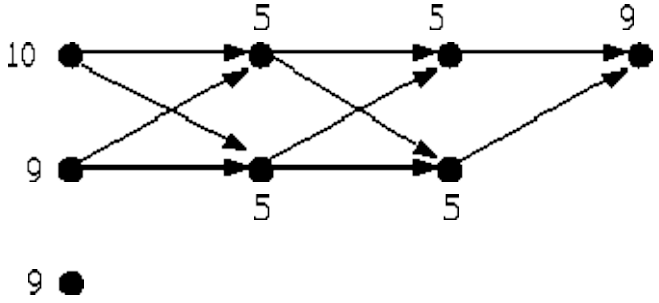
當 n=1000 $\quad \frac{d}{d_0} \le \frac{1}{2}\lceil \lg 1000 \rceil + \frac{1}{2} = \frac{11}{2}$

由此可知，以此近似演算法求出的值在 n 越小的時候誤差越小，n 變大時誤差會以很緩慢的比例增加，以所需的時間和所得結果來看的話，近似演算法確實很有用，在此例中也能看出當 problem size 很小的時後還可以選擇用全部算出的方法，但 problem size 一旦變大了，只能選擇近似演算法，否則可能永遠都算不完。

在很多科學領域中的問題都是 NP-Complete，也都要用近似演算法來處理，茲舉出幾個較爲有名的領域:

PERT　Chart (Programming Evaluation and Review Technique)
Transportation Scheduling
Process Control
Robotics
Avionics
Parallel Computation
Synthesis of VLSI Circuits

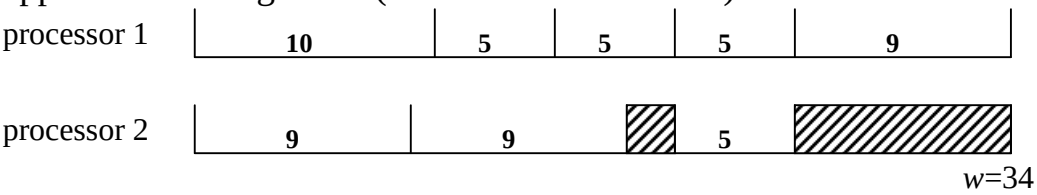
我們再看一個例子，Job Scheduling:
下圖代表八個工作所需的工作時間，以及須遵守的執行順序(Ex:最右邊的工作要做以前必須先完成兩個有箭頭指向它的工作，其餘依此類推)，有兩個處理器可以工作，但是一個工作只限一處理器做，問需多少時間才可完成所有工作？



此題也沒有人找出最佳解的演算法，所以要比過所有的方法後才能得到最短

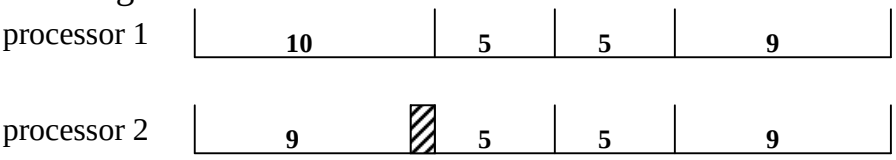
的時間，而近似演算法 no intentional idleness 卻能找出不錯的結果，當然近似演算法不需花太多的時間，這個演算法顧名思義就是只有在沒工作可做時才將處理器閒置，以下是近似演算法的結果和最佳結果的比較

approximation algorithm(no intentional idleness) result:



w=34

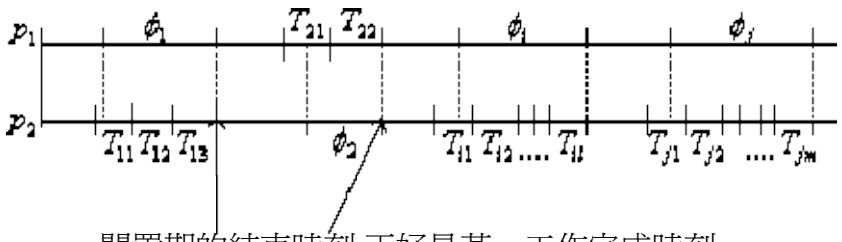
brute algorithm best result:



w_0 =29

注: 代表閒置中

w 和 w_0 的關係爲一定律 $\frac{w}{w_0} \le 2 - \frac{1}{n}$ ，n 爲處理器的數目



一閒置期的結束時刻,正好是某一工作完成時刻

Fig 2.1

證明:

爲了可以用簡化表示法，我們只證明 n=2 的情形，一般的情形可以同法證明，就 no intentional idleness 的輪工表來看，見圖 2.1，我們發現在一個處理器的閒置時間結束的同時必是另一個處理器執行完一個工作的時候(否則閒置時間不會結束)。設 ϕ_i 是一個處理器的其中一段閒置的時間，一個工作 T_{ij} 是和 ϕ_i 時間相衝的意思是，在輪工表中，一個處理器完成工作 T_{ij} 的時段跟 ϕ_i 相交。例如在圖 2.1 中，T_{i1}，T_{i2}，T_{i3} 和 ϕ_i 時間相衝。對閒置期 ϕ_i，令 T_{i1}，T_{i2}，….，T_{il} 爲與 ϕ_i 時間相衝的工作。欲證

$$T_{i1} \le T_{i2} \le T_{i3} \le ..... \le T_{il}$$

設此關係不成立，那麼 T_{i1}，T_{i2}，T_{i3}，這些工作就不必在 P_2 中依序執行了，而是可以利用 P_1 閒置的時候來執行。同樣的，跟據剛才的討論，如果 T_{j1}，T_{j2}，T_{j3}，…，T_{jm} 爲與另一閒置期 ϕ_j 時間相衝的工作，則

$$T_{j1} \le T_{j2} \le T_{j3} \le ..... \le T_{jm}$$

很清楚的是每件在T_{il}完成後才執行的工作一定是T_{il}的後繼工作(不然他們可以在ϕ_i期間以P_1執行)。由上可推得一原來工作群T的子集合ℓ，滿足：

1. ℓ是一鏈

2. $\sum_{T_k \in \ell} \mu(T_k) \geq \sum_{\phi_i \in \Phi} \mu(\phi_i)$

其中Φ爲輪工表中所有閒置期所成的集合。

因爲

$$\omega = \frac{1}{2}\left[\sum_{T_j \in T} \mu(T_j) + \sum_{\phi_i \in \Phi} \mu(\phi_i)\right] \leq \frac{1}{2}\left[\sum_{T_j \in T} \mu(T_j) + \sum_{T_k \in \ell} \mu(T_k)\right] \qquad (2.1)$$

又因 $\omega_0 \geq \frac{1}{2}\sum_{T_j \in \ell} \mu(T_j)$

並且 $\omega_0 \geq \sum_{T_k \in \ell} \mu(T_k)$

因此，(2.1)式可改寫成

$$\omega \geq \omega_0 + \frac{1}{2}\omega_0$$

即 $\frac{\omega}{\omega_0} \leq \frac{3}{2}$

這就是 n=2 時，定理的結論 #

當 $n \to \infty$ $\quad \frac{w}{w_0} \leq 2$

當 $n = 2$ $\quad \frac{w}{w_0} \leq \frac{3}{2}$

這個例子中的近似演算法的效果更好，即使在 problem size 趨近無窮大的情況下卻只和最佳解差在兩倍之內，看完這兩個例子後相信你已經對 NP-complete problems 和 approximation algorithm 有更深一層的認識了，NP-complete 讓我們知道該以什麼方式來處理問題，approximation algorithm 除了能算出近似解之外，更重要的是它給了我們一個解答的保證，只要有了 approximation algorithm 的結果後，可以再嘗試其他的 algorithm，如 randomize algorithm，heuristic algorithm 等等，會否得到的更好的結果，如果有，當然很好，若沒有也不怕會沒有可用的解答，至少有個只和最佳解差在一定程度的答案在備用，這會給我們更多研究的空間，來幫麻煩但很重要的 NP-complete problems 尋求更好的答案。