國立暨南國際大學資訊工程學系

碩士論文

指導教授：黃光璿博士

利用序列比對演算法
辨識抄襲之 C 程式作業

Cheating Catcher:
Using Sequence Alignment Algorithms to Identify Homologous C Programs

研究生：李昶宏

中華民國九十三年五月二十八日

國立暨南國際大學碩士論文考試審定書

資訊工程 學系（研究所）

研究生 李昶宏 所提之論文

利用序列比對演算法辨識抄襲之C程式作業

Cheating Catcher:

Using Sequence Alignment Algorithms to Identify Homologous C Programs

（中、英文題目）

經本委員會審查，符合碩士學位論文標準。

學位考試委員會

委員兼召集人

委員

委員

中華民國 93 年 5 月 28 日

博碩士論文授權書

（國科會科學技術資料中心版本 91.2.17）

本授權書所授權之論文為本人在 國立暨南國際 大學(學院) 資訊工程 系所＿＿＿＿＿＿組 92 學年度第 2 學期取得 碩 士學位之論文。

論文名稱： 利用序列比對演算法辨識抄襲之C程式作業

☑同意　□不同意 （政府機關重製上網）

本人具有著作財產權之論文全文資料，授予行政院國家科學委員會科學技術資料中心、國家圖書館及本人畢業學校圖書館，得不限地域、時間與次數以微縮、光碟或數位化等各種方式重製後散布發行或上載網路。

本論文為本人向經濟部智慧財產局申請專利(未申請者本條款請不予理會)的附件之一，申請文號為：＿＿＿＿＿＿，註明文號者請將全文資料延後半年再公開。

☑同意　□不同意 （圖書館影印）

本人具有著作財產權之論文全文資料，授予教育部指定送繳之圖書館及本人畢業學校圖書館，為學術研究之目的以各種方法重製，或為上述目的再授權他人以各種方法重製，不限地域與時間，惟每人以一份為限。

上述授權內容均無須訂立讓與及授權契約書。依本授權之發行權為非專屬性發行權利。依本授權所為之收錄、重製、發行及學術研發利用均為無償。上述同意與不同意之欄位若未鉤選，本人同意視同授權。

指導教授姓名: 黃光璿

研究生簽名: 李祖宏 (親筆正楷)　學號: 90321512 (務必填寫)

日期:民國 93 年 6 月 4 日

1. 本授權書（得自 http://sticnet.stic.gov.tw/sticweb/html/theses/authorize.html 下載）請以黑筆撰寫並影印裝訂於書名頁之次頁。
2. 授權第一項者，請確認學校是否代收，若無者，請個別再寄論文一本至台北市(106-36)和平東路二段 106 號 1702 室 國科會科學技術資料中心 王淑貞。(本授權書諮詢電話:02-27377746)
3. 本授權書於民國 85 年 4 月 10 日送請內政部著作權委員會(現為經濟部智慧財產局)修正定稿，89.11.21 部份修正。
4. 本案依據教育部國家圖書館 85.4.19 台(85)圖編字第 712 號函辦理。

博碩士論文電子檔案上網授權書

（提供授權人裝釘於紙本論文書名頁之次頁用）

本授權書所授權之論文爲授權人在 國立暨南國際 大學（學院） 資訊工程系所＿＿＿＿＿＿組 九十二 學年度第 二 學期取得 碩 士學位之論文。

論文題目：利用序列比對演算法辨識抄襲之C程式作業

指導教授：黃光璿

茲同意將授權人擁有著作權之上列論文全文（含摘要），非專屬、無償授權國家圖書館及授權人畢業學校之圖書館，不限地域、時間與次數，以微縮、光碟或其他各種數位化方式將上列論文重製，並得將數位化之上列論文以上載網路方式，提供讀者基於個人非營利性質之線上檢索、閱覽，或並下載、列印。

□讀者基於非營利性質之線上檢索、閱覽或下載、列印上開論文，應依著作權法相關規定辦理。

授 權 人：李昶宏

姓　　名：李昶宏　　（請簽名並蓋章）

中 華 民 國　93　年　5　月　28　日

誌謝

本論文可以順利完成，首先要感謝我的指導教授黃光璿 老師這兩年來給我的指導，不管在理論或是在系統架構建立上都給我不少的意見和思考方向，也非常感謝口試委員項潔以及洪政欣二位老師給予我於論文上莫大的建議與指正。

也要感謝研究室的學弟(展碩、光慶、明峰)，在課業上幫了我不少忙，在論文的界面上也給予我相當多的幫助。

感謝我周遭的好友們(俊賢、俊銘，敬育)以及女友不斷的鼓勵我、支持我、聽我訴苦、幫我分擔煩惱、陪伴我以及鼓勵我。 謝謝你們。

最後要感謝的就是我的家人，在這二十年來於背後無怨無悔的犧牲與奉獻，如果沒有他們的支持就不會有今天的我，最後僅將此論文獻給所有關心我的人。

李昶宏　　謹於

國立暨南國際大學

中華民國九十三年六月四日

目　錄

圖形目錄

表格目錄

方程式目錄

中文摘要

現今比對程式相似度的軟體很多，習慣上，程式碼差異度的比對一直以來都是由人在作，而目前最常用的是 Unix 上的 Diff 與 Windows 上的 windiff，但是它們並沒有考慮到多種問題，如:函式互換、變數替代、相關語法…等，而程式碼的改變使得人們必須花上很多的時間和精力去用眼睛做判斷，為了改善這種情況我們就開始著手對程式碼的差異程度去做分析比較。

我們利用改良過後的 DNA 比對演算法來進行比對，由於程式碼所造成的情況實在太多，若單純地利用程式間的 edit distance 來估算，可能會忽略許多問題，導致誤判產生，因此我們決定採用區域性比對來求得相似度，即 local alignment by dynamic programming，但由於程式碼相似部分可能有很多段，光是只找出其中一段相似程式碼是不夠的，必須要再找出其它相似的區段。很多文獻都有提及這方面的問題，但不外乎要重算或是找尋重疊相似程式片段，這樣需要花費許多計算時間，尤其當比對的程式碼很大的話，所用到的記憶體更是大的驚人，因此在有限的記憶體下，我們採用簡單的編碼技巧，以二個 bits 為一單位來紀錄 dynamic programming 所需的相關資訊，可把所需記憶體縮小成四分之一。在時間上我們也設計不需要重算的方法，取而代之的是採用“邊算邊紀錄”的方法，因此在計算 dynamic programming 的同時，便已得到相關程式碼每一區段的資訊，之後再配合一些處理機制，便可找出全部相似的區段，最後個別區段的相似度的加總即為兩程式碼所得到的**相似分數**。但利用相似分數來評估兩程式碼的相似度是很主觀的，如:我們可以說，“若兩程式碼有 60%相似，便稱它們相似”。我們採用**統計**來評估相似度，如此一來便可以很客觀地定義出程式的相似度。

但由於使用者有時需更快地找出相似的程式碼，我們提供一個更快，幾乎在線性時間便可完成的功能，所花費的記憶體也相對地少很多，運用一種稱為 FASTA 的 database search algorithm ，經過改良後便可很快速地比對相似程式碼，但精確度卻沒有比 local alignment 來得高，不過實驗上說明**改良過的**

FASTA 的精確度與 local alignment 差不了多少，另外我們也提供圖形介面讓使用者能看出程式碼哪些區段相似。

英文摘要

A lot of software tools (e.g., Diff and WinDiff) have been developed to identify the differences between text files. While being applied to compare programs, however, more situations need to be handled since renaming the identifiers, altering the function definitions, or replacing a block by a synonymous one can still make the program equivalent. In order to solve these problems, we investigate on how to analyze the similarity between programs.

We develop a system to estimate the similarity of two C programs by using Local Alignment Algorithm. Simply doing this can not meet our demand because similar but non-overlapping regions may scatter in different orders. Hence, some modification is necessary. We intend to identify all of the similar regions that are non-overlapping. We propose an idea to perform “computing and recoding” in order to achieve the goal efficiently. A straightforward way to accomplish this is to apply the dynamic programming (DP) repeatedly: once a similar region is identified, remove it and apply the DP again. However, this cannot work well. Another issue addresses the notion of similarity. We adopt statistic method for this. Memory limitation is also a problem. We propose a simple encoding technique to reduce third-quarters of the required memory space.

Trade-off between efficiency and accuracy is also considered in this thesis. We develop two versions of the system: one for the efficiency purpose and the other for the accuracy purpose. A FASTA-like algorithm is developed, although it is less precise than the full Local Alignment Algorithm. In addition, experiments show that our fast version can estimate the similarity well. Finally, the programs provide graphical illustration on how two programs are similar.

第一章 簡介

1.1 動機與目的

程式語言的設計風格經常都是根據程式設計師個人的習慣而對相同目的的程式語言有著不同的設計方法，無可避免的，在相同的目的下，不同程式設計師coding 的程式可能會有部份和其他程式設計師有著相同的程式碼，若兩者相似的部份比例太高，則會有抄襲的嫌疑存在，但是抄襲的程式設計師通常會對程式做某種程度上面的改寫，無論是在界面上或是程式內容的更改，使得單純的用眼睛判斷無法在第一時間內判斷出來。

一般來說，只能對於程式間的寫法來判斷兩兩間之相似度，而對於程式所要表達的功能，我們可以用理論證明其是不可能達到的[1]，所以大部份這方面相關的軟體只針對程式間的寫法，而沒有判斷程式間的功能。

一般比對兩個檔案相似度的差異使用最為頻繁的就是 Windows 之下的WinDiff，但若對程式碼中的變數名稱做替代或其他改寫程式的方法，雖然程式碼改變了，但站在程式碼相似的角度上看來並沒有任何的差異，因此DNA 演算法在這提供了判斷程式碼相式度一個很有利的方法，只要改良演算法的內容便可達到很好的效果。

在眾多的程式語言中，大部分的程式設計師都對C 語言有一定程度的接觸，雖然C 語言並不是現今最熱門的程式語言，但是到目前為止，C 語言依然有他存在的價值，而且現今大部分學校也都是用C 語言來授課，我們也可藉由學生的程式作業來判定學生間是否有抄襲的可能，於是我們選擇C 語言來做程式相似度比對系統的處理語言。

我們針對C 語言對相同目的的不同表示法做分析的動作，雖然利用單純的edit distance便可，但程式間有些相似度還是無法判斷，因此在本論文中採用

改良過的 DNA 演算法來作程式間比對分析，我們先轉換成事先已定義好的 token 字串再利用 DNA 演算法來作比對。

1.2 系統貢獻

一般來說對於要快速且精準地判斷程式碼的相似度機乎是不可能的，以 Windiff 來說，其對於程式碼相似判斷雖快但其精準度不是很高，原因是它是利用行來作比對工具，而我們的系統在此不僅對於速度且對於精確度也都有達到，我們利用改良過的演算法來提高其比對速度，且在實驗上其精確度也是很好的，且最後我們也提供圖形來說明相似部分。

1.3 論文章節概要

本篇論文主要可以分成下列幾個章節，內容大綱如下:

第一章: 簡介

簡介本論文的內容及貢獻。

第二章: 相關理論基礎與研究

詳細介紹DNA演算法，其在程式比對中的應用以及用到的DNA演算法的種類和理論，並對使用的方法做探討，且討論如何利用事先定義好的token來建立字串，最後針對C 語言的語法做討論以及現今程式比對工具的比較。

第三章: 實作系統所遇到的困難

主要探討實作系統所遇到的困難。

第四章: 解決困難的方法

詳細解說如何解決面臨的困難。

第五章: 系統實作

說明系統的架構以及系統的功能及功能的流程，並比較每個功能。

第六章:

測試程式比對系統實際比對出程式之間的差異以及效能。

第七章: 結論及未來展望

對本論文下一個結論以及未來系統可以改進效能的地方

第二章 相關理論基礎與研究

C語言程式比對不外乎是要先轉換為字串再加以比對分析，且我們也是利用edit distance為基礎，借用local alignment及 FASTA alignment方法把轉換後的字串加以比對分析，且我們也有一套計算相似度的方法。在這章節裡，我們將介紹DNA用在字串比對演算法的種類以及分別詳細介紹之。

2.1 序列字串 alignment 問題

給定一個字元集 {A、T、C、G}，則DNA序列就是任意在字元集字母的排列，例如: CGCTAAGCT。在二個字串比對中最簡單的方法就是只將二序列對齊，然後分別依字元逐字計算分數，例如從字元集中任意選定二序列:

TCATTACAACCGCT

CCATCAACAACCGCT

假設我們簡單 align 其結果是:

TCATTACAACCGCT

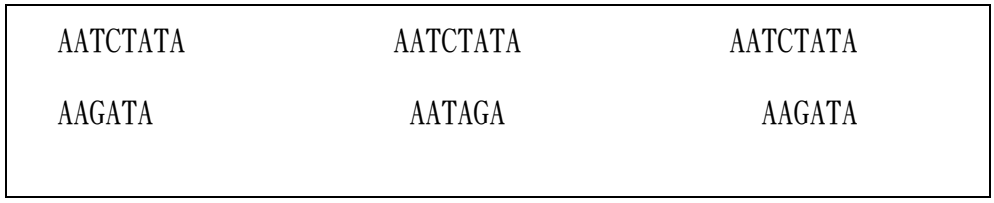
CCATCAACAACCGCT

令字元相同加 1 分，字元不同 -1 分，則我們說上列二序列 alignment 的結果分數是 4 分，假設若有 gap 存在，因此 align 的結果變:

TCATTA-CAACCGCT

CCATCAACAACCGCT

假設 gap 與字元 alignment 為 -1 分，因此二序列 alignment 的結果分數變為 9 分，但由於上列 alignment 的結果並非唯一，因此我們要如何知其最佳的 alignment(相當於二序列的最相似分數) 是幾分呢？且在 DNA 序列比對裡似乎沒這麼單純，因為可能每個字元又突變演化成其它的字元[2]，況且在程式比對中又更加地複雜，並不是簡單的 alignment 所能作出來的，因此就有幾個 alignment 演算法產生，而其中在 DNA 序列比對中最有名的演算法是 (1) Local alignment algorithm (2) Global alignment algorithm (3) FASTA alignment algorithm (4) BLAST algorithm 在這裡我們選擇了Local alignment algorithm 與 FASTA alignment algorithm 來從事相關的研究工作且在程式比對裡是允許有 gap 存在的。



圖表 一 :簡單的 DNA 序列 alignment(不唯一)

圖表 二 :簡單的 DNA 序列含 gap alignment(不唯一)

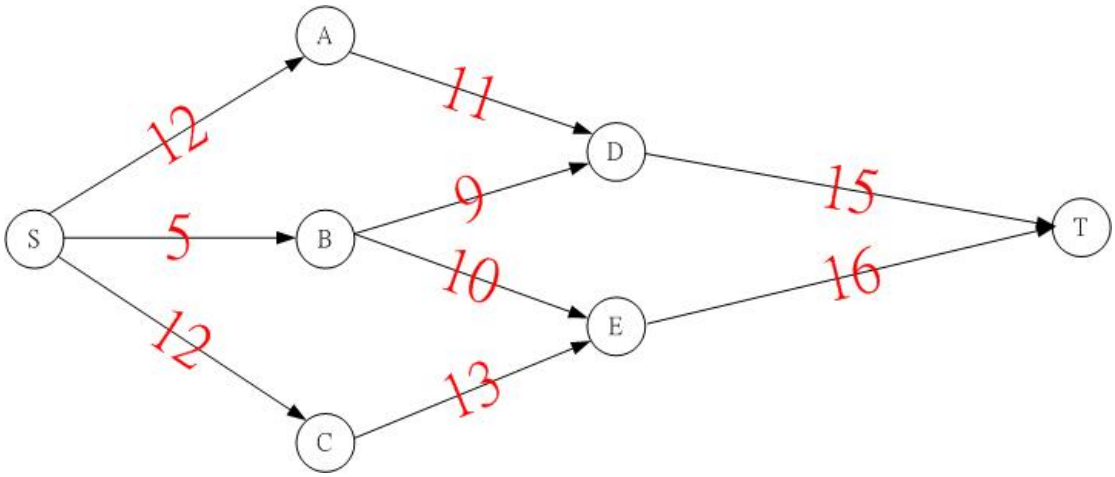
2.1.1 DNA 序列演化的三個 operator(insertions(插入)， deletions(刪除)，substitutions(置換))

在 DNA 序列比對裡 insertions、deletion、substitutions 扮演著演化的關鍵，insertion 就是從一個序列多出一個字元進而變成另一個序列，例如: 序列 AATTCC 可能演變成 AAGTTCC(多了 G) 而 deletion 就是從一個序列少掉一個字元進而變成另一個序列，例如: AATTCC 可能演變成 ATTCC(少了 A) ，最後 substitutions 是從一個序列中的一個字元變成另一個字元，例如: AATTCC 可以演變成 AGTTCC(A 變 G)，而在 C 程式比對中這三個 operator 也是判斷是否相似的關鍵，也就是一個程式的內容很容易就可以改寫成另一種程式內容，但是他們是相似的，因此在判斷是否有相似之處，這三個 operator 是不能少的。

2.2 Dynamic Programming 的概念

Dynamic programming 被廣泛地應用在各種演算法，例如: Dijkstra's algorithm(求最短路徑)[3]，在此我們將用一個求最短路徑問題的例子來介紹何謂 dynamic programming 並說明如何套用它至字串 alignment 上。

假設有一圖:



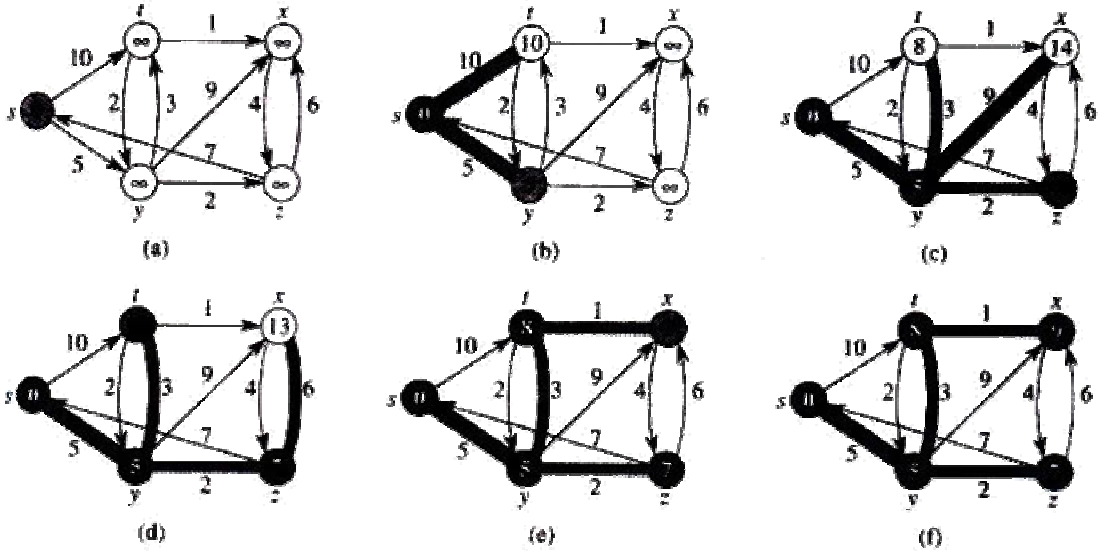
假設要求從 S 至 T 的最短路徑，套用 Dijkstra’s algorithm，我們很容易可以得到其最短路徑為 S -> B -> D -> T。

Dijkstra’s algorithm:

```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
S ←φ
Q ←V[G]
While Q!=φ
        do u ← EXTRACT-MIN(Q)
           S←SU[u]
            for each vertex v∈Adj[u]
               do RELAX(u,v,w)
```

```
INITIALIZE-SINGLE-SOURCE(G,s)
For each vertex  v ∈ V[G]
do d[v] ← ∞
 π[v] ← NIL
d[s] ← 0

RELAX(u,v,w)
If d[v]>d[u]+w(u,v)
        then  d[v] ←d[u]+w(u,v)
              π[v] ←u
```



圖表 三: 利用 Dijkstra's algorithm 的例子

2.2.1 Dynamic Programming 在 DNA 序列 alignment 的應用

由上章節例子可知，dynamic programming 的精神就是把一個較大的問題分成幾個較小相同的子問題，然而與 Divide-and-Conquer 不同的是 dynamic programming 的子問題是重覆的，而且其是利用一 recursive 通式不斷重複解決原來問題[4]。

因此假設有二個DNA序列: ATTCC，GTTAC，我們可以先分成較小的子問題，即先考慮二序列最後面的字元，則會有四種 alignment(考慮gap)的可能(@表示可以先不考慮的字元):

@@@@C	@@@@C	@@@@@-	@@@@@-
@@@@@-	@@@@C	@@@@C	@@@@@-
(一)	(二)	(三)	(四)

但其中由於第四種狀況無意義，故可以略去不考慮，最後字元討論完後，之後再討論倒數第二個字元，如此反覆下去，必定可以找到最佳的 alignment ，即有最高分數的 alignment，且我們稱之為 **“最佳相似”**(我們假設有最高分數的 alignment 即為最相似)，這樣的理論也可應用在程式 alignment 上。

2.3 C語言之 token 介紹

在此節裡我們將簡短介紹如何將 C 語言程式分成 token 字串，這並且是我們轉換 C 語言程式為字串的單位然後再說明何謂程式間的相似。

一般來說要作程式比對的工作，因為程式比 DNA 序列來得複雜許多，為了方便起見，不外乎先要轉換為字串再去分析比較，在此有的是以 “行” 為單位，有的是以 “區斷” 為單位，甚至有人是以 “函式”為單位去作比較。但為了更精確的比較，則在我們系統裡則是以 “符號” 為單位，如此一來程式間的演變可以更加地掌握(即是否由一個程式改成另外一個程式)，這在判斷程式間相似是很有幫助的。

例如考慮一簡短的 C 程式:

```
#include < stdio.h >
struct test
{
  int takes[6];
};
```

我們將其分成 [#]，[include]， [<]， [stdio.h]， [>]，[struct]，[test]，[{]， [int]， [takes[6]] [，]; [，] [}]， [;] 等 token。

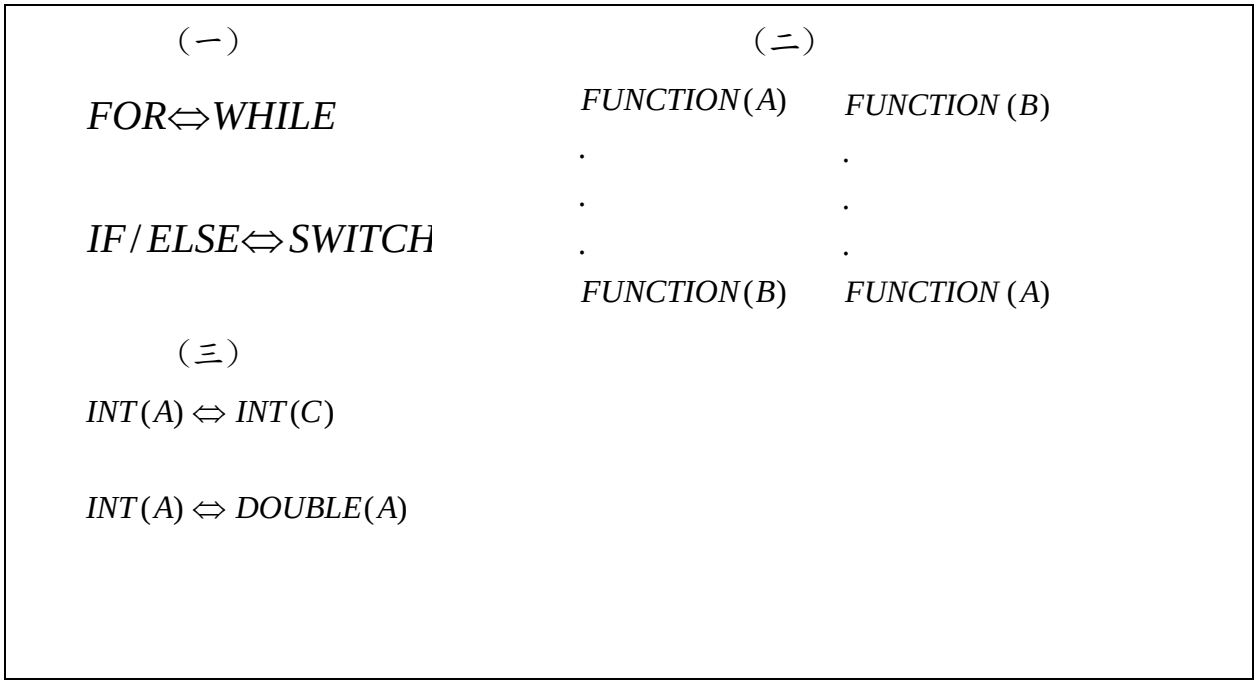
因此我們把 token 分為:

(一) keyword (二) 變數 (三) 標點符號 等三種

2.3.1 何謂程式間的相似

在此我們要定義何謂程式間的相似。

我們認為程式間相似有**（一）語法相似（二）函式互換（三）變數替換**等三種，另外，我們實作的系統也是根據這三種情況以來考慮程式間的相似度，且我們也對換行或空白都有處理。



圖表 四：三種相似的例子

2.4 DNA 序列 alignment 演算法的介紹

Dynamic programming 的處理理論有很多應用[5]，若單以暴力法來求 alignment ，則所花的時間必定驚人，因此便有幾種 algorithm 的出現，而在 DNA 序列比對上比較著名的有下列四種[6]，分別為 (1) Global alignment algorithm ; (2) Local alignment algorithm ; (3) FASTA alignment algorithm ; (4) BLAST alignment algorithm ，每一種都有特定的評分機制，而其中第 3 及第 4 種是針對欲搜尋大型資料庫而設計的，我們將針對此四種 algorithm 所花的時間複雜度和特點來作討論。

(1) Global alignment algorithm:

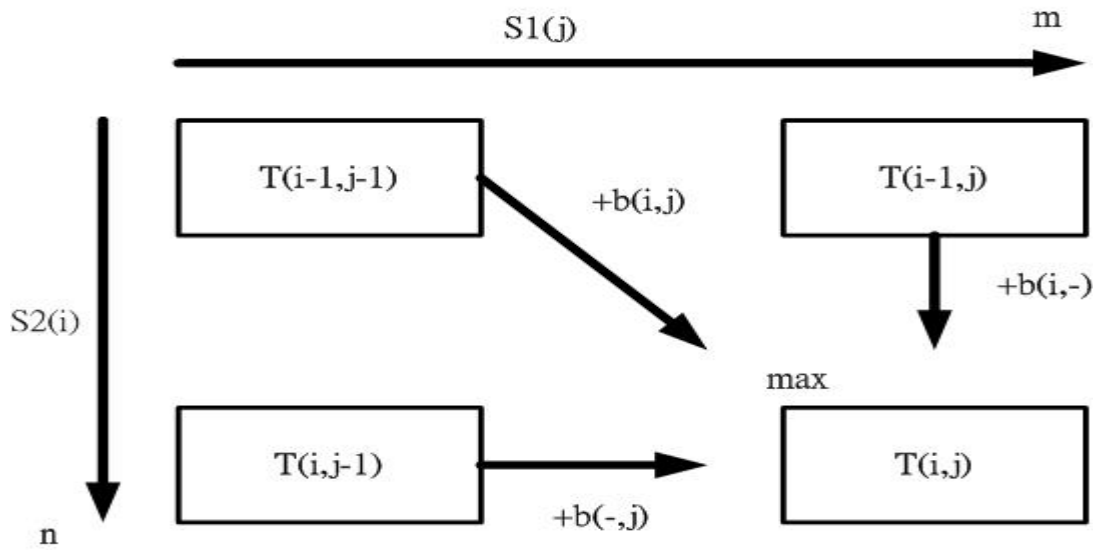
這方法是由 Needleman and Wunsch 所提出，其特點在於利用此方法，便可很有效率地求出 DNA 二序列間最佳的 alignment[7]，假設二個 DNA 序列長度分別為 m、n，則其時間複雜度為 O(nm)，且其評分機制為:

[1]

$$T(i,j)=\max\begin{cases}T(i,j-1)+b(-,j)\\ T(i-1,j)+b(i,-)\\ T(i-1,j-1)+b(i,j)\end{cases}$$

$$T(i,0)=-i,1\le i\le n$$
$$T(0,j)=-j,1\le j\le m$$

[1]其中b()為自行的定義分數而往回走須從 T(m,n) 走回。



圖表 五: Global alignment algorithm 評分機制

二 DNA 序列 CGT，CAGT
假設 字元相符 +1
子元不符 -1

		C	G	T
	0	-1	-2	-3
C	-1	1	0	-1
A	-2	0	0	-1
G	-3	-1	1	0
T	-4	-2	0	2

最佳的 alignment 爲
C - G T
C A G T

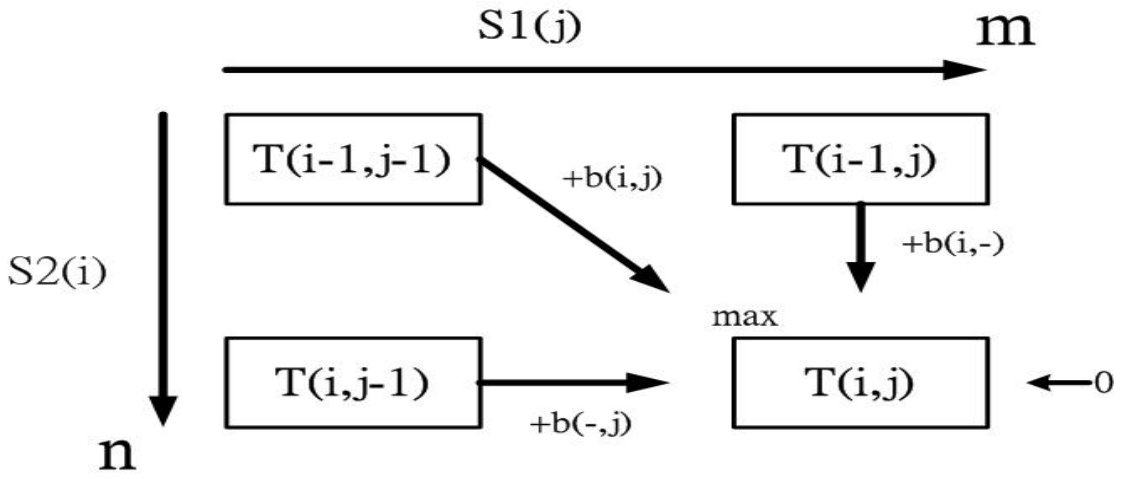
圖表 六: Global alignment algorithm 範例

(2) Local alignment algorithm

這方法是由 Smith-Waterman 所提出，其特點在於利用此方法，便可很有效率地求出 DNA 二序列間**局部**最佳的 alignment[8][9]，這個方法是用來尋找 DNA 中最相似的基因，假設二個 DNA 序列長度分別為 m、n，則其時間複雜度為 O(nm)，且其評分機制為:

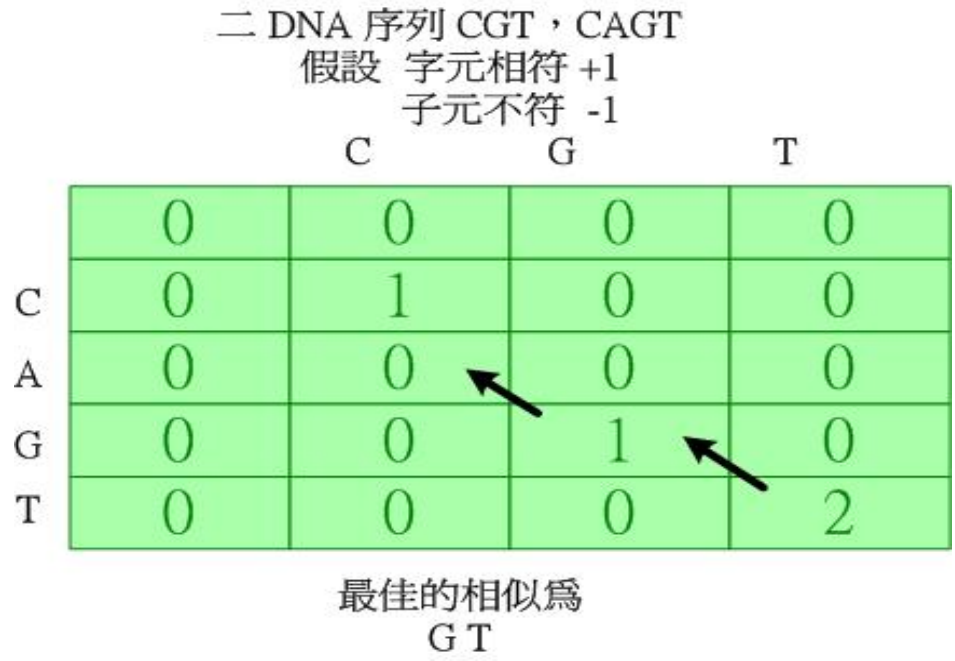
[2]

$$T(i,j)=\max\begin{cases}T(i,j-1)+b(-,j)\\ T(i-1,j)+b(i,-)\\ T(i-1,j-1)+b(i,j)\\ 0\end{cases}\qquad \begin{aligned}T(i,0)&=0, 1\le i\le n\\ T(0,j)&=0, 1\le j\le m\end{aligned}$$



圖表 七: Local alignment algorithm 評分機制

[2]**其中 b()為自行定義的分數，往回走時須從 T(m,n) 中最大分數走回:**



圖表 八: Local alignment algorithm 範例

(3) FASTA alignment algorithm

這方法是由 D. J. Lipman and W. R. Pearson 所提出，其特點在於利用此方法，根據經驗法則，便可很快地求出 DNA 二序列間**局部**最佳的 alignment[10]，這個方法是用來尋找 DNA 中最相似的基因，由於它速度之快，所以與前面二個不同是它是被套用在搜尋大型資料庫用的，現在的很多網站也都是用它來實作。

FASTA 是在計算好的表格，利用出現最高次數的”位移量”去作位移的選擇，因此它只能找到最佳基因，且位移量有**正、負**二種情況，正代表 target 序列往右平移，而負代表往左平移。

DNA 二序列爲
CTGACAG
GAC

Query Table

word	A	C	G	T
pos	4	1	3	2
		5	7	

Target Table

1	2	3
G	A	C
2	2	-2
6		2

由表格可得知其最相似的local DNA序列爲 GAC

圖表 九: FASTA alignment algorithm 範例

(4) BLAST alignment algorithm

這方法是由 W.Miller 等人 所提出，其特點在於利用此方法，便可很快地求出 DNA 二序列間**局部**最佳的 alignment[8]，這個方法也是用來尋找 DNA 中最相似的基因，所以與前面二個不同是它是被套用在搜尋大型資料庫用的，現在的很多網站也都是用它來實作，由於它速度比 FASTA 更快，原因是它有事先建好表格，在尋找資料庫時，只要依這些表格，便能很快地找出相似基因，所以現在大部份都是用此較多。

第三章 應用 DNA 演算法實作系統之侷限

在此章節裡，我們將介紹設計系統時所遭遇到的困難，在下一章節裡我們會分別將這些困難解決作詳細的介紹，由以上的章節知， DNA 序列的 alignment 有很多演算法可以利用，且由於程式的比對跟 DNA 很類似，因此這些演算法也可以相同地套用在程式比對上，例如程式的 **“函式”** 相對於 DNA 上的 **“基因”**，然而實際上，沒有這麼簡單就可完成，因為在 DNA 序列上，由於只有簡單的四個字母 (A，T，C，G)，而 C 程式裡比 DNA 序列較複雜許多(C 程式有 KEYWORD、變數、符號等)，因此，若要套用這些演算法，勢必會遭遇很多困難。

而，若單單只利用字串比對來比較 edit distance[11-15]，在程式中，必會漏失很多的資訊，以至於造成誤判的產生，接下來，我們將探討在設計系統時所遇到的困難。

3.1 序列中字母的限制

由於 DNA 是由簡單的四個字母(A，T，C，G)所組成，但 C 程式是由 KEY WORD 以及變數所組成的，況且在 C 語言裡光是變數就是好幾百個，更何況又是變數呢!而且光是把 C 程式碼轉為 token 時，如何再把 token 轉為更方便去比較的字母單位呢 ?尤其是 token 太長時。還有在 C 程式裡有很多特殊的寫法例如:

```
printf(“xxxxxx”,…) 或 printf
                        ("xxx");
```

或者是注解是否要考慮呢 ? 等等都是我們設計系統所面臨的。

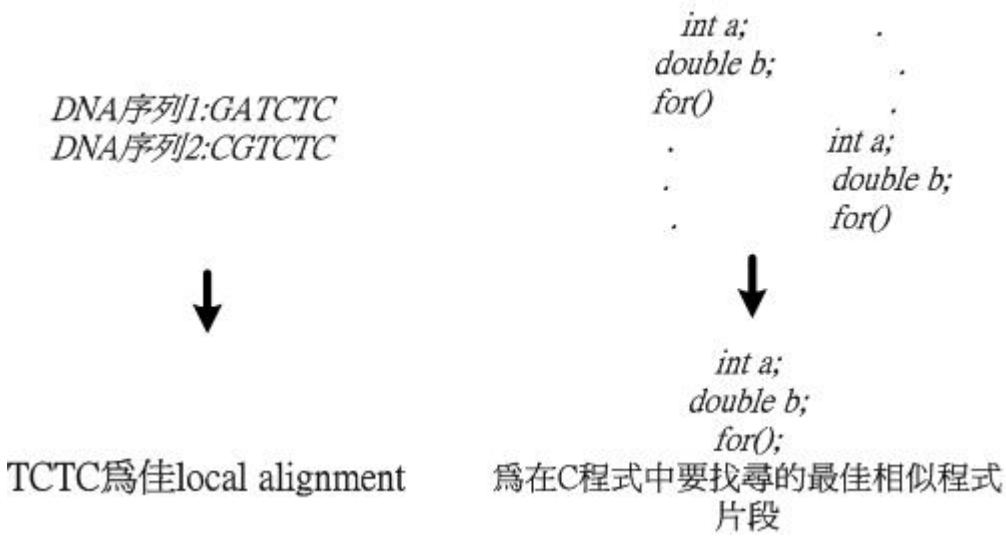
	DNA	C 程式
比較單位	A T C G	token
序列中的字母	A T C G	變數、key words
其它因素之考慮(換行，空白…等)	無	有(很多)

表格 1: DNA 序列比對與 C 程式比對的比較

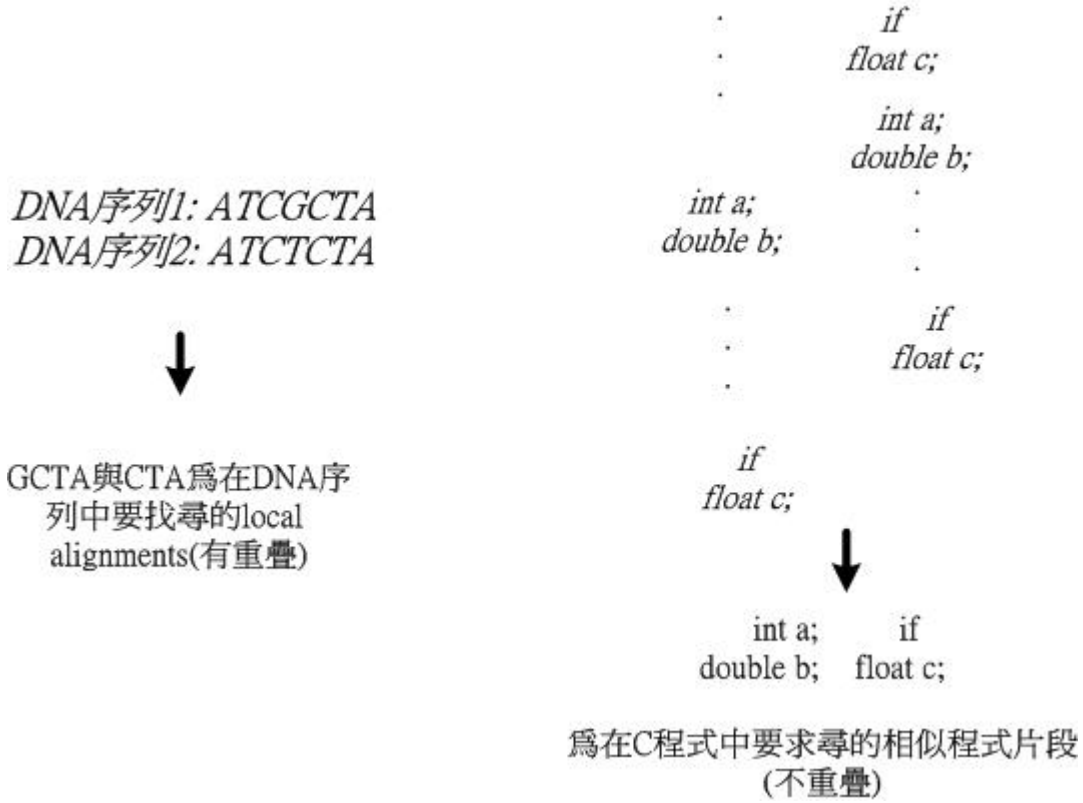
3.2 Local alignment algorithm 的限制

在上一章節有提到，Local alignment algorithm 可以用來找到二個 DNA 序列中一個最佳的 local alignment(分數最高)，這對於生物學家來說，這最佳的 local alignment 或許他們要尋找的結果，雖然在把 C 程式轉為 token 後，也跟 DNA 序列 一樣要找尋相似的程式片斷，但由於一個人在改寫程式時，有可能造成相似的程式片段有好幾段，因此光利用 Local alignment algorithm 來實作的話，會使得程式中其它相似程式片段無法找到，這也是我們所不希望的。

再者，雖然在 DNA 序列演算法裡有文獻提供找尋幾個 local alignments[16-18]，因為對於生物學家來說，或許還須找尋其它片段序列，例如:SARS 中的基因，但由於文獻所找出的 local alignments 是重疊的或是講求利用逼近最佳解法來找尋，且在我們的程式比對中，我們所要找尋的相似程式片段是不重疊的，因為我們不希望有無意義重疊相似程式片段的出現且我們對於程式的相似度上也不希望造成誤判，所以這也是我們面對的困難之一。



圖表 十: DNA 序列最佳 local alignment 範例



圖表 十一: 不重疊的相似程式片段的範例

3.3 Global alignment algorithm 的限制

在上一章節有提到，Global alignment algorithm 可以用來找到二個 DNA 序列中一個最佳的 alignment(分數最高)，這對於生物學家來說，這最佳的 alignment 或許就是他們要尋找的，雖然我們可以把全部 C 程式轉為 token 後，然後再跟另一個比，因為找出最佳的 alignment 相當於都幫我們找到了，但由於一個人在改寫程式時，有可能會把函式互換，這樣也是相似，然而光利用 Global alignment algorithm 來實作的話，會使得程式中函式互換無法找到，這也是我們所不希望的。

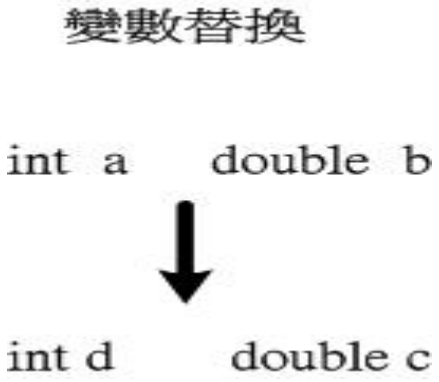
因此，對於函式互換 Global alignment algorithm 沒有辦法找出，因此它可以適用在沒有函式互換的程式，反之，就不能使用它，這也是我們遇到的難題之一。

圖表 十二: 程式中沒有函式互換的例子

圖表 十三: 程式中有函式互換的例子

3.4 變數間的替換

在上一章節有提到，WinDif 對於變數替換無法區分，因此就降低了準確度，然而辨別變數替換在程式比對中是必須的，而利用 DNA 演算法也無法滿足這方面的需求，所以這也是我們遇到的難題之一。



圖表 十四: 變數替換

3.4.1 變數間型態的替換

WinDiff 對於變數型態替換無法區分，因此就降低了準確度，然而辨別變數型態替換在程式比對中也是必須的，且 DNA 演算法還是無法滿足這方面的須求，所以這也是處理變數遇到的難題之一。

圖表 十五: 變數型態替換

3.5 FASTA algorithm 的問題

在上一章節有提到，WinDiff 在處理程式比對的速度是相當的快，但其精確度並不是理想，然而有時系統使用者也許想要很快且精確度也不能太差，因此在這方面 FASTA algorithm 便能擔任此角色，由於它是被廣泛地用在搜尋大型資料庫上。

它的原理是利用 **“位移”** 的觀念，因此它的速度很快，但 FASTA algorithm 跟之前的 global、local alignment algorithm 相似，所以想要套用之，必定會遇到一些問題，所以這也是我們遇到的難題之一。

DNA 二序列爲
CTGACTAG
GACAG

Query Table

word	A	C	G	T
pos	4	1	3	2
	7	5	8	6

Target Table

1	2	3	4	5
G	A	C	A	G
2	2	-2	0	-2
7	5	2	3	3

由表格可得知其最相似的 local 序列爲 GAC，但其中 local 序列 AG 沒有被找到

圖表 十六: 利用 FASTA algorithm 演算法的範例

3.6 BLAST algorithm 的問題

BLAST (Basic Local Alignment Search Tool)是一套在蛋白質資料庫或 DNA 資料庫中進行相似性比較的分析工具，BLAST 程序能迅速與公開資料庫進行相似性序列比較且 BLAST 結果中的得分是對一種對相似性的統計說明。

BLAST algorithm 與 FASTA algorithm 速度都是很快地，且 BLAST 比 FASTA 快，但精確度沒有比 FASTA 來得高，與 FASTA algorithm 不同的是 BLAST 採用統計方法來作 DNA 序列比對，同時也是被用在搜尋大型資料庫，況且 BLAST 有作過前置處理(建立演化 TABLE,如著名的 **PAM** 與 **BLOSUM**，之後再利用此來作胺基酸序列比對[19-20])，所以會比 FASTA 來得快，而現在很多 DNA 比對程式便是用它來實作。

因此在程式比對方面也許能利用 BLAST algorithm 來實作，但由於程式中的演化並沒有像 DNA 演化一樣地固定，若單純利用，其後來比對的結果，必定會造成誤判，且程式序列也比 DNA 多很多，因此在建表格上也很困難，所以想要套用之，必定會遇到一些問題，這也是我們所不希望的。

BLAST 也是跟 FASTA 一樣採用 **“位移”**，但其中有不一樣的地方，看下面的圖。

傳統 BLAST 比對步驟:

（一）把 query 序列分成 words。

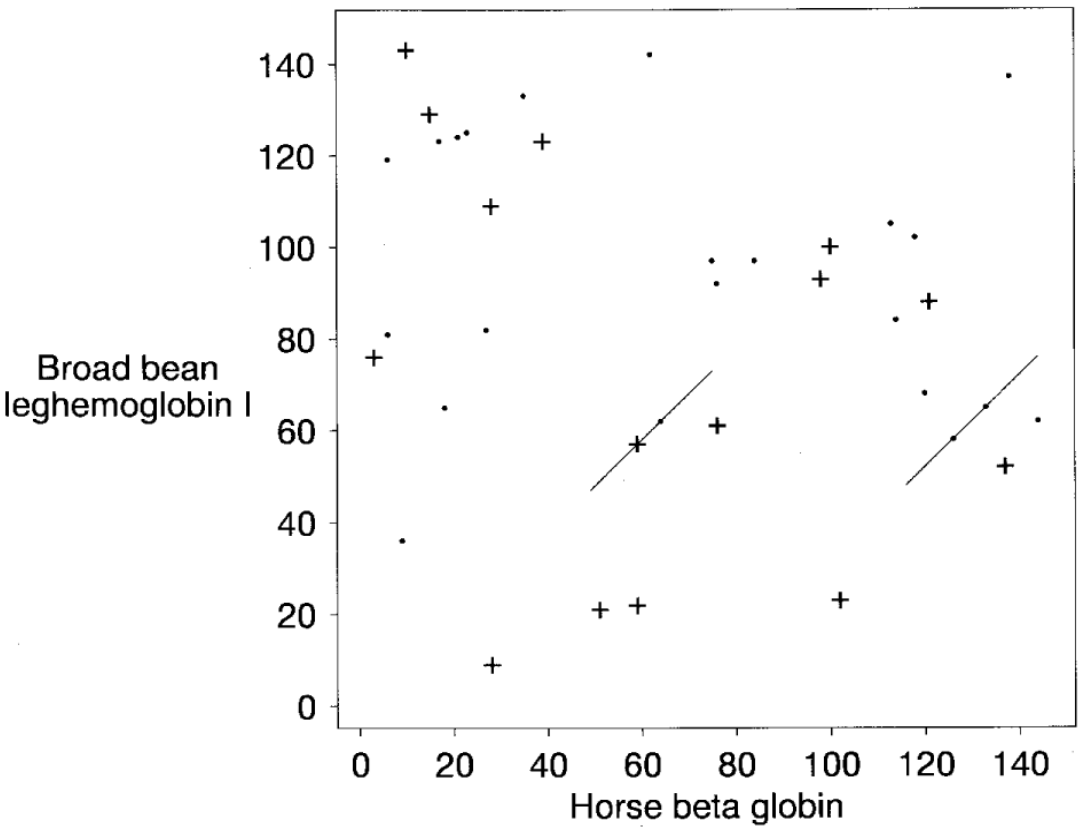
（二）開始在資料庫裡的序列(target 序列)尋求有無這些 words 的出現。

（三）若有一個 word 在 target 序列中有出現，則延伸此 word 的長度(word 在 query 序列後的字母加進去)，繼續比對下去。

圖表 十七: 利用 BLAST algorithm 演算法的範例

圖十七是 BLAST 傳統作法，然而若單純利用傳統作法，有可能找出二 DNA 序列間很多段相似短序列，而對於較長或最佳將無法找出。

如圖:



圖表 十八: 利用傳統 BLAST 作法的例子

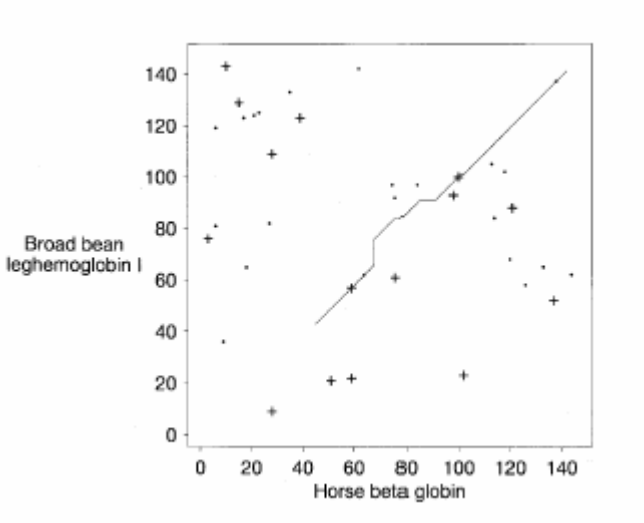
而最新 BLAST 的作法是[21]:

(1) 把 query 序列分成 words。

(2) 先利用二個 words 開始在資料庫裡的序列(target 序列)尋求有無這些 words 的出現。

(3) 若這二個 word 在 target 序列中有出現,且這二個 word 間的分數超過定義的門檻時,則分別延伸此二個 word 的長度,延伸完再利用 dynamic programming 找尋最佳的 alignment。

如圖:



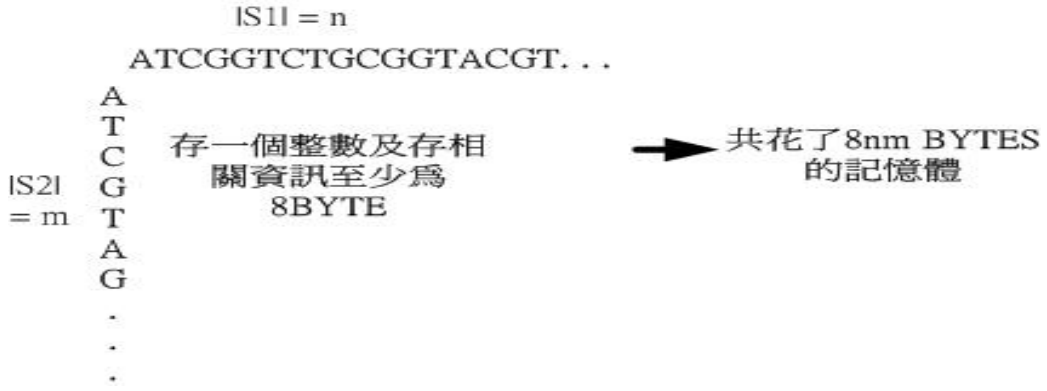
圖表 十九: 最新 BLAST 的作法的利用

3.7 記憶體的限制

由上面的章節可知，任何演算法，只要用到 dynamic programming 的，其必定需要相當量的記憶體，光是建立表格已經是很耗記憶體了，更何況還要紀錄相關的資訊，所以我們知道在記憶體有限的情況下，要實作 dynamic programming 是不可能的。

在此有文獻指出可用 Linear space 來完成 backtracking[22]，但由於我們必須還要找出其它的程式片段， 然而我們在設計程式比對系統的流程間，還是會用到 dynamic programming 去求出最佳解，因此，在有限的記憶體下，我們必須想一個辦法來克服此問題，這也是我們遇到的難題，因此在下一章我們會介紹一種用來結省記憶體的方法。

假設要比對的序列長度為 n ，另一個序列為 m ，所花費的時間為O(mn)若當 m 與 n 很小時就不太會影響，但反之必定會很大，如圖:



圖表 二十: 記憶體使用情況的範例

3.8 相似的評分標準

在程式比對完之後，接下來就要開始作分析比較，在這方面，每個人對於相似分數來評估兩程式碼的相似度是很主觀的，如:我們可以說，“若兩程式碼有60%相似，便稱它們相似，或者是只要某幾段相同便為相似”，所以很難有一套很客觀的講法。

或許有人會利用一個 “threshold”來找出相似的片段，之後在來計算分數，這樣作也不失一個辦法，但我們卻不知這 threshold 要給多少值呢 ？ 這個也是一個很主觀的想法。

然而，在程式比對上，評分機制是一定要作的，我們不想用主觀的方法，也不想造成誤判的產生，所以這也是我們遇到的困難。

第四章　困難的解決

在上一章節裡，我們提到關於設計系統所遇到的難題，然而在我們的系統還是須要用到上面提及的方法，因此，在這一章節裡，我們將詳細說明如何解決以上的困難。

4.1　C 語言 token 的處理

C 程式的 **key words** 與**變數**加起來比 DNA 來得多也來得複雜，因此在這裡我們先把 C 程式碼先轉換為 token，然後再轉換為數字，之後再拿數字序列去比對。

然而，我們必須對於 key words 先定義其所屬的數字表，而變數採用一計數器來紀錄並累加，因此整個處理 token 步驟如下:

(1) 把 C 程式轉為 token:

在這裡，我們視 C 語言的 [變數]，[符號]，[key word]等為 token，且對於**空白**一律不考慮，且**包含檔**(如#include <math.h>)也不考慮，另外**註解**也不考慮，原因是程式寫作者有時會加入空白以增進閱讀性，加入註解以提醒該行或該段欲作的事，另外，包含檔由於程式碼都一樣，所以若考慮的話會失去其準確度，因此整個 C 程式的 token 就產生出來了。

Ex :　printf
("xxx");

轉 token 變爲 [printf] [(] [xxx] [)] [;]

(2) 把 token 轉為數字

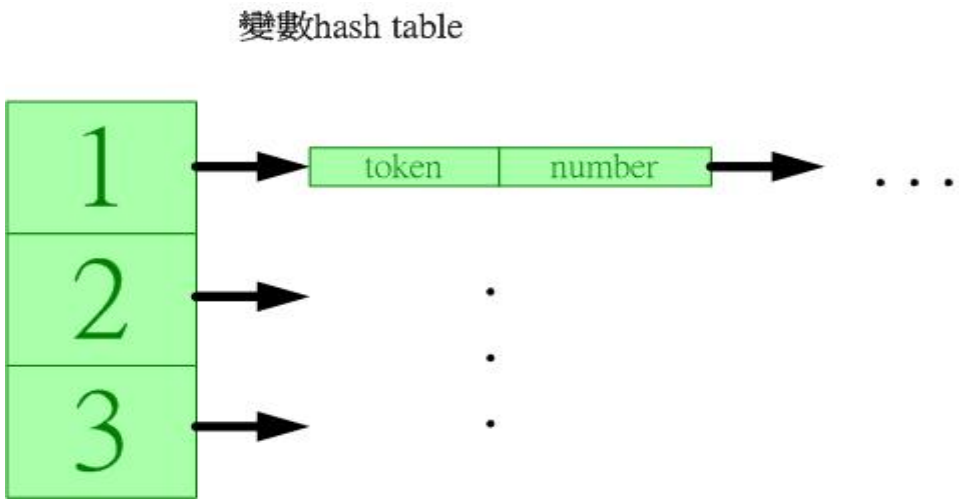
在此，我們將轉換後的 token 再轉為數字，由於 key word 是 C 語言固定的字，因此我們事先定義好一個表格(key word table(如表二)，然後再依照此表轉為數字(我們把 key word(含符號)定義為“負數”)，而變數採用一計數器累加(變數皆為正數(包含 0))，如此一來變把整個 toekn 都轉為數字了。

但由於我們轉換的不只一個 C 程式碼，所以難免會有重覆的變數名稱出現，為了處理這類的情況與增進系統的效率，我們利用 hash 的方法，把變數的 token 的每個字母相加以求得 hash 的索引，若索引相同，則利用指標來處理(如圖十九)。

利用此資料結構，在每次得到變數 token 時變去尋找有無重覆便可很容易得到所屬的 number。

Key word	number
int	-1
for	-2
switch	-3
;	-4
long double	-5

表格 2: keyword 表(事先定義)



圖表 二十一: 變數資料結構的範例

(3) 數字再存到表格裡

在這裡的表格，是存放每個程式碼轉換後的數字以供之後去分析比對，表格裡有[token]、[number]、[class]，其中 token 與 number 都以說明過，而 class 在之後的變數型態也會詳細的介紹。

4.2 C 程式碼的分群

在上節已說明如何把 token 轉為數字，轉換數字後接下來要幫每一個數字作分群，之所以要分群是為了利於檢查變數型態與變數替換，例如:

```
double a -> float b;
```

這行代表變數替換了且變數型態也替換了，但在語法上他們是相似的，所以為了有效判斷，我們利用了分群的技巧，把每個 token 給定一數字，代表它們是屬於第幾群。

例如:

int、long double 就屬同一群

在此，我們先把定義好的 token(key word)先分群，而變數一律全部給定一常數，但為了區分，變數給定的數字一定不會跟其它 token 一樣，所以分群的步驟如下:

(1) 先把 key word 的群組號碼先定義好

就是在把之前的 key word 表格(表二)，再多出一欄位來紀錄該行 token 所屬的群號，所以表二就以表三替換之:

number	class
-4	4
-5	1
0	5
2	5
3	5
5	5

表格 3: 群組表

(2) 變數則用一常數編組

因為事先已知道 key word 裡的群號，所以為了區分 key word 裡的群號，我們給定變數一常數群號。

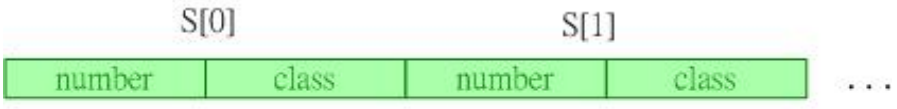
如此一來，C 程式碼已經轉換為我們想要的格式，其它的 C 程式碼也採用相同的方法，所以全部欲比對的 C 程式碼都已轉換完成(我們稱之為 number 檔)，其格式如表四:

token	number	class
int	-1	1
for	-2	2
switch	-3	3
;	-4	4
long doulbe	-5	1

表格 4: C 程式碼轉為 number 檔的內容範例

(3) 轉為字串

在一個 C 程式碼轉換為 number 檔之後，我們就可以拿此檔來編成數字序列，再利用一資料結構來儲存數字序列，資料結構如下圖:



S爲一序列的陣列

圖表 二十二: 序列資料結構的範例

然後再利用 S 陣列去作比對分析。

4.3 找出程式中不重疊部分的方法

在上一章裡，我們說明了 local alignment 的方法以及它在程式比對中的應用，在此有文獻指出方法可以尋找多段相似 DNA 序列，而且它們是**重疊**的，但在程式裡，我們也是必須尋求其它相似片段，不同的是它們是**不重疊**且**相似**的程式片段。

為了時間上的考量，在此，我們不打算利用 “**重算** dynamic programming” 的方法，取而代之的是 “**邊算邊紀錄**” 的方法。

因此，利用此方法，便可省下大量的時間，只要之後再配合一些機制，便可找出全部相似的片段。

所謂 “紀錄”，是紀錄相似程式片段的總分有超過門檻的(我們將在之後介紹我們定義的門檻)，而邊算代表計算 dynamic programming ，在這裡只要計算一次就好了，而在算的同時就紀錄下來超過門檻的片段，紀錄完之後，接下來利用排序的方法，且是依分數遞減的方式作排列，然後再一一尋找不重疊且相似的程式片段。

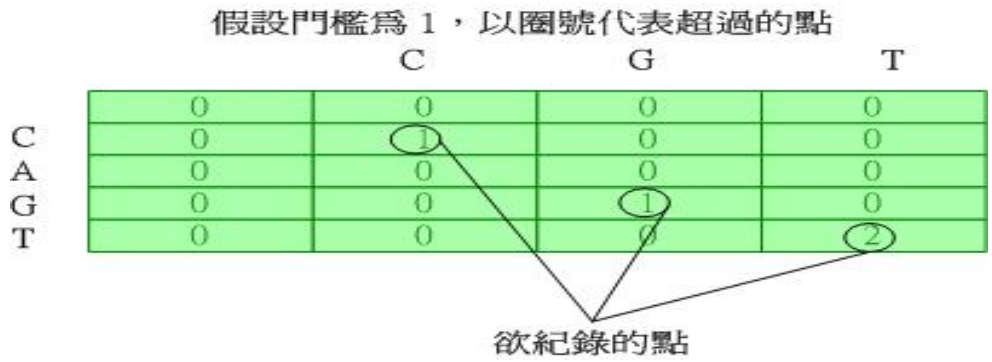
因此，整個計算 dynamic programming 的流程可以分為以下步驟:

(1) 利用字串陣列來作比對:

在此，我們利用上一章所提及的字串陣列(圖十八)去作比對，且程式是以兩兩去作比對(在此之前，全部欲比對的 C 程式碼已經轉為數字字串了)。

(2) 紀錄超過門檻的相似程式片段

我們先定義一門檻，然後再利用超過之來紀錄相似程式片段，而且，整個 dynamic programming 也是同樣地在計算，如圖:

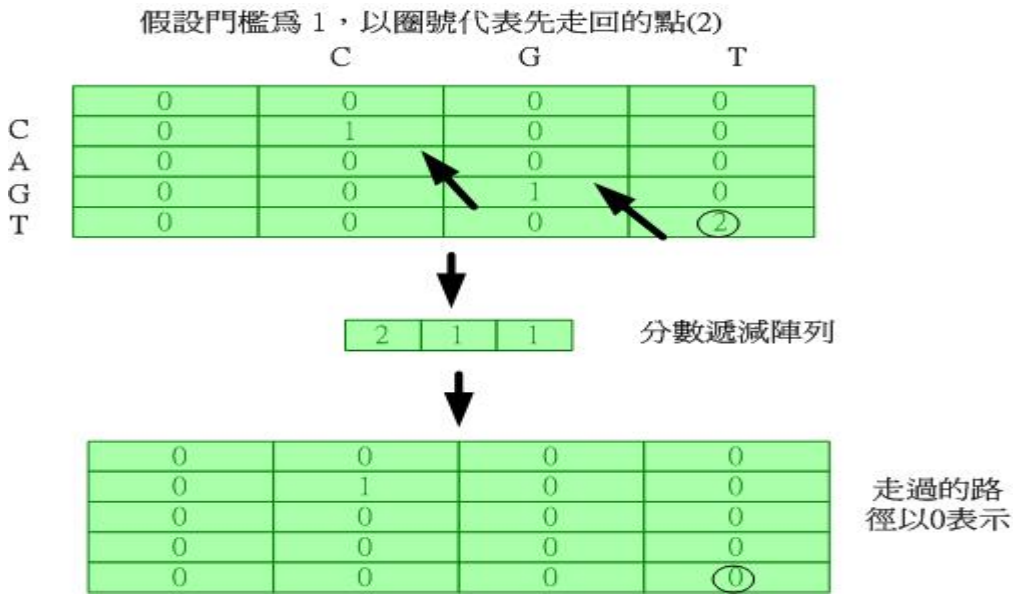


圖表 二十三: 邊算邊紀錄的範例

(3) 以分數遞減的方式排序被紀錄的點

在此，所有超過門檻的點已被紀錄了，接下來，再利用排序的方法，使分數以遞減方式排列，而我們就是從第一個(最高的分數)找起，若以圖二十的例子，其排例為 2 1 1 ，而我們就是從 2 先找起。

以上三個步驟便是我們 ”邊算邊紀錄”的方法，然而在從最高分數走回去時,我們還須先用一個簡單的方法來紀錄走過的程式片段,由於 local alignment 是以 0 來表示走回去時的終止條件，所以我們也以 0 來表示走過的路徑，之後若走相同路徑中的點便可以很容易地知道了,我們便是配合此機制來找出全部不重疊的相似程式片段，如圖:



圖表 二十四：邊算邊紀錄的範例(二)

4.4 FASTA alignment 套用

BLAST 雖然比 FASTA 來得快，但 BLAST 是依據 ”事先定好的表格(依演化的規則)”去分析比對，然而程式的 token 又來得比 DNA 多樣很多，其序列也更複雜，若想事先定義表格，那將是很困難，因為程式比對沒辦法像 DNA 一樣有著”演化的規則”，因此在我們的系統上，決定採用 FASTA alignment 演算法來比對分析。

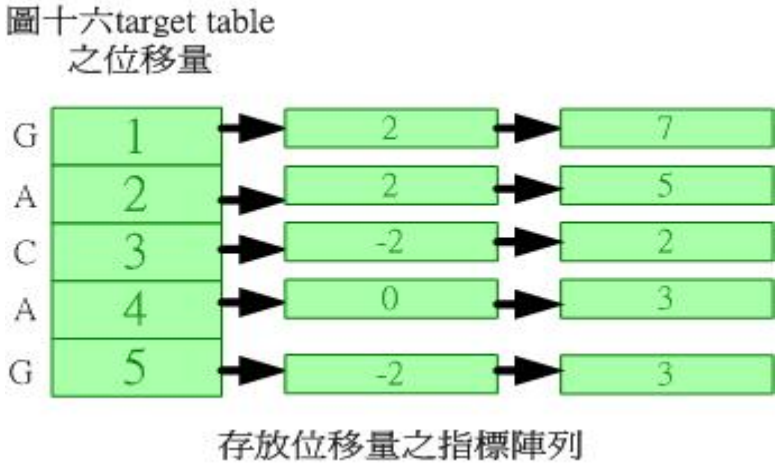
倘若一個人若想快速地完成比對分析，那麼，由於 FASTA alignment 是利用 ”位移”的觀念來作比對，所以將是不錯的選擇，但 FASTA 與前幾節講到的 local alignment 相似，因此，我們在這也是將位移量先紀錄，之後再將位移量依遞減的方式排列，只要再配合一機制便可將全部相似的程式片段找出。

在此，我們宣告二個一維 bool 陣列(長度為二個欲比對序列的長度)，用來紀錄已被走過的程式片段，因此，利用它，我們可以找出不重疊的相似程式片段。

因此，整個 FASTA alignment 的步驟為:

(1) 計算位移量

我們利用 query table 以及 target table，去計算位移量，並且用一資料結構存放之。



圖表 二十五: FASTA 存放位移量的資料結構範例

(2) 依位移量出現次數遞減方式排序

將位移量依圖二十二之資料結構建好之後，我們邊開始存放位移量，且紀錄每個位移量發生的次數，之後再依照發生次數依遞減方式作排列，而且我們還紀錄每個位移量在 target 序列中片段起點與終點位置。

如圖二十三，位移量為 2 的程式片段的起點位置為 1，而終點位置為 3，而位移量為 3 的程式片段的起點位置為 4，而終點位置為 5。

然而，我們觀察位移發生次數的變化，由於發生次數不會大於 target 序列的長度，因此，我們可以利用 index sort 來將次數作排列，這樣大大地減少時間複雜度，也提高了 FASTA 的可用性。

(3) 尋找全部不重疊的相似程式片段

由於我們要找到不只最高分數的程式片段，因此在此我們利用二個 bool 陣列來找出全部的相似程式片段，但由於 FASTA 的位移量有**正、負**二種因此利用一 general 數學公式:

方程式 1:

$$\text{target sequence} : x, y$$
$$\text{query sequence} : z + x, y + z$$

1. 位移量為正時:

例如二序:

query(S1) 序列: ATTCG

target(S2) 序列: TTGCA

經計算 table 結果知位移 2 在 target 序列中的起，終點位置是(1,3)，套用式子 1 得知在 query 序列所對應的位置是(3,5)，因此我們把序列 align 為:

ATTCT
  ATTCG

2.位移量為負時:

例如二序列:

query(S1) 序列: ATCGT

target(S2) 序列: CCATC

經計算 table 結果知位移 -2 在 target 序列中的起，終點位置是(3,5)，套用式子 1 得知在 query 序列所對應的位置是(1,3)，因此我們把序列 align 為:

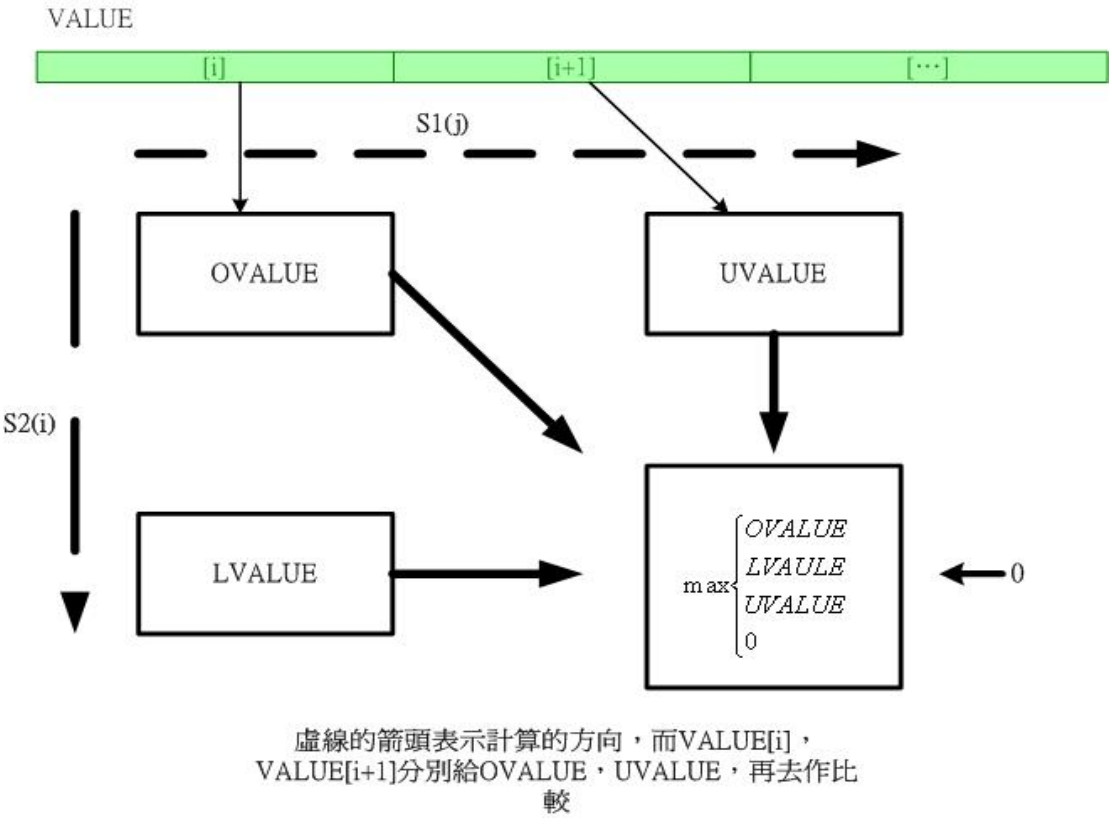
  ATCGT
CCATC

利用上面數學式子 1，再配合二個 bool 陣列，便可找出全部的相似程式片段。

4.5 解決記憶體之問題

在第三章我們曾提及，在作 dynamic programming 時，若在 score 矩陣(圖七)上要存下分數、反回的資訊等，必須要花費相當寵大的記憶體量，若比對的程式很大時，那麼記憶體不足的問題便要解決。

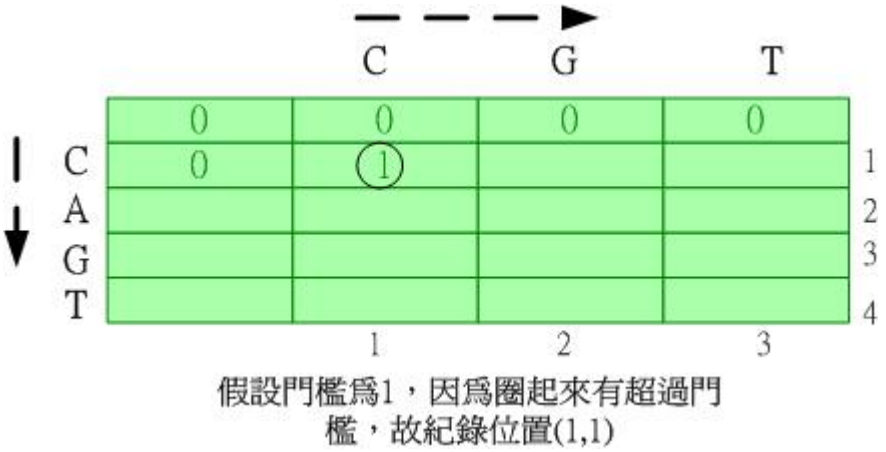
我們觀察在作 dynamic programming 時，由於我們只需紀錄返回的資訊，而對於 score 矩陣上的每一格我們不需紀錄其分數，在此我們先定義一門檻，只要 score 矩陣上的每一格的分數有超過之，便紀錄其位置（分別對應欲比對序列上的字母），如此一來，我們就可以只利用三個整數型態的變數以及長度為|s1|的整數陣列(整數陣列用來紀錄欲計算列的上一列的值)即可計算每格的分數，令三個整數為 LVAUE,UVALUE,OVALUE，而陣列為 VALUE 則其規則以圖來表示為:



圖表 二十六: 以三個整數計算 SCORE 矩陣

只要配合計**算的方向**與**一門檻**就能避免使用大量記憶體去存每一格的分數，如

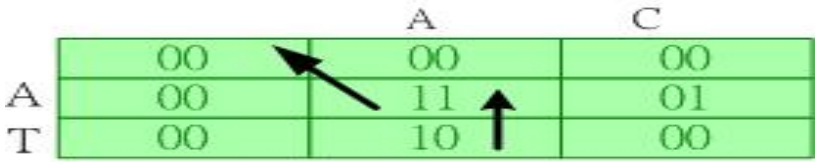
圖:



圖表 二十七: 紀錄有超過門檻之該點位置

再者，由於我們必須紀錄反回的資訊，故我們使用二個 bits 來紀錄[23]，我們以[00]、[10]、[01]、[11]分別來代表該方格要[終止]、[往上]、[往左]、[對角線]之方向，其中[00]代表該格分數為 0(終止)。

Ex:



圖表 二十八: 以二個 bits 來紀錄反回資訊

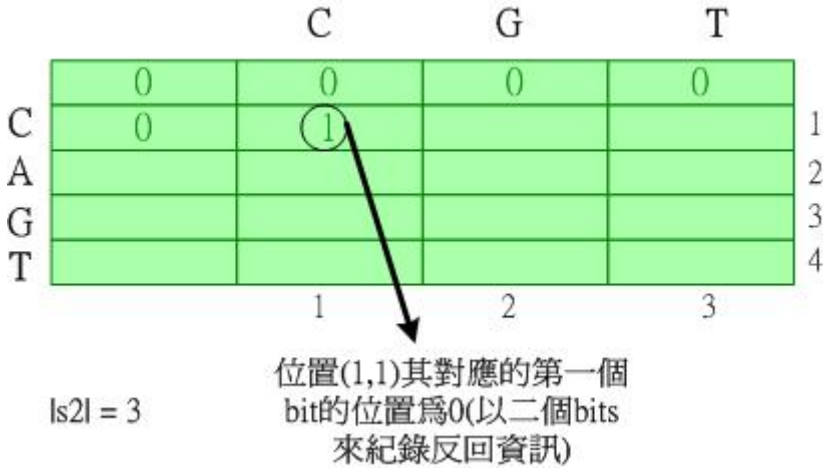
但由於我們的系統是以 C++ 語言來實作，它沒有支援 bit 的資料型態，因此我們定義一資料結構 bitset<32> {SizeOfBit}，其中 SizeOfBit 大小為 $\lceil |S1| \times |S2| \div 16 \rceil$，取 ceiling 整數即為所求的資料結構，有了此以後，再配合

此數學式:

第一個 bit 位置 = [(k * |S2| * 2) + (2 * 1 + 1))] / 32 (取其商數)

(1,k)為圖二十五所紀錄超過門檻的位置，例如，若一超過門檻的方格位置為(1,1)，套用式子便可獲得我們所要的位置，且為該方格的第一個 bit 位置。

如圖:



圖表 二十九: 位置的對應關系

如此一來，所需的記憶體大約為原來的 1/4，因此對於檔案較大的程式，也能比對。

4.6 門檻的選擇

正如前面章節所說，我們不想用主觀的方法來決定門檻的選擇，因為每個人的觀點都會不一樣，因此，我們打算利用統計的方法來決定門檻[24]。

在 DNA 序列比對中，有一統計公式用來計算 DNA 序列間的相似度，而此也是拿來作比對資料庫的方法，其統計公式為:

方程式 2: $$S = \log_2 \frac{K}{P} + \log_2 N$$

以二為底的 log 來分析 DNA 序列間的相似度，且其單位為 bit，其中 S 代表在二個序列中須要符合 S bits 才能稱此二序列相似，而 N 代表二序列長度的乘積，在 DNA 序列比對中 K 為一常數，接近 0.1，而若要使序列比對有意義，則 P 為 0.05，因此 $\log_2 N$ 便決定了 S 的值。

例如，若有二個 DNA 序列，其長度分別為 64 與 16，若要稱它們為相似，則須滿足 11bits(S = 11)，因此長度就決定了所需的相似度。

由於比對的單位為 bit，因此我們也要把評分的單位轉為 bit，其轉換的公式為:

方程式 3: $$x \times \frac{\lambda}{0.693}$$

其中 x 代表所要轉的數值，λ 為一正規化參數，0.693~ln2，其所套用的統計公式為:

方程式 4: $\sum_{i,j} p_i p_j e^{\lambda s_{ij}} = 1$

我們可以利用此公式導出 λ，或者是令 λ 為 1，使得所有評分分數為自然對數，或者是取 λ 為 ln 2，大約等於 0.693，使得評分 log 基底變成 2。

現假設我們取 λ 為 0.3465 則評分分數的轉換如下表:

	轉換前分數: x	轉換後分數(bit): $x \times \frac{\lambda}{0.693}$
相同字母	+5	+2.5
不相同字母	-3	-1.5
字母與 gap	-3	-1.5

表格 5: 原始分數轉為 bit 的範例

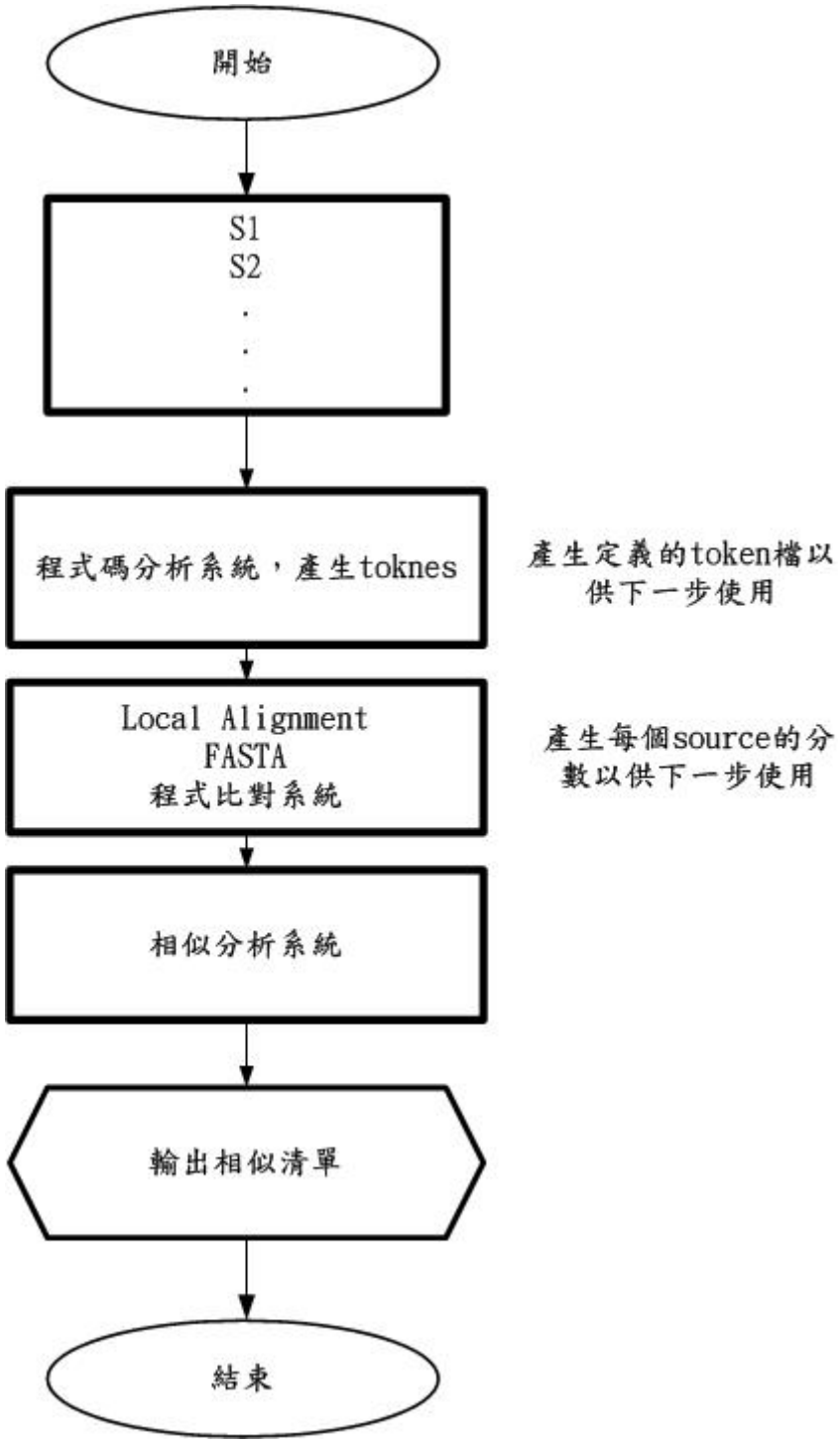
如此一來，我們已經把評分的分數加以定義了，往後只要利用方程式 2，便可知所需的門檻值。

然而，由於 λ 可隨一個人的習慣作調整，因此在我們的程式比對中，我們選取了一些學生的程式作業，然後套用方程式 3，求得 λ 為 0.1536，我們的系統便採取此 λ 值來分析比對。

由於我們是採用 Local Alignment 來比對，套用公式所求出的 S 代表二程式序列中其最高分數至少要 S bits(也就是擁有最高的分數程式片段)，因此我們還另外定義了一門檻，用來決定其它相似的程式片段，因此整個評分標準我們都加以定義了。

第五章　系統實作

5.1　系統架構流程圖



圖表 三十：系統流程圖

5.2 系統功能與界面

5.2.1 使用功能的目的

為了能讓使用者很方便地使用系統，我們採用視窗的形式來呈現界面，我們也提供快速比對與精確比對，另外，我們也提供圖形顯示的功能好讓使用者可以瞭解到任意二個 C 程式碼間的相似部位在哪，此外，我們也提供了一個界面讓使用者可以很清楚的選取欲比對的檔案或刪除檔案。

5.2.2 快速比對流程

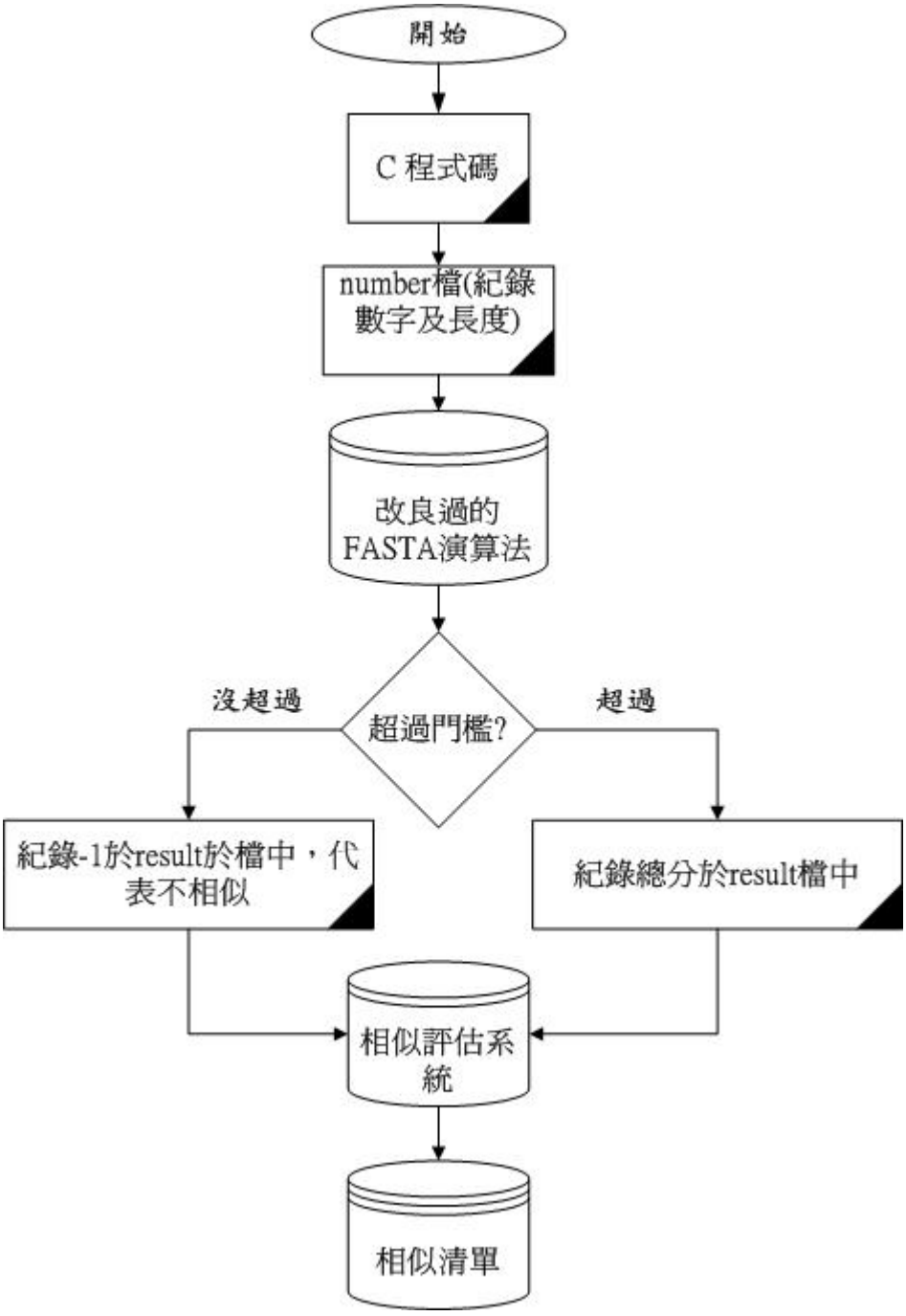
流程:

(1) 使用者可以利用此功能來進行 C 程式碼的快速比對，我們利用改良過的 FASTA 演算法來實作。首先，使用者必須選取欲比對的檔案，我們的系統允許多個 C 程式檔案一起作比較，然後系統會兩兩去作比對。在選取檔案錯誤時或選取相同檔案，系統都會產生訊息方塊來告知使用者。

(2) 程式一開始會將欲比對的 C 程式碼全部轉成 tokens ，並將轉換後的 tokens 全部轉為數字並儲存在一檔案中，我們稱之為 number 檔案，並且會紀錄數字 的數目(即 C 程式碼的長度)，利用 number 檔來形成一序列，我們稱為 C 程式碼的序列，系統利用此一序列來進行比對分析。

(3) 比對前我們利用序列的長度來快速過濾不可能相似的程式碼，之後為了尋找不重疊的相似程式片段，我們設定一門檻，來找出全部的程式片段。

快速比對功能全部的流程如圖:



圖表 三十一：快速比對流程圖

5.2.3 精準比對功能流程

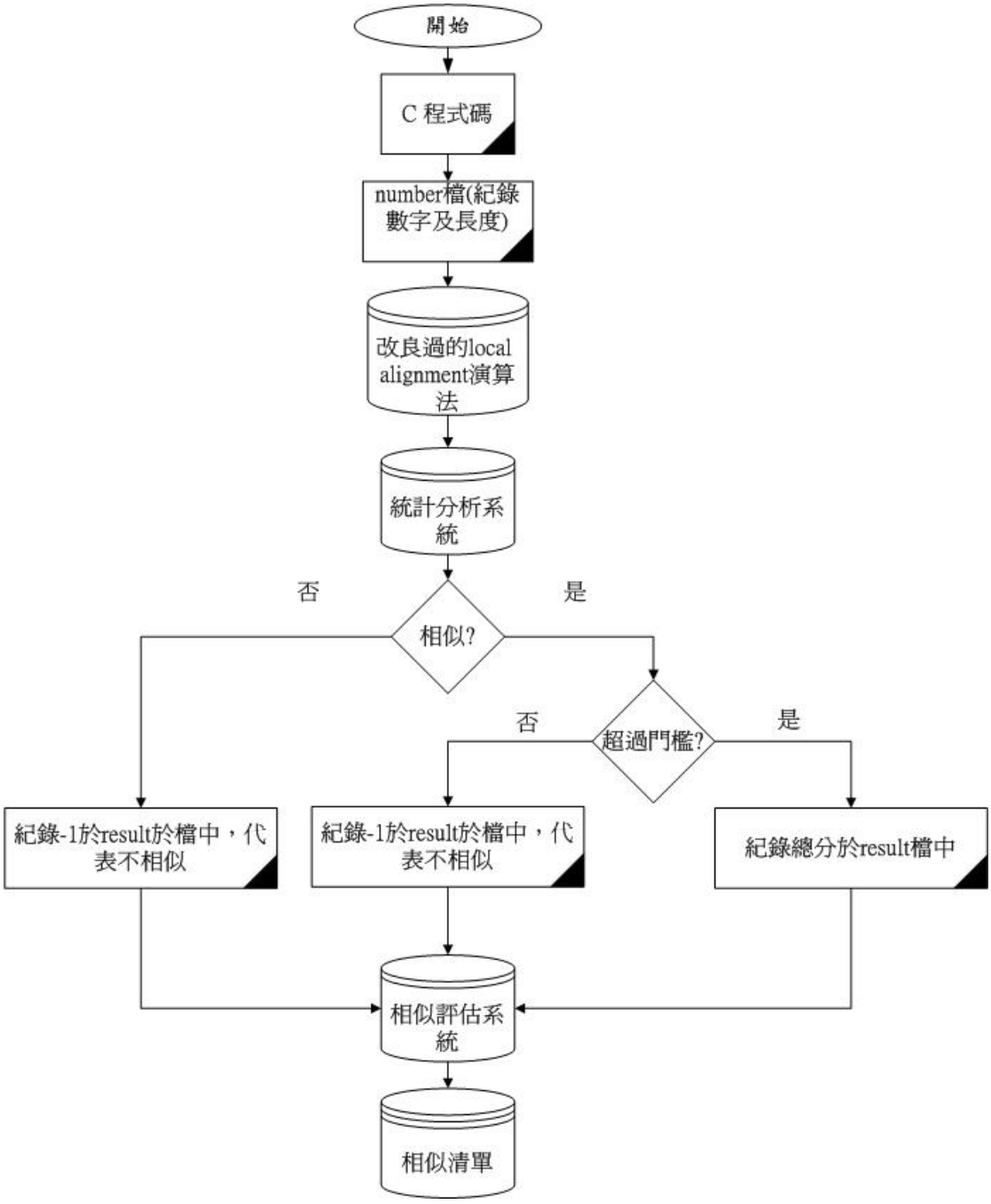
流程:

(1) 使用者可以利用此功能來進行 C 程式碼的精確比對，我們利用改良過的 local alignment 演算法來實作。首先，使用者必須選取欲比對的檔案，我們的系統允許多個 C 程式檔案一起作比較，然後系統會兩兩去作比對。在選取檔案錯誤時或選取相同檔案，系統都會產生訊息方塊來告知使用者。

(2) 程式一開始會將欲比對的 C 程式碼全部轉成 tokens ，並將轉換後的 tokens 全部轉為數字並儲存在一檔案中，我們稱之為 number 檔案，並且會紀錄數字 的數目(即 C 程式碼的長度)，利用 number 檔來形成一序列，我們稱為 C 程式碼的序列，系統利用此一序列來進行比對分析。

(3) 比對前我們也是利用序列的長度來快速過濾不可能相似的程式碼，但為了結省時間，與快速比對不同的是，我們利用**統記的方法**再過濾掉不可能相似的程式碼，之後再尋找不重疊的相似程式片段，我們設定一門檻，來找出全部的相似程式片段。

精準比對功能全部的流程:



圖表 三十二：精準比對流程圖

5.2.4 圖形顯示功能

我們提供了圖形顯示讓使用者知道兩兩相似的部位，使用者可以利用此功能了解相似部位在哪邊。例如使用者可以知道二個相似的函式，或者是二個相似的區段。

由於相似部分有多有少，因此我們只顯示超過 60%的部分，對於那些低於 60%的程式碼，在抄襲程式方面，我們認定它們並非抄襲，因此只顯示大過 60% 的程式碼。

5.3 系統功能比較

	快速比對	精確比對
時間	較快，採用index sort	較慢，採用dynamic programming
記憶體空間	較少	較多，但我們採用bit技巧，使其縮小為原來空間的 1/4
演算法	改良的FASTA演算法	改良local alignment演算法

表格 6: 系統功能優缺點比較

5.4 開發環境與使用的軟體

開發環境:

Windows XP 作業系統。

開發軟體:

Microsoft Visual C++ ， Borland C++ Builder。

開發硬體:

Intel P4 2.8G ， 512M RAM

第六章　系統效能

本章中，我們將測試系統效能，我們實驗的依據是運用比對程式來評估系統之效能。

並且我們利用二個觀念: Recall、Precision來評估系統的效能。

$$Recall = \frac{\text{系統偵測出來的結果中正確的數目}}{\text{實際上的結果}}$$

$$Precision = \frac{\text{測出來結果中正確有幾組}}{\text{系統測出來的結果}}$$

實驗共分三類:

一、比對學生作業:

我們將以三次的學生程式作業進行比對。

二、利用已知的結果來測試系統的效能:

我們利用手動方式將一程式作改變，並將其用來測試我們所實作到的功能，且我們將幾個不相似的程式混合，驗證是否可找出我們欲找的相似程式碼。

三、比對已知的相似的程式碼

我們將已知相似的程式碼先分類好，再與其它不相似的程式碼進行比對，驗證是否可找出之前已分類好的程式碼。

6.1 實驗一

我們測試學生所交的三次程式作業，實驗數據如下:

			精準比對				快速比對		
程式個數	平均大小 (bytes)		Time (s)	Recall	Precision		Time (s)	Recall	Precision
50	580		12	94.9%	100%		2	82.8%	100%
45	979		40	90.3%	96%		3	42.1%	94.2%
49	5495		1200	92.9%	98.3%		12	45.8%	93.2%

表格 7: 實驗一

精準比對		快速比對	
Recall	Precision	Recall	Precision
322/339	322/322	281/339	281/281
75/83	72/75	35/83	33/35
302/325	297/302	149/325	139/149

表格 8: Recall、Precision 的數據

6.2 實驗二

我們以手動方式修改一群些不相似的程式，測試我們所達到的功能，看是否可以找出我們欲找的相似程式碼，結果如下:

測試項目	是否能判斷	
	精準比對	快速比對
函式互換	是	是
變數替換	是	是
語法變換	是	是
資料型態的改變	是	是

表格 9: 實驗二

6.3 實驗三

共 12 組程式(每組有二個程式，其一為另一之變形)，我們利用這 24 個程式進行比對，評估是否達到預期的結果，結果如下:

測試項目	能否判斷	
	精準比對	快速比對
12組已知相似的程式	*是*	*找到8個*

表格 10: 實驗三

6.4 實驗結果討論

第一次實驗:

在此次實驗裡，我們測試了三次學生程式作業，在這三次比對裡，如意料之中，精確比對的精確度很高，但當檔案愈大時，所花的時間愈多，而快速比對所花的時間明顯的比精確比對來得少很多，因此當使用者想很快的知道比對結果但對結果的精確度要求不高時，則快速比對是不錯的選擇。

第二次實驗:

我們測試我們所實作的項目，方法是利用手動的方式產生二個相似程式，然後再與其它不相似的程式作比對。在實驗中，不論是精確比對與快速比對，皆能夠找到正確的結果。

第三次實驗:

我們利用 12 組(2 個程式為一組)相似的程式與不相似的程式混合作比對，我們發現，精確比對都能夠找到正確的結果，而快速比對則找到 8 組，但時間卻非常快。

由上列三個實驗可知，精確比對都能夠找到正確的結果，且其失誤率很低，而快速比對的精確度雖比精確來得低，但確非常快速。

第七章　結論與未來展望

7.1 結論

程式比對的方法大部份是利用Windows上的 WinDiff 或是Unix上的 Diff，但它們只依靠行的比對去分析程式碼的相似度，並沒有考慮相關的語法，況且每個人寫程式的習慣又不太一樣，因此在判定程式碼之間的相似度上面就會更加的困難。

在此，我們不僅對程式的語法加以考慮之外，也有考慮使用者可能的寫法(如函式互換、變數替換)，因此可大幅增加精確度。

此外，我們也利用統計的方法來過濾一些不可能相似的程式碼，不但能加速系統的比對速度，且能避免誤判的發生，且我們也提供快速比對與精準比對來供使用者來使用。

7.2 未來展望

由於比對C語言程式考慮的狀況很多，因此往後也有改善空間，我們把可以改善分為四點:

1. 將include檔一併處理:

由於使用者也許是使用標準函式庫或自已設計的include檔來設計程式，因此在比對之前，我們可以先比對是否是標準函式庫，若是，則不用考慮，若否，則將include一併考慮。

2. 更完美的評分機制:

也許將來可以定義一套評分機制來套用我們的程式比對,這樣也可以增加系統的精確性。

3. 用行去比對:

由於我們是採用tokens的方式去比對，為了使速度更快些，以後可以考慮將程式的每一行先轉成固定的語法(也即一行等於一個token)，之後再去作比對即可加速系統。

4. 更清楚方便的界面:

隨著使用者的需求，往後還可以再改善界面來達成系統的實用性。

5. 應用至其它語言:

往後我們可以利用此來比對其它程式語言，例如: C++, JAVA...。

參考書目

[1] H. Papadimitriou
Computational Complexity
Birkhäuser Verlag AG, 1994.

[2] Dan E. Krane, and Michael L. Raymer
Fundamental Concepts of Bioinformatics.
Benjamin Cummings, 2002.

[3] Thomas H. Cormen , Charles E. Leiserson, and Ronald L. Rivest
Introduction to Algorithms
MIT Press, 1998.

[4] M. S. Waterman
Introduction to Computational Biology
Chapman & Hall, 1995.

[5] A. Delcoigne, and P. Hansen
Sequence Comparison by Dynamic Programming
Biometrika, Vol. 62, 1975, 661-664.

[6] Yongqing Zhang
Sequence Alignment Methods, 2002.

[7] S. B. Needleman, and C. D. Wunsch
A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins
Journal of Molecular Biology, Vol. 48, 1970, 443-453.

[8] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. Lipman
A *Basic Local Alignment Search Tool*
Journal of Molecular Biology, Vol. 215, 1990, 403-410.

[9] F. F. Smith and M. S. Waterman
Identification of Common Molecular Subsequences
Journal of Molecular Biology, Vol. 147, 1981, 195-197.

[10] D. J. Lipman and W. R. Pearson
Rapid and Sensitive Protein Similarity Search
Science, Vol. 227, 1985, 1435-1441.

[11] W. J. Masek, and M. S. Paterson
A faster algorithm for computing string edit distances
Journal of Computer and System Sciences, Vol. 20, 1980, 18-31.

[12] D. E. Knuth, J. H. Morris, and V. R. Pratt
Fast Pattern Matching in Strings
SIAM Journal on Computing, Vol. 6,1977, 323-350.

[13] R. S. Boyer, and J. S. Moore
A Fast String Searching Algorithm
Communication of the ACM, Vol. 20, 1977, 762-772.

[14] E. Ukkonen
Finding Approximate Patterns in Strings
Journal of Algorithms, Vol. 6, 1985, 132-137.

[15] D. Gusfield
Algorithms on Stings, Trees, and Sequences
Computer Science and Computational Biology, 1997.

[16] G. J Barton
An Efficient Algorithm to Locate All Locally Optimal Alignments between Two Sequences Allowing For Gaps
Computer Applications in the Biosciences, Vol. 9, 1993, 729-734.

[17] Piotr Berman, Bhaskar DasGupta, and S. Muthukrishnan
Simple approximation algorithm for nonoverlapping local alignments
ACM-SIAM Symposium on Discrete Algorithms, 2002, 677-678.

[18] M. S. Waterman and M. Eggert
A New Algorithm for Best Subsequence Alignments with Application to tRNA-rRNA Comparisons
Journal of Molecular Biology, Vol. 197, 1987, 723-728.

[19] M. Dayhoff, R. M. Schwartz, and B. C. Orcutt
A Model of Evolutionary Change In Proteins
Atlas of Protein Sequence and Structure, Vol. 5, 1978, 345-352.

[20] S. Henikoff , and J. G. Henikoff
Amino Acid Substitution Matrices From Protein Blocks
Proc. Nat. Acad. Sci, Vol. 89, 1992, 10915-10919.

[21] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer1, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman
Gapped BLAST and PSI-BLAST: a new generation of protein database search programs
Nucleic Acids Research, Vol. 25, 1997, 3389–3402.

[22] Kun-Mao Chao
Computing All Suboptimal Alignments in Linear Space
Lecture Notes in Computer Science, Vol. 807, 1994, 31-42.

[23] O. Gotoh
An Improved Algorithm for Matching Biological Sequences
Journal of Molecular Biology, Vol. 162, 1982, 705-708.

[24] Stephen F. Altschul
Amino Acid Substitution Matrices From An Information Theoretic Perspective
Journal of Molecular Biology, Vol. 219, 1991, 555-565.