第四章 系統實作與結果分析

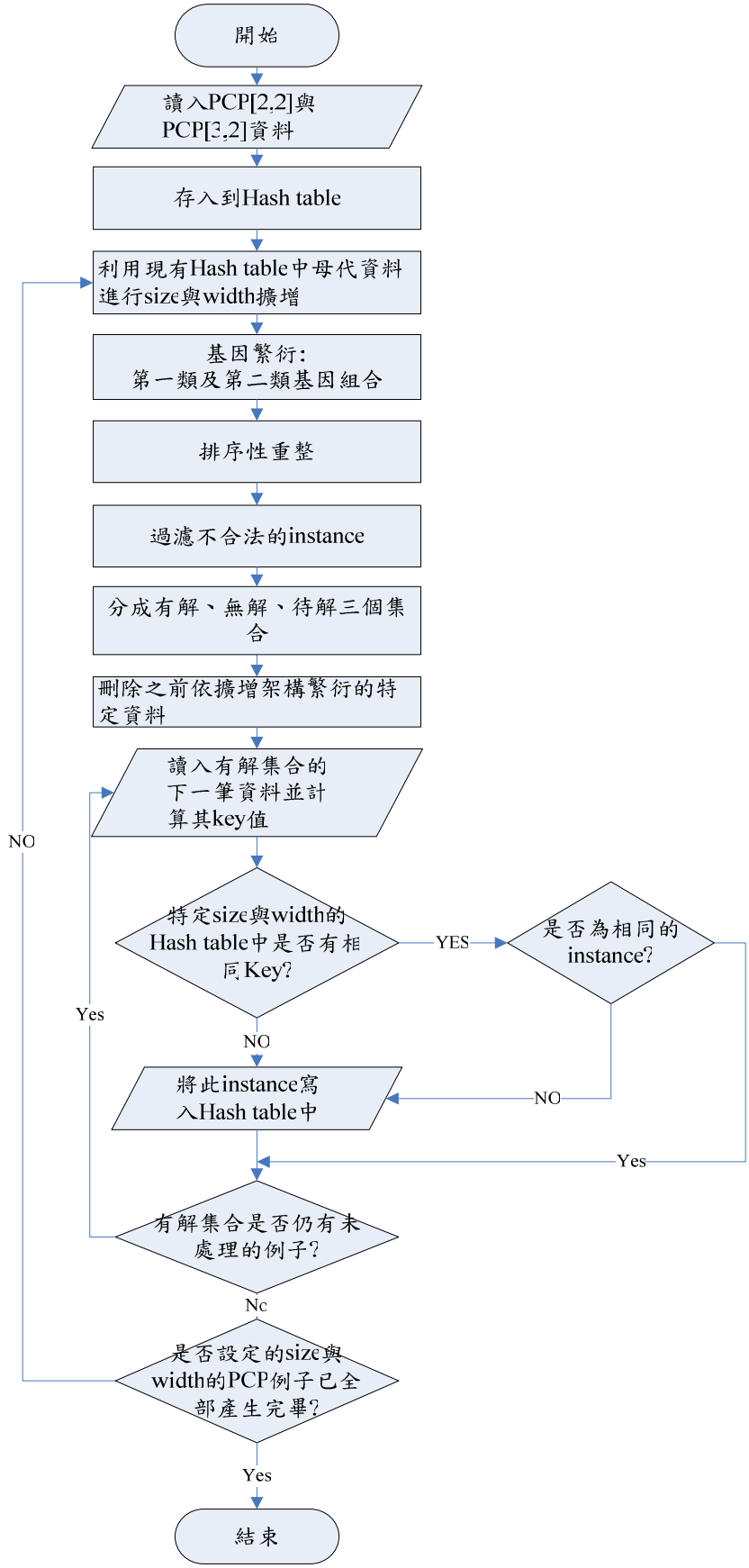
第一節 系統實作

為了實現「PCP 基因重整繁衍」演算法，並且落實重複性的篩檢，在本論文中實作一套名為「PCP 之困難例子產生器」的程式，利用原 Solver 中的 PCP[2,2]的三筆例子與 PCP[3,2]的 20 筆例子，繁衍到特定 size 與 width 且不重複的 PCP 集合。本研究實作環境之軟、硬體規格如下：

項次	規格名稱	規格描述
1	程式開發語言	Java SDK 1.5
2	作業系統	Windows Vista Premium 家用進階版
3	主機 CPU	Intel Core 2 Quad Q9300(2.5GHz)LGA775 處理器
4	記憶體 RAM	2G bytes
5	硬碟容量	320G Bytes

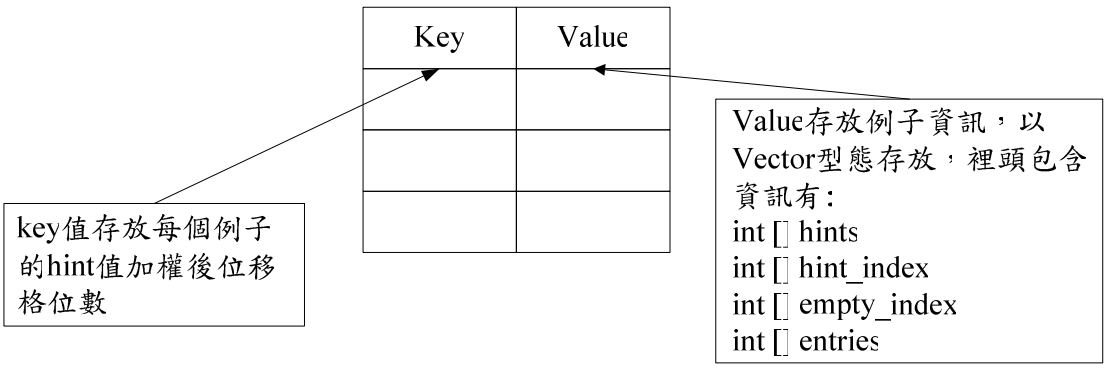
表一 系統規格

PCP 之困難例子產生器的程式，首先將讀入的母體資料做擴增架構符合特定 size 與 width，再去做基因重整繁衍出子代候選例子檔案，接著篩檢合法性，然後透過「Hard index solver」產出合法有解的 PCP instance 後，再次重新讀入到該特定 size 與 width 的資料結構中，如此透過相同的流程循序產生特定 size 與 width 的合法有解的 PCP instance，藉此產生一龐大的 PCP 資料庫，便是開發「PCP 之困難例子產生器」的最終目標。系統流程圖如圖十六：



圖十六：藉由 PCP[2,2]與 PCP[3,2]產生特定 PCP 集合之流程圖

程式一開始，首先讀入 Solver 中系統化法產生的 PCP[2,2]的 3 筆母代資料與 PCP[3,3]的 20 筆母代資料，將之存入到 Hash table 中。接著利用這兩個母代資料再以列為主(row-major ordering)循序繁衍出特定 size 與 width 的特定子集合的合法有解例子，直到繁衍到特定的 size 與 width 為止。母代的資料讀入到程式後，將被轉換為 Hash table 的資料結構存放，利用 Hash table 的特性，將每筆例子依 Key 值作為分類，相同 Key 值所對應到的所有例子都存放在 Value 欄位之中，協助後續演算法中快速的比對及運作。其中 Key 值存放的是每組例子根據其內含的提示數經過加權後與格位數位移所產生的數值；Value 欄位則是存放該例子的相關資料，如圖十七所示。由於繁衍例子的數量非常龐大，所計算出來的 Key 值可能會重複，所以在 Value 的型態上必須設定為 Vector，使得每次取得新例子時，相同的 Key 值會累增在相同的 Value 位置而不被覆蓋。Hash table 的實作則由 Java 語言的 Collection 結構中的 Map 類別完成。Map 的特性即「鍵-值」(Key-Value)匹配，故 Hash table 的維護則交由 Map 來維護。



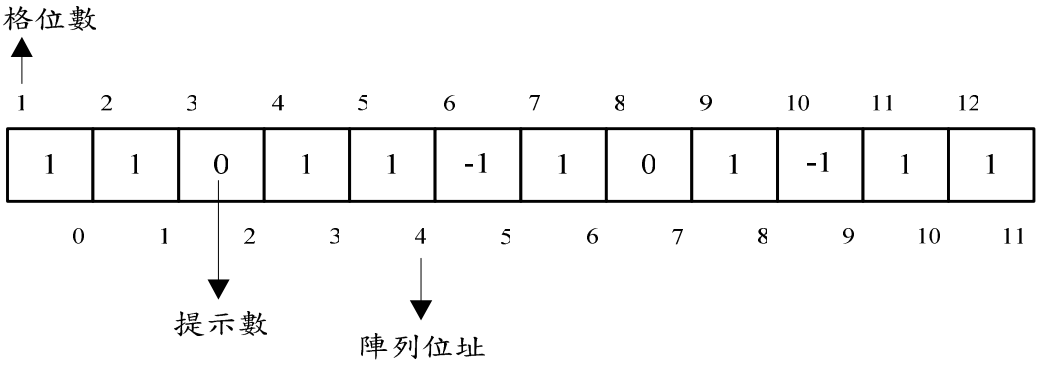
圖十七：Hash table

每個 PCP 例子都以整數陣列格式存放，在本論文中將存放完整例子的陣列命名為「entries」(原始例子陣列)，以圖十八的 PCP[3,2]例子為例：

1	1	0	1	1	
1	0	1		1	1

圖十八：PCP[3,2]

例子中有 size*width*2=3*2*2=12 個格位數，第一格位數中的提示數將會存放於陣列的第一個位置，如果格位內並無提示數，則提示數以“-1”取代，儲存陣列如圖十九：



圖十九：PCP[2,3]所存放之陣列格式

依照圖十七的儲存結構，將提示數 1、0、-1 先給予加權；1 的給定權值為 5，0 的給定權值為 3，-1 的給定權值為 1，當遇到提示數時，便將格位數往左位移加權後的數值加總：1<<5+2<<5+3<<3+…+10<<1+11<<5+12<<5 ＝$1*2^5+2*2^5+3*2^3+4*2^5+5*2^5+6*2^1+\ldots+10*2^1+11*2^5+12*2^5$ ＝ 1928，則 1928 就代表此 PCP[3,2]例子特有的編號，在本論文中將此編號設為 Key，提示數陣列則存在 Value 中的 Vector。

除了整個例子的陣列外，另一個整數陣列專門存放提示數的內容，在本論文

中稱之為「hint」(提示數陣列)，以圖十八的 PCP[3,2]例子為例，例子中共有 10 個提示數，則逐一將提示數存放到 hint 陣列中，第一個提示數為 1，第二個提示數為 1，第三個提示數為 0，以此類推，紀錄所有的提示數，陣列長度為 10，如圖二十。

1	1	0	1	1	1	0	1	1	1

圖二十：hints 陣列

而另一個整數陣列專門存放提示數的所在格位數，在本論文中稱之為「hint_index」（提示數格位陣列），舉例而言，依舊是以圖十八的 PCP[3,2]例子為例，第一個提示數(非-1 之提示數)的所在格位數為 1，所以 hint_index 陣列所存放的第一個值就是 1，第二個提示數的所在格位數為 2，所以 hint_index 陣列所存放的第二個值就是 2，以此類推，紀錄所有提示數的所在格位數如圖二十一。

1	2	3	4	5	7	8	9	11	12

圖二十一：hint_index 陣列

最後一個整數陣列存放的則是非提示數的所在格位數，也就是-1 所存放的格位數，在本論文中稱之為「empty_index」（空白數格位陣列），以圖十八的 PCP[3,2]例子為例，格位數 6 中所存放的正是非提示數(也就是-1)，所以 empty_index 陣列所存放的第一個值就是 6，格位數 10 中存放的亦是非提示數，所以 empty_index 陣列所存放的第二個值就是 10。紀錄所有非提示數的所在格位數，陣列長度為 2，如圖二十二所示。

6	10

圖二十二：empty_index 陣列

將 PCP[2,2]與 PCP[3,2]所有的例子讀入並且轉換成上述的資料結構之後，接著就要開始以列為主「Row-Major」（以列為主）執行「width 擴增架構」演算法，在第二章第二節中有提到，此演算法將新增擴增完的陣列，再把較小 size 與 width 的陣列資料儲放到新擴增完的陣列，再填補到新的陣列，擷取部份程式碼如下：

```
01  宣告 width 擴增完陣列的大小
02   For  每個比 width 小一的原始例子陣列
03      For(I =0 , J = 0;J<擴增陣列(size*width*2);J++)
04           一開始將擴增陣列每一元素皆設為空白(-1)
05           If(該格位數%width= =1)
06                        continue;
07           ELSE
08                擴增架構的元素內含值= 原始例子的元素內含值
09                      I++;
10           EndIF
11      EndFor
12   EndFor
13   擴增架構  第一個 width 填補為 1(將 pair 1 的空白基因補足為 1)
14   擴增架構最後一個空白基因也填補為 1
```

圖二十三：擴增架構之 width 擴增程式碼

圖二十三中的程式虛擬碼描述的是擴增架構中的 width 擴增，主要目的就是當 size 相同時，由比 width 小 1 的母代擴增到 pair 數相同的 width 中，再將其第一個 pair 的 upstring 與最後一個 pair 的 downstring 的空白基因補足為 1，並將新的組合存放到「新生例子陣列」中。由於只是將較小 width 的所有例子加入擴增

到新生盤面中，故在擴增完後會產生舊有 size 與 width 原有的例子個數，以本文第二章第二節中的圖八為例說明，圖八(b)為圖八(a)經由擴增演算法執行之後的結果。

擴增架構的 size 擴增架構則是當 width=2，size 不為 2 或 3 時，所必須執行的擴增演算法，在第二章第二節中也有提到。擷取部份程式碼如下：

```
01  宣告 width 擴增完陣列的大小
02  以 vector 型態存放 size 為 2,width=2 的所有例子，以 vector_first 來命名
03  以 vector 型態存放新 size 減 2,width=2 的所有例子，以 vector_second 來命名
04  FOR 每個 vector_first 的所有原始陣列中的數值
05    FOR 每個 vector_second 的所有原始陣列中的數值
06        將每個 vector_first 的原始陣列的前二分之一元素放入新擴增陣列中
07        將每個 vector_second 的原始陣列的前二分之一元素放入新擴增陣列中
08        將每個 vector_first 的原始陣列的後二分之一元素放入新擴增陣列中
09        將每個 vector_second 的原始陣列的後二分之一元素放入新擴增陣列中
10        將新擴增陣列輸入到相對應 hash table 中存放
11    ENDFOR
12  ENDFOR
```

圖二十四：擴增架構中的 size 擴增程式碼

藉由上面擴增架構的 size 與 width 擴增演算法，則可以使得 PCP 困難例子產生器可以只需讀入 PCP [2,2]與 PCP [3,2]的 23 筆資料例子後，便可以繁衍到特定的 size 與 width 的子集合。同樣再以本文第二章第二節中的圖九為例說明，圖九(c)為圖九(a)與圖九(b)經由擴增演算法執行之後的結果。

將母代資料讀入再藉由擴增架構轉到特定 size 與 width 之後，接著就要開始執行「基因重整繁衍」演算法，在第二章第三節中有提到，此演算法有兩類基因組合方式；第一類基因組合為：刪除一個基因，然後在其他空白基因中補一個基因，擷取部份程式碼如下：

```
01  FOR 每個提示基因格位陣列中的數值          //hint_index
02   FOR 每個空格基因格位陣列中的數值         //empty_index
03       FOR 數字 0 到 1 的每個數值             //2 candidates
04          FOR 總格位中的每個格位           //size*width*2 grids
05             IF 該格位數等同於空格基因格位陣列中的數值 THEN
06               填入“0 到 1 的每個數值”於新生例子陣列的該格位中;
07             ELSE IF 格位數等同於提示基因格位陣列中的數值 THEN
08               填入數值 “-1”(空白)於新生盤面陣列的該格位中；
09             ELSE
10               填入“原始盤面陣列該格位中的數值”於新生盤面陣列
11               的該格位中；
12          ENDIF
13         ENDFOR
14       ENDFOR
15    ENDFOR
16  ENDFOR
```

圖二十五：基因重整繁衍法之第一類基因組合程式碼

圖二十五中的程式虛擬碼描述的是基因重整繁衍法中的第一類基因組合，主要目的就是在提示基因中依序挑選出一個提示基因使之變成空白基因，然後再依序挑選一個空白基因並填入數字 0~1 使之成為提示基因。利用巢狀迴圈的方式，逐一搜尋每個提示基因格位陣列(hint_index)中的數值（即提示基因的格位數），以及每個空白基因格位陣列(empty_index) 中的數值（即空白基因的格位數），在每輪中掃描所有(size*width*2)格位，將提示基因一一替換為空白基因，另外空白基因也逐一由數字 0~1 取代成為新的提示基因，並將新的基因組合存放到「新生例子陣列」中。以本論文第二章第三節中的圖十一為例說明，圖十一(b)為圖十一(a)經由基因重整繁衍演算法執行第一類基因組合之後所會產生的其中一組候選例子。

第二類基因組合則是將提出來的提示數設定為非提示數的另外 1 個數字，也就是在原始被刪除的提示數位置中填入與原提示數不同的 1 個數字，簡單來說就

是將 0 轉成 1，將 1 轉成 0。擷取部份程式碼如下：

```
01  FOR 每個提示基因格位陣列中的數值          // hint_index
02    FOR 數字 0 到 1 的每個數值                //candidates
03          IF 數字 0 到 1 的數值與提示基因陣列中的數值不同 THEN
04              FOR 總格位數中的每個格位
05                填入“原始盤面陣列中該格位數的數值”於新生盤面陣列
06                的該格位中；
07              ENDFOR
08                填入“0 到 1 不同於提示基因的每個數值”於新生盤面陣
09                列的該格位中；
10        ENDIF
11    ENDFOR
12  ENDFOR
```

圖二十六：基因繁衍法之第二類基因組合程式碼

圖二十六描述的是基因重整繁衍法中的第二類基因組合，主要目的就是將原提示基因由其它數字取代成為新的提示基因。作法為挑選出一個提示基因，再從數字 0~1 中剔除與原提示基因相同之數字，將其餘 1 個可能數字填入此提示基因之位置，產生的新組合存放於新生盤面陣列之中，以本文第二章第三節中的圖十二為例說明，圖十二(b)為圖十二(a)經由基因重整繁衍演算法執行第二類基因組合之後所會產生的其中一組候選例子。

接下來便是「合法性」之篩檢了，由於基因重整的方式，有可能會產生不符合 size 與 width 的例子，所以必須特別做合法性的篩檢動作。而要作合法性篩檢前須先將新生例子陣列依照 size 與 width 的規格重新排序。在本論文第三章第一節中提到「合法性的排序重整」，也就是將每個繁衍完的新生例子以 pair 主分類處，上下 string 中的陣列重新排序，將空白基因排到每個 string 的最後一個元素。擷取部份程式碼如下：

```
01  FOR (int k = 0; k <  傳入陣列的大小; k += width) {
02          boolean flag = true;
03          FOR (int i = k; i < (k + width) && flag; i++) {
04              flag = false;
05              FOR (int j = k; j < (k + width) - 1; j++) {
06                  IF(number[j] ==  空白) {
07                      swap(number, (j + 1), j);
08                      flag = true;
09                  ENDIF
10              ENDFOR
11          ENDFOR
12  ENDFOR
```

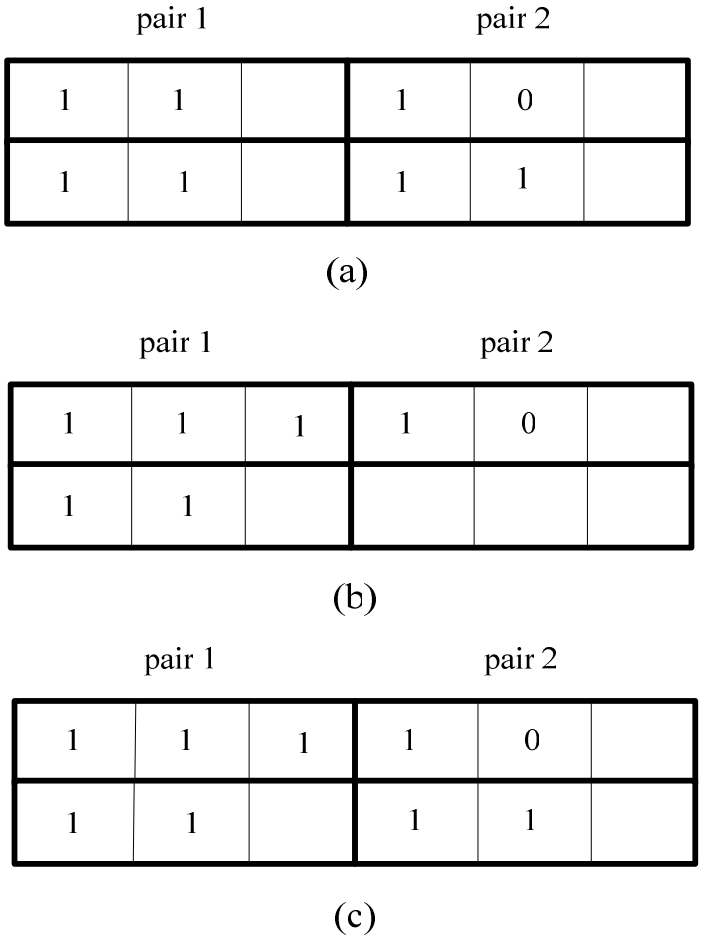
圖二十七：排序性重整

我們以本文第三章第一節中的圖十四為例說明，圖十四(b)為圖十四(a)經由排序性重整所排序後的結果。而重整後便可接著做合法性判斷：第一個 pair 的 string 是否滿足長度為 width，與其它 pair 的上下 string 是否滿足空白基因個數介於 0 到 width-1 的規定。擷取部份程式碼如下：

```
01  boolean addflag = true;              //判斷合法性 flag = true
02  int   judgefirst = 0;                //第一個 pair 的 upstring
03  int   judgeothers = 0;               //其它 pair 的 upstring 與 downstring
04  runloop:
05      FOR(int k = 0; k < number.length; k += width) //針對每個新生例子陣列
06          FOR (int j = 0; j < width; j++) {
07            IF (在 width 個數前，若有空白) 則將 judgefirst 累加
08              IF (當 judgefirst 的個數大於零時) 跳出整個迴圈
09            ENDIF
10            IF(開始紀錄其他 string 的空白個數)
11              IF(若 string 的空白個數= =width 個數)  judgeothers 累加
12            EndIF
13          ENDFOR
14          IF(判斷 judgeother 個數是否等於 width 長度)
15            若是則跳出整個迴圈
16          judgeother 設定為 0;
17       ENDFOR
18       IF(假如 judgefirst>0 或是 judgeother= =width)
19          addflag = false;
```

圖二十八：合法性判斷

合法性判斷藉由重新排序完新生例子陣列後，再以 pair 為分隔線，上下 string 為單位，判斷每一個 string 中的空白基因個數是否介於 0 到 width-1，當第一個 pair 的 upstring 需符合空白基因個數為 0，若不符合則就不須檢核其他 string 是否合法了。同理，當找到一組 string 不合法時則不需再檢核該新生例子陣列。底下以圖二十九來說明之，其中圖二十九(a)中，其第一個 pair 的 upstring 的空白基因個數為 1，所以(a)在合法性判斷的第一組 pair 的 upstring 便被篩檢出不合法了，故(a)中的例子後續其它組 pair 的 string 便不需再檢核。圖二十九(b)中，在最後一組 pair 的 downstring 的空白基因個數不符合介於 0 到 width-1 之間，故也被篩檢出不合法。只有(c)才能被判定為合法的子代候選例子。



圖二十九：(a)與(b)皆為不合法例子，(c)為合法例子

當經過合法性篩選後,新生例子陣列就會存放到另一個以Map實作Hash table的資料結構中存放，其中的 Key 值計算方式也是和之前存到特定 size 與 width 的 Hash table 一樣，當相同的 Key 值會累增在相同的 Value 位置而不被覆蓋。等繁衍完某一特定 size 與 width 的 PCP instance 之後，便將所有的新生例子輸出至文字檔，再利用 Hard index solver 作為「解題」之篩檢器。

Hard index solver 是改寫 Solver 程式而來，主要功能在於可針對已知文字檔，去計算或判斷該文字檔中每一例子，並將解題結果分為：無解、有解、或待解（在搜尋到 320 層後還找不到答案時）三種情況。但因為本論文所繁衍的例子有可能是一檢查就知道是無解的，所以先改寫為針對基因繁衍的例子先過濾掉不可能有

解的例子，即先使用 Solver 中的過濾函數。由於 Solver 程式是以 C++語言開發，而本論文所發表的「PCP 困難例子產生器」則是以 Java 語言開發[11,13,14]，為了要整合兩種不同語言程式，在開發過程中，本論文使用 Java 所提供的 Runtime.getRuntime().exec()函式，於 Hard index solver 最終產生的執行檔（包在 BAT 檔案內）中，順利將兩者不同語言之程式結合。透過 Hard index solver 的篩檢，會將產生的候選例子先經過過濾，之後才去判定與解題。

第二節 實驗分析

本論文針對 Ling Zhao 所收集的兩百個困難例子所在的五個 PCP 子集合—依序為 PCP[3,3]、PCP[3,4]、PCP[3,5]、PCP[4,3]、PCP[4,4]做為實驗對象，Ling Zhao 在這些集合的收集來源為(1) Richard J. Lorentz[5]；(2)在 PCP AT Home 的網站[8]；與(3)使用 Solver 系統化法與亂數產生法產生，而 Solver 除了在 PCP[3,5]、PCP[4,4]是由亂數法產生之外，其餘皆是透過系統化的方式產生。故本論文除了重做 Solver 其系統化產生方法(節點數量沒有設門檻值與有設門檻值)外，再加入本論文所提及的基因重整繁衍法(節點數量有設門檻值)作一比對。實驗一為實作前人 Solver 系統化法與我們的基因重整繁衍法，其 PCP[3,3]的實驗所產生的 PCP 例子結果如下：

	Solver 系統化法	Solver 系統化法	基因重整繁衍法
門檻值設定	無	一百萬個節點	一百萬個節點
產生子代數量	127303	127303	780(母體為 PCP[3,2]，有 20 個)
過濾後數量	2002	2002	114
有解數量	1958	1958	104
無解數量	1	1	0
待解數量	43	43	10
解題時間（秒）	23.399	23.46	46.783
執行總拜訪節點數	1019964	1019964	98087972

表二 PCP[3,3]Solver 系統化法與基因重整繁衍法比較

設定門檻值的理由在於：實驗 PCP[3,3]基因重整繁衍法過濾後的子代再經由 Solver 解題時，發生所耗費的解題時間超過於兩倍於系統化法解題耗費的時間，卻仍然跑不出結果的情形，為了考量實驗的時間成本，故將系統化法產生的子代與基因繁衍的子代於解題時多設了一個門檻值。設定門檻值的方式為：修改 Solver 在解題時，在拜訪前 20 層後，多加一個拜訪總節點數是否超一百萬個的判斷，若有則先宣判此例子為待解，若無則維持之前 iterative DFS 方法，再加深 20 層重頭拜訪繼續解題（意即重新拜訪至前 40 層），如此每次加深 20 層，直到拜訪總節點數超過一百萬或 Solver 所設定的 320 層為止。而待解部分將等未來有較充裕的時間時，再以沒有設定門檻值限制的方式進行解題。

由表二可看到 Solver 系統化法(沒設門檻值)與 Solver 系統化法(有設門檻值)

的差異並不大，其有解、無解、待解的個數，與總拜訪節點數皆一樣，唯一的差異在於解題時間。我們在此分批跑了數次後，歸納出設有門檻值的解題時間會較慢一點，且差異多在一分鐘以內，探究其原因為設有門檻值相對於沒設門檻值的部份多了一行判斷所致，故我們判定兩者解題時間的差異在可容忍的範圍以內，意即有無設定門檻值，就 Solver 系統化法而言，對於困難度的判定是毫無影響的。

在設定門檻值後，PCP[3,3]基因重整繁衍法部份的解題結果為：由 PCP[3,2]循序所產生的 20 個母體，先經由 PCP 基因重整繁衍法繁衍出 780 個子代，再經過濾後的子代數目為 114 個，其中 104 個有解、0 個無解、與 10 個待解。

基於基因重整繁衍法的平均解題時間與平均拜訪總節點數，皆較傳統 Solver 系統化法為高的理由，我們可以大膽的假設基因繁衍的待解例子中，其中應該存在有比系統化法產生的待解例子較為困難的例子。底下表三是 PCP[3,4]的實驗結果：

	Solver 系統化法	Solver 系統化法	基因重整繁衍法
門檻值設定	無	一百萬個節點	一百萬個節點
產生子代數量	13603334	13603334	3315 (母 體 為 PCP[3,3]，有 104 個)
過濾後數量	65846	65846	493
有解數量	61122	61122	355
無解數量	1309	1309	5
待解數量	3415	3415	133
解題時間（秒）	12693.509	12727	1180.057
執行總拜訪節點數	107897406	107897406	2461753342

表三 PCP[3,4]Solver 系統化法與基因重整繁衍法比較

同樣的在表三可看到 Solver 系統化法(沒設門檻值)與 Solver 系統化法(有設門檻值)的差異並不大，其在有解、無解、待解的個數是一樣的，而在總拜訪的節點數也是一樣的，唯一的差異是在其解題時間，我們在此分批跑了數次，這兩者解題時間互有快慢，但真正的差異其實只有在一分鐘以內，故將其原因歸屬於可以容忍的範圍之內。理由也是和上面 PCP[3,3]一樣，且因為其所解過濾後的子代相較於 PCP[3,3]來的多，故在解題時間會有較大的一些差異。而在 PCP[3,4]基因繁衍部份是由 PCP[3,3]基因繁衍有解的 104 個母體再經由 PCP 基因重整繁衍算法所繁衍，繁衍的子代數目為 3315 個，經由過濾後的數目為 493 個，而其中有 355 個有解、5 個無解、與 133 個待解。

基於基因重整繁衍法的平均解題時間與平均拜訪總節點數，皆較傳統 Solver 系統化法為高的理由，我們可以大膽的假設基因繁衍的待解例子中，其中應該存

在有比系統化法產生的待解例子較為困難的例子。底下表四是 PCP[4,3]實驗結果：

	Solver 系統化法	Solver 系統化法	基因重整繁衍法
門檻值設定	無	一百萬個節點	一百萬個節點
產生子代數量	5587598	5587598	3740(母體為 PCP[4,2]，有 75 個)
過濾後數量	123314	123314	1233
有解數量	106264	106264	1155
無解數量	1502	1502	0
待解數量	15548	15548	78
解題時間（秒）	36811.457	37159.703	9568.586
執行總拜訪節點數	65239371721	65239371721	3357400837

表四 PCP[4,3]Solver 系統化法與基因重整繁衍比較

PCP[4,3]是由 PCP[4,2]所繁衍，而 PCP[4,2]則是由 PCP[2,2]經由擴增架構中的 size 擴增演算法所繁衍。PCP[4,3]的實驗結果也和上面的系統化法測試一樣，在 Solver 系統化法部分無論有無設定門檻值，皆不會影響其有解、無解與待解的例子個數，只有在解題時間有所差異，其原因在於若過濾後的子代例子個數越多，則解題時間的差異越大。而在 Solver 系統化法上所找的三個集合部份，在過濾完的數量都和 Ling Zhao 所做的實驗一致，但在有解數量、無解數量、待解數量上卻有些微不同，我們後來重做其待解部份例子，發現可以找出其所收集到的困難例子。這部份我們和他實驗內容的差異猜測是 Ling Zhao 於 2002 年發表文章所使用的為第一版 Solver，而在 2003 年四月發展成第二版，而我們所使用的是

2003 年十一月所公開下載的 Solver 第三版本，也許是因為版本的問題，但詳細情況我們並不知道原因出在哪裡。

在 PCP[3,5]與 PCP[4,4]兩個子集合部分，本論文於重新實作時，依據 Solver 所提供的功能—即輸入 size、width、希望輸出 instance 的解答長度、與亂數的重覆次數等四個參數，我們在「希望輸出 instance 的解答長度」參數設定為 1，在「亂數的重複次數」參數則分別設定 11034 與 25087（即基因重整繁衍所產生的子代數量），解題結果則放入其 hard.txt 中（原本 Solver 的動態產生功能上並無顯示其過濾後節點，與一些拜訪的總節點數，只有一個輸出檔為 hard.txt）。

基於亂數產生的原因，在實際測試上，若只執行一次則過於草率，故我們針對同一集合皆實驗五次，以期能有較完整的測試結果，而在使用基因重整繁衍法時，則是由 PCP[3,4]基因重整繁衍法的 355 個有解個數所繁衍。

底下表五是剩下的 PCP[3,5]與 PCP[4,4]分別由動態產生法與基因重整繁衍法的比較：

	動態產生法	基因重整繁衍法
門檻值設定	無	一百萬個節點
產生子代數量	11034	11034(母 體 為 PCP[3,4]，有 355 個)
過濾後數量	無資料顯示	1643
有解數量	約 387 個(五次平均)	1010
無解數量	無資料顯示	65
待解數量	無資料顯示	568
解題時間(秒)	8.9118	7526.1

表五 PCP[3,5]之動態產生法與基因重整繁衍法

底下表六則是 PCP[4,4]的實驗結果：

	動態產生法	基因重整繁衍法
門檻值設定	無	一百萬個節點
產生子代數量	25087	25087(母 體 為 PCP[4,3]，有 1155 個)
過濾後數量	無資料顯示	9622
有解數量	約 2225 個(五次平均)	8089
無解數量	無資料顯示	71
待解數量	無資料顯示	1502
解題時間(秒)	128.6734	41908.925

表六 PCP[4,4]之動態產生法與基因重整繁衍法

由於 Solver 系統化法在有無設定門檻值的待解個數上並無差異，我們在後續追蹤下發現每一個例子先必須得每拜訪 20 層後才去判斷是否節點數超過門檻值，若有超出門檻值則先宣告為待解例子，若無則繼續累加到 40 層，同樣的若有超出門檻值也宣告為待解例子，若無則繼續拜訪到 60 層，以此類推一直到所設定的 320 層後，若有超出門檻值或是沒有超出門檻值，此時 Solver 程式都將先宣告為待解例子。而由 Solver 系統化法所產生的待解例子全部都是達到 320 層後又有些被檢測出超過門檻值，故在以上實驗一中的 Solver 系統化法無論有無設定門檻值的總拜訪節點數，與待解個數都完全相符。基於上訴的理由，在以下實驗二中將只探討基因重整繁衍法部份的待解例子。

實驗二則是針對上面實驗一的基因重整繁衍法待解例子部份，看看是否待解例子有多少比例超過門檻值，我們針對待解部份的文件撰寫一小程式，將待解例子的編號、size、width、拜訪節點數與例子題目等等資訊寫入到 Microsoft Access 資料庫當中，以利後續分析。

待解例子個數	超出門檻值的例子個數
10	3

表七 PCP[3,3]基因重整繁衍法待解例子超出門檻值圖表

由表七可看出在 PCP[3,3]中，10 個當中有 3 個超出門檻值。針對基因重整繁衍的過濾例子解題時，在一開始若沒有設定門檻值，則其會使得解題時間變的較難控制。底下則是 PCP[3,4]的待解例子超出門檻值的個數：

待解例子個數	超出門檻值的例子個數
133	75

表八 PCP[3,4]基因重整繁衍法待解例子超出門檻值圖表

而在 PCP[3,4]當中，133 個例子當中就有 75 個例子有可能還沒到達 320 層就被宣告為待解例子。底下為 PCP[3,5]待解例子圖表：

待解例子個數	超出門檻值的例子個數
568	372

表九 PCP[3,5]基因重整繁衍法待解例子超出門檻值圖表

而在 PCP[3,5]當中，568 個例子當中就有 372 個例子有可能還沒到達 320 層就被宣告為待解例子。底下為 PCP[4,3]待解例子圖表：

待解例子個數	超出門檻值的例子個數
78	55

表十 PCP[4,3]基因重整繁衍法待解例子超出門檻值圖表

而在 PCP[4,3]當中，78 個例子當中就有 55 個例子有可能還沒到達 320 層就被宣告為待解例子。底下為 PCP[4,4]待解例子圖表：

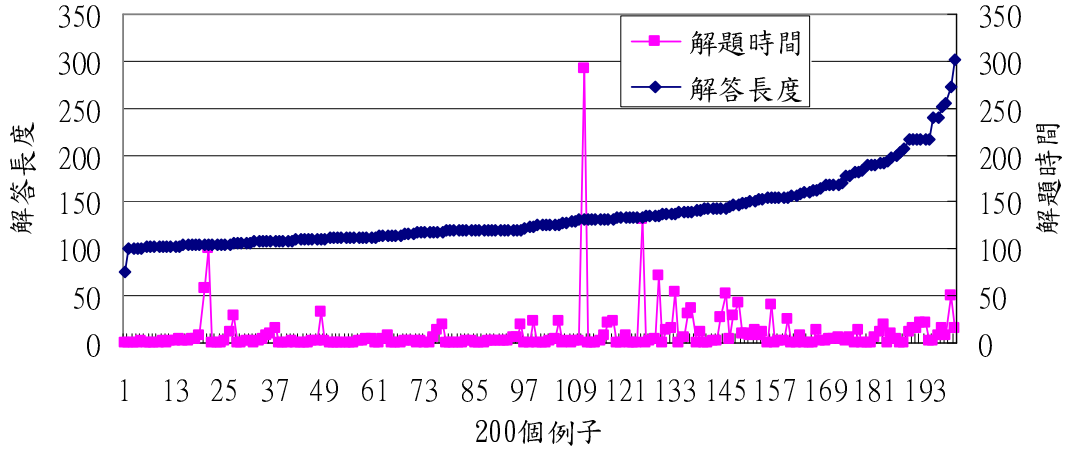
待解例子個數	超出門檻值的例子個數
1502	1065

表十一 PCP[4,4]基因重整繁衍法待解例子超出門檻值圖表

而在 PCP[4,4]當中，1502 個例子當中就有 1065 個例子有可能還沒到達 320 層就被宣告為待解例子。

從上面表七到表十一這五個集合當中可以發現，除了表七的 PCP[3,3]的待解超出門檻值的例子佔待解例子總數不到一半之外，其他集合中佔的比率都超過一半以上，說明了我們在待解部分的例子超出門檻值的例子相對於 Solver 系統化法在待解部份例子較為困難，且隨著集合數越大的情況之下，透過基因重整繁衍所產生出的例子有越來越難的趨勢。

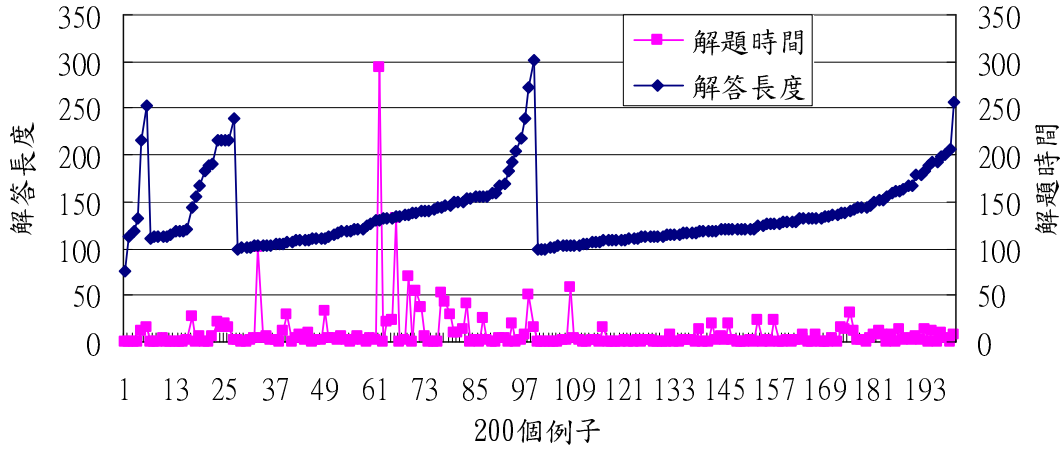
而在實驗三的部份我們將針對 Ling Zhao 收集的 200 個困難例子中，其解答長度與解題時間做一探討，我們先依解答長度由小到大排序如圖三十所示，可以發現在 Ling Zhao 以長度做為困難度收集的兩百個例子中，其解答長度(圖三十藍色線圖)與解題時間(圖三十粉紅色線圖)並無一定的關連，並沒有因解答長度越長相對的解題時間變的較久，表示以解答長度做為困難度指標並不合理。



圖三十：Zhao 收集 200 個困難例子中解答長度與解題時間對照圖

而在圖三十一則是針對 Zhao 收集的 200 個困難例子首先以集合名稱排序，再以解答長度由小到大排序，同樣也可以看出解題時間並沒有隨著解答長度增加

而在解題時間有較久的正相關趨勢。



圖三十一：Zhao 收集 200 個困難例子中解答長度與解題時間對照圖

而在實驗四的部份將會針對基因重整繁衍法所繁衍的母體中依我們所定義的困難度指標，挑選 200 個例子相對應於 Ling Zhao 所收集的 200 個困難例子做一實驗。分別在五個集合所挑選的個數如下：

集合名稱	Zhao 於該集合挑選例子個數	我們所挑選例子個數
PCP[3,3]	1	1
PCP[3,4]	5	5
PCP[3,5]	21	21
PCP[4,3]	72	72
PCP[4,4]	101	101
總和	200	200

表十二 200 個困難集合例子分布特定集合表

我們寫了一個小程式，將每一集合中的 sol.txt 中的例子標題、size 個數、width 個數、例子題目、有解長度、有解個數、解題時間，與我們所定義的困難度讀入到 Microsoft Access 資料庫中，並按照困難度大小排序寫到另一文字檔中，以利

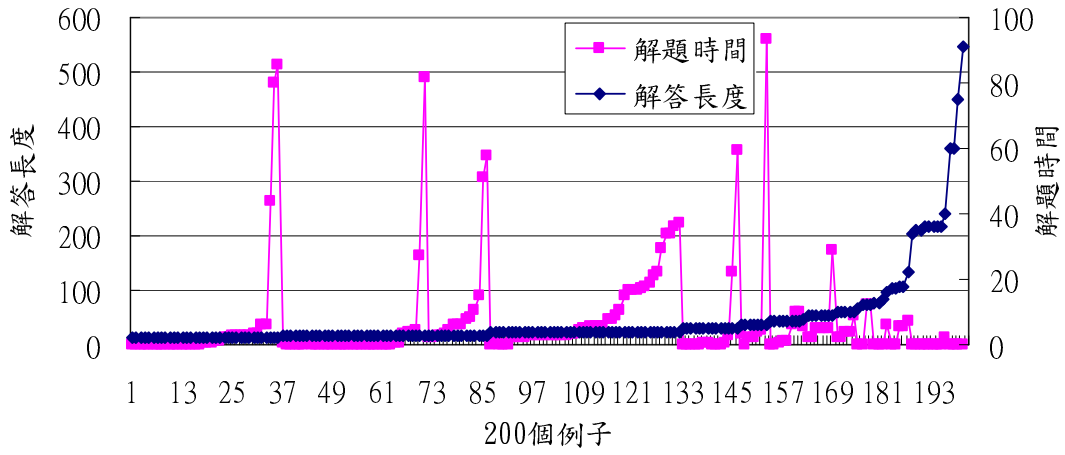
後續研究者挑選困難例子，藉此可以快速挑選與 Ling Zhao 相對應的 200 個困難例子。

接著針對(1)總個數、(2)平均解答長度、(3)平均解答個數(4)平均困難度指標(5)平均執行時間等項目，對我們所挑選的困難例子與 Ling Zhao 相對應的 200 個困難例子作一探討。

	Ling Zhao 的 200 個困難例子	我們的 200 個困難例子
總個數	200	200
平均解答長度	135.965(=27293/200)	7.5(=1500/200)
平均解答個數	1.13(=226/200)	1.09(=218/200)
平均困難度指標	8.4418(=1688.364/200)	39.1595(=7831.8921/200)
平均解題時間(秒)	9.0775(=1815.493/200)	41.1881(=8237.62/200)

表十三 Zhao 的 200 個困難例子與我們所挑選的 200 個困難例子比較表

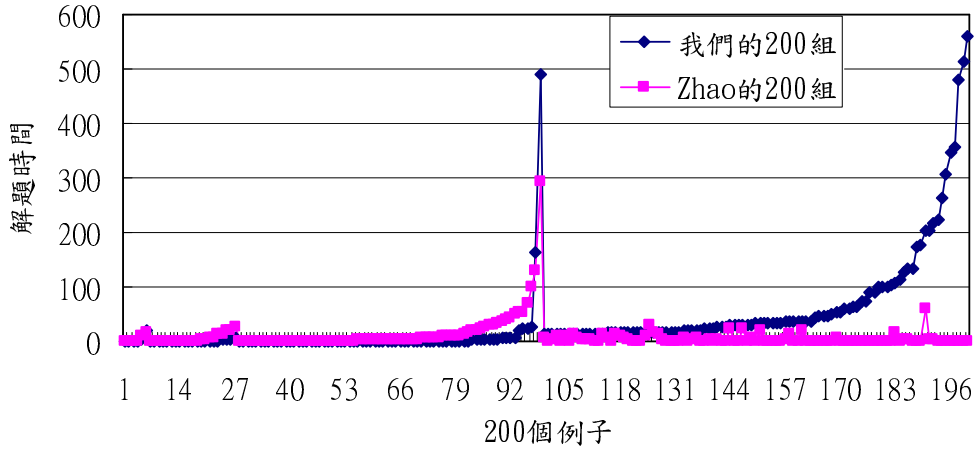
可以看出我們在平均解答長度並沒有超過 Ling Zhao 收集的例子解答長度，但在解題時間上都相較於 Ling Zhao 的解題時間久，而我們以解答長度由小到大排序從基因重整繁衍法所收集的 200 個困難例子如圖三十二，可以清楚看出解答長度與解題時間並沒有因為長度越長而於解題時間就越久的一種正相關聯情形發生，所以可以更加確認的是，Ling Zhao 以解答長度當作困難度指標的依據並不客觀。



圖三十二：我們收集 200 個困難例子中解答長度與解題時間對照圖

接著由圖三十三(200 個例子先以集合名稱排序，再以解題時間由小到大排序)可以看出在 size 與 width 較小的集合中，Ling Zhao 依解答長度做為困難度所挑選的例子小小超越我們所挑選的例子，說明了解答長度做為困難度在較小集合中所挑選的例子具有一定的困難度，但隨著 size 與 width 較大的集合中，並沒有因此具有一定的困難度，顯示以解答長度當作困難度較適合在較小 size 與 width 中做為困難度指標，我們所挑選的兩百組在大部份集合下的解題時間都超過 Ling Zhao 所挑選的兩百個例子，也說明了我們基因重整繁衍法所產生的例子有一定的困難度。

在附錄 A 中我們也列出使用我們的方法所產生的 20 個例子。



圖三十三：Zhao 所挑選 200 個困難例子與我們所挑選的 200 個困難例子