國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

並行 Java 程式測試環境中並行路徑選取方法之研究

A Study of Concurrent Path Selection Problem in Concurrent Java Program

研 究 生：孫維孝

指導教授：鍾乾癸

中 華 民 國 九 十 四 年 六 月

並行 Java 程式測試環境中並行路徑選取方法之研究

A Study of Concurrent Path Selection Problem in Concurrent Java Program

研 究 生：孫維孝　　Student : Wei-Shiau Suen

指導教授：鍾乾癸　　Advisor : Chyan-Goei Chung

國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

A Thesis

Submitted to Department of Computer Science and Information Engineering College of Electrical Engineering and Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science and Information Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中 華 民 國　九 十 四　年　六　月

並行 Java 程式測試環境中並行路徑選取方法之研究

研究生：孫維孝　　指導教授：鍾乾癸

國立交通大學資訊工程學系研究所

摘要

執行一並行 Java 程式，若所執行之 main()函式會啟動一或多個主動物件之 run()函式，則這些 run()函式將與 main()並行執行，所有可能並行執行的路徑組合均需驗證其正確性，方可保證這條 main()函式路徑的正確性，如何選擇最少並行路徑組合予以測試是一個重要課題，唯目前尚無系統化方法被提出，本研究之目的在提出一系統化方法以解決此問題。

欲以最少數目的並行路徑來測試上述執行行為，需保證每一 run()之路徑及每一由 shared object 所產生之不同 run()的路徑互動行為皆被驗證。本研究提出以直接資料流來代表二個不同 run()之路徑呼叫同一 shared object/class 函式之互動行為，由於一條路徑可能呼叫多個不同 shared object/class 的函式，因此二直接資料流將產生互動影響而構成間接資料流。運用資料流關係可以找出所有有互動關係之路徑組合，這些路徑組合皆需被測試；這些路徑組合與所有未呼叫 shared object 的函式之路徑均需被測試。

尋找最少組合之並行路徑來驗證所有路徑組合與所有未呼叫 shared object 的函式路徑之問題竟然與 graph coloring 之問題相等，無法在 polynomial time 內找出最佳解，本研究採用 gaming tree 之 look-ahead 策略來尋找並行路徑，其計算複雜度為 $O(N^4)$。

經用多個範例驗證，本研究所提方法所找出並行路徑數目確為最佳解，且可在短時間內找出，證實本研究所提方法實用性與有效性。

A Study of Concurrent Path Selection Problem in Concurrent Java Program

Student : Wei-Shiau Suen Advisor : Chyan-Goei Chung

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University

Abstract

In concurrent Java program, the main() method may create one or more threads to execute the run() methods of active objects. Thus, the run() methods would be executed parallel with the main() method. There are many different combinations of paths that may be executed parallel, and a combination is named a concurrent path. In order to ensure the correctness of the main() method, it is necessary to verify the correctness of all different concurrent paths. In concurrent program testing, it is a very important issue how we select the fewest concurrent paths to test. Unfortunately, there is not a systematic measure to solve this problem. So this research is about to propose one systematic measure to solve this problem.

If we want to test all execution behaviors of a concurrent program with fewest concurrent paths, we need to ensure every run() path and every inter-thread interaction could be verified during testing. In this research, we propose that there might exist data-flows between two paths belonging to different run() methods when they call the same shared object's functions. We also notice that a path may call different shared objects' functions. Thus one directly data-flow may affect another. This is so-called indirectly data-flow. We can find all combinations of paths which have interactions by using data-flows, and all combinations that have been found should to be tested. These combinations and other paths without shared objects should to be tested.

The problem of finding the fewest path combinations is equivalent to the graph

coloring problem. It is impossible to find the optimal solution in polynomial time. Thus, this research adopts the look-ahead policy of the Gaming Tree mechanism to find the concurrent paths, and it has time complexity of $O(N^4)$.

Via various examples, it shows that the proposed mechanism in this research can find the optimal solution for this problem quickly and ensures the practicality and validity.

誌謝

首先我要感謝指導教授鍾乾癸老師，感謝兩年多以來不厭其煩地策勵與鞭策，並在百忙之中來回奔波，悉心指導我做研究應有的態度與方法。讓我能順利完成碩士班的學業。老師以身作則的做事態度更讓我了解到對任何事物都應抱持著認真、用心的態度盡力去完成。

其次要感謝靜紋學姐，在我研究與寫作論文的期間，犧牲自己的時間來給予我寶貴的指導。並且要感謝實驗室的同袍昆灝、嘉鑫、志忠及志宇，在研究上與學業上與我彼此勉勵與扶持。

除此之外，還要感謝所有大學的好同學們，健文、旻錚、明曄、明福、宗杰、俊廷、俊逸、賢麟、和晉、還有其他許多的好朋友，在我在歷經人生最低潮的時候，是你們的存在，讓我沒有放棄自己。謝謝你們，我很感謝有你們這一大群一輩子的好朋友，如果沒有你們，就沒有今天的我。

最後感謝我最敬愛的父親、母親，還有我的大哥，謝謝你們永遠在我的背後支持我。謹以此論文獻給所有我需要感謝的人，謝謝他們。

目錄

第１章 緒論.....1
１.１ 研究動機與背景.....1
１.２ 章節介紹.....2
第２章 背景知識與相關研究.....3
２.１ 循序 Java 程式的測試策略.....3
２.２ 並行 Java 程式的執行機制.....9
２.３ Concurrent Java Program 的測試策略.....15
第３章 Concurrent Path 的選取方法.....19
３.１ Paths 間依存關係之分析.....19
３.２ 尋找所有 paths 間依存性的方法.....23
３.３ 藉由 dependent path set 選取測試單元 concurrent path.....35
３.４ 圖形著色問題.....50
第４章 範例.....54
第５章 結論.....61
References.....63

圖目錄

圖２-１ Java 元件與元件間關係表示圖 4
圖２-２ 循序 Java 程式範例..... 5
圖２-３ Java 程式之結構模型圖 6
圖２-４ main()的 expanded path 範例..... 7
圖２-５ 執行路徑與路徑中呼叫函式示意圖..... 7
圖２-６ 並行 Java 程式範例，兩種主動類別..... 9
圖２-７ Run 與 path 圖例 10
圖２-８ 主動類別與共享類別圖例..... 11
圖２-９ 並行 Java 程式範例..... 12
圖２-１０ 並行 Java 程式結構模型範例..... 13
圖２-１１ 並行 Java 程式中同時執行的 expanded paths 範例..... 14
圖２-１２ expanded path 與建立 thread 示意圖 15
圖２-１３ 並行 Java 程式與存取 shared object 範例..... 17
圖３-１ 兩條 path A1 與 B1 共同存取一個 shared object S1 20
圖３-２ A1 與 B1 共同存取 S1 時的所有可能情況 21
圖３-３ 存在間接相關性的系統範例..... 22
圖３-４ 四條 paths 間的間接相關性..... 22
圖３-５ shared object 資料結構示意圖 25
圖３-６ 直接 data-flow 資料結構示意圖..... 25
圖３-７ 演算法：尋找直接資料流..... 26
圖３-８ 直接與間接資料流之關係..... 27
圖３-９ 資料流串鏈..... 27
圖３-１０ 資料流串鏈示意圖..... 28
圖３-１１ path 中的存取點資料結構示意圖 29
圖３-１２ 資料流串鏈資料結構示意圖..... 29
圖３-１３ 演算法：尋找資料流串鏈..... 31
圖３-１４ 以位元陣列儲存 dependent path set 32
圖３-１５ 演算法：判斷兩 dependent path set 是何包含關係..... 33
圖３-１６ 演算法：尋找 dependent path sets 34
圖３-１７ partial concurrent path A1B1 應與 C2 結合 36
圖３-１８ gamming tree 示意圖 38
圖３-１９ P_i 與 P_1 及 P_2 的關係對新集合可組合情況的影響 39
圖３-２０ partial concurrent path 以陣列方式儲存 40
圖３-２１ 可結合矩陣 A 與組合方式陣列 C 的關係示意圖..... 41
圖３-２２ XRLF 演算法[20]..... 51
圖３-２３ DIMACS 圖形著色例圖各演算法表現之比較 53

圖 3-24 隨機圖形各演算法表現之比較.....53
圖 4-1 範例一.....54
圖 4-2 範例一：path B1 上儲存所有流至 B1 的直接資料流參考.....55
圖 4-3 範例二.....57
圖 4-4 範例三.....58
圖 4-5 範例四.....59

第 1 章 緒論

1.1 研究動機與背景

由於分散式系統與多緒多工機制技術的成熟，運用並行式程式(concurrent program)以充分利用這些硬體資源來處理複雜問題，已是今日軟體系統開發之重要技巧。

Java 是一個物件導向的程式語言，具有跨平台執行的性質，內建 multithreading 的能力，並且能夠透過全球資訊網頁直接執行[1][2]，這些優點使得利用 Java 語言來開發的軟體日益增多；透過 Java 語言內建的 multithreading 機制，撰寫並行式程式(concurrent program)更為簡便，因此愈來愈多的並行程式選擇 Java 語言來開發。

為了保證軟體的正確性與可靠度，測試是軟體開發必需的步驟。欲測試一軟體，首先需準備測試個案(test case)，包括輸入資料與預期的輸出結果，然後依每個測試個案一一執行待測程式，以驗證程式的正確性。

並行式 Java 程式執行時，不同的平行執行個體並非各自獨立，而是會透過呼叫共用的物件存取共同變數來作相互間的同步及交換資料，如此方能合作以完成共同目標。輸入一組數據以執行並行程式，每一個不同的執行個體都會執行一連串指令，此稱為執行個體的一條路徑，而所有執行個體執行路徑組合稱為一條並行路徑。輸入不同資料，每個執行個體可能會執行不同的路徑，因而形成不同的並行路徑，產生不同的執行行為。

從軟體測試的角度來看，可依每種可能執行的並行路徑來產生測試個案，測試員需針對每一條並行路徑來設計測試個案。當執行一條並行路徑的各個執行個體的執行順序不同，其結果也可能不同，使測試並行 Java 程式更為複雜。需要測試的程式執行行為與需要產生的測試個案數量增加，這使得並行程式的測試複雜度與成本皆增加。

今日軟體系統愈來愈複雜，並行 Java 程式的並行路徑數目快速增加，如果要將所有並行路徑均測試完，是一件非常複雜的工作。為了提升測試效率與降低成本，如何從所有並行路徑中選出一個最小的子集合來測試，且能保證測試的完整性，是所有 Java 程式測試研究者的目標，目前已有的研究成果尚缺乏一套系統化方法，且無法說明是否為最佳解。

1.2章節介紹

本論文章節安排如下，首先在第二章介紹循序式 Java 程式的測試策略與並行式 Java 程式的並行機制與程式架構，並介紹現有的並行 Java 程式測試方法。第三章分析並行 Java 程式的執行個體間因存取共同物件而產生的依存性關係；再藉由共同物件的存取情形，尋找因不同執行個體間會互相影響的路徑而找出應一起被測試的配對；最後提出依這些依存性關係找出可以涵蓋所有執行行為的並行路徑的方法。第四章提出範例說明本研究所提之方法確然可行並可找到最少之測試個案。第五章總結本研究之貢獻。

本研究希望能提出一套系統化方法來產生平行執行個體間最少並行路徑，以減少測試個案的準備；本研究分析存取同一共同物件的多條不同執行單位的路徑之執行行為相互影響關係，藉此找出互有影響的路徑集合；在由上述路徑集合組合最少的並行路徑時，發現它與圖形著色(graph coloring)的問題相類似，屬於 NP-Completed 問題，無法在有限時間內找出最佳解；本研究提出以 gaming tree 之 look-ahead 測略來產生最少並行路徑，雖無法證明這方法保證產生最佳解，但分析多個不同的 Java 程式後，發現其為最佳解，且計算時間短。

第 2 章 背景知識與相關研究

本章於 2.1 節介紹循序 Java 程式之測試方法。2.2 節介紹並行 Java 程式的執行機制，與找出並行 Java 程式的執行單元 concurrent path 的方法。2.3 節介紹目前並行 Java 程式的測試方法，並說明因各個並行的執行個體因存取 shared object 而產生互動，故必需測試所有可能的 concurrent paths 以驗證整個並行 Java 程式的正確性，但根據 shared object 的存取情形，發現只需測試部份的 concurrent paths 即可涵蓋完整的執行情形。故本研究提出可分析 shared object 的存取情形，來減少測試的 concurrent paths 的數量，增進並行 Java 程式的測試效率。

2.1 循序 Java 程式的測試策略

Java 程式分為 Java application 與 Java applet 兩種，Java application 是可獨立執行的程式，applet 則是需透過網頁整合瀏覽器執行的元件，除了安全性與執行環境的差異之外，兩種程式之結構幾乎相同，其測試方法亦大致相同，因此本研究以 Java application 為探討對象。

Java 是一個並行式的物件導向的語言，具備物件導向的性質及並行執行的能力。一個 Java 程式若未呼叫 Thread 物件的 start()函式，將只會有一個 thread 在系統中執行，因此稱之為循序式 Java 程式(sequential Java program)。反之，若有 Thread 物件的 start()函式被呼叫，每次呼叫 Thread 物件的 start()函式系統將建立一條新的 thread 來執行該 Thread 物件的 run()函式，因此稱為並行式 Java 程式(concurrent Java program)。

一個並行式 Java 程式被執行時，將有兩個以上的執行單元同時執行，每一個執行單元均是循序執行，可視為一個循序 Java 程式，所以討論並行 Java 程式的測試方法之前，有必要先探討循序 Java 程式的測試方法。

分析 Java 程式之結構可了解其各元件間的關係，依此程式結構模型可幫助測試人員建立測試個案(test case)。Java 程式是由包裹(Package)、類別(Class)、與

介面(Interface)等三種元件組成[10]，一個 Java 程式可以包含一個或多個包裹，一個 Java 程式的包裹是一組功能相近或性質相關的類別和介面所成的集合。Java 的類別是由一組屬性(attribute)和作用在這些屬性上的函式(method)所成的集合，包括屬性的宣告(attribute declaration：包括屬性的型態和名稱)、函式的宣告(method declaration：包括函式的名稱、傳回值之型態、以及傳入值的型態和名稱)和函式的定義(method definition：除了函式的宣告外，還包括函式程式碼的主體)。而 Java 的介面是只有常數宣告以及函式宣告而沒有函式定義的類別。

Java 元件間關係有繼承(inherit)、實作(implement)、與使用(use)等三種。類別與介面間可以有實作關係或使用關係，類別與類別間可以有繼承關係或使用關係，而介面與介面間只能有繼承關係。一個介面可被多個類別繼承，若一個類別使用介面 *I*，則可能使用任一實作 *I* 的類別或繼承 *I* 的介面。Java 程式之包裹、類別、與介面等三種元件，與繼承、實作、與使用等三種關係，可以用圖 2-1 所示的符號來代表。

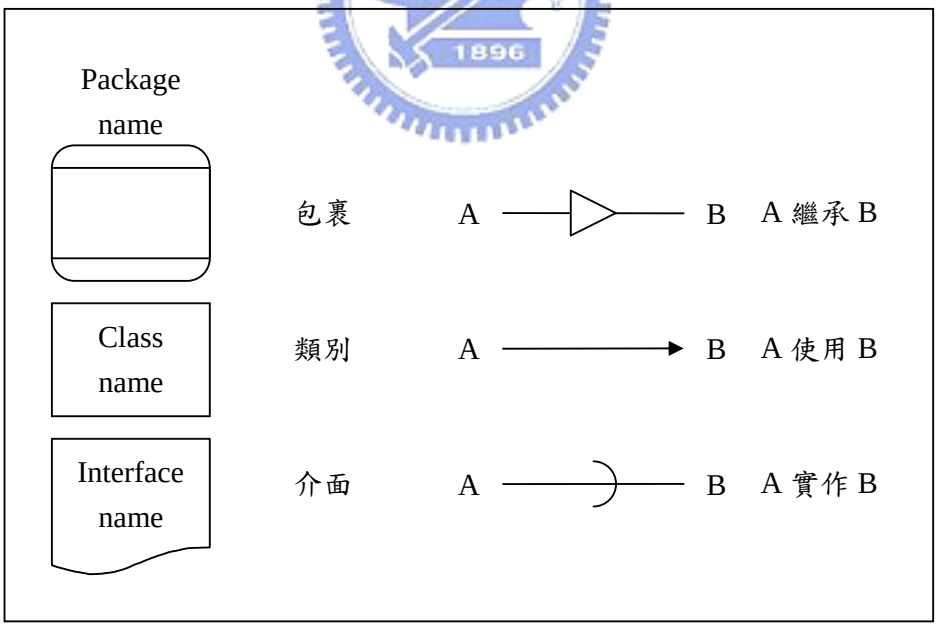

圖 2-1 Java 元件與元件間關係表示圖

利用這些符號可以表示循序 Java 程式之組成元件及元件間的關係，例如圖 2-2 之 Java 程式可以以圖 2-3 之結構模型來表示。

CA.java:

```
package A;
import B.*;
interface IA extends IB {...}
class CB implements IA {...}
class CC extends CE implements IB {...}
public class CA {
  public static void main(String args []) {
    CB cb = new CB();
    CC cc = new CC();
    CD cd = new CD(); ...
  }
}
```

IB.java:

```
package B;
public interface IB {...}
interface IC {...}
interface ID {...}
interface IE extends IB, IC {...}
```

CE.java:

```
package B;
public class CE {...}
```

CD.java:

```
package B;
import C.*;
public class CD implements ID, IE {
   public void foo() {
      CE ce=new CE();
      IG ig;
      CX cx;
      CV cv = new CV();
      ...
      switch(I) {
      case 1: ig = new CS(); break;
      case 2: ig = new CT(); break;
      }
      ...
      switch(I) {
      case 1: cx = new CX(); break;
      case 2: cx = new CY(); break;
      }
      cv.fee(cx, ig);
  }
}
```

CV.java:

```
package C;
public class CV {
  public void fee(CX cx, IG ig) { cx.m1(ig); }
}
```

IG.java:

```
package C;
public interface IG {...}
interface IF extends IG {...}
```

CS.java:

```
package C;
public class CS implements IF {...}
```

CT.java:

```
package C;
public class CT implements IG{...}
```

CX.java:

```
package C;
public class CX {
   public void m1(IG ig) {...}
}
```

CY.java:

```
package C;
public class CY extends CX {
   public void m1 (IG ig) {...}
}
```

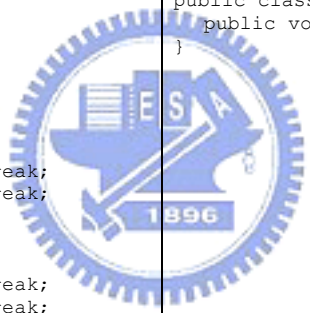

圖2-2 循序 Java 程式範例

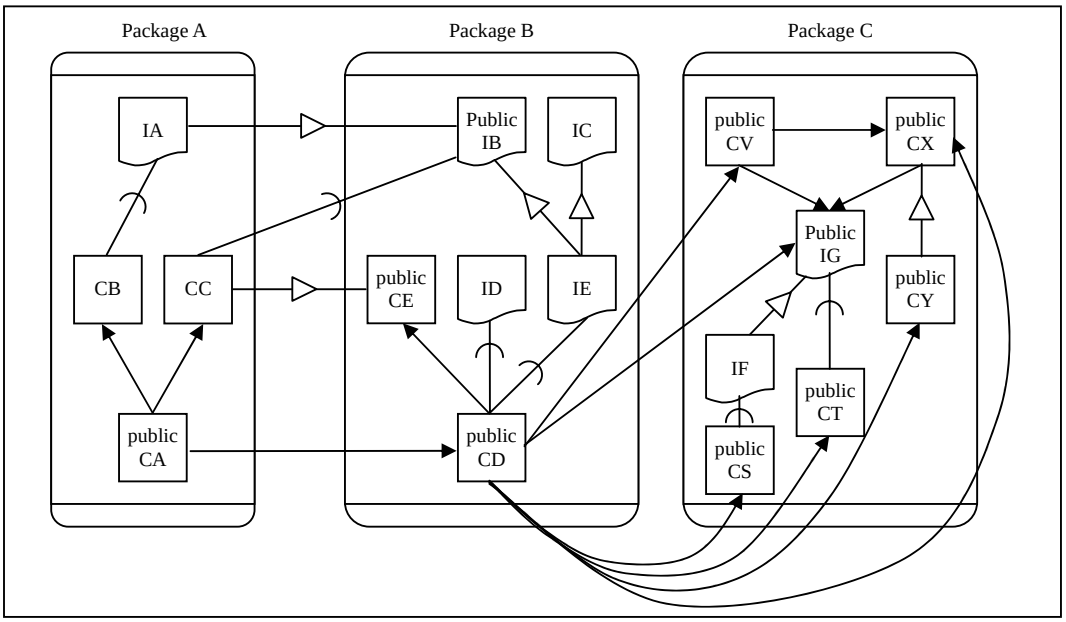

圖 2-3 Java 程式之結構模型圖

當一個循序 Java 程式被執行時，系統自動建立一條 thread 去執行 main()函式，從 main()函式的開頭執行，經過一連串的指令，直到結束或回到特定點，這一連串指令稱之為 main()函式之一條 path。因為條件判斷會使程式執行不同的指令，一個 main()函式可擁有多條 paths，每次執行只有一條 path 被執行。main()函式的一條 path 可能包含產生某個物件或呼叫某個物件的函式的指令，執行此指令時會將控制權轉移至該物件的函式；一個物件的函式從開頭直到 return 指令亦可能有多種 path，但每次執行只會執行其中一條 path。一物件的函式也可能有產生其它物件或呼叫其它物件函式的指令，因此定義一個 main()函式的 expanded path 為一 main()函式的一條 path 與執行此 path 時因呼叫其它物件函式而執行的所有 paths 的組合。圖 2-4 表示一個 main()函式的一條 path 的範例，main()的 path 呼叫物件 O1 的 m1()函式與物件 O2 的 m2()函式，而 O2.m2()被執行的 path 又呼叫 O3 的 m3()函式，因此圖中依 1 至 7 的執行順序所經過的循序指令即為 main()函式的一個 expanded path。若 main()與各個被呼叫函式的執行路徑都已被找出來，則 main()的 expanded path 即可由 main()的 path 與被呼叫的函式的 paths 予以組合產生。

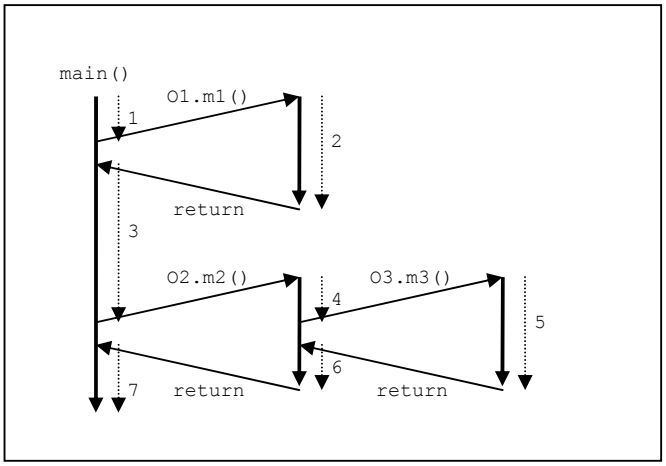

圖2-4 main()的 expanded path 範例

每次執行一個 Java 程式，即為執行 main()函式的一條 expanded path。所有的 Java 函式包含的執行路徑，皆可使用傳統 procedure-oriented 測試方法來找出。在 main()與每個被呼叫的函式任取一條 path 產生的組合，都可形成一條 expanded path。如圖 2-5 標記每條 path 呼叫了哪些函式之後，程式所有的 expanded path 就可以輕易地找出來[6]。

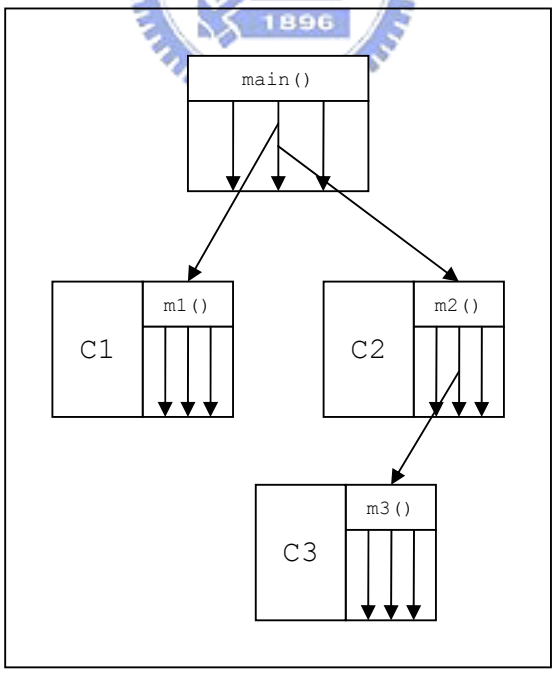

圖2-5 執行路徑與路徑中呼叫函式示意圖

測試一個循序 Java 程式相當於測試 main()函式的所有 expanded path 的行

為，直接測試所有的 expanded path 之缺點為(1)錯誤發生時不容易鑑別發生錯誤是在哪一個函式中的哪一條 path 中，(2)一個函式中相同的 path 可能會被測試多次。

main()函式的一條 expanded path 包含了一條 main()函式的 path 與這條 path 所呼叫的函式的 paths，若被呼叫到的函式的 paths 都已先測試驗證正確，當測試這條 expanded path，其結果若不正確，則一定是在 main()函式的 path 發生錯誤。依照這個概念，若類別 *A* 使用了類別 *B*，則先測試被使用的類別 *B*，驗證 *B* 的正確性之後，再測試 *A*。

根據之前所述的結構模型，可定出循序 Java 程式中元件的測試順序，沒有使用其它類別的類別應最先被測試，然後是只使用到這些類別的類別。對於有繼承關係的類別，則先測父類別，因為子類別可呼叫父類別的函式。若有互相都使用對方的多個類別存在，則先測試使用未測試的類別數量最少的類別，並使用 stub 來模擬它使用到但尚未被驗證過的類別。這個測試方法稱為 bottom-up 測試方法[11][12]。

同一個類別中含有多個函式時，一個函式可能會呼叫其它函式，依照 bottom-up 的概念，同樣先測試被呼叫的函式。當一個函式 *F* 呼叫其它已驗證的函式 *G* 時，可將呼叫 *G* 的動作視為正確的指令，因此驗證函式 *F* 的正確性只需測試 *F* 本身的 paths 即可。一個函式可以包含多條 paths，執行一個函式呼叫會依據物件狀態(object state，物件包含的屬性的值所成的集合)和函式的傳入值，來決定執行哪一條 path，並產生新的物件狀態以及函式傳回值，依照這些資訊來判斷這條 path 是否正確。欲測試一個函式需為每一條 path 準備測試個案(test case)，包括上述物件狀態、函式傳入值以及預期執行完畢後的物件狀態及傳回值等資料，以驗證這條 path 的正確性。

依 bottom-up 測試策略，main()函式會最後被測試。當 main()函式的 paths 都被驗證時，即完成循序 Java 程式的測試工作。

2.2 並行 Java 程式的執行機制

Java 虛擬機器允許一個程式有多個 thread 同時在執行。當一個 Java 程式開始執行時，Java 虛擬機器會自動地產生一個 thread 來執行 main()函式。main()的執行過程中若呼叫一個 Thread 物件的 start()函式，Java 虛擬機器會自動產生另一 thread 來執行該 Thread 物件之 run()函式。可以產生 thread 的物件，稱之為主動類別(active class)。循序 Java 程式僅有一個 thread，而在並行 Java 程式中，一個程式在執行時，會有多個 thread 同時執行，以進行不同的任務。

在 Java 中，一個類別若是(1)繼承自 java.lang.Thread 類別或(2)實作 java.lang.Runnable 介面，則稱為主動類別。如圖 2-6 所示。

```
class SensorThread extends Thread {
  long time;
  SensorThread(long time) {
    this.time = time;
  }
  public void run() {
    // check the sensor status every
    // time mini-second
  }
}
public class Main {
  public static void main(String argv[]) {
    SensorThread st = new SensorThread(1000);
    st.start();
  }
}
```

```
class SensorThread implements Runnable {
  long time;
  SensorThread(long time) {
    this.time = time;
  }
  public void run() {
    // check the sensor status every
    // time mini-second
  }
}
public class Main {
  public static void main(String argv[]) {
    SensorThread st = new SensorThread(1000);
    new Thread(st).start();
  }
}
```

圖 2-6 並行 Java 程式範例，兩種主動類別

一個 Thread 物件的 start()函式只能被呼叫一次，Thread 物件之 start()函式被呼叫後會啟動 run()函式執行，假如在 run()函式執行未結束前再次呼叫此 Thread 物件的 start()函式，則會造成 IllegalThreadStateException。若該 thread 已經執行結束後，該 Thread 物件的 start()函式再次被呼叫，則不會再建立 thread 來執行 run()函式。一旦 thread 被啟動，這個 thread 會一直執行直到 run()函式被執行完或者 stop()函式被執行為止。

每個 thread 本身都是循序執行的執行個體，在上一節即討論過循序 Java 程式的測試策略。在本研究中，將這些循序執行的執行個體稱為 Run，並行 Java 程式中存在有兩個以上的 Run，以大寫英文字母命名加以區分，如第一個 Run 稱為 Run A，接著是 Run B，依此類推。每一個 Run 因皆為循序執行，可以循序 Java

程式的測試策略找出其須測試的 paths，因每個 Run 可有多條 paths，path 以 Run 名稱中的字母加上數字編號代稱，如 Run A 的第一條 path 為 A1，第二條為 A2 等等，其圖例如下所示。

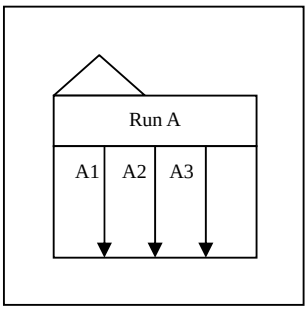

圖2-7 Run 與 path 圖例

在 Java 中，兩條 threads 可能是獨立執行的，也可能存在下面三種互動方式：(1)透過 stop()、suspend()與 resume()函式中止、暫停或繼續另一個 thread，(2)以 join()函式等待某個 thread 執行完畢，(3)呼叫同一個物件的函式以存取共同變數。其中 stop()函式可能導致錯誤的物件狀態，而呼叫 suspend()及 resume()可能會導致死結的發生，因而 Java 已不建議使用這些函式。

Java 的 thread 提供了 join()函式，當一個 thread 呼叫另一個 Thread 物件的 join()函式，若被呼叫的 Thread 物件的 thread 執行 run()尚未結束，則呼叫的 thread 必須等到被呼叫的 thread 執行完畢之後才能繼續執行；若被呼叫的 Thread 物件之 run()函式已執行完畢，則呼叫 join()函式的 thread 不需等待。此互動方式影響 threads 的執行先後順序，但 threads 之間不會互相影響執行結果。可依循序 Java 程式的測試方法測試各個 Run。

Java 虛擬機器允許多個 thread 並行執行，這些 thread 共用一份共同的記憶體空間，因此多個 thread 可能會呼叫同一個物件的函式來做溝通，會被多個 thread 同時呼叫使用的物件稱為 shared object。

當不同的 thread 同時呼叫一個 shared object 的函式，是以 interleaving 的方式執行被呼叫的函式，若這些函式間將存取相同資料成員，則會因執行順序之不同而產生不一樣的結果，即所謂的 race condition 的問題。為了避免 race condition

發生，Java 程式語言提供了 monitor 作為執行單元之間的同步機制，即同一時間只有一個執行緒可以執行受 monitor 保護的程式碼。我們可用 lock 的觀念來解釋 Java 所提供的 monitor 機制，在 Java 虛擬機器中每一個物件都有一個對應的 lock，在函式宣告時若加上 synchronized 關鍵字，則此函式被稱為同步函式 (synchronized method)，一條 thread 執行一個物件的同步函式前必須取得此物件的 lock，如果此 lock 已被其他的 thread 取得，則此 thread 將不會執行此同步函式，直到取得此物件的 lock；當 thread 執行完一個物件同步函式並從同步函式返回時，此物件相對應的 lock 將被釋放。Java 利用這種機制來保證同時只有一個 thread 在執行 shared object 中的同步函式，避免 race condition 的問題。

建立並行 Java 程式的結構模型可看出各個類別間的關係，而排出類別的測試順序。一個 shared object 必定會以參數的方式傳遞到主動類別中，因此在做 Java 程式的靜態分析時，可以很輕易地找出會產生 shared object 的共享類別(shared class)。在結構模型圖中以下列方式代表主動類別與共享類別。

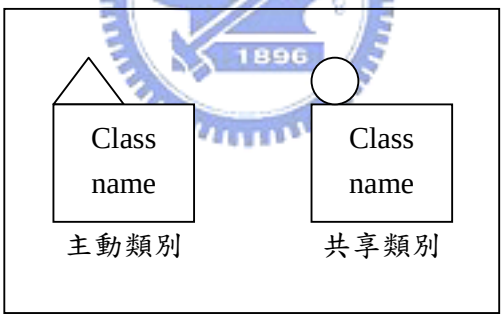

圖 2-8 主動類別與共享類別圖例

如圖 2-9 的並行 Java 程式經過靜態分析可得如圖 2-10 的結構模型圖。

C1.java

```
package A;
class GC1 {
  public void m1() { ... }
}
class A1 extends Thread {
  private GC1 gc1;
  A1() { gc1 = new GC1(); }
  Public void run() { gc1.m1(); }
}
class S1 {
  public synchronized m1() { ... }
  public synchronized m2() { ... }
  public synchronized m3() { ... }
}
class A2 extends Thread {
  private S1 s1;
```

C2.java

```
package B;
class GC2 {
  public void m1() { ... }
  public void m2() { ... }
  public void m3() { ... }
}
class A3 extends Thread {
  private GC2 gc21;
  A3() { gc21 = new GC2(); }
  public void run() { gc21.m1(); }
}
class A4 extends Thread {
  private GC2 gc22;
  A4() { gc22 = new GC2(); }
  public void run() { gc22.m2(); }
}
```

```
  A2(S1 s) { s1 = s; }
  public void run() {
    if( ... )
      s1.m1();
    else
      s1.m2();
  }
}
public class C1 {
  private A1 a1;
  private A2 a21;
  private A2 a22;
  private S1 s1;
  private C2 c2;
  private C3 c3;
  C1() {
    a1 = new A1();
    s1 = new S1();
    a21 = new A2(s1);
    a22 = new A2(s1);
    c2 = new C2();
    c3 = new C3();
  }
  public void main(String argv[]) {
    a1.start();
    a21.start();
    a22.start();
    s1.m3();
    c2.m1();
    c3.m1();
  }
}
```

```
public class C2 {
  private A3 a3;
  private A4 a4;
  private GC2 gc23;
  C2() {
    a3 = new A3();
    a4 = new A4();
    gc23 = new GC2();
  }
  public void m1() {
    a3.start();
    a4.start();
    gc23.m3();
  }
}
```

C3.java

```
package C;
class S2 {
  public synchronized m1() { ... }
  public synchronized m2() { ... }
  public synchronized m3() { ... }
}
class A5 extends Thread {
  private S2 s21;
  A3(S2 s) { s21 = s; }
  public void run() { s21.m1(); }
}
class A6 extends Thread {
  private S2 s22;
  A3(S2 s) { s22 = s; }
  public void run() { s22.m2(); }
}
public class C3 {
  private A3 a3;
  private A4 a4;
  private S2 s23;
  C3() {
    s23 = new S2();
    a3 = new A3(s23);
    a4 = new A4(s23);
  }
  public void m1() {
    a3.start();
    a4.start();
    s23.m3();
  }
}
```

圖2-9 並行 Java 程式範例

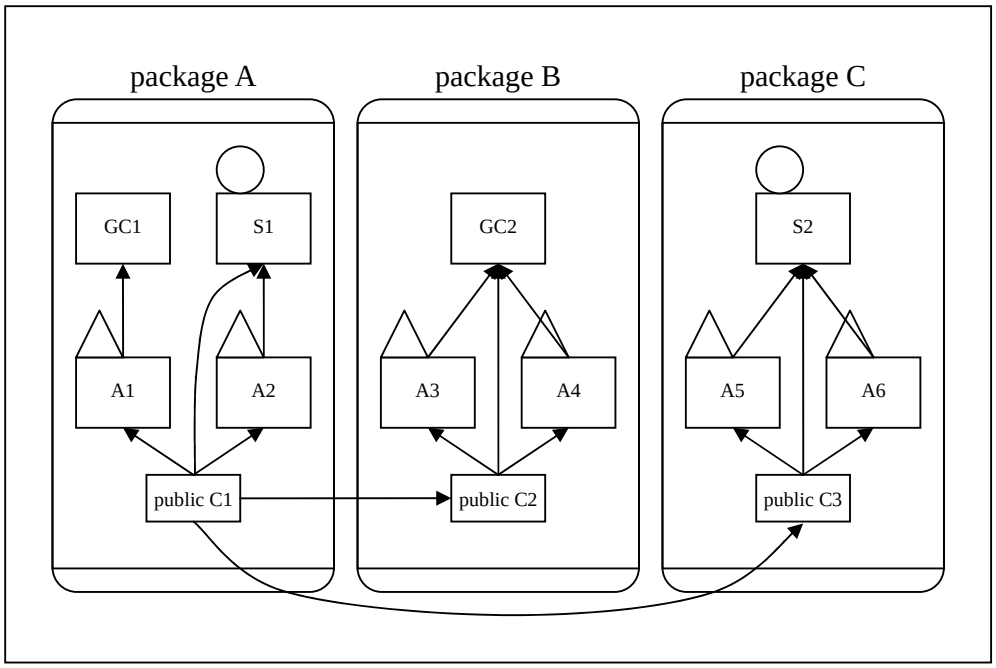

圖 2-10 並行 Java 程式結構模型範例

當一個並行 Java 程式開始執行時，Java 虛擬機器會自動地產生一個 thread 來執行 main()函式的一條 expanded path。若這條 expanded path 沒有呼叫主動類別的 start()函式，則這條 expanded path 僅是一群循序執行的指令的集合，可用循序 Java 程式的測試方法測試。

若 main()函式的 expanded path 中有呼叫 Thread 物件的 start()函式的指令，每一個 start()呼叫會建立一條 thread 來執行這個 Thread 物件 run()函式的一條 expanded path，run()函式中的 expanded path 也可因呼叫別的 Thread 物件的 start()函式再建立一條 thread，執行被呼叫的 Thread 物件 run()函式的一條 expanded path。main()函式的 expanded path 與其它被建立的 threads 的 expanded paths 會同時在系統中執行，它們所成的集合，稱為 main()的一條 concurrent path[6]。

以圖 2-11 為例是一個並行 Java 程式的執行情況，main()函式的 path 依序呼叫了 Thread 物件 TO1、TO2 與 TO3 的 start()函式，分別建立了三條 thread。main()的 path 還呼叫了 TO1 的 foo()函式，形成依 1 至 3 循序執行的一條 expanded path。TO1.run()函式的 path 執行標記為 11 的循序指令。TO2.run()與 TO3.run()

函式的 paths 分別呼叫了 shared object SO 的 m1()與 m2()函式，TO2.run()的 expanded path 依 21 至 23 的順序執行，而 TO3.run()的 expanded path 依 31 至 33 的順序執行。圖中以虛線箭頭表示呼叫 Thread 物件的 start()函式建立一條新的 thread。main()的 expanded path 與 TO1.run()、TO2.run()和 TO3.run()的 expanded paths 的集合即為 main()的一條 concurrent path。

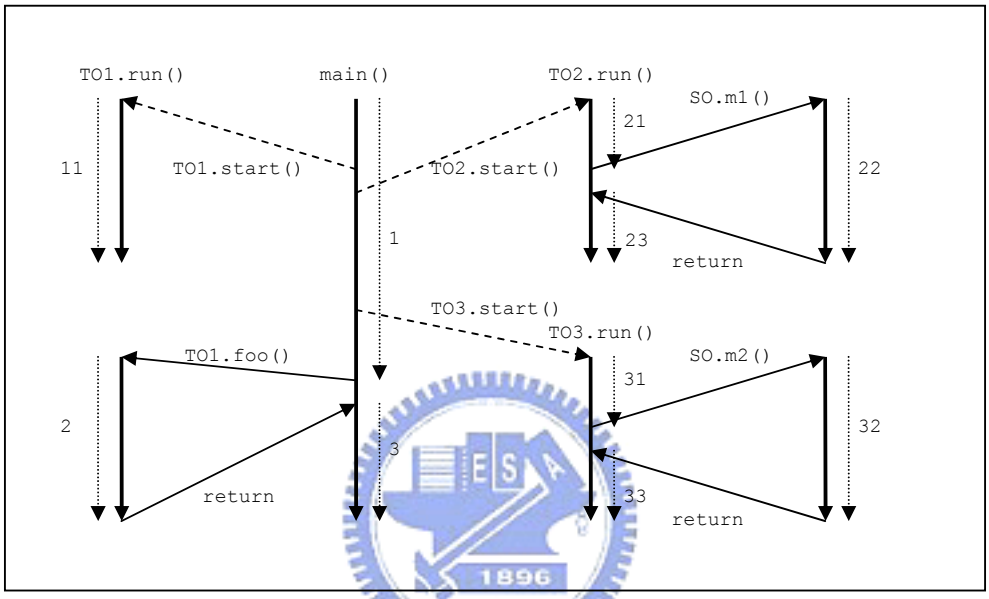

圖 2-11 並行 Java 程式中同時執行的 expanded paths 範例

執行一個並行 Java 程式，即為執行 main()函式的一條 concurrent path。main()函式與被建立的 threads 各任取一條 expanded path 的組合，皆可形成一條 concurrent path。找出 main()函式與所有的主動類別 run()函式的 expanded path，如圖 2-12，組合 main()函式與呼叫的所有主動類別的 paths 即可構建該 main() path 的所有 concurrent paths。

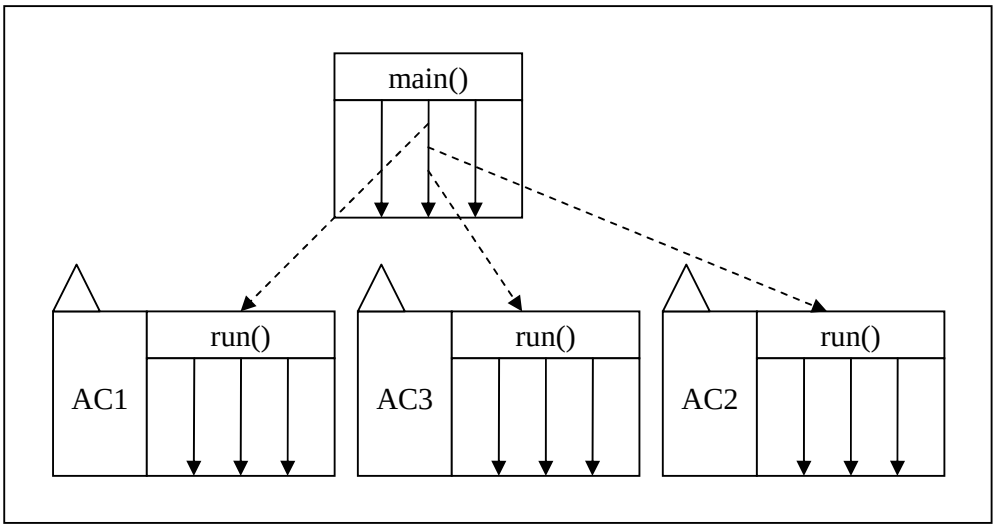

圖2-12 expanded path 與建立 thread 示意圖

一個並行 Java 程式被執行時，依據輸入資料不同，可能會執行任何一條 concurrent path，因此測試一個並行 Java 程式正確與否，應驗證所有可能執行的 concurrent path 的執行行為都正確無誤。

2.3 Concurrent Java Program 的測試策略

由2.2節的並行程式之行為可知，測試一個並行 Java 程式，等於測試 main() 函式的所有展開的 expanded paths 與 concurrent paths。若直接測試這些展開的 expanded paths 及 concurrent paths，則有兩個缺點：(1)concurrent path 含有數個 expanded paths， expanded path 又含有多個函式的 path，錯誤發生時，難以鑑別發生錯誤的指令位置；(2)同一個函式的 path 可能會出現在許多不同 expanded paths，將被重複測試多次。因此可運用 bottom-up 測試的觀念，來降低測試之複雜度與時間。

在 main()函式或主動類別的 run()函式的 expanded path 中，若沒有建立新的 thread，這條 expanded path 可視為循序執行，可用循序 Java 的測試方法測試之，因此並行 Java 程式可先測試其循序執行性質，再測試其並行性質，其測試方法可為四個步驟[6]：

(1) 以 main()函式或主動類別的 run()函式為單位，根據 Java 程式的結

構模型，找出 main()函式或 run()函式使用到的所有類別。

(2) 忽略呼叫 Thread 物件的 start()函式的指令，以２.１節介紹之循序 Java 程式的 bottom-up 測試方法驗證 main()函式或 run()函式使用到的類別，最後循序測試 main()函式或 run()函式每一條 expanded path。

(3) 以２.２節描述的方法組合被 main()函式用來建立 thread 的主動類別 run()函式的 expanded paths，建構 main()函式所有的 concurrent paths。

(4) 依序測試每一條 concurrent path，驗證每一條 concurrent path 的執行行為。

上述步驟(1)與步驟(2)主要在驗證所有 main()及 run()函式的循序的執行行為，與測試所有的 expanded paths 相比，有較低的測試複雜度與偵錯的成本，且可必免重複測試。

main()函式或主動類別的 run()函式可能有多條 expanded path，當一條 expanded path 執行時有呼叫 Thread 物件的 start()函式，會產生一個 thread 以執行該 Thread 物件的 run()函式，當給定一輸入資料以執行 main()函式的這條 expanded path，此 expanded path 所產生的其它 threads 將各自執行一條 expanded path，它們的組合即為一條 concurrent path，如圖２-１３，main()的一條 expanded path 利用呼叫 Thread 物件的 start()函式產生了兩個新的 thread，依上節對執行個體的命名稱為 Run A 與 Run B，而 Run A 與 Run B 各有三條 expanded paths。main()函式 expanded path 的輸入資料不同，Run A 與 Run B 也會執行不同的 expanded path，因此構成不同的 concurrent path。因此只要 main()函式的 expanded path 含有一個或多個 start()指令，將會產生多條 concurrent path；由圖２-１３之範例，Run A 與 Run B 各有三條 expanded path，每一個 Run 的 expanded path 均應被測試過，且每一 shared object 的交互行為也均應被測試過，如何選擇最少組合數目來完成上述兩個目的，即為本研究的主題。

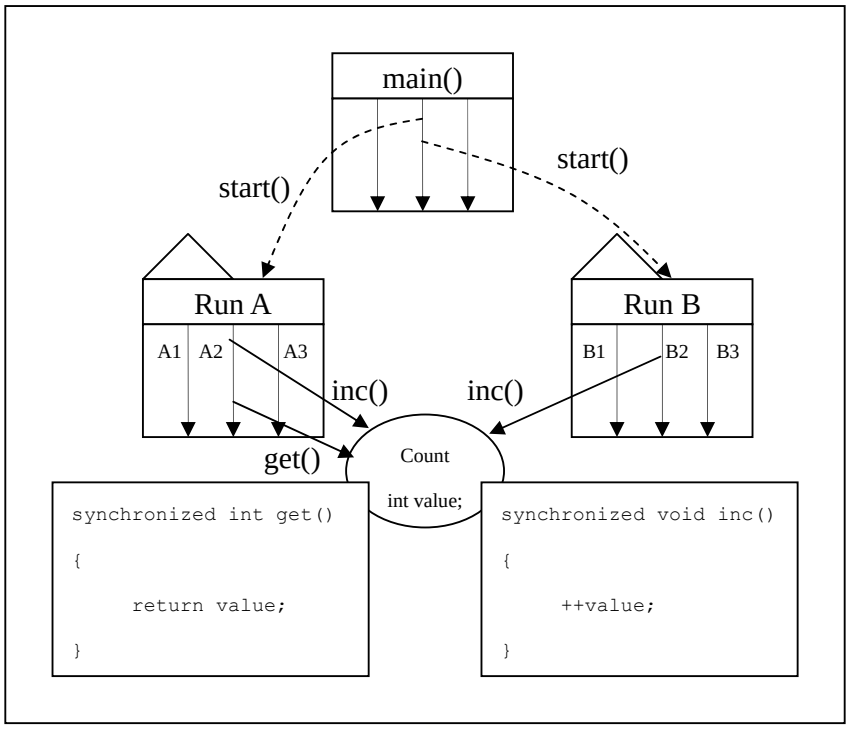

圖2-13 並行 Java 程式與存取 shared object 範例

當執行一含有 shared object 的 concurrent path 時，當 thread 之間相對執行速度不一時，各 thread 呼叫執行 shared object 的先後順序亦可能不同，因此可能產生不同結果，此即為 concurrent path 之不確定執行行為[7]。如圖 2-13 中 Run A 與 Run B 共享 Count 這個 shared object，假設 value 的初值為 0，若三個訊息到達的順序為 A's inc()→B's inc()→A's get()，則 Run A 呼叫 get()所得到的值為 2；若順序為 A's inc()→A's get()→B's inc()，則 Run A 呼叫 get()所得到的值為 1。因此測試一個含有 shared object 的 concurrent path 時，需測試每種存取 shared object 的順序，才能夠達到驗證此 concurrent path 所有行為之目的。現有的一些測試工具，如 BugNet [8]或 TAP[9]，均提供控制各 thread 間執行順序的機制，因此可以控制 shared object 訊息到達的順序，要測試時，需先列舉所有可能存取 shared object 的順序，控制訊息到達順序，才能得到預期目的[6]。

在圖 2-13 的範例中，驗證此並行程式所有的行為，應測試過所有九條可能的 concurrent path，分別記為 A1B1、A1B2、A1B3、A2B1、A2B2、A2B3、A3B1、A3B2、A3B3。其中 A2 與 B2 因有 shared object 使這兩條 paths 會互相影

響，因此需以 A2B2 這條 concurrent path 測試它們的互動情形，其它使用到這兩條 paths 其中之一的 concurrent path，如 A1B2 執行了 B2，因為 A1 與 B2 間沒有互動，這兩條 paths 僅會獨立循序執行，而 B2 獨立執行的執行行為已在 concurrent path A2B2 中被測試過，只要測試另一可包含 A1 的循序執行行為的 concurrent path，如 A1B1，則 concurrent path A1B2 就不需測試，因為它的執行行為都已被測試過。在此例中若選出 A1B1、A2B2、A3B3 這三條 concurrent paths 來測試，即可測試到 A2 與 B2 互相影響的情況，同時亦可測試到所有六條 paths 循序執行的執行行為，因此其它 concurrent paths 便可以不必測試亦可保證其行為已被驗證。由此可知，對於並行 Java 程式，可以測試部分的 concurrent paths 即可驗證所有的執行行為，因此有必要找到一個系統化的方法來自動產生最少的 concurrent paths 加以測試。

第 3 章 Concurrent Path 的選取方法

執行並行 Java 程式將有多個 threads 同時執行，且這些 threads 間有訊息交換而影響其執行過程及結果，在進行並行 Java 程式測試，必須完整測試所有 threads 間互動情況，才能保證其正確性；如何選擇最少的測試個案來驗證所有並行執行 thread 間的互動關係，以降低測試成本，目前尚無良好方法，因此只能用暴力法(exhaustive)來產生所有 concurrent paths 而加以測試。並行執行的 threads 間主要是透過 shared object 來交換資料，本章首先分析存取同一 shared object 的 path 的資料流關係，找出 concurrent path 之不同 path 間之相互影響關係，進而找出需並行測試的不同 thread 的 paths 的最少組合，利用這些組合去產生所需的最少 concurrent paths。

本章 3.1 節說明不同 thread 之 paths 間存取 shared object 而產生之依存的關係；3.2 節提出找尋有依存關係 path 組合之方法；3.3 提出找出涵蓋所有需測試之 path 組合之 concurrent paths 之方法。

3.1 Paths 間依存關係之分析

不同 thread 的 paths 呼叫同一 shared object 時，各 thread 對 shared object 存或取的順序不同，將影響 shared object 的狀態，而導致執行結果之不同。如圖 3-1 所示，path A1 與 B1 分屬不同 Run，且呼叫同一 shared object S1。若 A1 先執行存取 S1 的 statement，且有寫入動作，則 A1 在寫入之前所執行指令的結果將影響 S1 的屬性值；當 path B1 從 S1 讀取資料時， B1 後續指令的執行結果將受到 A1 先前指令執行結果之影響，此可視為有一從 A1 經由 S1 流至 B1 的資料流(data-flow)，影響 B1 的執行結果。

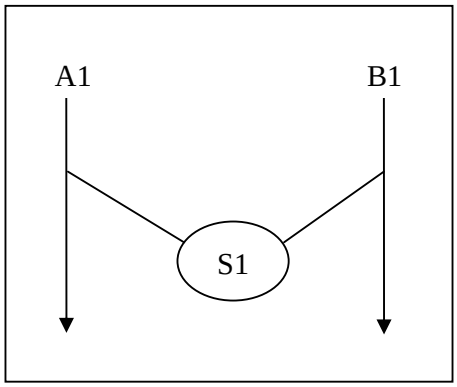

圖 3-1 兩條 path A1 與 B1 共同存取一個 shared object S1

若 A1 的輸入資料可經由 S1 流往 B1，而影響 B1 後半部分的執行。則稱 B1 的後半部執行結果依存於(dependents on)A1 的前半部的執行結果，或 A1 前半部執行結果影響(influence)B1 後半部的結果。我們將這種因 shared object 資料流動順序而造成影響的情形記為 A1 → B1。根據 A1 或 B1 何者先呼叫 S1，及 A1 或 B1 對 S1 之共用屬性變數作讀或寫的動作之不同，其相互影響關係如圖 3-2 所示，由此表可知，當 A1 與 B1 對 S1 的共同屬性變數的動作同為讀取或同為寫入時，A1 與 B1 間將無相互依存性；只有當一條 path 對 S1 共用屬性變數作讀取動作，另一條為寫入動作，且寫入較讀取早發生時，A1 與 B1 才具有依存性。然而若要知道各 path 對 shared object 的存取情況時，必需對所有 shared object 各個 method 的指令作分析，耗費甚多工夫，因此可假設存在 A1 → B1 及 B1 → A1。

先存取 S1 的 path 為 A1 或 B1	A1 的動作	B1 的動作	影響
A1	R	R	
A1	W	R	A1 → B1
A1	RW	R	A1 → B1
A1	R	W	
A1	W	W	
A1	RW	W	
A1	R	RW	
A1	W	RW	A1 → B1
A1	RW	RW	A1 → B1
B2	R	R	
B2	W	R	

B2	RW	R	
B2	R	W	B1 → A1
B2	W	W	
B2	RW	W	B1 → A1
B2	R	RW	B1 → A1
B2	W	RW	
B2	RW	RW	B1 → A1

圖 3-2 A1 與 B1 共同存取 S1 時的所有可能情況

Path 與 path 間也可能非因存取同一個 shared object 而產生依存性。一條 path A1 可能因為與另一個 Run 的 path B1 存取同一個 shared object S1 而有直接依存性，使 B1 的資料影響了 A1 存取 S1 之後的行為；若 A1 在對 S1 的存取點之後，又有一個對另一個 shared object S2 的存取點，而 A1 與另一 path C1 因 S2 而存在直接依存性，進而影響 C1 執行的結果。在這樣的情形下，B1 的行為直接影響了 A1(B1 → A1)，之後 A1 又直接影響了 C1(A1 → C1)。於是 B1 經由 A1 的執行而間接影響 C1 的執行結果，稱之為間接依存性關係，記為 B1 → A1 → C1。Paths 會因為間接依存而互相影響執行結果，因此測試時也需被包含在一個 concurrent path 中。

Paths 間的依存關係應考慮每條 path 呼叫 shared object 指令的位置，才能正確判斷其相互影響之資料流關係，先將每條 path 呼叫 shared object 的 statement 標出，以此為斷點而將 path 分為數個區段，稱為 path segment。

以圖 3-3 為例，一個擁有四個 Run 的系統共用三個 shared objects，其存取情形如圖所示，任一 path 的號碼代表其呼叫 shared object 的位置。此圖中 Run A 之兩條 paths 與 Run B 之 path 呼叫一同 shared object S1，Run B 之 path 又與 Run C 呼叫同一 shared object S2，Run C 之 path 又與 Run D 之任一 path 呼叫同一 shared object S3，究竟 Run A 之 path 與 Run D 之 path 是否會相互影響呢？

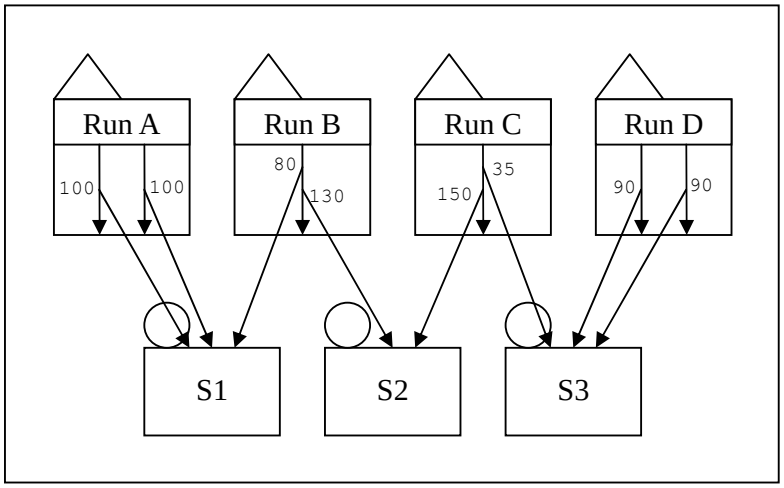

圖 3-3 存在間接相關性的系統範例

若 Run A 與 D 各選第一條 path 與 Run B 及 Run C 之 path 共組一 concurrent path，分別以 A1、B1、C1、D1 表示之。當 A1 之第 100 個 statement (記為 A1[100]) 對 S1 的呼叫將對共同屬性變數作寫入動作，而 B1[80]對呼叫 S1 則對此共用屬性變數作讀取動作；而對於 S2，B1[130]有寫入共同屬性變數的動作，C1[150]則有讀取共同屬性變數的動作；若 A1[100]較 B1[80]早執行，且 B1[130]較 C1[150]早，則透過存取 S1，使 A1[100]將資料流往 B1[80] (A1[100] → B1[80])，同一條 path 中的 B1[80]將資料流往 B1[130] (B1[80]→B1[130])，再透過存取 S2 將資料流往 C1[150] (B1[130] → C1[150])，因此會有一個資料流 A1[100] → B1[80] → B1[130] → C1[150]，因此 A1，B1 及 C1 之關係可表示為 A1 → B1 → C1。

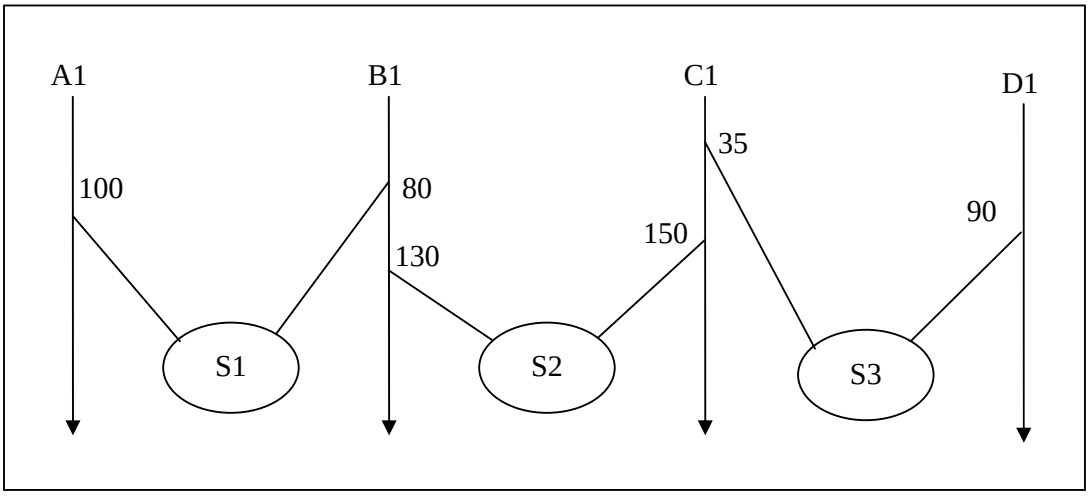

圖 3-4 四條 paths 間的間接相關性

A1，B1，C1 與 D1 等四條 paths 之中所有可能的資料流包括(1) A1[100] → B1[80] → B1[130] → C1[150]，(2) B1[80] → A1[100]，(3) C1[150] → B1[130]，(4) C1[35] → D1[90]，與(5) D1[90] → C1[35] → C1[150] → B1[130]；因此其依存性關係為(1) A1 → B1 → C1，(2) B1 → A1，(3) C1 → B1，(4) C1 → D1，(5) D1 → C1 → B1。換言之， A1 與 D1 皆不會影響彼此的執行。有依存關係之 path 組合需被測試，才能驗證其交互行為之正確性。

依存性 A1 → B1 → C1 之交互行為若用包含 A1B1C1 這三條 paths 的 concurrent path 來測試，但此 concurrent path 也可同時測試 B1 → A1 與 C1 → B1 這兩個依存性。同理，依存性 D1 → C1 → B1 需以包含 B1C1D1 這三條 paths 的 concurrent path 來測試，且這條 concurrent path 亦可同時測試 C1 → D1 的依存性。由此可知，需產生包含 A1B1C1 與 B1C1D1 此二組 path 集合的 concurrent paths 來測試。

在圖 3-3 中，A1 及 A1 與 D1 及 D2 相互間不會影響彼此的執行，因此只需選出包含 A1B1C1、B1C1D1、A2B1C1 與 B1C1D2 這四組 path 集合的 concurrent paths 來測試。選擇包含此四個 Run 的 concurrent path A1B1C1D1 與 A2B1C1D2，即可驗證所有 paths 間互相影響的執行行為。

由上例可知，由不圖 path 間之資料流可找出有相互依存關係的不同 Run 之 path 組合，這些 path 組合一定要被測試，因此要找出最少數目的 concurrent path 來測試由一條 main()的 path 所呼叫並行執行的 Run 的所有行為，必需先找出有依存關係之 path 組合後，才能找出滿足(1)所有 path 皆可被測試及(2)所有有依存關係之 path 組合皆可被測試的並行路徑組合。尋找 path 組合之方法將於 3.2 節敘述，產生 concurrent path 的方法將於 3.3 節敘述。

3.2 尋找所有 paths 間依存性的方法

上節已說明不同 Run 的 paths 呼叫同一 shared object 的 method，則這些 path 間有直接依存關係，由不同的 shared object 所找出之有直接依存關係的 path 組合

間又可能再構成間接依存關係；有間接依存關係的 path 組合可能包含二個以上的依存關係之 path 組合，需將它們過濾掉，才能找出最少組合。因此，這個 phase 又分成三個步驟：(1)先找出所有因存取 shared object 而產生的直接資料流；(2)再由直接資料流所有可能的連接方式找出間接的資料流；(3)過濾多餘的依存關係 path 組合。

當我們允許存取點的動作為存或取時，不同 Run 上的 path，只要共同存取同一個 shared object，就會產生雙向的直接資料流。因此針對每一個 shared object，將存取它的 paths 兩兩配對，只要配對的兩條 paths 分屬於不同的 Run，就會在它們之間產生雙向的直接資料流。

若兩個存取點 $P_1[NOS_1]$與 $P_2[NOS_2]$對同一個 shared object 有存取動作，且 P_1 與 P_2 屬於同一個 Run，則有兩種可能性：(1) P_1 與 P_2 為同一條 path，不會因 shared object 而獲得額外其它 Run 的資料；(2) P_1 與 P_2 為同 Run 上的不同 path，由於並行程式執行時，一個 Run 僅會執行一條 path，因此 P_1 與 P_2 不可能同時被執行。由此可知，存取同一個 shared object 的 paths 屬於同一個 Run 時，不會因 shared object 而交換資料。考慮直接資料流時，應排除屬於同一個 Run 的 paths。

要找出 paths 間的資料流，首先必需對欲測試之 Java 程式碼作靜態分析，以找出所有共享類別的 Run、各個 Run 所含的 expanded paths，以及每一條 path 呼叫 shared object 的函式的 statement，而後對每一個 shared object/class，找出呼叫這個 shared object/class 的二條 paths。如圖 3 - 5 所示，對每一個 shared object/class，建立存取點的相關資料，包括存取該 shared object 的 paths 及對應的存取點位置，以 path[NOS] (NOS, number of statement)表示之。

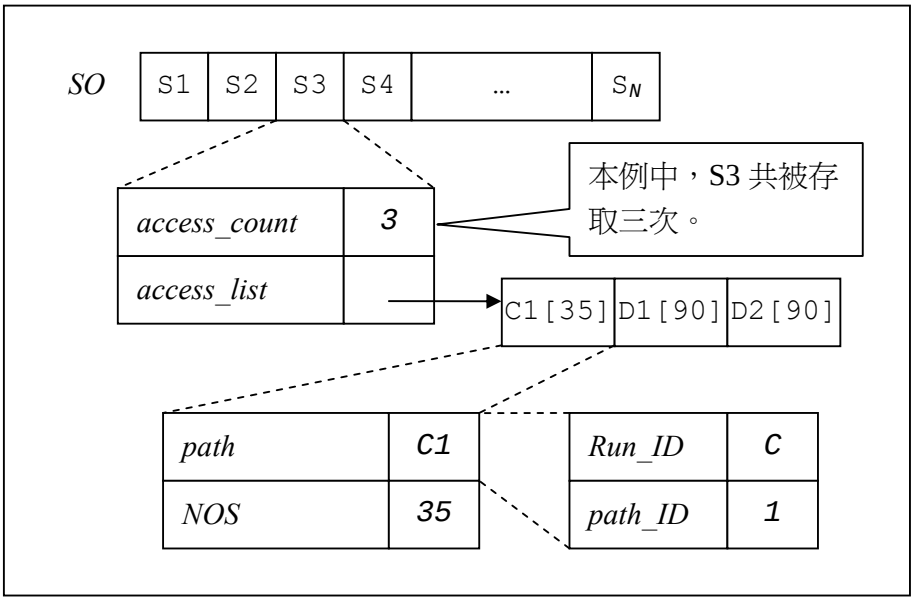

圖 3-5 shared object 資料結構示意圖

依序對一個 shared object S_i 之任二存取點 AP_j, AP_k 組成一個直接依存關係組合，並將二條資料流 $AP_j.path[AP_j.NOS] \rightarrow AP_k.path[AP_k.NOS]$ 與 $AP_j.path[AP_j.NOS] \rightarrow AP_k.path[AP_k.NOS]$ 儲存於陣列 *DF* 中，如圖 3-6 所示，儲存的內容包括這二個存取點所屬的 Run，path，及 statement number。找出直接資料流的演算法則如圖 3-7 示。

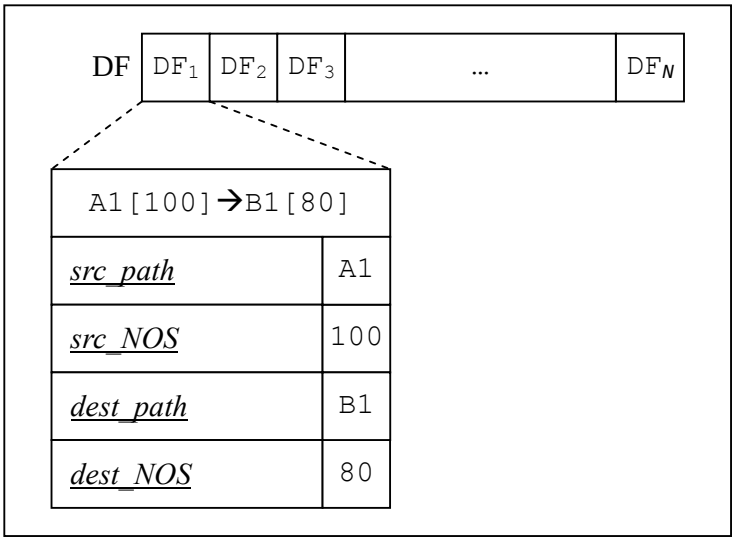

圖 3-6 直接 data-flow 資料結構示意圖

```
TARGET:
```

```
    找出所有的直接 data-flow
INPUT:
    Array SO; /* 儲存 shared object 的資訊的陣列 */
OUTPUT:
    Array DF; /* 儲存直接 data-flow 的資訊的陣列，初始為空陣列 */
ALGORITHM:
{
    for all shared object SO_i in SO
    {
        for all access point AP_j and AP_k accessing SO_i,
            0 < j < k ≤ number of access point of SO_i
        {
            if AP_j.path and AP_k.path belongs to different Runs
            {
                new direct data-flow DF_j→k with
                    DF_j→k.src_path = AP_j.path;
                    DFj(k.src_NOS = APj.NOS;
                    DFj(k.dest_path = APk.path;
                    DFj(k.dest_NOS = APk.NOS;
                add DFj(k into DF;
                new direct data-flow DF_k→j with
                    DF_k→j.src_path = AP_k.path;
                    DF_k→j.src_NOS = AP_k.NOS;
                    DF_k→j.dest_path = AP_j.path;
                    DF_k→j.dest_NOS = AP_j.NOS;
                add DF_k→j into DF;
            }
        }
    }
}
```

圖 3-7 演算法：尋找直接資料流

間接的資料流是經由二直接資料流有一相同 path 而連接產生，如圖 3-4 的

間接資料流 A1[100] → B1[80] → B1[130] → C1[150]，是由 A1[100] → B1[80]與 B1[130] → C1[150]連接而成的，第一條直接資料流之 *dest_path* 的 *NOS* 必需小於第二條直接資料流之 *src_path* 的 *NOS*。兩者的關係可以類似 BNF 的表示法表示如下：

```
DIRECT-DATA-FLOW ::=
    PATH[NOS]→PATH[NOS]

INDIRECT-DATA-FLOW  ::=
    DIRECT-DATA-FLOW→DIRECT-DATA-FLOW
    | DIRECT-DATA-FLOW→INDIRECT-DATA-FLOW
```

圖 3-8 直接與間接資料流之關係

據此，可使用資料流串鏈(data-flow chain)的方式來表示直接與間接的資料流，如圖 3-9。

```
DATA-FLOW-CHAIN  ::=
    DIRECT-DATA-FLOW
    | DIRECT-DATA-FLOW→DATA-FLOW-CHAIN
```

圖 3-9 資料流串鏈

資料流串鏈係由一直接資料流與另一條資料流串鏈連接而成，可使用 linked-list 資料結構予以表示，如圖 3-10 所示。圖中 DFC_1 表含四條直接資料流的資料流串鏈 A1[100] → B1[80] → B1[130] → C1[35] → C1[150] → D1[90] → D1[110] → E1[55]，而 DFC_4 表示僅含一條直接資料流的資料流串鏈 D1[110] → E1[55]。有 *n* 條資料流的資料流串鏈，稱其長度為 *n*。

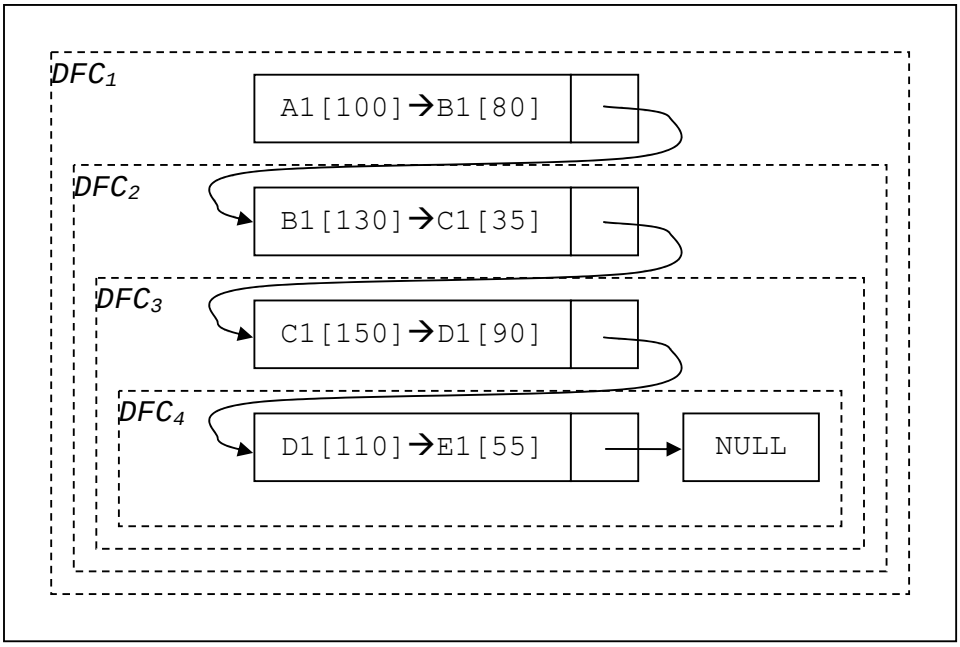

圖 3-10 資料流串鏈示意圖

將所有直接資料流依 *dest_path* 及 *dest_NOS* 排序後，便可依序從每一條直接資料流之 *src_path* 尋找可串接之直接資料流，若有找到則組成一資料流串鏈，再用串鏈第一個直接資料流之 *src_path* 去尋找可再串接之直接資料流，依此類推，即可找到所有的資料流串鏈。上列資料結構，如圖 3-11 所示，在 path B1 上有三條流入的直接資料流，*data_flow_list* 中表示流入順序。當所有間接資料流均找完後，可用圖 3-12 之資料結構來儲存，以方便過濾有包含關係的資料流，圖中 DFC_1 是一個代表資料流串鏈 A1[100] → B1[80] → B1[130] → C1[35]的 linked-list item，起始於直接資料流 A1[100] → B1[80]，並連接至另一資料流串鏈 DFC_2：B1[130] → C1[35]。在 DFC1 中以兩個陣列記錄資料流串鏈在每一個 Run 所經過的 path(*path_of_run*)，以及每條經過的 path 上最早的存取點位置(*path_NOS*)。尋找資料流串鏈的演算法如下圖 3-13 所列。

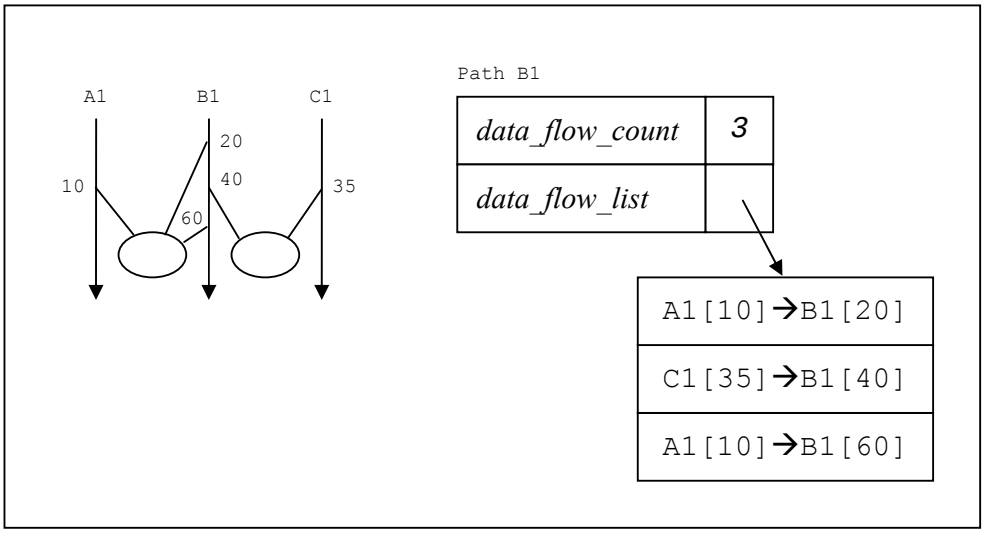

圖3-11 path 中的存取點資料結構示意圖

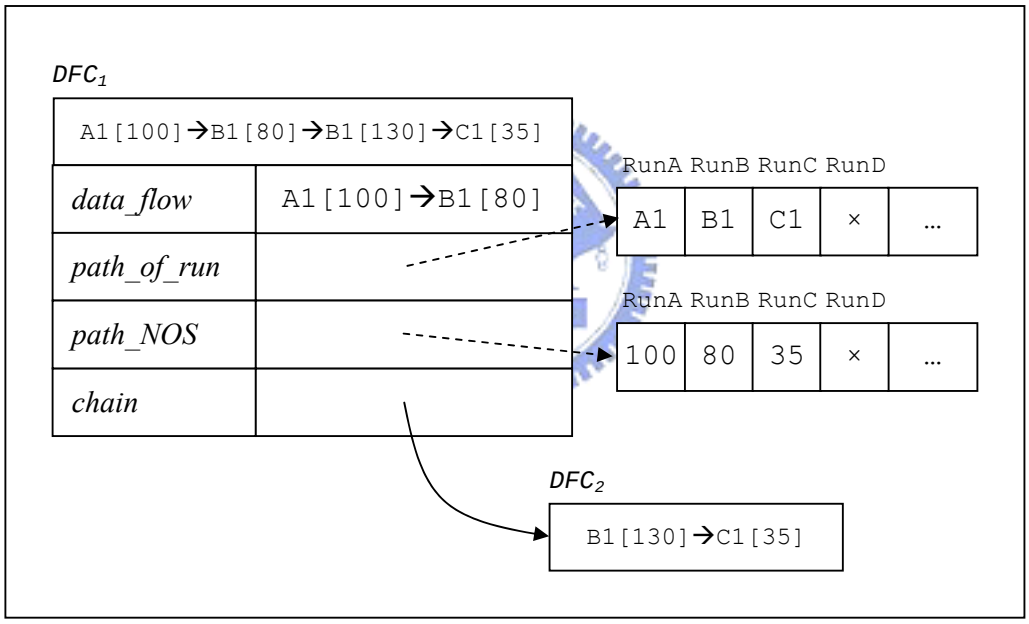

圖3-12 資料流串鏈資料結構示意圖

```
TARGET:
    找出所有的 data-flow chain
INPUT:
    Array DF; /*儲存直接 data-flow 的陣列 */
OUTPUT:
    Array DFC;     /* 每一個陣列元素 DFC^i 也都是一個陣列， */
                   /* 儲存所有長度為 i 的 data-flow chain。 */
```

```
                    /* 初始時，所有的 DFC^i 都為空陣列。 */
ALGORITHM:
{
    // 首先從直接 data-flow 產生長度為 1 的 data-flow chain
    for all data-flow DF_i in DF
    {
        new data-flow chain DFC^1_i = (DF_i → NULL);
            set all items in DFC^1_i.path_of_run to NULL;
            DFC^1_i.path_of_run[run of DF_i.src_path] =
                DF_i.src_path;
            DFC^1_i.path_NOS[run of DF_i.src_path] = DF_i.src_NOS;
            DFC^1_i.path_of_run[run of DF_i.dest_path] =
                DF_i.dest_path;
            DFC^1_i.path_NOS[run of DF_i.dest_path] =
                DF_i.dest_NOS;
        add DFC^1_i into DFC^1;
    }

    // 使用 DP 觀念，每次從長度為 i 的 data-flow chain 產生長度為
    // i + 1 的 data-flow chain。
    for(i = 1; DFC^i is not empty; i = i + 1)
    {
        for all data-flow chain DFC^i_j in DFC^i
        {
            // 連接條件(1)
            Extract source path of DFC^i_j as P;
            for all data-flow DF_k in P.data_flow_list,
                0 ≤ k < P.data_flow_count, and
                DF_k.dest_NOS ≤ source NOS of DFC^i_j
            {
                // 連接條件(2)
                if DFC^i_j has not passed any other paths belonging
                to run of DF_k.src_path {
```

```
                // 連接條件(3)
                if (DFC^i_j has not passed DF_k.src_path) or (DFC^i_j
                has passed DF_k.src_path and DF_k.src_NOS <
                DFC^i_j.path_NOS[run of DF_k.src_path]) {
                    new data-flow chain DFC^(i+1)_n = DF_k → DFC^i_j;
                    copy all items in DFC^i_k.path_of_run into
                        DFC^(i+1)_n.path_of_run;
                    copy all items in DFC^i_k.path_NOS into
                        DFC^(i+1)_n.path_NOS;
                    DFC^(i+1)_n.path_of_run[run of DF_j.src_path] =
                        DF_j.src_path;
                    DFC^(i+1)_n.path_NOS[run of DF_j.src_path] =
                        DF_j.src_NOS;
                    DFC^(i+1)_n.path_of_run[run of DF_j.dest_path] =
                        DF_j.dest_path;
                    DFC^(i+1)_n.path_NOS[run of DF_j.dest_path] =
                        DF_j.dest_NOS;
                    add DFC^(i+1)_n into DFC^(i+1);
                }}
            }
        }
    }
}
```

圖3-13 演算法：尋找資料流串鏈

每一個資料流串鏈代表的即為系統中一條可能的資料流，因此經過的 paths 都會產生依存性關係，測試時必須被包含於同一條 concurrent path 中，以保證這些 paths 同時執行的情況被測試過。針對一個資料流串鏈，其經過的所有 paths，可以形成一組有依存性關係的 paths 的集合，稱為 dependent path set。如上節圖3-4的範例，其依存性有(1) A1 → B1 → C1，(2) B1 → A1，(3) C1 → B1，(4) C1 → D1，(5) D1 → C1 → B1，可產生的 dependent path sets 為(1) {A1, B1, C1}，(2) {A1, B1}，(3) B1, C1}，(4) {C1, D1}，(5) {B1, C1, D1}。

兩個 dependent path sets 間可能有包含關係，如集合{A1, B1, C1}包含了集合{A1, B1}，若一條 concurrent path 包含集合{A1, B1, C1}，必然也會包含集合{A1, B1}，測試這條 concurrent path 可以同時驗證這兩個 dependent path sets。因此若兩個 dependent path sets 互有包含關係，可以不考慮較小的集合。

檢查兩 dependent path sets 是否有包含關係，應對兩者包含的每一條 path 進行檢查，原是相當耗時的工作。為了增進這個檢查的效率，可將 dependent path set 儲存為位元陣列的形式，每一條可能的 path 佔一個位元，若 dependent path set DPS_i 包含這條 path，則將此位元設為 1，否則設為 0。如圖 3-14 所示。

	A1	A2	A3	B1	B2	B3	C1	C2	C3	D1	D2	D3
A1B1C1	1	0	0	1	0	0	1	0	0	0	0	0
A2B2C1	0	1	0	0	1	0	1	0	0	0	0	0
A1D2	1	0	0	0	0	0	0	0	0	0	1	0
B3C2D1	0	0	0	0	0	1	0	1	0	1	0	0

圖 3-14 以位元陣列儲存 dependent path set

若 dependent path set $DPS_i \subseteq DPS_j$，對於每一條 path P，在位元陣列分別以 b_i 與 b_j 代表，數對(p_i, p_j)應為(1, 1)、(0, 0)或(0, 1)，亦即具有 $p_i \wedge p_j = p_i$ 的性質。藉由位元運算指令，我們可以很快速地判斷 DPS_i 與 DPS_j 是否具有包含關係。

```
TARGET:
    判斷 dependent path sets 是否有包含關係
INPUT:
    BitArray DPSi, DPSj; /* 兩個 dependent path set */
OUTPUT:
    Relation; /* DPSi 與 DPSj 的包含關係 */
ALGORITHM:
check_DPS_inclusion(DPSi, DPSj)
{
    make a bit-and operation between DPSi and DPSj, and store
```

```
        the result as R;
    if R = DPS_i {
        Relation = INCLUDE_POST;    /* 表示前者包含後者
                                       (DPS_i⊆DPS_j) */
    } else if R = DPS_j {
        Relation = INCLUDE_PRIOR;   /* 表示後者包含前者
                                       (DPS_j⊂DPS_i) */
    } else {
        Relation = NO_INCLUSION;    /* 表示兩者無包含關係 */
    }
    return Relation;
}
```

圖 3-15 演算法：判斷兩 dependent path set 是何包含關係

根據上述原則，產生 dependent path sets 時，應對每一個資料流串鏈 DFC^i_j 加以處理，首先利用資料流串鏈中記錄的 *path_of_run* 資訊，可擷取出資料流串鏈 DFC^i_j 所經過的 paths 的集合，將之儲存於 DPS_{new} 中。一個新找到的 dependent path set DPS_{new} 應檢查是否與已儲存在陣列 *DPS* 中的每一個 dependent path set DPS_k 有包含關係。其可能情形有三：(1) $DPS_{new} \subseteq DPS_k$，則新的 DPS_{new} 可以不予考慮。(2) $DPS_k \subset DPS_{new}$，原來的 DPS_k 可不予考慮，所以從 *DPS* 中刪除之，而 DPS_{new} 則應儲存至陣列 *DPS* 中。(3) DPS_{new} 與任一 DPS_k 皆無包含關係，則 DPS_{new} 應新增至陣列 *DPS* 中。最後產生的陣列 *DPS* 即為所有互不包含的 dependent path sets，演算法如下所列：

```
TARGET:
    找出所有的 dependent path sets
INPUT:
    Array DFC;     /* 儲存 data-flow chain 的陣列 */
OUTPUT:
    Array DPS;     /* 儲存所有 dependent path set 的陣列， */
                   /*  初始為空陣列。 */
```

```
ALGORITHM:
{
    for all data-flow chain DFC^i_j,
        1 ≤ i and DFC^i is not empty,
        1 ≤ j ≤ number of data-flow chain in DFC^i
    {
        extract paths from DFC^i_j.path_of_run and set as DPS_new;
        set ADD_NEW_DPS flag;   /* Initially assume DPS_new
                                   should be added into DPS */
        for all dependent path set DPS_k in DPS
        {
            // 呼叫前述演算法以檢查包含關係
            Relation = check_DPS_inclusion(DPS_new, DPS_k);
            // 包含情況(1)
            if Relation = INCLUDE_POST {  /* DPS_new⊆DPS_k */
                clear ADD_NEW_DPS flag;
                exit for-loop;
            }
            // 包含情況(2)
            if Relation = INCLUDE_PRIOR { /* DPS_k⊂DPS_new */
                remove DPS_k from DPS;
            }
        }
        // 在包含情況(2)與(3)皆應將 DPS_new 加入 DPS 中
        if flag ADD_NEW_DPS is set {
            add DPS_new into DPS;
        }
    }
}
```

圖 3-16 演算法：尋找 dependent path sets

經過上述處理可找出所有 dependent path set，一 dependent path set 代表這些分屬不同 Run 的 paths 有相互影響關係，需被同時測試。

3.3 藉由 dependent path set 選取測試單元 concurrent path

經上一階段的處理，已得到一些有依存關係的 path 組合，每一 path 組合為一 dependent path set。一個 dependent path set 的 path 均屬不同 thread。

本階段則是利用使用這些有依存關係的 path 組合以及沒有呼叫 shared object 的 independent path 以組合產生最少數目的 concurrent path 來建構一條 main() 的 path 應測試的 concurrent paths。

一個 concurrent path 由參與 concurrent execution 的每一個 Run 之一條 path 所組成，前面所述的有依存關係的 path 組合，可能只包含部分 Run 的一條 path，因此稱為一個 partial concurrent path。一個 partial concurrent path P_1 為另一 partial concurrent path P_2 的子集合，則 P_2 含有 P_1 中所有的 Run，且這些 Run 的 path 均相同，則稱 P_2 包含 P_1。

二個 partial concurrent path P_1 與 P_2 可以互相合併而共同執行，合併的條件是 P_1 與 P_2 中屬同一個 Run 的 path 均相同。若 P_1 與 P_2 中有相同的 Run，且 P_1 與 P_2 在此 Run 中之 path 是不同的，則無法合併成一個 partial concurrent path，我們稱此 partial concurrent paths P_1 與 P_2 互斥。

但參與 concurrent execution 的 Run 中有一些 path 並沒有存取任何 shared object，這些 path 稱為 independent path，這些 path 需被測試，因此可將 independent path 視為一條 partial concurrent path，因此本階段的目的是由這些 partial concurrent paths 來產生一組最少個數的 concurrent paths，藉由這些 concurrent paths 的執行可驗證所有的 partial concurrent path 的正確性。

如圖 3-17 所示，一個 main()的 path 啟動了三個 Run：Run A、Run B 及 Run C，此一條 partial concurrent path A1B2 若要成為 concurrent path 尚缺少 Run C 的 path。選取 Run C 中 path C2 加入這個 partial concurrent path 則與 main()的 path 組合而成 concurrent path，測試此 concurrent path 即可保證原本的 partial concurrent path A1B2 會被測試。也可由 Run C 選擇 C1 path 而組成 concurrent path，但在

Run C 中選取 C2 較 C1 為佳，因為 C2 沒有存取任何 shared object，較不會對 shared object 產生副作用。

由上例可知，partial concurrent path set 的任一 partial concurrent path 均需被測試，而一條 concurrent path 可能同時測試二個以上的 partial concurrent path，因此如何挑選出可同時被測試的 partial concurrent path 組合是本階段的重點。

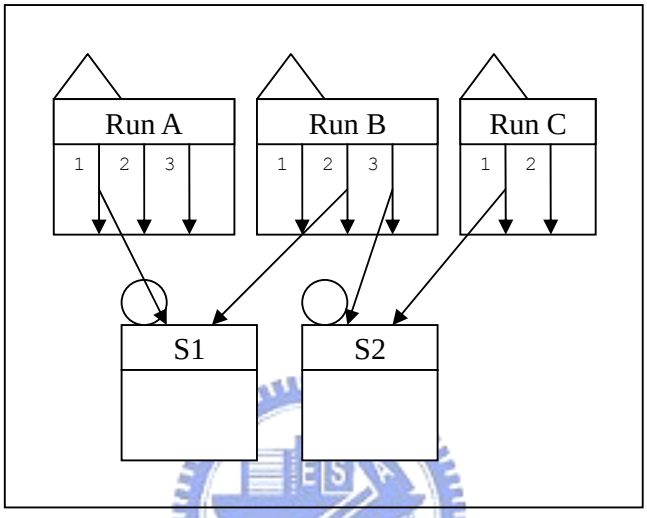

圖 3-17 partial concurrent path A1B1 應與 C2 結合

若二條 partial concurrent path 是互斥，則無法同時被一條 concurrent path 測試，因此可將這些 partial concurrent path 集合以 graph 表示，每一個 node 代表一條 partial concurrent path，若二 partial concurrent paths 互斥，則在對應的 nodes 間建立一條 edge 以表示之，以 N 代表所有 partial concurrent path 所對應的 node 的集合，E 代表各 partial concurrent paths 間互斥關係 edge 的集合，這組 partial concurrent path set 即可用 graph $G = (N, E)$ 來代表。由 G 選出沒有 edge 相連的 nodes 形成 node partition，這些 nodes 所對應的 partial concurrent paths 即可合併成一個新的 partial concurrent path 而同時測試。因此尋找最少組數的 partial concurrent path 的問題可視為將圖 G 切割為最少組數不含 edge 的 node partition。

將同一個 partition 的 nodes 著以同樣顏色，不同 partition 的顏色不重複，有 edge 相連的相鄰 node 都著上了不同顏色，形成圖論中 graph coloring 問題，因此

「尋找最少組合之可同時被測之 partial concurrent path set 組合」與「在 G 上找出最少顏色的著色方法」的問題相等。在圖論中，graph coloring 問題的最佳著色法已被證明為 NP-Completed 問題，換言之，對於一個現有的 partial concurrent path 集合，無法找出一個 polynomial time 運算時間的方法使其合併為最少數目之兩兩互斥 partial concurrent path，因此只能找出一個有效的演算法，使其在有限運算次數與時間內找出趨近於最佳解的方法。

為了實現這個方法，我們觀察 partial concurrent path 合併的性質。

Partial concurrent path set 中可以兩兩合併的對象可能有很多，一 partial concurrent path 也可能與多個其它 partial concurrent paths 可以結合，但每一次的合併所產生的 partial concurrent path 勢必影響下一階段的合併動作之 partial concurrent path 選取，因此可選擇還可作更多組合的一組來合併；由一 partial concurrent path 的集合中選擇兩個 partial concurrent path 來結合，所產生的新 partial concurrent path set 的元素個數將比原來的集合少一，再由這個新的集合中挑選兩個來結合，直到這集合中所有 partial concurrent path 皆兩兩互斥為止，因此可採用 stage-by-stage 的方式，使其經過的 stage 愈多，則集合中的元素個數愈少，代表最後所得之互斥 partial concurrent path 之數目愈少，愈有機會找出最佳解。

每一個 stage 可作合併的 partial concurrent path 組合可能很多，應選擇可使新組合有最多合併方式的那一種較為恰當，這樣能在下階段有更多機會進行合併，較易使組合的 stage 之次數增多，因此可採用 gaming tree 之 look ahead 方式來挑選當前最佳的結合方式，針對每一種合併方法，先產生下一階段其可能發生的組合情況，並計算其可能的組合數，進而選出下階段可產生最多組合數的合併方法，如圖 3-18 所示。

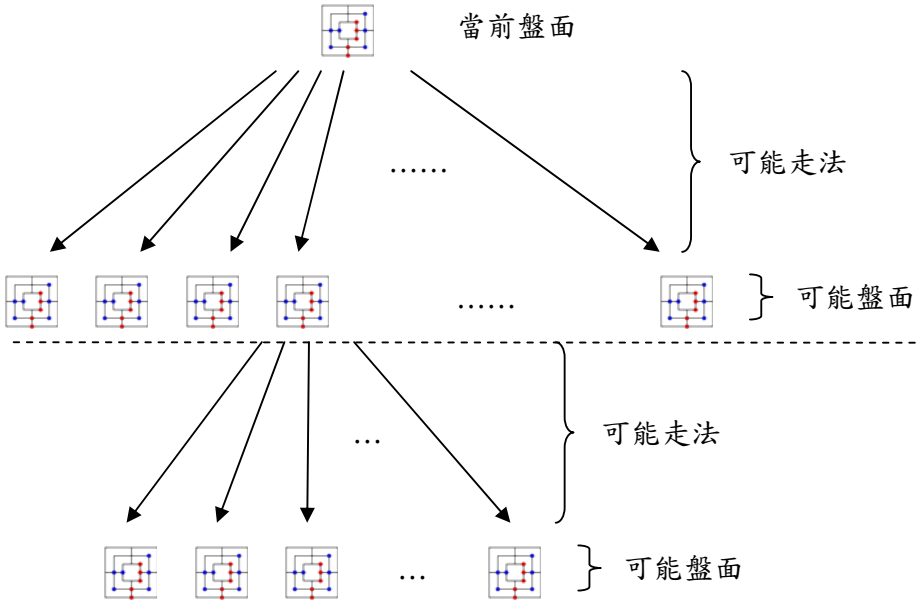

圖 3-18 gamming tree 示意圖

當 partial concurrent path set 中任二 partial concurrent path 皆為互斥時，即對每一個 partial concurrent path 分別產生可包含所有 Run 的 concurrent path，測試這些 concurrent path，即可保證各個 independent path 及所有 shared object 的各種組合情況皆可被測試。

若一 stage 有 N 個 partial concurrent path，有 M 種結合方式，則下一個 stage 將有 $N-1$ 個 partial concurrent path，利用 look ahead 方法需一一比對這 $N-1$ 個 partial concurrent path 兩兩是否可以合併；比對 $N-1$ 個 partial concurrent path 有多少合併組合需要 $C_2^{N-1} = (N-1)(N-2)$ 次比對，時間複雜度為 $\theta(N^2)$；有 M 個候選結合，因此選出最佳走法所需之時間複雜度為 $\theta(M \cdot N^2)$，計算量非常龐大，實有必要減少其計算量。若能由當前 stage 的狀態以少量計算推算出下一步驟較有利的結合，而不必算出一個結合的評分，則可減少許多計算量。

假設一組 partial concurrent path set 中，二個 partial concurrent path P_1 與 P_2 可合併為 partial concurrent path P'，新產生的 partial concurrent path 集合比原 partial concurrent path 集合少了 P_1 與 P_2 而多出 P'，P' 與此集合中其它任一 partial

concurrent path 是否可合併可由 P_i 與 P_1 及 P_2 之關係推論如下列四種情形：

1) 若 P_i 可與 P_1 合併但不可與 P_2 合併。

   P' 包含 P_2 的所有元素且 P_i 與 P_2 互斥，因此 P_i 將與 P' 互斥。

2) P_i 可與 P_2 合併但不可與 P_1 合併。

   同第一種情況，P_i 與 P' 互斥。

3) P_i 與 P_1 及 P_2 不能合併。

   由上二種情況可推論出 P_i 與 P' 互斥。

4) P_i 可跟 P_1 及 P_2 合併。

   P' 包含 P_1 及 P_2 之所有元素，而 P_i 不會與 P_1 及 P_2 互斥，故 P_i 將不會與 P' 互斥。

假設 N 個 partial concurrent path 中有 M 個組合情況，若二個 partial concurrent path P_1 及 P_2 已合併為 P'，因 P_1 與 P_2 已不存在，故 M 個組合將減少一，再由任一 P_i 與 P_1 及 P_2 之關係，可看出對 M 之影響。根據上述關係，可推論 P_i 在新 partial concurrent path 集合與其他 partial concurrent path 之合併數目可推論如下：

P_i 可與 P_1 結合	P_i 可與 P_2 結合	P_i 可與 P' 結合	對新集合中可組合數目之影響	
✓	×	×	-1	刪除 $P_i \bullet P_1$
×	✓	×	-1	刪除 $P_i \bullet P_2$
×	×	×	不變	
✓	✓	✓	-1	刪除 $P_i \bullet P_1$ 刪除 $P_i \bullet P_2$ 加入 $P_i \bullet P'$

圖3-19 P_i 與 P_1 及 P_2 的關係對新集合可組合情況的影響

其可組合數目可用下式表示：

$$score(P_1 + P_2) = (M-1) - \sum_{i}^{N} F(P_i, P_1, P_2)$$

$$F(P_i, P_1, P_2) = \begin{cases} 0 & if\ P_i\ cannot\ be\ combined\ with\ P_1\ nor\ P_2 \\ 1 & if\ P_i\ can\ be\ combined\ with\ P_1\ or\ P_2 \end{cases}$$

由於只檢查任一 P_i 與 P_1 及 P_2 之關係，只需 $2\times(N-2)$ 次比對即可，故 M 個

結合候選需要 $\theta(M \cdot N)$ 的時間複雜度，並且比對僅需對可結合矩陣進行查表動作，較原本需要檢查每個 Run 是否有不同的 path 快上許多。在選擇了一個結合之後，下一個 stage 所需的可結合方式以及可結合矩陣，也可以利用上表，從這個 stage 的可結合方式以及可結合矩陣產生，只需 $O(N)$ 的時間。

Partial concurrent path 的儲存方式如圖 3-20 所示，假設系統中存在有 RunA 至 RunN，則可使用一個陣列儲存 partial concurrent path 中的所有 path，陣列中每一個 item 儲存這個 partial concurrent path 在每個 Run 所包含之 path 的 index，例如在 RunA 的 item 中儲存 1 則表示這個 partial concurrent path 有 A1 這條 path。倘若 partial concurrent path 不包含某個 Run 的 path，則儲存 0，如圖 3-20 RunC 與 RunD 的 paths 皆未包含在 PCP_i 中。因為 partial concurrent path 最多只會在每個 Run 含有一個 path，不會無限制的增長，因此使用陣列而不使用 linked-list，可以省去配置記憶體的麻煩。

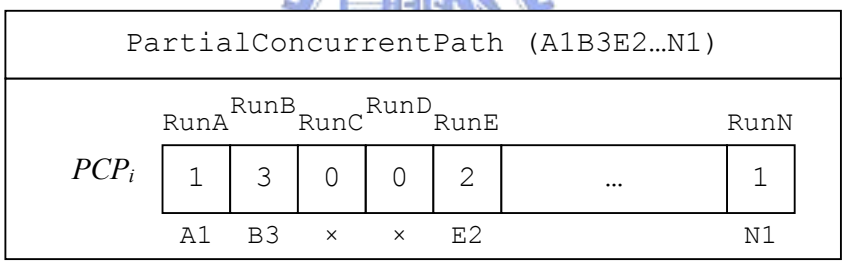

圖 3-20 partial concurrent path 以陣列方式儲存

為了組合 partial concurrent path，因此在進行這個演算法之前會需要先產生所有初始的 partial concurrent path，包括第一階段產生的 dependent path set 所組成的 partial concurrent paths 與來自於 independent paths 的 partial concurrent paths。將這些 partial concurrent path 存在陣列 $Array_P$ 中，共有 N 個，作為這個運算方法的輸入資料。

為了快速比對兩 partial concurrent path 可否合併，第一個 stage 應將 N 個 partial concurrent path 實際兩兩比對，若可以合併，則將兩個 partial concurrent path 在陣列 $Array_P$ 中的索引儲存到陣列 C 中，並將這個組合在陣列 C 中的索引存在

可結合矩陣 A 相對應的欄位裡，或是以-1 表示兩個 partial concurrent path 互斥不能合併，如圖 3-21 所示，儲存這些資料可以使刪除陣列 C 中的組合的動作較快，並且矩陣 A 是一個左下與右上互相對稱的矩陣，如此便於表格的查詢。

在每個 stage 一開始，會對 C 中所有的組合都計算分數，找出最佳的組合。假設最佳的組合是 $P_1 \bullet P_2$，則將這兩個 partial concurrent path 合併產生新的 partial concurrent path P'，暫存於 $Array_P$ 的末端，接著利用圖 3-19 的推論，查詢可結合矩陣 A，推論出 P' 與每一個 P_i 可否可以合併，以及組合方式的改變，自陣列 C 中刪除或增加組合，並將與 P' 相關的可結合矩陣暫存於矩陣 A 最末 Row 與 Column，如圖 3-21 所示。最後才自陣列 $Array_P$ 刪除 P_1 與 P_2。

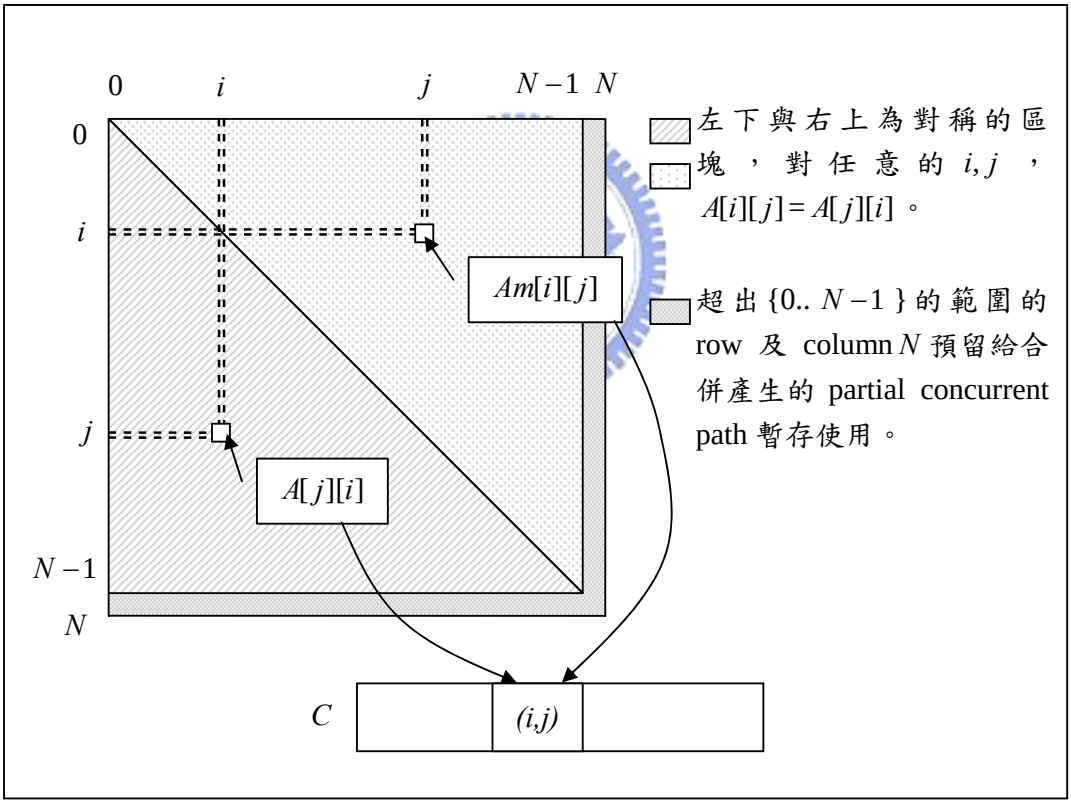

圖 3-21 可結合矩陣 A 與組合方式陣列 C 的關係示意圖

因為第一個 stage 有 N 的 partial concurrent path，再加上預留給 P' 的位置，矩陣 A 所需的空間為 $(N+1)\times(N+1)$。

所有的刪除動作均使用將陣列最後一個元素搬移到欲刪除的位置，再捨棄最

後一個元素的方式快速進行。當所有 stage 結束後再為所有的 partial concurrent path 產生 concurrent path。所有處理的演算法如下所列：

```
TARGET:
    組合 partial concurrent path 並產生最後的 concurrent path
IPNUT:
    Array  Array_P;    /* partial concurrent path set */
OUTPUT:
    Array  Array_P;    /* 最終的 concurrent path set */
INTERMEDIATE DATA:
    Integer   N;  /* N 為 partial concurrent path 的數量 */
    Integer   M;  /* M 為每一個 stage 可結合的方式的數量 */
    Matrix A;  /* 可結合矩陣 A */
    Array  C;  /* 記錄可結合方式的陣列，每一元素為兩個可相結合的
                  partial concurrent path 的索引產生的數對 */
ALGORITHM:
{
    N = number of item in Array_P;
    allocate an integer matrix A with size (N+1)×(N+1);
    /*
    初始化第一個 stage 的可結合矩陣，同時將所有可合併的組合
    存在陣列 C 中，以變數 M 計算其個數。
    */
    initialize M = 0;
    for all partial concurrent path pair P_i, P_j in Array_P,
        0 ≤ I < j < N
```

```
{
    if Pi can be combined with Pj {
        // 能合併則將數對(i,j)存入 C 中
        add (i,j) into C with index M;
        // 將索引存在可結合矩陣 A 中
        A[i][j] = A[j][i] = M;
        increase M by 1;
    }
    else
    {
        // 若不能合併則在矩陣 A 中填入代表不合法的索引-1
        A[i][j] = A[j][i] = -1;
    }
}

/*
若尚有可以合併的 partial concurrent path，
則開始一個 stage 的選取。
*/
while M > 0
{
    // 尋找最大分數的組合方式，因為分數必≥0，故最大值初始化為-1
    initialize maximum score as -1;
    for all combination Ct in C
    {
```

```
    // 提取出其 partial concurrent path 索引的數對(i,j)
    extract Ct as (i,j);
    // 依公式計算每一個組合的分數
    score = M - 1;
    for all partial concurrent path Pk in ArrayP,
        0 ≤ k < N
    {
        if (A[i][k] ≥ 0) or (A[j][k] ≥ 0) {
            /* 若可結合矩陣存的值≥0 則表示可結合 */
            // 依之前推論，此時分數應減 1
            score = score - 1;
        }
    }
    // 檢查是否為目前最大分數
    if score is bigger than maximum score {
        update maximum score to score;
        Cmax = Ct;  // 將擁有最大分數的組合記錄在變數 Cmax 中
    }
}
//  組合 Cmax 擁有最大分數，這個 stage 決定使用這個組合
extract Cmax as (i,j);
lookup index i and j in ArrayP as Pi, Pj;
combine Pi and Pj into partial concurrent path P';
// 將 P'暫存在 ArrayP 的最後面, index = (N-1)
add P' into ArrayP with index N,
```

```
increase N by 1;
/*
根據之前的推論更新結合候選陣列 C 並計算 P'的可結合情形
*/
// 從 C 刪除 Pi 與 Pj 的組合，應一併更新矩陣 A，演算法後述
delete Cmax from C;
for all partial concurrent path Pk in ArrayP,
    0 ≤ k < N, i ≠ k, j ≤ k
{
    if (A[i][k] ≥ 0) and (A[j][k] ≥ 0) {
        // 將 Pk 與 P'的組合加進組合候選中
        add (k, N-1) into C with index M;
        A[k][N-1] = A[N-1][k] = M;
        increase M by 1;
    }
    else {
        // 標記 Pk 與 P'不能合併
        A[k][N-1] = A[N-1][k] = -1;
    }
    if A[i][k] ≥ 0 {
        delete index A[i][k] from C;
    }
    if A[j][k] ≥ 0 {
        delete index A[j][k] from C;
    }
```

```
        }
        /* 刪除 Pi 與 Pj，應一併搬移可結合矩陣 A，演算法後述 */
        delete Pi and Pj from ArrayP;
    } // stage 結束，開始下個 stage

    /*
    已無可以合併的組合，由 partial concurrent path 產生 concurrent
path
    */
    for all partial concurrent path Pi in ArrayP
    {
        /*
        每個 partial concurrent path Pi 是一個陣列，
        儲存每個 Run 的 path 的索引，或以 0 表示該 Run 沒有 path，
        則應選擇該 Run 中存取 shared object 最少次的 path，
        以減少測試時的負擔。
        */
        for all Run in system Runr
        {
            if Pi[r] = 0 {
                Pi[r] = path index that access shared objects
                            fewest times in Runr;
            }
        }
    }
```

```
}
```

```
TARGET:
    自組合候選陣列 C 中刪除一個組合，並更新相關的資料
INPUT:
    Integer   index; /* 欲刪除的組合在陣列 C 中的索引 */
    Array     C;  /* 組合候選陣列 C */
    Integer   M;  /* 組合候選數目 */
    Matrix    A;  /* 可結合矩陣 */
OUTPUT:
    Array     C;  /* 更新後的組合候選陣列 C */
    Integer   M;  /* 更新後的組合候選數目, 應為輸入的 M - 1 */
    Matrix    A;  /* 更新後的可結合矩陣 */
ALGORITHM:
{
    // 取出組合數對(i,j)
    extract C[index] as (i,j);
    // 在可結合矩陣 A 中標記這兩個 partial concurrent paths
    // 的組合已不存在
    A[i][j] = A[j][i] = -1;
    // 只有在被刪除的組合不是最後一個才需做搬移的動作
    if index < M - 1 {
        // 將最後一個組合搬到欲刪除的位置上
        C[index] = C[M - 1]
        //更新被搬移的這個組合在可結合矩陣 A 上儲存的資料
        extract C[index] as (i,j);
```

```
        A[i][j] = A[j][i] = index;
    }
    // 已刪除一個組合
    decrease M by 1;
}
```

```
TARGET:
    自陣列 ArrayP 中刪除一個 partial concurrent path，
    並更新相關資料
INPUT:
    Integer   index; /* 欲刪除的 partial concurrent path
                        在 ArrayP 中的索引 */
    Array     ArrayP;    /* partial concurrent path set */
    Integer   N;  /* 目前 partial concurrent path 的數量 */
    Matrix    A;  /* 可結合矩陣 */
    Array     C;  /* 結合候選陣列 */
OUTPUT:
    Array  ArrayP;    /* 更新後的 partial concurrent path set */
    Integer   N;  /* 刪除後 partial concurrent path 的數量，
                     應為輸入的 N - 1 */
    Matrix    A;  /* 更新後的可結合矩陣 */
    Array     C;  /* 更新後的結合候選陣列 */
ALGORITHM:
{
    // 只有在被刪除的 partial concurrent path
    // 不是最後一個時才需搬移
```

```
if index < N − 1 {
    // 搬移可結合矩陣 A 中對應的 row 與 column
    Array_P[index] = Array_P[N − 1];
    for all other partial concurrent path P_i in Array_P,
        0 ≤ i < N − 1
    {
        if i = index {
            // 自己跟自己不能合併
            A[i][i] = -1;
        } else {
            A[i][index] = A[i][N − 1];
            A[index][i] = A[N − 1][i];
                // 若有個組合使用了被搬移的 partial concurrent
                // path，則應更新它所儲存的索引
            if A[i][index] ≥ 0 {
                extract C[A[i][index]] as (x,y);
                if x = N − 1 { x = index; }
                if y = N − 1 { y = index; }
                store new (x,y) back to C[A[i][index]];
            }
        }
    }
}
// 已刪除一條 partial concurrent path
decrease N by 1;
```

```
}
```

3.4 圖形著色問題

圖形著色(Graph Coloring)問題自被人提出至今，已有許多解決方法被提出來，但由於其為 NP-Completed 問題，因此現存的解法皆為試探性質(heuristic)，無法證明為最佳解。評價這些演算法，乃以現有的公認具代表性質的圖[13]，或利用 vertex 數量與 edge 存在機率為參數所隨機產生的圖[14]，視演算法產生的結果做為評斷依據。

上節已證明組合 partial concurrent path 產生最少數量 concurrent path 相當於圖形著色問題，並提出以 gamming-tree 中的 look-ahead 概念加以解決。這個演算法亦能直接使用於一般的圖形著色問題。

目前圖形著色問題的演算法，最常使用的概念，先將部份圖形著色，每次增加一個 vertex 著上合法顏色，直到完整的圖都被著色為止，這個概念稱為 successive augmentation[15][16][17][18]。這類演算法通常步驟簡單，將 vertex 加以排序後循序著上合法的顏色即可。Smallest-Last 演算法(SLC)[19]是一般實作上常用的演算法，基本精神為在一圖 G 中，degree 最小的 vertex v_k 最後著色，扣除 v_k 後產生圖 $G'=G-v_k$，繼續在 G' 中尋找 degree 最小的 vertex，直到原圖中所有的 vertex 著色順序皆決定。依此順序每次為一個 vertex 著上一種已使用的顏色，或已使用的顏色皆被該 vertex 的某個鄰近 vertex 使用，則著上一個新的顏色。這個演算法僅需 $O(|V|+|E|)$ 的時間複雜度，其實作簡單且複雜度極低的優點，使它被廣泛使用。

其它演算法中，最常在相關研究中被拿來做為比較標準的是 XRLF 演算法[20]，因其在現有的演算法中，可找到極佳的解，但相對複雜度也較高。圖 3-22 為 XRLF 演算法的基本架構。

1. Call READ_INSTANCE() to read input, compute an upper bound c^* on the optimal solution value, and return the average neighborhood size N.
2. Call INITIAL_SOLUTION() to generate an initial solution S and return $c = cost(S)$.
3. Choose an initial temperature $T > 0$ so that in what follows the *changes*/*trials* ratio starts out approximately equal to *INITPROB*.
4. Set *freezecount* = 0.
5. While *freezecount* < *FREEZE_LIM* (i.e., while not yet "frozen") do the following:

   5.1 Set *changes* = *trials* = 0.
   While *trials* < *SIZEFACTOR*·*N* and *changes* < *CUTOFF*·*N*, do the following:

   5.1.1 Set *trials* = *trials* + 1.
   5.1.2 Call NEXT_CHANGE() to generate a random neighbor S' of S and return $c' = cost(S')$.
   5.1.3 Let $\Delta = c' - c$.
   5.1.4 If $\Delta \le 0$ (downhill move),
   Set *changes* = *changes* + 1 and $c = c'$.
   Call CHANGE_SOLN() to set $S = S'$ and, if S' is feasible and $cost(S') < c^*$, to set $S^* = S'$ and $c^* = cost(S')$.
   5.1.5 If $\Delta > 0$ (uphill move),
   Choose a random number r in [0,1].
   If $r \le e^{-\Delta/T}$ (i.e., with probability $e^{-\Delta/T}$),
   Set *changes* = *changes* + 1 and $c = c'$.
   Call CHANGE_SOLN().

   5.2 Set $T = TEMPFACTOR \cdot T$ (reduce temperature).
   If c^* was changed during 5.1, set *freezecount* = 0.
   If *changes*/*trials* < *MINPERCENT*, set *freezecount* = *freezecount* + 1.

6. Call FINAL_SOLN() to output S^*.

圖 3-2 2 XRLF 演算法[20]

XRLF 演算法會先產生一個著色法作為初始解法。其做法是先尋找一組不相鄰的 vertex 集合，標上一種顏色，並且重複這個動作直到 vertex 皆已著色為止。尋找最大的不相鄰 vertex 集合本身即是 NP-hard 問題，XRLF 先隨機選取一個 vertex，然後從不與已選取之 vertex 相鄰的 vertex 中選取一個 vertex 加入集合中，直到可選擇之 vertex 少於一定量，則採列舉的方法尋找局部最大的不相鄰 vertex 集合。XRLF 演算法複雜處在於利用前述方法產生一種著色方法後，再改變部份的著色方式，產生相臨圖(neighbor graph)，使全圖著色方式的花費(cost)降低，直到最後花費無法再降低為止，這個做法稱為 annealing。尋找相鄰圖的實作方法會影響整個演算法的複雜度，如使用 2-swap 方法，亦即相鄰圖間至多僅有二個 vertex 改變顏色，則會使每次尋找較佳相鄰圖需耗時 $O(|V|^2)$，而尋找次數有 $O(|V|^2)$

次。由此可知，使用 k-swap 則會使其時間複雜度成為 $O(|V|^{2k})$，愈高的 k 值可使 XRLF 演算法找出的解愈接近最佳解，但所需的時間複雜度也相對提高許多。

本研究所提之演算法，套用在圖形著色問題上，提出以 look-ahead 的概念尋找可出著相同顏色的 vertex，加以合併。根據上節的說明，本演算法的時間複雜度為 $O(|V|^2(|V|^2-|E|))$，或 $O(|V|^4)$。以 DIMACS 網站[13]提供之目前具代表性的圖加以實驗，其數據如圖 3-23 所列。表中 k 代表此圖的最佳解，N 為本研究所提演算法所得解，前述兩種現有演算法所得結果分別列於其後，其中 XRLF 演算法以 6-swap 實作，其時間複雜度為 $O(|V|^{12})$。圖 3-24 列出三種演算法在隨機圖形 125 個 vertex 以及 edge 出現機率為{0.1, 0.5, 0.9}的情況下的表現。

filename	k	N	SLC	XRLF
flat300_20_0	20	38	46	20
flat300_26_0	26	38	47	28
flat300_28_0	28	39	46	32
fpsol2.i.1	65	65	65	65
fpsol2.i.2	30	30	30	30
fpsol2.i.3	30	30	30	30
inithx.i.1	54	54	54	54
inithx.i.2	31	31	32	31
inithx.i.3	31	31	32	31
latin_square_10	?	124	219	100
le450_15a	15	16	18	17
le450_15b	15	17	18	17
le450_15c	15	23	26	21
le450_15d	15	23	26	21
le450_25a	25	25	25	25
le450_25b	25	25	25	25
le450_25c	25	28	31	28
le450_5a	5	8	11	5
le450_5b	5	7	12	5
le450_5c	5	8	11	5
le450_5d	5	6	15	5
mulsol.i.1	49	49	49	49
mulsol.i.2	31	31	32	31
mulsol.i.3	31	31	32	31
mulsol.i.4	31	31	32	31
mulsol.i.5	31	31	31	31
school1	?	23	15	14
school1_nsh	?	22	21	14
zeroin.i.1	49	49	49	49
zeroin.i.2	30	30	30	30
zeroin.i.3	30	30	30	30
anna	11	11	11	11
david	11	11	11	11
huck	11	11	11	11
games120	9	9	9	9
miles1000	42	43	43	42
miles250	8	8	8	8
miles500	20	20	20	21

miles750	31	31	32	32
queen15_15	?	18	24	17
queen16_16	?	20	26	18
myciel7	8	8	8	8

圖 3-2 3 **DIMACS** 圖形著色例圖各演算法表現之比較

Parameter	**N**	**SLC**	**XRLF**
R(125, 0.1)	6.24	7.31	6
R(125, 0.5)	20.24	24.04	18.2
R(125, 0.9)	47.79	54.32	45.8

圖 3-2 4 隨機圖形各演算法表現之比較

根據上述實驗可以發現，針對一般性的圖形著色問題，可在複雜度不高的情況下有較 SLC 演算法為佳的表現。在部份案例中雖較 XRLF 略差，但大多亦能找到最佳解，且有部份案例可找到較 XRLF 為佳的著色法。顯示本研究之演算法所提，使用 look-ahead 策略與合併 vertex 之方法，確能產生不錯的解，但其選擇每個 stage 所使用合併方式的策略，則仍有改進的空間。

第 4 章 範例

本章以範例說明在各種情況下，以本研究所提之方法皆能找到較少的 concurrent paths 測試所有的執行情況，減少測試的負擔。

➢ 範例一：

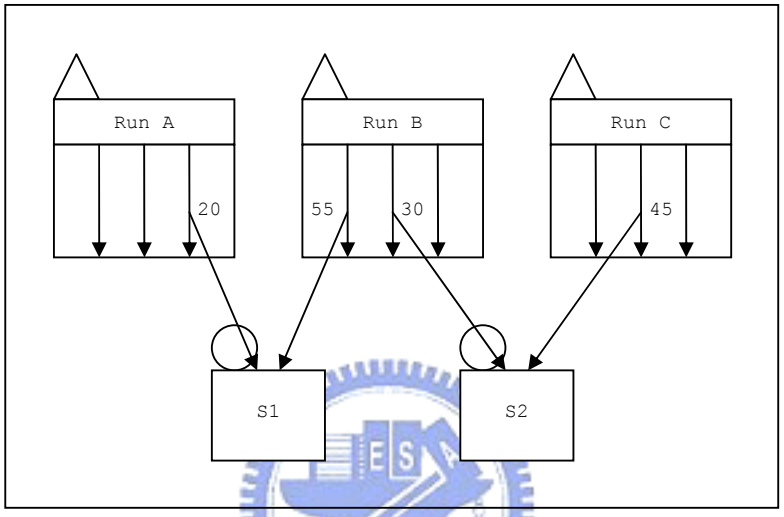

圖 4-1 範例一

範例一是一個單純的系統，由 main()產生三個 Run：Run A、Run B 及 Run C 各有三條 expanded paths，共用兩個 shared objects，如圖 4-1 所示。

根據本研究提出之方法，首先由每一個 shared object 的存取情形產生直接資料流。S1 共有兩個存取點，A3[20]與 B1[55]，分別屬於不同的 Run，兩兩配對共有 C_2^2=1 種方式，產生 $C_2^2 \times 2$=2 條直接資料流：A3[20] → B1[55]，B1[55] → A3[20]。同理，對 S2 亦有兩個存取點，產生兩條直接資料流 B2[30] → C2[45]，C2[45] → B2[30]。

上述四條資料流除統一儲存於一陣列中，尚應在流至的 path 上儲存一份參考(reference)，以供產生資料流串鏈使用，如 3.2 節所述，例如 path B1 的資料結構中會以一個陣列儲存所有流至 B1 的直接資料流，在本例中僅有 A3[20] → B1[55]一條，如圖 4-2 所示。由上可知，*N=4* 條資料流需(((直接資料流)×1+參

考×1)×4)的記憶體空間，空間複雜度為 *O(N)*，與直接資料流的個數成正比。

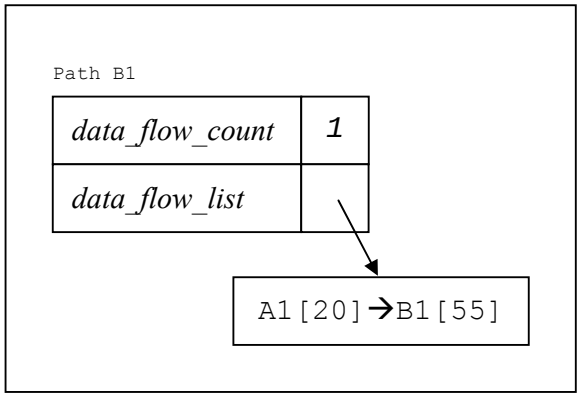

圖 4-2 範例一：path B1 上儲存所有流至 B1 的直接資料流參考

本例中並無法產生長度超過 1 的資料流串鏈，故僅需 4 個資料流串鏈 linked-list 的 item，每一個 item 含兩個陣列記錄串鏈經過的 path 與 NOS，長度均與為 Run 的數量相同。

由於系統三個 Run 共有九條 path，因此每個 dependent path set 需 9 個 bits 來儲存，上述四條資料流可產生兩組 dependent path sets，{A3, B1}與{B2, C2}，共需 18 個 bits 的記憶體空間。

本例中有許多沒有呼叫 shared object 的 paths，組成 independent path set {A1, A2, B3, C1, C3}，使用 dependent path sets 與 independent path set，可產生 A3B1, B2C2, A1, A2, B3, C1, C3 等七組 partial concurrent paths。

在組合 partial concurrent paths 時，N_1 = 7 組 partial concurrent paths 需建立 $N_1 \times N_1$ = (7+1)×(7+1)的可結合矩陣，在第一個 stage 檢查兩兩 partial concurrent paths 是否可結合，共有 $C_2^{N_1} = C_2^7$ = 42 種配對，因每個 partial concurrent path 最多包含三個 Run 的 paths，最多需比較 3 次才能確認兩 partial concurrent paths 是否互斥，因此最多需比對 42×3 次，此例中可找出 M_1 = 12 種可結合方式；計算每種結合方式的評分共需 $M_1 \times ((N_1\text{-}2)\times 2) = 12\times((7\text{-}2)\times 2)$次查表，因此最糟情況的計算次數為 $\theta(N_1 \times M_1) = O(N_1^3)$。但只有在所有的 partial concurrent paths 皆可以互相結合時，M_1 有最大值 $N_1 \times (N_1\text{-}1)$，而這種情形只可能發生在所有並行執行

個體皆只有一條 path 時。通常 M_1 有遠小於 N_0^2 的性質，因此實際計算次數也會遠小於 N_1^3。

在 42 種配對中，於其中選擇分數最大的 B2C2 與 A1 進行組合，成為 A1B2C2。第二個 stage 時 partial concurrent paths 剩下 $N_2 = 6$ 組，而可結合方式剩下 $M_2 = 7$ 種，計算分數需 7×((6-2)×2)次查表，選擇組合 A3B1 與 C1 形成 A3B1C1；第三個 stage，$N_3 = 5$，$M_3 = 3$，經 3×((5-2)×2)次查表後，組合 A2 與 B3，形成 A2B3；在第四個 stage 剩餘 $N_4 = 4$ 與 $M_4 = 1$ 結合方式，由 A2B3 與 C3 組合形成 A2B3C3，剩餘的 partial concurrent paths A1B2C2，A3B1C1，A2B3C3 都無法再互相結合。

上述每個 stage 的計算皆需 $M_i \times (N_i-2) \times 2$ 次查表動作，經過 t 個 stage 共需 $\sum_{i}^{t} M_i \times (N_i - 2) \times 2$ 次查表，當所有 partial concurrent paths 最後合成為一個時，t 最大可為 N_0-1，因此查表總計時間複雜度為 $O(N_0^4)$。另外，若有 N 個 partial concurrent paths，刪除其中之一須對可結合矩陣進行 $\theta(N)$的搬移動作，因此對該矩陣總搬移動作的複雜度為 $\theta(t \times N_0) = O(N_0^2)$；從 M 個組合中刪除一個的僅須 constant time，且 phase 2 最後必使 $M = 0$，因此其時間複雜度為 $\theta(M_0)$。

根據上列結果，採用本方法將建議測試者對 A1B2C2，A3B1C1，A2B3C3 等三條 concurrent paths 進行測試，而非傳統並行程式測試方法所需的全部 27 條 concurrent paths。測試這三條 concurrent paths 足以測試每一條 path，並能驗證透過 S1 與 S2 使不同 Run 之間的 paths 產生的互動關係。在本例中，由於每個 Run 都有三條 paths，至少必需以三條 concurrent paths 才能涵蓋一個 Run 的所有 paths，因此上述三條 concurrent paths 已是最少數量可完整測試所有執形情況的 concurrent paths。

- 範例二：

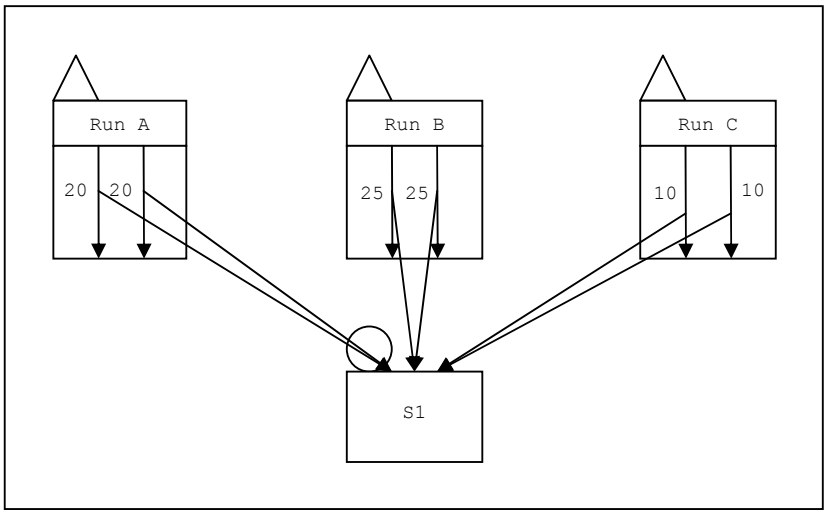

圖4-3 範例二

範例二中，所有的 paths 皆會呼叫同一 shared object，因此勢必要測試所有的 concurrent paths 方能驗證所有的互動行為。根據本研究之方法，應先找出直接資料流如下表所列：

A1[20]→B1[25]	A1[20]→B2[25]	A1[20]→C1[10]	A1[20]→C2[10]
A2[20]→B1[25]	A2[20]→B2[25]	A2[20]→C1[10]	A2[20]→C2[10]
B1[25]→A1[20]	B1[25]→A2[20]	B1[25]→C1[10]	B1[25]→C2[10]
B2[25]→A1[20]	B2[25]→A2[20]	B2[25]→C1[10]	B2[25]→C2[10]
C1[10]→A1[20]	C1[10]→A2[20]	C1[10]→B1[25]	C1[10]→B2[25]
C2[10]→A1[20]	C2[10]→A2[20]	C2[10]→B1[25]	C2[10]→B2[25]

根據上列直接資料流，可產生的間接資料流共 48 條：

A1[20]→B1[25]→C1[10]	B1[25]→C1[10]→A1[20]	C1[20]→A1[20]→B1[25]
C1[10]→B1[25]→A1[20]	B1[25]→A1[20]→C1[10]	A1[20]→C1[10]→B1[25]
A1[20]→B1[25]→C2[10]	B1[25]→C2[10]→A1[20]	C2[20]→A1[20]→B1[25]
C2[10]→B1[25]→A1[20]	B1[25]→A1[20]→C2[10]	A1[20]→C2[10]→B1[25]
A1[20]→B2[25]→C1[10]	B2[25]→C1[10]→A1[20]	C1[20]→A1[20]→B2[25]
C1[10]→B2[25]→A1[20]	B2[25]→A1[20]→C1[10]	A1[20]→C1[10]→B2[25]
A1[20]→B2[25]→C2[10]	B2[25]→C2[10]→A1[20]	C2[20]→A1[20]→B2[25]
C2[10]→B2[25]→A1[20]	B2[25]→A1[20]→C2[10]	A1[20]→C2[10]→B2[25]
A2[20]→B1[25]→C1[10]	B1[25]→C1[10]→A2[20]	C1[20]→A2[20]→B1[25]
C1[10]→B1[25]→A2[20]	B1[25]→A2[20]→C1[10]	A2[20]→C1[10]→B1[25]
A2[20]→B1[25]→C2[10]	B1[25]→C2[10]→A2[20]	C2[20]→A2[20]→B1[25]
C2[10]→B1[25]→A2[20]	B1[25]→A2[20]→C2[10]	A2[20]→C2[10]→B1[25]

A2[20]→B2[25]→C1[10]	B2[25]→C1[10]→A2[20]	C1[20]→A2[20]→B2[25]
C1[10]→B2[25]→A2[20]	B2[25]→A2[20]→C1[10]	A2[20]→C1[10]→B2[25]
A2[20]→B2[25]→C2[10]	B2[25]→C2[10]→A2[20]	C2[20]→A2[20]→B2[25]
C2[10]→B2[25]→A2[20]	B2[25]→A2[20]→C2[10]	A2[20]→C2[10]→B2[25]

在這些直接與間接資料流中，僅有 8 組相異的 dependent path sets，分別為{A1, B1, C1}，{A1, B1, C2}，{A1, B2, C1}，{A1, B2, C2}，{A2, B1, C1}，{A2, B1, C2}，{A2, B2, C1}，{A2, B2, C2}。在本例中不存在 independent paths，且八條 partial concurrent paths A1B1C1，A1B1C2，A1B2C1，A1B2C2，A2B1C1，A2B1C2，A2B2C1，A2B2C2 都不能互相結合，因此這八組 partial concurrent paths 即為建議使用者測試的 concurrent paths。

在這個極端範例中，但為了確認每一條 concurrent paths 皆存在其它 concurrent paths 無法涵蓋的執行行為，本研究所提之尋找資料流的方法，需要最大的空間與時間，方能組合出需測試的所有 path 組合，並不如傳統直接將各個 Run 的 paths 加以組合產生所有 concurrent paths 迅速，測試成本會較傳統方法略高。

➢ 範例三：

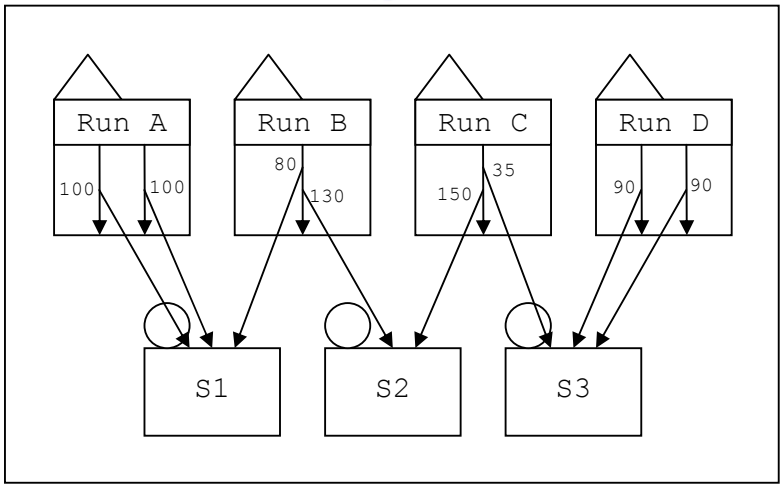

圖 4-4 範例三

在範例三的系統中，Run A 與 Run B 共用 shared object S1，Run B 與 Run C 共用 S2，Run C 與 Run D 共用 S3，看似所有 paths 間都會互相影響。但實際分

析後，可發現存在直接與間接的資料流如下所列：

A1[100]→B1[80]	B1[80]→A1[100]	A2[100]→B1[80]	B1[80]→A2[100]	B1[130]→C1[150]
C1[150]→B1[130]	C1[35]→D1[90]	D1[90]→C1[35]	C1[35]→D2[90]	D2[90]→C1[35]
A1[100]→B1[80]→B1[130]→C1[150]		A2[100]→B1[80]→B1[130]→C1[150]		
D1[90]→C1[35]→C1[150]→B1[130]		D2[90]→C1[35]→C1[150]→B1[130]		

由這些資料流，可產生的 dependent path sets 有{A1, B1, C1}，{A2, B1, C1}，{B1, C1, D1}，{B1, C1, D2}，可看出 Run A 的 paths 與 Run D 的 paths 不會互相影響。本例中亦無 independent paths，故僅有四組 partial concurrent paths A1B1C1，A2B1C1，B1C1D1，與 B1C1D2。組合 partial concurrent paths 可找出能同時測試多個 partial concurrent paths 的 concurrent paths，因此經過組合產生 A1B1C1D1 與 A2B1C1D2 兩條 concurrent paths，可涵蓋所有可能的執行情況。傳統並行程式測試方法應測試所有的 concurrent paths 共四條，而藉由資料流概念確認 Run A 與 Run D 不會互相影響的性質，可以有效地減少需測試的 concurrent paths 的數量。

➢ 範例四：

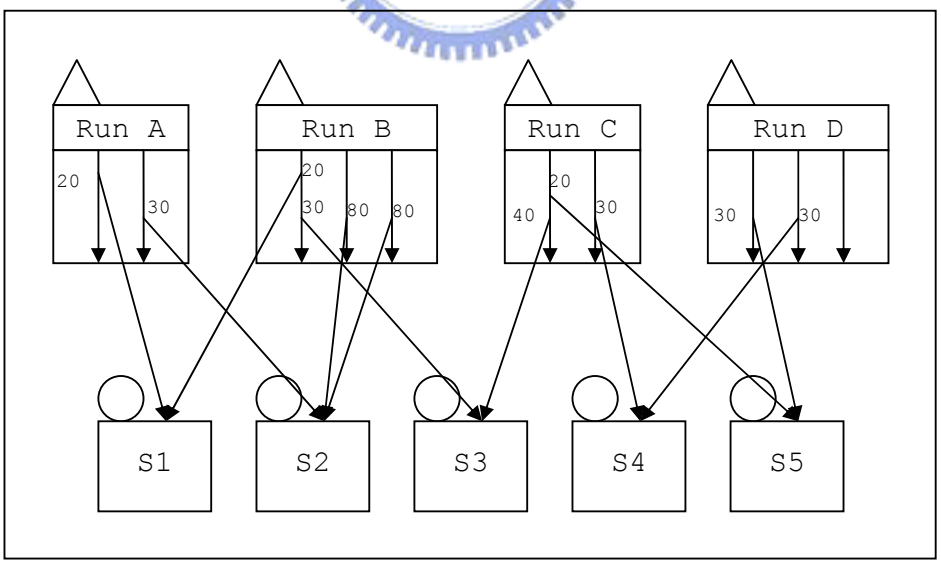

圖4-5 範例四

範例四是一個較為複雜且較符合真實一般並行 Java 程式的系統，共有四個 Run 共同使用五個 shared object。根據 shared object 的存取情形，可產生如下所

列的資料流：

A1[20]→B1[20]	B1[20]→A1[20]	A2[30]→B2[80]	B2[80]→A2[30]	A2[30]→B3[80]
B3[80]→A2[80]	B1[30]→C1[30]	C1[30]→B1[30]	C2[30]→D2[30]	D2[30]→C2[30]
C1[20]→D1[30]	D1[30]→C1[20]			
A1[20]→B1[20]→B1[30]→C1[40]		D1[30]→C1[20]→C1[40]→B1[30]		

上列 14 條資料流，共會產生五組 dependent path sets，分別為{A1, B1, C1}，{B1, C1, D1}，{A2, B2}，{C2, D2}，與{A2, B3}。由於 path D3 沒有存取 shared object，因此為 independent path。由 dependent path sets 與 independent path set 產生的 partial concurrent path 計有 A1B1C1，B1C1D1，A2B2，C2D2，A2B3，與 D3 共六組。組合時，根據分數最高優先結合的原則，第一個 stage 會先選擇組合 A1B1C1 與 B1C1D1 形成 A1B1C1D1；stage 2 組合 A2B3 與 D3，形成 A2B3D3；最後 stage 3 組合 A2B2 與 C2D2 成為 A2B2C2D2。最後所得的互斥 partial concurrent path sets 為{A1B1C1D1, A2B3D3, A2B2C2D2}等三組。其中 A2B3D3 缺少 Run C 的 path，因此應選擇呼叫 shared object 次數較少的 C2 與之結合。在本例中，所有的 concurrent paths 共有 2×3×2×3=36 條不同的可能，但依本研究之方法，會建議使用者測試 A1B1C1D1、A2B3C2D3、與 A2B2C2D2 這三條 concurrent paths，即足以涵蓋此並行程式所有執行行為的測試個案，且無法找出更少數量的 concurrent paths 完成此程式的測試。

根據上述範例可看出，無論在何種並行執行的情況下，本研究所提之方法找出的 concurrent paths 確能涵蓋所有的執行行為，測試這些 concurrent paths 即能保證每一條 path 都被測試過，不同 Run 間的互動行為也都被測試到。並且依本研究的方法找出之 concurrent paths，確能以最少數量的測試個案完整測試並行程式的執行行為。

第 5 章 結論

Java 程式語言是一物件導向程式語言，具有跨平台及多緒執行的能力，因此已有許多軟體工程師運用 Java 語言開發並行程式。輸入一組數據，重複執行一並行 Java 程式，將因平行執行單位執行順序之不同而導致產生不同結果，此種不確定行為導致並行程式之測試更為複雜，因此有必要將所有並行互動行為一一驗證，以確定並行 Java 程式之正確性。

一個 Java 程式之 main() path 可以啟動多個主動類別之 run()函式而平行執行，依據不同的輸入資料而產生多種並行執行路徑，若單由這些 run()函式之路徑來組合成不同之並行路徑，其組合數目非常龐大，測試工程師需對這些並行路徑準備測試數據，所費工夫甚為龐大，因此如何產生最少組合的並行路徑以驗證其執行行為，是一重要課題，唯目前尚無系統化方法被提出，本研究即在探討此問題。

欲驗證一組由一條 main()路徑所產生的 run()執行行為之正確性，首先要確定每一個 run()的 path 皆已被測試，其次要確定不同 run()路徑共同呼叫相同 shared object 之函式的互動行為皆要被測試，而一條 run()路徑可能呼叫多個不同 shared objects 的函式，更增加其互動行為的複雜度。本研究依據呼叫同一 shared object 之不同 run()路徑的資料交換關係，提出以直接資料流來代表其互動關係，而二直接資料流間若有相同 run()路徑，則其資料會透過共同路徑而傳至第三個 run()，此種交互影響關係稱為間接資料流，進而再與另一直接資料流交互影想而形成資料流串鏈。每一組直接與間接資料流均需被測試。

將每一個不含呼叫 shared object 函式的路徑及所有資料流串鏈(包含直接資料流及間接資料流，且任二資料流串鏈不具包含關係)，視為一個 partial concurrent path，這些 partial concurrent paths 均應被測試，這些 partial concurrent paths 有些可同時執行，有些則不能，其問題在於每一相同 run()均應選用同一條 path，因此二條 partial concurrent paths 若對同一 run()有不同 path 則稱為互斥。尋找最少

組合數量將可並行執行之 partial concurrent paths 予以組合的問題與 graph coloring 之問題相同，且無法在 polynomial time 內找出最佳解。本研究提出用 gaming tree 之 look-ahead 策略來作 partial concurrent paths 組合之尋找，其計算複雜度為 $O(N^4)$。

本研究所提演算法可套用在一般性的 graph coloring 問題，時間複雜度較 XRLF 演算法低，可以較小的計算成本，在大部份的案例中得到最佳解或趨近最佳解。雖計算複雜度較 SLC 演算法高，但表現可較 SLC 演算法出色許多。

References

[1] D. Flanagan, *Java in a nutshell*, 5th Edition, O'RELLY, 2005.

[2] A. V. Hoff, S. Shaio, and O. Starbuck, *Hooked on Java*, Addison-Wesley, 1996.

[3] P. Niemeyer and J. Knudsen, *Learning Java*, 2nd Edition, O'RELLY, 2002.

[4] B. Beizer, *Software testing techniques*, Data System Analysts Inc., Pennsanken, New Jersey, 1984.

[5] N. F. Schneidewind, "Application of program graphs and software development and testing," *IEEE Transactions on Software Engineering*, Aug. 1979, pp. 192-198

[6] 李正國, "最佳測試個案選取之研究," 博士論文, 國立交通大學資訊工程研究所, July 1990

[7] 陳家豪, "並行 Java 程式不確定執行行為之測試方法," 碩士論文, 國立交通大學資訊工程研究所, 1999

[8] R. Curtis, and L. Wittie, "BugNet: a debugging system for parallel programming environments," in *Proceedings of the 3rd International Conference on Distributed Computing Systems*, 1982.

[9] A. J. Gordon, and R. A. Finkle, "TAP: a tool to find timing errors in distributed systems," in *Proceedings of Workshop on Software Testing*, 1986, pp. 154-163.

[10] J. Gosling, B. Joy, and G. Steele, *The Java language specification*, Addison-Wesley, 1996.

[11] T. J. Cheatham and L. Mellinger, "Testing object-oriented software systems," in *Proceedings of the 18th ACM Annual Computer Science Conference*, ACM Inc., New York, 1990, pp. 161-165.

[12] J. G. Lee and C. G. Chung, "Structural testing method for C++ programs," in

Proceedings of COMPSAC'94, 1994, pp. 123-123.

[13] Michael Trick: Network Resources for Coloring Graphs, http://mat.gsia.cmu.edu/COLOR/color.html , Last revision: October 26, 1994.

[14] B. Bollobas and A. Thomason. Random graphs of small order. *Annals of Discrete Math.*, 28:47-97, 1985.

[15] Brélaz D. New methods to color the vertices of a graph Commun. ACM 22, 4 (Apr. 1979), 251-256.

[16] D.J.A. Welsh and M.B. Powell. An upper bound on the chromatic number of a graph and its application to timetabling problems. *Computer Journal*, 10:85-86, 1967.

[17] D. C. Wood. A technique for coloring a graph applicable to large scale time-tabling problems. *The Computer Journal*, 3:317-319, 1969.

[18] M. M. Halldórsson. A still better performance guarantee for approximate graph coloring. Technical Report 91-35, DIMACS, New Brunswick, NJl, 1990.

[19] D. W. Matula and L. L. Beck. Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. *Journal of the Assocmuon for Computing Machinery*, Vol. 30, No. 3, July 1983.

[20] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378-406, 1991.