第一章.　緒論

隨著科技進步、國際市場間的藩離消失及顧客的需求改變等種種因素，使企業的目標、方法以及基本的組織規則，不能再一成不變，企業不再是大量生產以滿足市場需求，而是為了滿足顧客需要而採多樣性的產品變化，為因應環境的變化，企業若沒有適當的應變策略，很可能無法生存。因此，有遠見的現代企業、政府機關都積極的思考如何才能增強自已的競爭能力，希望能將成本降至最低、並盡量縮短處理各種事件的時間，以適應瞬息萬變的社會。

而在企業、政府機關中，存在有許多的流程(business processes)，這些流程便是由組織中眾多的成員所共同參與，利用企業的資源來協力執行。組織便是透過這些流程的執行來達成組織的目標(business goal)。因此，這些流程品質的良窳，代表了組織中資源的使用方式、服務品質、競爭力的優劣。因此，如何了解、進而改善企業的流程就變得非常重要了。

所以，近年來，改造企業中的程序流程(Business Process Reengineering)受到許多企業的重視，並有許多企業紛紛投入此一行列，例如：GM、Pepsi、U.S. Sprint、AT&T、IBM 等[Mana94]。然而，流程的改造，顧及其牽動組織的範圍大，人力資源有限，必須經過多方面的評估與考量，方可成行。尤其對規模較大，動輒含有上百甚至上千個的活動流程，改變流程的策略不太能輕易實施。基於以上種種原因，如何尋求科技的幫助，使改變流程變得更為可行、有效，便成為當前重要的課題。

而工作流程管理，便是起源於利用資訊科技對工作流程的支援，目的在促進工作間的協調，使工作流程更具效率與彈性。由於電腦與電腦網路的普及，原本皆由人負責的工作流程，現在可由電腦控管，不但有效率，且更方便追蹤與修改。因此，目前有許多領域對於工作流程的相關研究、技術皆感到莫大興趣。

本研究便是著眼於利用資訊科技對流程的支援，希望藉由所搜集工作流程的各式資料，從中萃取相關資訊，以協助流程定義之修改，提昇流程的工作品質，使企業的運作便為順暢、資源利用更有效率。

第二章. 研究動機與目的

隨著全球性的發展，現今的企業必須對於變革有著快速反應的能力，以便能迅速的提供新的服務、發展新的產品，同時改善生產力、提高品質以及降低成本的花費。企業流程改造，為許多企業視為達成這些目標的一大利器。

根據 Davenport 和 Short 的定義，「流程改造」是設計並且分析組織內與跨組織的工作流程[Dave93]，主要可以分為以下 5 個步驟：

1.認清與了解企業中流程的整體目標。

2.找出需要改造的流程。

3.了解並評估現有的流程，避免犯下同樣的錯誤。

4.腦力激盪來提出新的流程。

5.測試並部署新的流程。

從上述步驟，可以很容易看出，要進行流程改造，必須先了解、找出目前既有的流程，從而進行分析、改造。然而，要找出既有的流程並不容易[Hamm93]。再者，對改造流程的工作人員而言，要找出既有的流程，通常就代表了一連串與管理者、員工、各種不同群體工作人員的許多會議與溝通協調，時間、人力成本甚為可觀。而且，不同的改造人員，所繪出的企業流程，並未標準化，可能會有許多不同之處。所以，一個有趣的問題：是不是可以藉由某些架構機制、方法演算，來自動找出組織中的既有流程呢？本研究的主要動機便是著眼於此--透過資訊科技的幫助，自動找出組織中的既有流程。

進一步思考，組織中的既有流程，可能是以人工、半自動化或自動化的方式執行，當這些流程執行時，組織裡通常會留下一些記錄，這些記錄以各種不同的方式存在。例如，以一些人工紙上作業留下記錄。當流程執行時，會有工作進度表以記錄、稽核工作的執行時間、進度，收集這些工作進度表，就可以得到目前既有流程執行的時間資料。

或者有一些專案管理工具，可以在既有流程執行的過程中，自動地留下一些事件(event)、時間的記錄。如[Brad94]的研究中，即設計了流程監督工具(process monitor)，來記錄流程執行過程中，所發生的事件及其時間。收集這些自動記錄下來的事件時間記錄，也可以得到目前既有流程執行的時間資料。在這些被記錄下來的資料裡，其實隱含著許多流程的相關資訊，如果可以透過

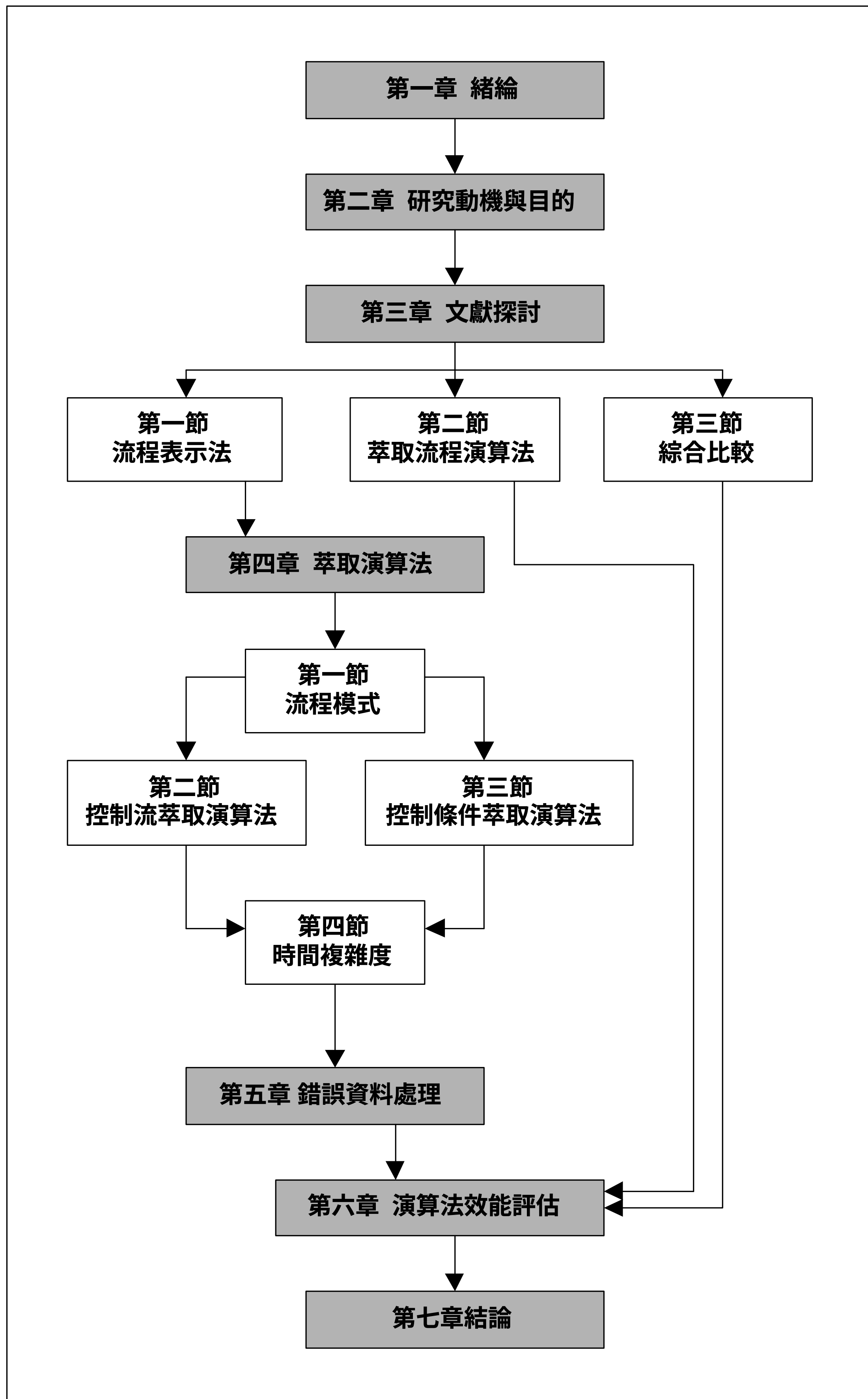
適當的方法，是不是可以由這些歷史資料中找出既有流程的定義呢？

再則，目前市面上已有一些工作流程管理系統、工具，可以依據使用者給定的流程定義，實際執行流程，當然也可以留下流程實際執行過程的相關資訊。然而，使用者一開始所定義的流程並不見得完全正確，或者隨著環境的變遷、條件的改變，實際上的流程執行已與原有流程定義不相符。這時，是不是也可以由這些工作流程管理系統、工具，所實際記錄下來的流程執行相關資訊，來找出目前既有、實際的流程定義呢？

因此，由這些人工紙上作業、專案管理工具、工作流程管理系統所記錄下來的資料中，我們應該可以收集到許多與流程相關的資訊。如何利用這些資訊，解決前述不易找出既有流程的問題，便成了本研究的目標。

回應前述的研究動機，本研究的主要目的是：利用所記錄下來的歷史資料，透過演算法的萃取，自動化地找出流程的定義，以提供專家或改造人員參考，使得流程的模式化，更為容易。對於工作人員而言，也可以在更短的時間內，以更少的人力、資源、更有效率地找出企業既有流程，以為進一步分析、改造之用。

本研究論文分為七大章。第一章與第二章，敘述研究背景與研究的動機、目的，讀者可以從中了解本研究的研究概念。第三章是相關研究，說明萃取流程定義演算法的相關文獻。第四章則闡述本研究所採用的流程模式(process model)和萃取演算法，讀者可以由此章了解本研究的方法架構與演算法。第五章則說明本研究對於錯誤資料(noise)的處理方法，以使演算法更具實用性。第六章，則以演算法實作雛形系統，並與其它演算法進行效能比較評估。第七章，總結研究成果及未來研究的方向。綜合以上所述，將各章的關係以圖 2-1 表示。



圖<2-1>

第三章. 文獻探討

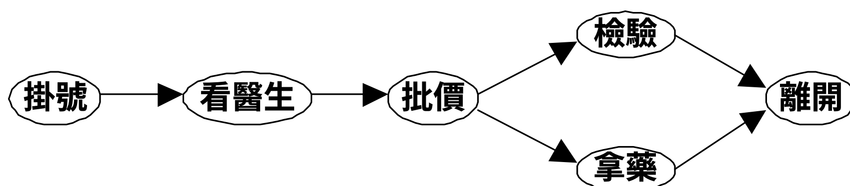
目前已有一些研究，對於流程萃取方法進行探討，但這些方法所使用的流程模式、表示法各有不同。因此，本研究首先於第一節流程表示法，敘述常見、或於不同萃取方法中，所使用的流程表示法。第二節萃取流程演算法，則分別針對這些流程萃取方法進行分析。並綜合比較各萃取方法的特性、缺點於第三節綜合比較。

第一節 流程表示法

流程表示法是用來表達某一特定流程，方便使用者了解這個流程的方法，因此流程表示法必須選取簡單、清楚的描述方式。表示方法有下列數種被提出：

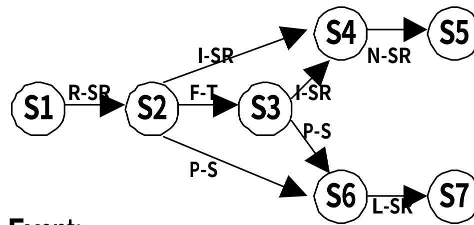
1.以溝通為基礎的表示法(Communication based Methodologies)[Geor95]：這個方法以"Action Model"為代表[Memo93]。它是將每一個流程或活動都視為是一執行者(Performer)和一客戶(Customer)間的溝通。而溝通又可分為四階段：準備、協商、處理和接受，流程中的活動全部以這四種型態來表示。

2.以活動為基礎的表示法(Activity-based Methodologies)[Geor95]：這種表示法著重在描述流程中各組成活動間的關係，而非人與人之間的溝通。如圖 3-1 是一個生病求診的流程。



圖<3-1>

3.以有限狀態圖 FSM(Finite State Machine)表示流程的方法[Cook95]：這種表示方法是由流程執行的事件(event)和狀態(state)來表示，如圖 3-2 所示。圖 3-2 是一個電腦中心提供服務給使用者的流程，圖中的 S1、S2…S7 是狀態，而如 R-SR、F-T…L-SR 便是事件。其中，若流程位於狀態 S1，在事件 R-SR 發生後，會改變流程的狀態，而成狀態 S2(這個電腦中心，在收到使用者要求後，準備進行處理)。也就是說，以 FSM 圖來表示流程時，主要在強調流程可以執行的事件，及這些事件執行時流程狀態的改變。



Event:
R-SR(Recieve Support Request)
F-T(Forward Request to Technician)
P-S(Perform Service Tasks)
I-SR(Invalidate Support Request)
N-SR(Nullify Support Request)
L-SR(Log Support Request)

圖<3-2>

在本研究的模式中，將採用以活動為基礎(Activity Based)的模式，這是因為此種模式是比較便潔的方式，精確地描述出活動之間的關係，對於流程中活動的執行分析具有重要的價值。

而在以活動為基礎的表示法中，本文參考[WfMC98]所訂定的標準。WfMC (Workflow Management Coalition)，工作流程管理協會，是由許多此領域內企業、組織所共同成立的團體，其研究著重於自動化工作流程的標準制定。在 WfMC 提出的標準裡，包括了 Workflow Enactment Service、Workflow Client/Server Invoked Application、Workflow Interoperability、Process Definition Interchange[WfMC98]。在 1995-1998 年更彙集了各家的說法，統一專有名詞的定義及應用程式的標準[WfMC94][WfMC96_1][WfMC96_2]。

在 WfMC 的定義中，工作流程是一部份或全部自動化的企業流程。依據[WfMC98]所制定的標準，流程係以下面七種元件來描述：(1)流程(Process)：企業中依照一定規則進行一連串工作步驟，以達成一特定目標。這一群為達成一定目標所執行的工作及工作間的傳遞，共同形成一個流程。(2)活動(Activity)：是指流程中的每個邏輯步驟，它可以是一個自動、非自動或半自動的活動。(3)遞移(Transition)：描述不同活動之間的相互關係，活動之間的連接就是依靠資訊的傳遞(Transition Information)。(4)參與者(Participant)：就是流程中各活動的執行、參與人員以及系統。(5)組織模式(Organization Model)：即參與者在流程中所扮演的角色。例如，業務經理。(6)應用程式(Application)：流程執行中所包括、使用的應用程式。(7)參考資料(Relevant Data)：流程執行中所產生出來並在往後的流程被使用的資料。

而且，依據流程中常見的進行方式，[WfMC98]的流程表示法包括了以下四種遞移：

1.循序(Sequence)：活動與活動的執行順序是循序進行的，當前一個活動執行完成後，下一個活動才能開始執行。

2.平行(Parallel)：流程中，可以同時有一個以上的活動被執行。

3.迴圈(Loop)：流程中，部份活動可以被重覆執行，即這些活動可以執行一次以上。

4.子流程(Subflow)：流程執行時，有些活動可以再被細分成許多更小、相關的活動。

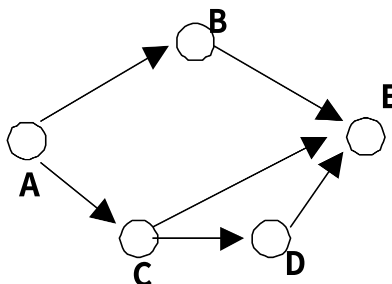
在上述 WfMC 所提出的流程進行方式中，我們選擇了前三者。因為本研究主要是透過使用者介面，提供使用者所有基本、功能性的活動，當使用者準備執行該項活動時，會點選這個活動，而執行中，這個活動的相關資訊就會被記錄下來。接下來，本研究則試圖由這些記錄下來的資訊中，找出流程裡眾多活動之間的執行關係。因此，一個中介(intermediary)、可以再被細分的活動，其實已包含這個被細分的活動和細分後活動之間的關係，使用者可以直接將這個被細分的活動和細分後活動視為不同層級、不同群的活動來處理。而同一群的活動中，當然就只剩下前三項的遞移方式了。

第二節 萃取流程演算法

藉由資訊科技的幫助，在流程執行一段時間後，便可以記錄許多歷史性的資料。如何利用這些資料，找出流程的定義，是目前許多研究([Agra98][Cook95][Datt98])的目標。這些研究所使用的模式、演算法都各有不同，因此，以下各節將分別針對 1.Agrawal 演算法 2.KTAIL 演算法 3.Markov 演算法，這三個不同的方法，敘述其假設、模式與演算法，並對各法的特性、缺點作一簡單比較。其中，Agrawal 演算法所找出的是一有向圖(directed graph)，類似本論文所採用的模式；而 KTAIL 和 Markov 演算法，則找出的是有限狀態圖(FSM)，類似一般編譯程式(compiler)所根據的機制。

1.Agrawal 演算法

在[Agra98]的研究中，一個流程是由許多的活動所組成，這些活動是某些較小單位的工作。假設這些活動的執行時間是"一瞬間(instantaneous)"、沒有活動的開始時間相同。在這樣的假設下，流程一次執行的記錄(稱為一個流程例)，就會成為一列(list)的活動記錄。如圖 3-3，假設這個流程某次執行活動 A、活動 B、活動 C、活動 E，則所記錄的一個流程例是 ABCE。因此，當收集許多的流程例後，所得到的資料是許多列的活動記錄，如{ABCE、ACDBE、ACDE…}。



圖<3-3>

因此，在[Agra98]研究中，是將每一次流程執行的記錄分開(成各次流程例)，而每一個流程例所記錄的是一列活動執行先後次序(total order)。所以，收集了所有流程例資料後，如{ABCE、ACDBE、ACDE、…}，就成為演算法的輸入(input)資料了。

那麼，演算法的輸出呢？演算法的輸出是一個有向圖，圖上的每一個點(Vertex)代表一個活動，點與點間的邊(Edge)則代表這兩個活動的執行順序。(因為一個點代表一個活動，所以一個活動在有向圖上只有一點)。如圖 3-3 就是一個可能輸出的有向圖。

在這樣的輸入假設與輸出要求下，Agrawal 演算法主要的想法(idea)是這樣的：由流程執行記錄(Workflow log)中的每一個流程例，找出兩兩活動的執行先後順序的相依性(dependence)，若整理全部流程例後，活動間有雙向的相依性，則刪去這些相依性。利用最後剩下的相依性，就可以找出一相依性的有向圖(direct graph)，即全部活動，也就是整個流程的控制流(control flow)了。

將 Agrawal 演算法列示如下：

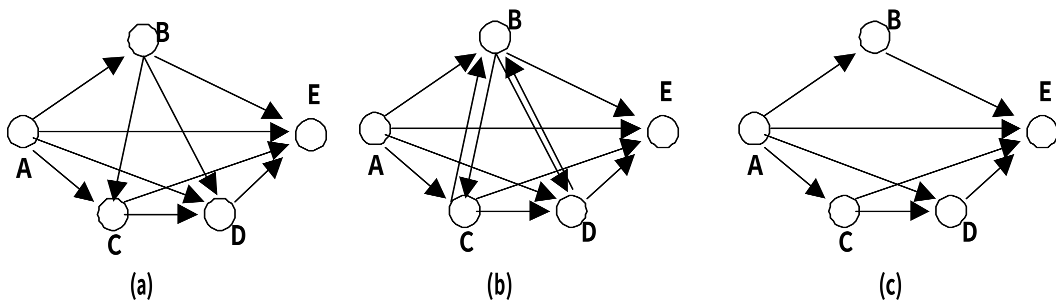
1. Start with the graph G=(V, E), with V being the set of activities of the process and

E=ψ. (V is instantiated as the log is scanned in the next step)

2. For each process execution in L, and for each pair of activities u, v such that u terminates before v starts, add the edge (u,v) to E.
3. Remove from E the edges that appear in both directions.
4. For each strongly connected component of G, remove from E all edges between vertices in the same strongly connected component.
5. For each proecss execution in L:
   (a) Find the induced subgraph of G.
   (b) Compute the transitive reduction of the subgraph.
   (c) Mark those edges in E that are present in the transitive reduction.
6. Remove the unmarked edges in E.
7. Return(V,E).

上述演算法，可以用一個例子來說明。假設有一個流程 P，其中有五個活動，分別為 A、B、C、D、E。當這個流程執行一段時間後，得到下列流程資料：{ABCDE、ACBE、ADBE}。利用 Agrawal 演算法，首先，step1 設有向圖的點為所有活動，也就是 A、B、C、D、E，而邊則為空集合。

接下來的 step2 則將每一個流程例中，兩兩活動的相依性找出，成為有向圖的邊。如第一個流程例(ABCDE)，可以找出的相依性是 A→B、A→C、A→D、A→E、B→C、B→D、B→E、C→D、C→E、D→E。因此，由第一個流程例所繪出的有向圖，如圖 3-4(a)所示。

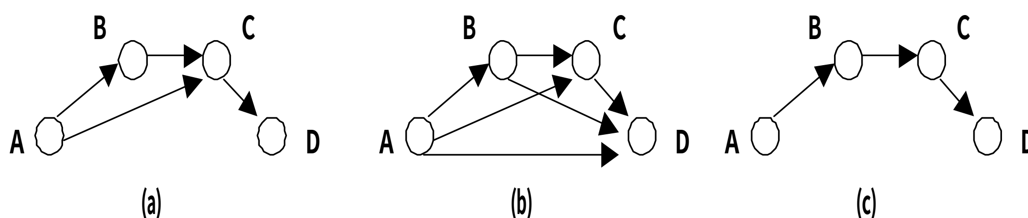


圖<3-4>

同樣的，將第二個流程例(ACBE)、第三個流程例(ADBE)所找出的相依性繪出。所以 step2 完成後，有向圖如圖 3-4(b)所示。由圖 3-4(b)中，可以很明顯

發現，活動 B 與活動 C 之間的邊是雙向的，也就是活動 B 與活動 C 有雙向的相依性。這是因為在某些的流程例中，活動 B 在活動 C 之前執行，而在另外某些流程例中，活動 C 卻在活動 B 之前執行，所以產生了雙向的相依性。換句話說，活動 B、C 之間並沒有一定的執行順序，所以應將活動 B、活動 C 之間的相依性刪去。同樣的道理，如果有數個活動，在有向圖中形成迴圈，則這些活動也難以肯定彼此間的先後次序，也應刪去。演算法中的 step3、step4 便基於這種特性，將有向圖中，活動之間沒有固定執行順序的邊刪去。所以圖 3-4(b)在 step3、step4 執行完成後，如圖 3-4(c)所示。

很明顯的，執行至 step4，演算法已掃描過全部流程例一次，而找出所有活動兩兩之間的相依性，成一有向圖。接下來，演算法必須考慮，有向圖中有些邊是因為遞移才產生的。例如，如果活動 X 執行完成，活動 Y 才能開始執行，活動 Y 執行完成，活動 Z 才能開始執行，則 step4 所完成的有向圖中不僅有(X→Y)、(Y→Z)的邊，也會有(X→Z)的邊。這是因為活動 X 執行的時間當然也會比活動 Z 開始的時間要早。但實際執行時，並不能直接由活動 X 跳至活動 Z，所以(X→Z)的邊是多餘的，必須去除。但是不是可以由 step4 中完成的有向圖，直接將活動間遞移的邊刪去就可以呢？答案當然也是否定的。這可以用圖 3-5(a)的例子來說明。

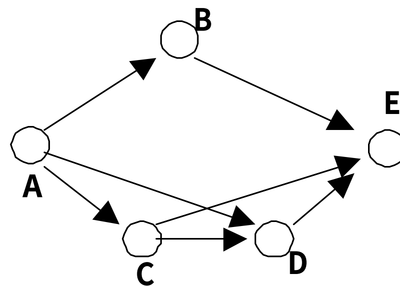


圖<3-5>

如果有個流程例是活動 A 執行完成，執行活動 B，活動 B 執行完成，執行活動 C，活動 C 執行完成，執行活動 D。而另外一個流程例是活動 A 執行完成，執行活動 C，活動 C 執行完成，執行活動 D。則可以利用前述的演算法(執行至 step4)找出的有向圖如圖 3-5(b)。如果這時將圖 3-5(b)中所有遞移邊刪去如圖 3-5(c)，則第二個流程例是不能執行的。也就是說，在活動 A 與活動 C 之間，其實有執行的順序關係，可以在活動 A 完成後，直接跳至活動 C，因此活動 A、C 之間應有(A→C)的邊，(A→C)邊並不是遞移。由這個例子，(A→C)邊在第一個流程例中是遞移，但第二個流程例卻不是，所以有向圖中的邊是不是因為遞移

而產生，應針對每一個流程例去考慮。

因此，Agrawal 演算法中，step5、step6 即針對每一個流程例去考慮，找出有向圖中每一個流程例的子圖(subgraph)，將不是遞移的邊給予一記號(mark)，待所有流程例處理完後，所有擁有記號的邊自然都是在某些流程例中不是遞移的邊。如圖 3-4(c)在 step6 完成後，就可以將遞移的邊刪去，而如圖 3-6 所示。



圖<3-6>

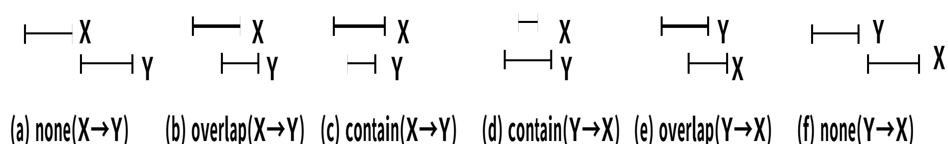
因此，Agrawal 演算法須掃描全部流程例二次，step2 至 step4 掃描第一次以找出所有活動的相依性，若全部有 m 個流程例，而有 n 個活動，則 step2 至 step4 的時間複雜度(tiem complexity)為 $O(mn^2)$。Step5 至 step6 掃描第二次以去除多餘的遞移相依性，這部份的時間複雜度則為 $O(mn^3)$。所以，整個流程的時間複雜度為 $O(mn^3)$。

還有一個值得考慮的問題，若流程中有迴圈呢？Agrawal 演算法對於迴圈的處理方式，主要是將重覆出現的同一個活動視為不同的活動。例如，假設有一流程例為{ABCBCD}，活動 B 與活動 C 重覆執行兩次，則將此流程例改為 $\{AB_1C_1B_2C_2D\}$，第一次執行的活動 B 被改為活動 B_1，第二次執行的活動 B 被改為活動 B_2，活動 B_1、活動 B_2 被視為不同的活動(活動 C 也作同樣的處理)。所以接下來只要利用同樣的方法找出有向圖，最後再把重覆執行的活動合併就可以了，修改後的演算法如附 1 所示。

綜合上面的說明，本研究認為利用 Agrawal 演算法來萃取流程定義，主要有下列缺點：

(1)模式假設將產生多餘相依性：在 Agrawal 演算法中，假設活動的執行時間是”一瞬間(instantaneous)”、沒有活動的執行開始時間相同，所以記錄下來的流程例是一系列活動先後次序(total order)，再由一系列的活動先

後次序找兩兩相依性。這樣的作法會產生許多多餘且不必要的相依性，試以圖 3-7 為例：



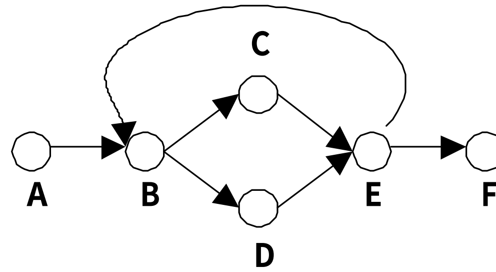
圖<3-7>

假設活動 X 與活動 Y 是任意兩個活動，則這兩個活動的執行時間可能存在圖 3-7 的六種關係。若活動 X 與活動 Y 的執行時間是圖 3-7(b)(c)(d)(e)中的一種，則活動 X 與活動 Y 必然沒有相依性，兩活動是獨立的。但如果活動執行時間假設為”一瞬間”呢？則圖 3-7(a)(b)(c)都會認為有 X→Y 的相依性，而圖 3-7(d)(e)(f)都會認為有 Y→X 的相依性。也就是說，若活動 X 與活動 Y 的執行時間是圖 3-7(b)(c)(d)(e)的關係時，以 Agrawal 演算法的假設，就會多出一些相依性。雖然這些相依性可能可以利用其它流程例，而在兩活動間有雙向相依性時，刪去這些多餘的相依性，可是當然這樣就需要較多的流程例才能辦到。

再者，活動實際上執行一段時間。因為執行時間有交集的活動，不可能存在相依性，所以如果能夠善用這些執行時間有交集則兩者獨立的特性，應該能夠更正確、更快地將流程例中多餘的相依性刪去。但若假設活動的執行時間是一瞬間，則當然活動執行時間不會有交集，就不能利用這個特性了。

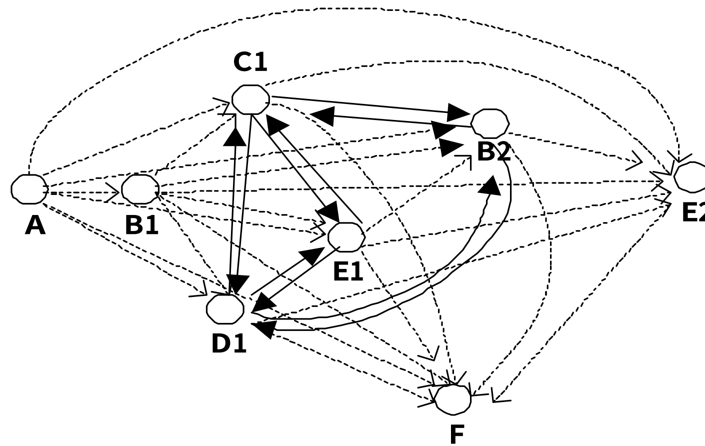
(2)迴圈處理只適用於簡單迴圈：在 Agrawal 演算法中，對於迴圈的處理方法，主要是將重覆執行的活動視為不同的活動，在利用同樣的方法找出相依性後，再合併。但是這樣的處理方式對於迴圈中所有的活動不一定全部執行時，可能會將許多正確的相依性刪去。

如以圖 3-8 為例，圖 3-8 為一包含迴圈的流程，其中活動 B、C、D、E 形成一迴圈，可能重覆執行。且活動 C 與活動 D 是平行、OR_Split 的兩個活動，也就是說，活動 B 執行完成後，可能執行活動 C 之後就執行活動 E，也可能活動 B 執行完成後，執行活動 D 之後就執行活動 E，當然也有可能，活動 B 執行完成後，執行活動 C、活動 D，再執行活動 E。



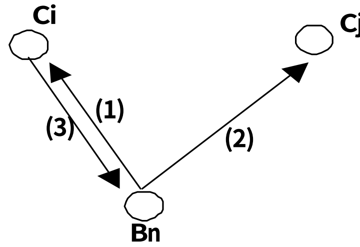
圖<3-8>

因此，假設圖 3-8 二次執行所收集的流程資料為{ABCEBDEF、ABDEBCEF}，則利用附 1 的演算法執行至 step3 所產生的有向圖如圖 3-9 所示。其中可以很明顯地發現活動 C_1 與活動 B_2、活動 C_1 與活動 E_1、活動 D_1 與活動 E_1、活動 D_1 與活動 B_2 之間都有雙向的相依性，所以演算法中，step4 將會將這些相依性刪去。以活動 C_1 和活動 B_2 為例，在第一次的流程例中，活動 C_1 比活動 B_2 早執行，所以會有 $C_1 \rightarrow B_2$ 這個相依性，但在第二次的流程例中，活動 B_2 比活動 C_1 早執行，所以會有 $B_2 \rightarrow C_1$ 這個相依性，因此雖然這兩個相依性都是對的，但由於雙向，將會被刪去。



圖<3-9>

這個問題可以利用圖 3-10 的概念來解釋，圖 3-8 迴圈中的活動 B 是迴圈執行時每次都會執行的活動，因此當迴圈執行第 n 次時，活動 B 會被展開成 B_n。但活動 C 在迴圈中並不一定每次執行，所以在迴圈執行第 n 次時，假設活動 C 執行第 i 次($i<n$)，因此，活動 C 被展開成 C_i，也因此，活動 B_n 會在活動 C_i 之前執行，而有 $B_n \rightarrow C_i$ 的相依性，如圖 3-10 中(1)的相依性。



圖<3-10>

但是在另一個流程例中，假設迴圈執行第 n 次時，活動 C 執行 j 次($i<j<n$)，活動 C 會被展開成 C_j，也因此，活動 B_n 會在活動 C_j 之前執行，而有 $B_n \rightarrow C_j$ 的相依性，如圖 3-10 中(2)的相依性。但因為 $i<j$，所以活動 C_i 必在活動 B_n 之前執行，而有 $C_i \rightarrow B_n$ 的相依性，如圖 3-10 中(3)的相依性，所以如果考慮這二個流程例，則 B_n、C_i 之間會有雙向的相依性，而會被去除，因而有不正確的結果。

所以，這個問題的原因是因為，當迴圈中的某些活動執行次數不固定時(如有些活動在每一次迴圈執行時不一定執行，或者有巢狀迴圈使得有些活動在每一次迴圈執行時重覆執行數次)，這些活動相對於迴圈中另外一些活動(如圖 3-8 中的活動 B、活動 E)，就有可能在某些流程例中比較早執行，但在某些流程例中則比較晚執行，使得有向圖有雙向的相依性，而必須刪去，雖然這些相依性其實都是正確的。因此，Agrawal 演算法對於迴圈的處理方式，並不適用於迴圈中部份的活動執行次數不定的情形。

2.KTAIL 演算法

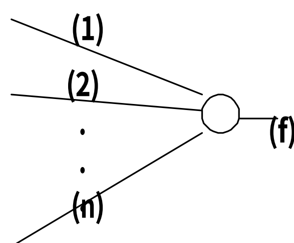
KTAIL 演算法[Cook95]是由 Biermann 和 Feldman[Bier72]的演算法(以下簡稱為 B_F 演算法)修改而來。B_F 演算法原是為了解決 grammar discovery 的問題(即針對某一種語言，以一些句子為例，由其中找出這個語言的文法)，所以 B_F 演算法以句子的各單元字(token)的字串為輸入，而產生 FSM(Finite State Machine)為輸出。

在 KTAIL 的演算法中，認為一個流程是由許多的活動所組成，這些活動是某些較小單位的工作。每一個活動執行時，會執行一段時間，在這段時間內可能會記錄下一些事件(event)，如活動開始事件(begin event)、活動結束事件(end event)、資源等待事件(wait event)…等等。這些事件是在某一個時間點發生

的，沒有事件的時間點相同。所以如果收集流程執行過程發生事件的記錄，就會得到一連續的事件流(event stream)。

接下來，如果把這些事件流(event stream)中每一個事件(event)視為一個個的單元字(token)，就可以將由這些事件流中找出流程定義的問題轉為前述 grammar discovery 的問題。因此，KTAIL 演算法以事件流(event stream)為輸入，而以找出的 FSM 圖為輸出，但為了簡化起見，KTAIL 的演算法也假設每一個活動執行只記錄一個事件(event)。

因此，在以事件流(event stream)為輸入，FSM 圖為輸出的要求下，KTAIL 演算法的主要想法(idea)是：FSM 圖中有許多狀態，每一個狀態可以由將來(future)會發生的行為來定義。以圖 3-11 的概念來解釋，假設圖 3-11 中(1)、(2)、..(n)都是一些連續的事件執行(以下簡稱事件列)，但這些連續的事件執行後，接著都會執行某些事件，如圖 3-11 中的(f)。所以，可以認為，圖 3-11 中(1)(2)…(n)的執行可能到達一個相同的狀態，所以接著才會有相同的執行事件(f)。因此，可以利用這個特性來定義狀態(state)，只要將事件流中會有相同將來行為的事件列分為一類，並用這一類的事件列代表一個狀態，就可以將 FSM 圖中的狀態找出來了。



圖<3-11>

將 KTAIL 演算法列示如下:

1. All the equivalence classes C_1、C_2、…C_s are extracted form the given sample event streams that characterize the process. Each equivalence class C_j is represented as a unique state S_j of the process. A graph is created with nodes $S_1,\ldots,S_s$.
2. For each C_j (input state), transitions to destination states C_k upon each event A_i are determined as follows:
   (a) For each history h in C_j, create a concatenated event sequence q, where q = h.A_i
   (b) Now, if q is found to belong to some other equivalence class C_k (i.e., if q ε C_k), then C_k is termed as a destination state of event A_i from state C_j.
   (c) An edge is created from state S_j to each destination state S_k corresponding to C_k, and it is labeled as "A_i".
   (d) If no such destionation state can be found, then no transition exists out of state C_j upon the occurrence of event A_i. (It is also possible that there are more than one transitions from an input state upon an event).
3. Merge the initial FSM，then the resultant graph is the final FSM for the process. Each of the states S_j in the FSM is mapped to a set of histories in H.

上述演算法，可以用圖 3-12 來說明。假設圖 3-12 是一個流程執行執行一次(一個流程例)所收集的事件流(event stream)。由這個事件流可以找出許多如圖 3-11 中(1)(2)…(n)的先前事件列(prefix)，如：{A、AB、ABC、ABCA、…、ABCABCBACBACABCBACBACABCBACABCABCBACBA}。

ABCABCBACBACABCBACBACABCBACABCABCBACBA

圖<3-12>

因此，利用 KTAIL 演算法，在 step1 中，首先要將這些先前事件列(prefix)依這些事件列之後的行為分類。但"之後的行為"如何定義呢？其實最簡單的說法就是這些事件列之後會執行那些事件(event)。在 KTAIL 演算法中需給予一參數 K，"之後的行為"由事件列(prefix)之後的 K 個事件(event)來定義。以圖 3-12 為例，假設 K 為 3，則可以由圖 3-12 的事件流(event stream)中找出長度為

3 的事件集合：{ABC、BCA、CAB、BCB、CBA、BAC、ACB、ACA} 等 8 種。因此，接下來，就可以很容易地把各先前事件列(prefix)分類了。如先前事件列 A 之後的事件是 BCA，所以先前事件列 A 就被分到 BCA 這類，而先前事件列 AB 之後的事件是 CAB，所以先前事件列 AB 就被分到 CAB 這類。依同樣的方法，將所有先前事件列分類。

由分類結果，可以很明顯地發現，先前事件列(prefix)的分類中，{ABC、ABCABCBACBAC、ABCABCBACBACABCBACBAC、ABCABCBACBACABCBACBACABCBAC、ABCABCBACBACABCBACBACABCBACABC}這些事件列之後都會執行事件 ABC，所以這些先前事件列(prefix)都被分到同一類。這也就是說，這些先前事件列就如同圖 3-11 的(1)(2)…(n)一般，都到達同一個狀態，所以之後的行為(f)就接著執行事件 ABC。因此，就把這一個分類稱為一個相同類別(equivalence class)，設為 C_1，也就利用這一個分類(C_1)來代表一個狀態(S_1)。

同樣的方法，由圖 3-12 的事件流中，共可找出 8 個相同類別(C_1，C_2，…C_8)，分別代表 8 個狀態(S_1，S_2，…S_8)。因此，可以利用這 8 個相同類別畫出 FSM 圖中 8 個狀態。

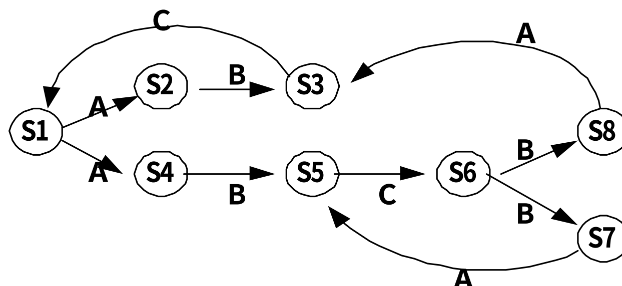
接下來，只需要找出 FSM 圖中各狀態之間的轉換(transition)就可以了，所以 KTAIL 演算法的 step2，目的便在找出各狀態之間的轉換(transition)。尋找轉換(transition)的方法很簡單，只要在相同類別(C_j)的既有事件列(history)之後加上所有可能的事件(A_i)，成為新的事件列($h.A_i$)，若這個新的事件列屬於另一種相同類別(C_k)，則代表這個事件列原來所屬的相同類別(C_j)可以在某個事件(A_i)發生後變成新的類別(C_k)、新的狀態(S_k)。只要利用這個特性，考慮相同類別中所有的事件列可能使狀態發生的變化，就可以找出這個類別所代表的狀態與其它狀態的關係，即與其它狀態之間的轉換。

以相同類別 C_1 其中既有的事件列 ABC 為例，如果 ABC 之後加上各種可能的事件，則可能成為新的事件列是 ABCA、ABCB 及 ABCC。而其中 ABCA 屬於相同類別 C_4，代表相同類別 C_1 可以在事件 A 發生後變成新的類別 C_4、新的狀態 S_4。所以，以相同類別 C_1 為例，演算法 step2(a)中對各既有事件列加上所有可能事件，整理如表 3-1 中”$h.A_i$”欄所示，step2(b)中找出新的事件列所屬的相同類別，則如表 3-1 中”C_i”所示。

History	h.A_I	C_i
ABC	ABCA ABCB ABCC	C_4 - -
ABCABCBACBAC	ABCABCBACBACA ABCABCBACBACB ABCABCBACBACC	C_4 - -
ABCABCBACBACABCBACBAC	ABCABCBACBACABCBACBACA ABCABCBACBACABCBACBACB ABCABCBACBACABCBACBACC	C_4 - -
ABCABCBACBACABCBACBACABCBAC	ABCABCBACBACABCBACBACABCBACA ABCABCBACBACABCBACBACABCBACB ABCABCBACBACABCBACBACABCBACC	C_2 - -
ABCABCBACBACABCBACBACABCBACABC	ABCABCBACBACABCBACBACABCBACABCA ABCABCBACBACABCBACBACABCBACABCB ABCABCBACBACABCBACBACABCBACABCC	C_4 - -

表<3-1>

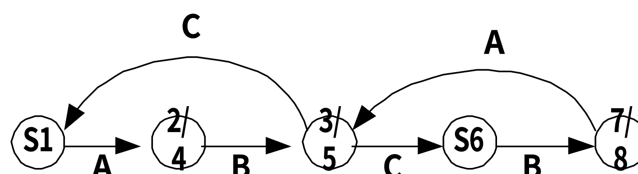
而 step2(c)則利用 step2(a)、(b)所找出的結果，在 FSM 圖中，由狀態 S_1 畫出邊(edge)至狀態 S_4，標示為 A，同理，也由狀態 S_1 畫出邊(edge)至狀態 S_2，標示為 A。依同樣的方法，將所有相同類別處理完成後，就可以得到 FSM 如圖 3-13 所示了。



圖<3-13>

為了使如圖 3-13 的 FSM 圖減少複雜性，KTAIL 演算法加入 step3(B_F 演算法無此步驟)，將圖 3-13 的 FSM 圖中一些狀態合併。合併的方法是：如果一個狀態 S_1，有標示為”t”的轉換(transition)邊至狀態 $S_2…S_n$，而狀態 $S_2…S_n$ 連出

去的轉換(transition)邊標示皆相同，則狀態 $S_2 \ldots S_n$ 合併。以圖 3-13 為例，S_2 和 S_4 合併，而 S_3 和 S_5 合併，最後 S_7 和 S_8 再合併如圖 3-14 所示。



圖<3-14>

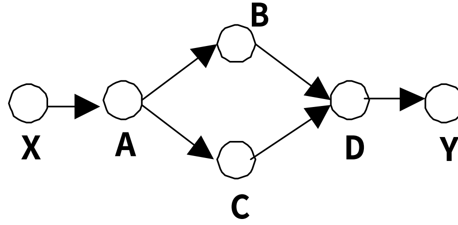
因此，由上述的演算法，就可以由流程執行所記錄下來的事件流資料中找出流程(以 FSM 圖表示)的定義了。但有一點值得注意的是，在前述 KTAIL 演算法中所使用的是正確的事件流資料，也就是說並未考慮錯誤資料(noise)的問題。

所以，在[Datt98]的研究中，針對錯誤資料(noise)的問題，修改 KTAIL 演算法。主要的不同是加上一個參數 C，作為臨界值(confidence value)。當原來 KTAIL 演算法 step1，由所有事件流中找出所有的相同類別後，[Datt98]的研究則再加上，計算相同類別中每一個事件列的出現機率，只有超過臨界值者才留下，不到臨界值的事件列則刪去，而產生新的相同類別。自然，在經過這個修正後所產生的相同類別，才繼續執行原來 KTAIL 演算法的 step2。

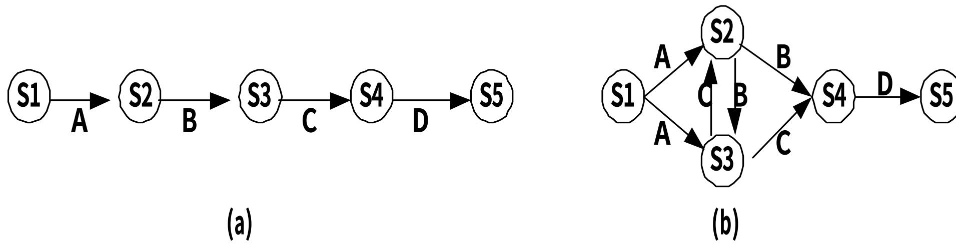
綜合以上的說明，本研究認為利用 KTAIL 演算法來萃取流程定義，主要有下列缺點：

(1)模式假設將產生多餘的狀態及狀態轉換: 在 KTAIL 演算法中，雖然認為一個活動是由一些事件所組成，但為了簡化起見，也假設一個活動只有一個事件。但如果一個活動只記錄一個事件，就會如同 Agrawal 演算法的假設(假設活動的執行時間是”一瞬間(instantaneous)”、沒有活動的執行開始時間相同)，產生許多多餘的相依性(KTAIL 演算法則產生許多多餘的狀態及狀態轉換)。

如圖 3-15，設圖 3-15 為一個流程，而這個流程的一次流程例為 XABCDY，則利用 KTAIL 演算法(設 K 值為 1)所找出的 FSM 圖如圖 3-16(a)所示。由圖 3-16(a)中，可以明顯發現，會有 A-B-C-D 的執行順序關係。但是，實際上，活動 B 與活動 C 之間是獨立的，並沒有一定的執行順序，當然，狀態 S_3 也是多餘的。



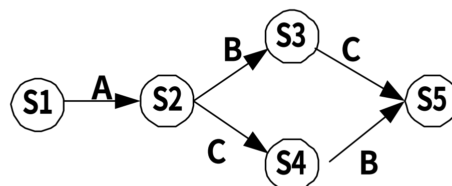
圖<3-15>



圖<3-16>

而如果有許多的流程例，對於活動 B 與活動 C 而言，這兩個活動之間可能是活動 B 之後執行活動 C，或者是活動 C 之後執行活動 B(因為只記錄事件，事件只有一個時間點，不會有二個事件同時發生)，不論何者，都不能使這個多餘的狀態及狀態間的轉換刪去。例如，圖 3-15 的流程如有另一流程例為 XACBDY，則利用這兩個流程例，以 KTAIL 演算法所繪出的 FSM 圖如圖 3-16(b)所示，仍有 B→C 或 C→B 的轉換(transition)，並未刪去。所以，KTAIL 演算法在流程有平行處理，即有活動彼此之間是獨立、平行的時候，將產生許多多餘的狀態及狀態間轉換。

(2)參數值設定不易：KTAIL 演算法中需給定一參數值，來決定一個狀態之後的行為(由狀態之後 K 個事件所定義)。然而，同樣的流程，記錄同樣的事件，只要 K 值設定不同，則所產生的 FSM 圖並不相同。同樣以圖 3-15 的流程及圖 3-16 的流程例為例，如果令 K 值為 2，則所繪出的 FSM 圖如圖 3-17 所示。



圖<3-17>

由圖 3-16 與圖 3-17 比較，可以發現兩圖有相當程度不同。因此，KTAIL 演算法要求 K 值的設定必須由對流程有經驗、熟悉的專家來決定，而且不同

的流程可能就須要設定不同的 K 值。所以，參數 K 值的設定將明顯影響最後所找出的 FSM 圖，如何適當設定並不容易。

(3)有些狀態不容易合併，模式複雜: KTAIL 演算法以 FSM 圖為輸出，FSM 圖中表達了狀態及狀態間的轉換。而演算法中，狀態是由"之後的行為"來定義，所以只要之後執行某些相同的行為，就認為不論前面如何執行，都到達同一種狀態。由圖 3-17 的例子，可以找到 A-B-C 及 A-C-B 的事件序列，都到達同樣的狀態 S_5，但以流程而言，並不見得真正執行完成事件 A-B-C 與事件 A-C-B 會到達相同的狀態。所以，狀態 S_5 並沒有辦法提供更多的資訊，來整理事件 A-B-C、事件 A-C-B 等事件執行順序。

所以，FSM 圖中只是把可能的部份事件順序找出來而己，若可能的事件順序很多(如有其它事件可以跟事件 B、C 平行處理)，則找出的可能事件順序就有多種可能，而變得複雜，難以進一步整理、合併(merge)，而會有一個較為複雜的 FSM 圖(如圖 3-17 的狀態 S_2 到狀態 S_5 就難以再整理、合併)。

因此，KTAIL 演算法所找出的 FSM 圖是一個較為複雜，提供對流程有經驗者作進一步整理、分析的工具。

而且，進一步考慮，如果流程中的活動包含更多的訊息，如流程中活動實際上包括一個以上的事件時，以同樣的演算法，因為所找出的是流程部份的事件執行順序，所以可能一個活動的數個事件被其它活動的事件所隔開，則要找出同一個活動的事件就更為困難了，當然就更難加以整理、合併了。

3.Markov 演算法

在[Cook95]的研究中，與 KTAIL 演算法類似地，仍認為一個流程是由許多的活動所組成，這些活動是某些較小單位的工作。每一個活動執行時，會執行一段時間，在這段時間內可能會記錄下一些事件(event)，如活動開始事件(begin event)、活動結束事件(end event)、資源等待事件(wait event)...等等。這些事件是在某一個時間點發生的，沒有事件的時間點相同。所以如果收集流程執行過程發生事件的記錄，就會得到一連續的事件流(event stream)，而為了簡化起見，Markov 的演算法也假設每一個活動執行只記錄一個事件(event)。在記錄了流程執行所產生的事件流後，希望能利用演算法由事件流中找出 FSM 圖，即找出流程的定義來。

因此，在以事件流(event stream)為輸入，FSM 圖為輸出的要求下，Markov 演算法的主要想法(idea)是：找出最可能的事件序列(event sequence)，並利用這些事件序列，轉成 FSM 圖中的狀態和狀態間的轉換。因此，演算法首先要由事件流中找出較可能的事件序列。而要找出較可能的事件序列，可以由所有事件組合出許多長度固定的事件列，並計算這些事件列之後執行某事件的機率，找出其中機率較大者，自然就是較有可能的事件序列了。

將 Markov 演算法列示如下：

1. The event-sequence probability tables are constructed by traversing the event stream.
2. A directed graph, called the event graph, is constructed from the probability tables in the following manner. Each event type is assigned a vertex. Then, for each event sequence that exceeds the threshold probability, a uniquely labeled edge is created from an element in the sequence to the immediately following element in that sequence.
3. The previous step can lead to over-connected vertices that imply event sequences that are otherwise illegal. To correct this, over-connected vertices are split into two or more vertices. (The general definition of this step works by finding disjoint sets of input and output edges for a vertex that have some non-zero sequence probability, and splitting the vertex into as many vertices as there are sets.)
4. The event graph G is then converted to its dual G’ in the dollowing manner. Each edge in G becomes a vetex in G’ marked by the edge’s unique label. For each in-edge/out-edge pair of a vertex in G, an edge is created in G’ from the vertex in G’ corresponding to the in-edge to the vertex in G’ corresponding to the out-edge. This edge is labeled by the event type.

同樣以圖 3-12 的例子來說明 Markov 演算法。圖 3-12 是一個流程執行執行一次(一個流程例)所收集的事件流(event stream)。因此，首先由事件流中找出較可能的事件序列。然而，要找出較可能的事件序列，需要先由所有事件組合出許多長度固定的事件列，所以 Markov 演算法也需要需給予一參數 K，事件

列的長度由 K 值決定。

以圖 3-12 為例，其中只有 A、B、C 三種事件，所以假設 K 值為 2，則可以找出所有的事件列為：{AA、AB、AC、BA、BB、BC、CA、CB、CC}等九種事件列。接下來，則需要計算這些事件列之後執行某事件的機率，因此以這九種事件列作為表格的縱軸，而以所有可能的事件(A、B、C)作橫軸，繪出表格如表 3-2 所示。

	A	B	C
AA	0.00	0.00	0.00
AB	0.00	0.00	1.00
AC	0.50	0.50	0.00
BA	0.00	0.00	1.00
BB	0.00	0.00	0.00
BC	0.33	0.67	0.00
CA	0.00	1.00	0.00
CB	1.00	0.00	0.00
CC	0.00	0.00	0.00

表<3-2>

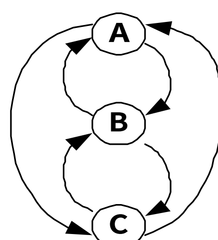
接下來，自然要算出表格中各種事件序列的機率了。如表格中第二列為事件列 AB，圖 3-12 的事件流中事件列 AB 出現 6 次，且之後都執行事件 C，所以事件列 AB 之後執行事件 C 的機率為 1，而之後執行事件 A、B 的機率皆為 0，如表 3-2 第二列所示。又如表格中第六列為事件列 BC，圖 3-12 的事件流中事件列 BC 也出現 6 次，但之後執行事件 A 有 2 次，所以事件列 BC 之後執行事件 A 的機率為 0.33，而之後執行事件 B 有 4 次，所以事件列 BC 之後執行事件 B 的機率為 0.67，當然，之後執行事件 C 的機率就是 0 了，如表 3-2 第六列所示。餘類推，完成如表 3-2 所示。

因此，完成表 3-2 後，就可以得到所有可能事件列之後執行某個事件的機率了。也就是說，Markov 演算法 step1 執行完成，可以得到如同表 3-2 的事件序列機率表(event-sequence probability table)，由表中就可以得知各種事件序列發生的機率。

當然，接下來只要找出機率較大者，就是較有可能的事件序列了。所以 Markov 演算法 step2，首先由事件序列機率表中找出機率較大的事件序列。因此，Markov 演算法需要另一參數 C，作為門檻值(threshold)，只取機率值大於等於這個門檻值的事件序列。

以表 3-2 為例，設門檻值 C 為 0.5，則可以找到的事件序列是{ABC、ACA、ACB、BAC、BCB、CAB、CBA}。因此，以這些事件序列畫事件圖(event graph)。即以事件為點(vertex)，事件序列中每一個事件依序相連，繪出圖形如圖 3-18 所示。

圖<3-18>



但由圖 3-18 中可以發現，雖然這些事件和事件之間的邊，是由事件序列機率表取超過門檻機率值而繪出的，但仍會多出許多的事件序列。例如，可以由圖 3-18 中找出事件序列 C-B-C，但這個事件序列發生機率並未超過門檻值，所以應將此事件序列由圖 3-18 中去除。

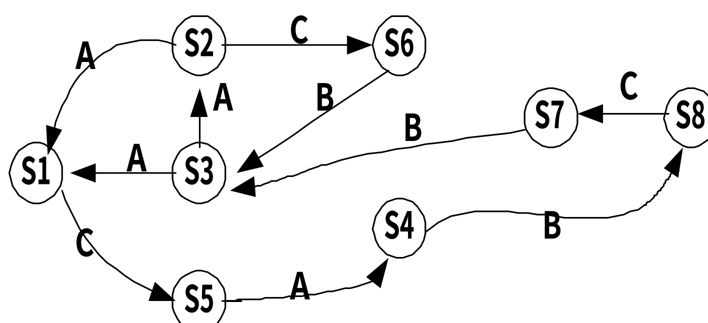
但如何去除呢？以圖 3-18 上點 B 為例，有連進來的邊(ingoing edge)A→B、C→B，而有連出去的邊(outgoing edge)B→A、B→C，這些邊是由不同的事件序列所繪出，所以只要將不同的連進來和連出去的邊分類，使分類後的邊沒有多出來的事件序列，再將點 B 分裂成數點，以連接分類後的邊即可。

所以，Markov 演算法 step3 的目的，在將如圖 3-18 的事件圖去除多餘的事件序列。所使用的方法則是藉由圖中的點分裂，使各點所連的邊，不會產生多餘的事件序列。因此，當 step3 完成，就可以得到一個包含所有超過門檻機率值的事件序列，但無多餘事件序列的事件圖了。

因此，接下來 step4 只要將這個事件圖轉成 FSM 圖就可以了。因為如圖 3-18 中的點代表事件，所以圖中的邊便代表事件執行前後的狀態了。因此，利用圖 3-18 分裂後的每一個邊(共有 8 個)，繪出 FSM 圖中 8 個狀態。接下來，需要找出 FSM 圖中狀態之間的轉換(transition)。如(1)-C-(5)，便代表了狀態(1)執行事件 C 後成為狀態(5)，所以，FSM 圖中便有一邊由狀態 1 至狀態 5，標示為 C。這也就是說，由圖 3-18 所完成的分裂圖中，只要找出各點連進來(ingoing

edge)和連出去(outgoing edge)邊的兩兩組合，就可以在 FSM 圖中繪出由連進來邊所代表的狀態 至連出去邊所代表狀態 而標示為這個點的轉換(transition)了。最後，完成轉換如圖 3-19 的 FSM 圖(可以再利用一些 FSM 圖合併的方法予以合併，此處不再討論)。

圖<3-19>



因此，執行 Markov 演算法後，也可以由所記錄下來的事件流，找出流程定義(以 FSM 圖表示)。且因為 Markov 演算法是利用門檻值的方式，取較有可能的事件序列來繪 FSM 圖，所以，對於錯誤資料(noise)的問題，已可以處理。但 step3 由事件圖刪去多餘事件序列的方法，在點數較多、K 值較大時，將變得複雜，所以在[Datt98]的研究中對多餘事件序列刪去的方法作了修改，演算法如附 2 所示。

綜合上面的說明，本研究認為利用 Markov 演算法來萃取流程定義，主要有下列缺點：

(1)模式假設將產生多餘的狀態及狀態轉換: 在 Markov 演算法中，也假設一個活動只有一個事件。因此，如同 Agrawal 演算法、KTAIL 演算法，會產生許多多餘的狀態及狀態轉換。

(2)參數值設定不易: 在 Markov 演算法中需給定二參數值，K 值決定事件列的長度，而 C 值則決定門檻值。這兩個參數值的設定，對於最後所產生的 FSM 圖都有很大的影響。因此，Markov 演算法對於參數值 K、C 的設定也必須由對流程有經驗、熟悉的專家來決定，而且不同的流程可能就須要設定不同的 K、C 值。所以，參數值的適當設定並不容易。

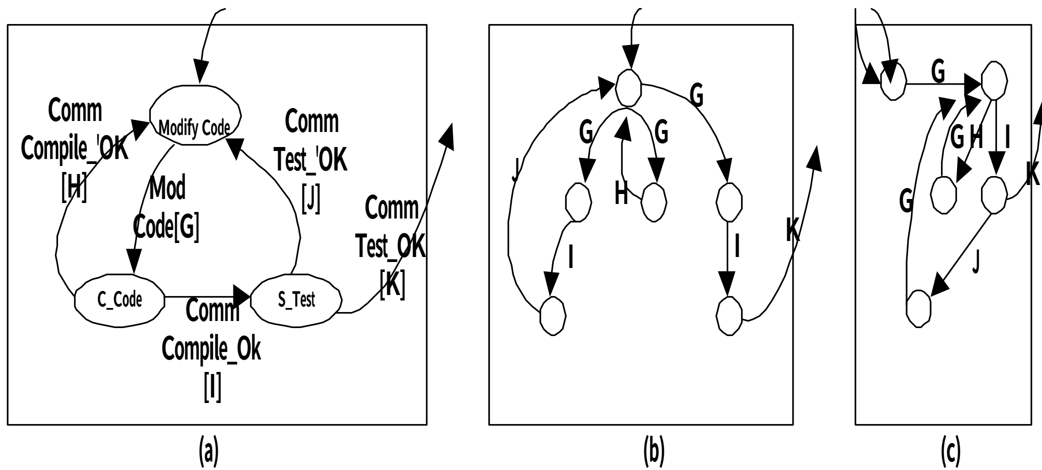
(3)有些狀態不容易合併，模式複雜: 就如同 KTAIL 演算法一般，Markov 演算法也需要對找出的 FSM 圖再合併(merge)、整理。Markov 演算法是取固定長度的事件列來找尋可能的事件序列，即片段的事件順序找尋事件間順序，

所以初步找出的 FSM 圖會很複雜，也需要更進一步的合併、整理。因此，Markov 演算法所找出的 FSM 圖也是一個較為複雜，提供對流程有經驗者作進一步整理、分析的工具。

第三節 綜合比較

綜合上節三個演算法的說明，本研究認為在流程模式的假設上，如果假設一個活動的執行只有一瞬間、一個時間點，不論上述那一種演算法，由記錄下來的流程歷史資料，來萃取流程定義，都會產生許多多餘的相依順序關係，而且導致找出的流程定義更為複雜。所以，本研究試圖針對此點進行改善，提供一個更簡潔的萃取方法。

此外，KTAIL 和 Markov 演算法(及其改良方法[Datt98])，所產生的有限狀態圖(FSM)，在很多情況下並不易了解。試以圖 3-20(a)、(b)、(c)為例，圖 3-20(a)，是一個程式撰寫流程([Cook95]，ISPW 6/7 process)的一部份。當程式人員撰寫修改程式(G)後，會編譯程式，如果編譯不成功(H)，則重新撰寫修改。若成功(I)，則測式輸出資料是否正確，若不正確(J)，同樣重新撰寫修改，當然，如果輸出結果是正確的(K)，則進入下一階段。



圖<3-20>

圖 3-20(b)是利用 KTAIL 演算法所找出來的流程，而圖 3-20(c)是利用 Markov 演算法所找出來的流程。由圖 3-20(b)中，可以清楚發現，KTAIL 演算法所找出的流程，可以表達當程式人員撰寫程式(G)後，編譯成功(I)，但輸出不正確(J)，重新撰寫修改的流程意義(G-I-J)。或是撰寫程式(G)後，編譯不成功(H)，重新

撰寫修改的流程意義(G-H)。當然，也表達了當程式人員撰寫程式(G)後，編譯成功(I)，輸出正確(K)，而進入下一階段的流程意義(G-I-K)。雖然利用 KTAIL 演算法所找出的流程，表達了這些流程意義，但這些流程意義的表達與原來圖 3-20(a)相較，是相當複雜而難以合併、了解的，必須由對這個流程熟悉的專家來作進一步的解釋與整理。

同樣的情形也發生在 Markov 演算法所找出的流程。在圖 3-20(c)中，也表達了如 G-I-J、G-H、G-I-K...等流程執行上的意義(而且更複雜)，但與圖 3-20(a)相較，同樣難以合併、了解，必須由專家解釋。或者說，這二個方法，只是提供了一個較為初級的流程定義資訊，來供對該流程有經驗的專家參考。這二個方法所找出的流程較為複雜且必須再由有經驗的專家作進一步的整理。

最後，因為現今商用的工作流程管理系統也很少接受以有限狀態圖所表達的流程定義。因此本研究將重點放在找出類似有向圖(directed graph)的流程模式，而且試圖以更簡潔、有效率的方法找出更高品質的流程定義。

第四章. 萃取演算法

藉由資訊科技的幫助，應用程式的開發，在流程執行一段時間後，便會留下一些歷史性的資料(Workflow log)，展現和分析這些資料，也很重要。本研究試圖由這些歷史性的資料中，找出流程的定義，並參考第三章第一節，[WfMC98]所訂定的流程定義標準，而著重於找出流程的控制流(control flow)部份。

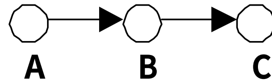
所以，以下簡單將本章分為三個部份：(1)流程模式(Process Model)，說明本研究中對流程、活動、控制流的定義及假設。(2)控制流萃取演算法，利用這個演算法找出流程的控制流定義，也就是流程中活動執行的順序關係。(3)控制條件萃取演算法，流程中活動的執行，除了順序關係外，控制條件可以決定活動是否執行，因此也很重要，利用這個演算法以萃取找出流程的控制條件。

第一節 流程模式

企業中有許多的流程(process)，執行這些流程可以達成某些企業目標。而一個流程是由許多的活動(activity)所組成，這些活動是某些較小單位的工作。例如，保險公司有"客戶索求保險給付"的處理流程，當客戶要求保險給付時，就可以依這些流程進行處理。這個流程中可能包括"客戶填寫表單"、"表單資料審查"、"保費費率計算"、"專業審查"...等等的單位工作，這些單位工作就稱為活動。因此，一個流程可以包含許多的活動，而流程的一次執行則稱為此流程的一個流程例。

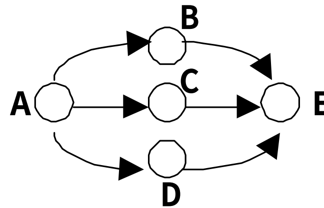
本研究以一個有向圖來代表流程中活動的執行，圖中的每一個點(Vertex)都代表一個活動，點與點間的邊(Edge)，則代表這兩個活動的執行順序(Dependence)。同時，邊上也含有一個布林函數(boolean function)。如果，流程執行時，這個函數值為"真"(true)，則下一個活動就會被執行；如果是"假"(false)，則下一個活動就不會被執行了。所以，流程中活動的執行，可以分為下列三種情形：

1. 循序(Sequence)：流程中的活動依序往下執行，如圖 4-1 所示。活動 A 執行完成後，活動 B 才開始執行，活動 B 執行完成後，活動 C 才開始執行，以此類推。也就是說，在這樣的有向圖中，這些活動與活動間的邊上所附的布林函數值都是"真"。因此，這些活動依序往下執行。



圖<4-1>

2.平行(Parallel)：如圖 4-2 所示，活動 A 執行完成後，會測試 A→B 邊上的布林函數，如果值為"真"，則執行活動 B，如果為"假"，當然活動 B 就不執行了。同樣地，也會測試 A→C、A→D 邊上的布林函數以決定是否執行活動 C 和活動 D。



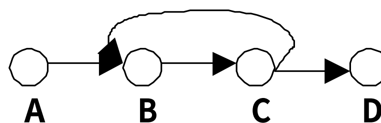
圖<4-2>

也就是說，當活動有數個連出去(outgoing)的邊時，每一個邊上的布林函數都必須被測試，以決定所連的下一個活動是否執行。而且，這些布林函數測試結果是獨立的，不影響其它布林函數的成立與否。如 A→B 邊上的測試結果不影響 A→C 邊上的測試結果。所以當一個活動所有可能的下一個活動都會被執行(函數測試值皆為"真")時，這種情形稱為 AND；也可能總是只有一個被執行，這種情形稱為 XOR；當然也有可能介於上述二者之間，一次有數個不等的活動執行，稱為 OR。

在上述的定義中，活動 B、C、D 的執行與否是彼此獨立的，端視其連進來邊上的布林函數值是否為真，本研究此項定義是參考[WfMC98]中 AND_SPLIT 的定義而來。當然，目前也有一些不同的定義方法，例如，[WfMC98]中有另一種 XOR_SPLIT 的定義，在這種定義裡，活動之間的執行不是彼此獨立的(但此種定義也可以用 AND_SPLIT 的定義來處理)。而在[Atti93]的研究中，更詳細的定義了多種活動之間可能的關係。目前有許多定義活動間關係的研究提出[Atti93][Adam98]，本研究不再於此贅述，但這些活動間關係較為複雜、適用於特定狀況，所以本研究採用較為一般化，且廣為工作流程管理系統所接受的平行活動獨立的假設。

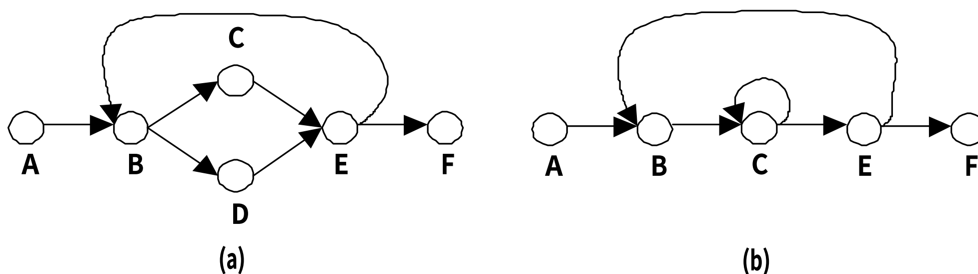
另外，上述有許多連出去(outgoing)邊的活動，是一個分支(Split)的活動，必有一相對應活動，有許多的連進來(incoming)的邊，可以將前面

分支出去的邊合併(Join)，如圖 4-2 中的活動 E。這兩個分支(Split)、合併(Join)活動之間的部份流程，就視為一個區間(Block)。
當然，合併也有兩種情形，一種是必須等前面所有分支出去的流程皆執行完成，才能往下執行，這種情形稱為 AND_Join。如圖 4-2 中活動 A 執行後，活動 B、C 執行，而活動 E 必須等到活動 B、C 皆執行完成，才能開始執行。另一種是只要前面所有分支出去的流程其中之一執行完成，就可以往下執行，這種情形稱為 XOR_Join。如圖 4-2 中活動 A 執行後，活動 B、C 執行，而活動 E 只要等到活動 B 或活動 C 執行完成，就可以開始執行了。
3.迴圈(Loop)：流程中的活動可以重覆執行，如圖 4-3 所示。活動 A 執行完成後，執行活動 B，活動 B 執行完成後，會執行活動 C，當活動 C 執行完成後，有 C→B、C→D 邊上的條件會被測試，若 C→B 邊上條件為”假”，C→D 邊上條件為”真”，則執行活動 D。反之，則重覆執行活動 B、活動 C，再測試條件，以此類推。因此活動 B、C 可能重覆執行多次。



圖<4-3>

如圖 4-3 中 C→B、C→D 邊上的控制條件，稱為迴圈條件，這兩個條件彼此不交集，一個成立(為”真”)，另一個必不成立(為“假”)，且在迴圈執行完成後測試。至於活動 B、活動 C，則構成迴圈的主體，也將迴圈主體視為一個區間。在圖 4-3 中的迴圈主體(活動 B、活動 C)，是順序執行的，當然迴圈主體也可以是平行、迴圈(巢狀)的執行方式，如圖 4-4 所示。



圖<4-4>

流程中，每一個活動會執行一段時間，當活動執行期間會留下一些事件(event)記錄。事件是"瞬間(instantaneous)"發生的、沒有事件的發生時間相同。在本研究中，並非記錄所有事件，而是對每一個活動，記錄這個活動的開始(begin)、結束(end)、寫入(write)變數事件。其中的開始、結束事件，每一個事件以三個資料項(3-tuple)來表示，記錄成[InsNo,AcNo,TS]。其中 InsNo 是流程例代號、AcNo 是活動代號，而 TS 則是這個事件發生的時間。因此就可以將相同流程例、相同活動的開始、結束時間找出，整理如表 4-1 所示。由表 4-1，就可以得知每一個流程例中，各活動的開始、結束時間。本研究便以表 4-1 的資料作為第二節控制流萃取演算法的輸入資料，以找出流程定義(以有向圖表示)。

Instance_No	Activity_No	Begin_time	End_time

表<4-1>

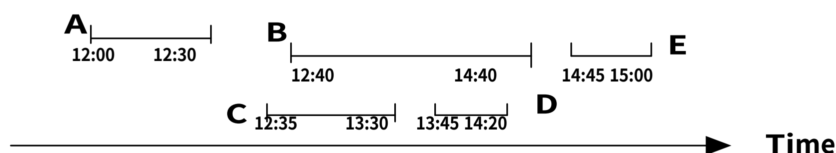
而寫入變數的事件，則以四個資料項(4-tuple)來表示，記錄成[InsNo,AcNo,VarName,Value]。其中 InsNo 是流程例代號、AcNo 是活動代號，而 VarName、Value 則是這個事件所寫入變數的變數名稱和變數值。本研究記錄了所有寫入事件的資料後，便以此作為第三節控制條件萃取演算法的輸入資料。

因此，在本研究中，首先由使用者列出流程中所有的活動，並假設每個流程中都可以找到一個開始與結束的活動。如果實際上，流程中的開始活動(start activity)不只一個，則使用者可以加上一個空(null)的開始活動，並由這個空的開始活動連至原來的開始活動。同樣的，若流程中的結束活動(end activity)不只一個，則使用者可以加上一個空(null)的結束活動，並由原來的結束活動連至這個新的結束活動，而使得流程中只有一個開始與一個結束活動。當使用者列出所有的活動後，工作流程管理系統便可以去收集、記錄如前所述的事件資料，本研究則試圖由這些資料，利用下節的演算法以找出流程定義。

第二節 控制流萃取演算法

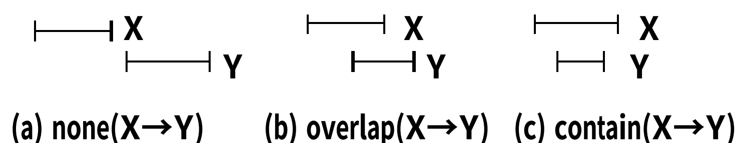
為了便於解釋，我們先以圖 4-2 為例，說明流程中活動執行時間的關係。假設圖 4-2 的流程，某次執行時，活動 A 在 12:00 開始執行，12:30 結束，接下

來，活動 B、C、D 可以平行執行，所以活動 B 在 12:40 開始執行，14:40 結束，而活動 C 在 12:35 開始執行，13:30 結束，活動 D 在 13:45 開始執行，14:20 結束，最後，活動 E 則在 14:45 開始執行，15:00 結束，利用時間為橫軸，將這些活動的執行時間表示如圖 4-5 所示。



圖<4-5>

從圖 4-5 中，可以很容易發現，活動與活動的執行時間可能存在三種關係：(1)不交集(none)，如活動 A 執行完成後，活動 B 才開始執行，所以這兩個活動的執行時間並未有交集。(2)部份交集(overlap)，如活動 C 與活動 B，活動 C 先開始執行，但兩活動有一部份的時間是一樣的(因為活動 B、C 是平行的，兩活動可能平行執行)。(3)包含(contain)，如活動 B 和活動 D 的執行時間，因為活動 B 執行較久，執行時間可以完全包含活動 D 的執行時間。所以從圖 4-2 與圖 4-5 中可以知道，兩兩活動執行時間會有如上述的三種關係。將這三種執行時間關係整理如圖 4-6 所示。



圖<4-6>

從上述的例子中，可以發現非常重要的一點：在有向圖中，如果活動 X 與活動 Y 之間有一邊(X→Y)，則活動 X 的執行開始時間必然比活動 Y 的執行開始時間要早，且這兩個活動的執行時間，必沒有交集(none)。如圖 4-2 中，活動 A、活動 B 有一邊 A→B，而圖 4-5 中活動 A 的執行開始時間是 12:00，比活動 B 的執行開始時間 12:40 要早，且活動 A、B 的執行時間不交集。同樣的情形也發生在活動 A 與活動 C、活動 A 與活動 D、活動 B 與活動 E、活動 C 與活動 E、活動 D 與活動 E。這些都是因為，如果有向圖中有一邊(設為 X→Y)，則必須活動 X 執行完成後，活動 Y 才能開始執行，所以會有這樣的關係。

然而，在圖 4-5 中，活動 A 與活動 E 也滿足這樣的條件，但在圖 4-2 中，

卻沒有 A→E 邊。這是因為在圖 4-2 中，活動 E 在其它活動(如活動 B、C、D)之後執行，而這些活動則在活動 A 之後執行。換句話說，活動 E 是因為遞移，而在活動 A 之後執行。因此，要找出有向圖中，與活動 A 相連(A→X)的活動 X，應該由滿足上述條件的活動中，選擇最接近活動 A 的活動。因為在所有滿足條件的活動中，越靠近活動 A 就越有可能在有向圖中有這個邊。

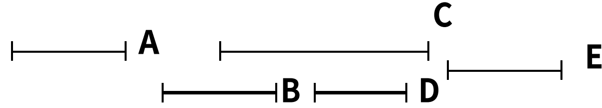
因此，我們有個想法：在一個活動的執行後，不交集而緊接著執行的活動，較有可能是這個活動的下一個活動(即有向圖中，這兩個活動間有一邊存在)。所以，是不是可以利用這樣的想法，由每一個流程例中，找出一個活動可能的下個活動，最後再將所有流程例合起來，應該就可以找出一個活動所有可能的下個活動了。

將上述想法說明，整理成演算法 1 如下:

Algorithm 1

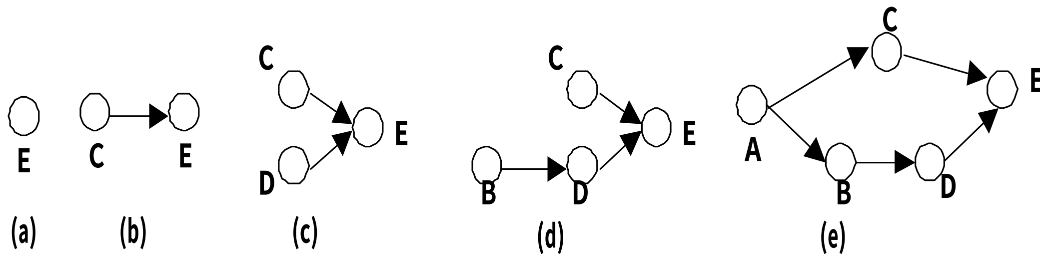
```
Step0. Let Nonegraph = (V,E), V=ϕ, E=ϕ
Step1. Let Obverse_list = the ascendant order of all activities by their start time
           Reverse_list = the descendant order of all activities by their end time
Step2. Get first activity(e) from Reverse_list, add node e to Nonegraph
Step3. Get an activity(u) from Reverse_list, add node u to Nonegraph
Step4. Search Obverse_list, let I = the position of activity u+1
Step5. Test = 0
       While test=0 do
           Read I_th activity from Obverse_list as activity v
           If end time of activity u < start time of activity v
           Then
               Test = 1; Add edge (u,v) to Nonegraph; max= end time of activity v
           Else
               I = I+1
       End
Step6. Test = 0; J= I +1; Read I_th activity from Reverse_list as activity t
       While test=0 and J<=n do
           Read J_th activity from Obverse_list as activity o
           If start time of activity o < end time of activity t
           Then
              J = J+1
              If start time of activity o < max
                  Then add edge (u,o) to Nonegraph
                  Else Test= 1
              If end time of activity o < max then max = end time of activity o
           Else
              Test = 1
       End
Step7. Goto step3 untio no other activity in the Reverse_list
Step8. Return E
```

我們可以利用圖 4-7 的例子來說明這個演算法。假設圖 4-7 是一次流程執行後所有活動執行時間的記錄，如下所示:



圖<4-7>

演算法 1 中，step0 首先將 Nonegraph 設為一空圖形，而 step1 則先將所有活動依開始時間由小到大排序，所以，Obverse_list 為{A、B、C、D、E}，也將所有活動依結束時間由大到小排序，所以，Reverse_list 為{E、C、D、B、A}。接下來，step2 讀取 Reverse_list 中第一個活動 E，並將這個節點 E 加入 Nonegraph 中，如圖 4-8(a)所示。



圖<4-8>

同樣地，step3 由 Reverse_list 中讀取活動 C，將節點 C 加入 Nonegraph 中。接著執行 step4，搜尋 Obverse_list，找到活動 C 在 Obverse_list 中所處的位置。因為活動 C 是 Obverse_list 的第三個活動，所以 I 值被設為 4。接下來的 step5，目的在找出開始時間晚於活動 C 的結束時間、且最接近的活動。因為 Obverse_list 是依所有活動的開始時間由小到大來排列的，所以尋找符合目的的活動，只要由活動 C 所處的位置繼續往前找即可。因此時 I 值為 4，所以讀取 Obverse_list 中第四個活動 D，因為活動 D 的開始時間比活動 C 的結束時間早，所以並不符合要求。I 值加上 1 成為 5 且重覆 step5 迴圈，讀取 Obverse_list 中第五個活動 E，因為活動 E 的開始時間比活動 C 的結束時間晚，所以，活動 E 是第一個找到符合要求的活動，所以也是最接近的。將 C→E 邊加入 Nonegraph 中，如圖 4-8(b)所示。接著執行 step6，step6 的目的，在找出所有比活動 E 晚開始、執行時間與活動 E 有交集、且與活動 C 之間並非遞移的活動。要找出比活動 E 晚開始的活動，而現在的 I 值，是活動 E 在 Obverse_list 中的位置，所以也只要由位置 I 繼續往前找即可。因此，設 J 值等於 I+1(而為 6)。此時，因為 J 值為 6，超過這個流程例中活動個數(設為 n)，代表沒有活動比活動 E 晚開始，所以直接結束 step6 迴圈。接著執行 step7，而回到 step3。

接下來，step3 中由 Reverse_list 中讀取活動 D，將節點 D 加入 Nonegraph 中。執行 step4，搜尋 Obverse_list，找到活動 D 在 Obverse_list 中所處的位置。因為活動 D 是 Obverse_list 的第四個活動，所以 I 值被設為 5。接下來的 step5，因此時 I 值為 5，所以讀取 Obverse_list 中第五個活動 E，因為活動 E 的開始時間比活動 D 的結束時間晚，所以，活動 E 是找到最接近而符合要求的活動。將 D→E 邊加入 Nonegraph 中，如圖 4-8(c)所示。接著執行 step6，因此時，J 值等於 I+1 而為 6，代表也沒有活動比活動 E 晚開始，所以直接結束 step6 迴圈。接著執行 step7，而回到 step3。

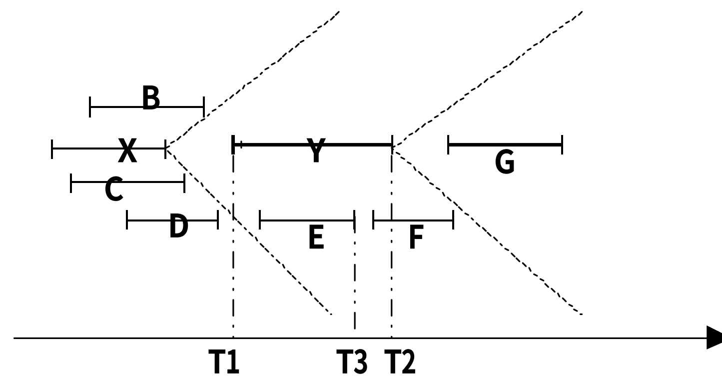
此時，step3 由 Reverse_list 中所讀取的活動是 B，將節點 B 加入 Nonegraph 中。執行 step4，搜尋 Obverse_list，找到活動 B 在 Obverse_list 中所處的位置。因為活動 B 是 Obverse_list 的第二個活動，所以 I 值被設為 3。接下來的 step5，因此時 I 值為 3，所以讀取 Obverse_list 中第三個活動 C，因為活動 C 的開始時間比活動 B 的結束時間早，不符合要求，所以 I 值加 1 成為 4 且重覆迴圈。因此時 I 值為 4，所以讀取 Obverse_list 中第四個活動 D，因為活動 D 的開始時間比活動 B 的結束時間晚，所以活動 D 是找到最接近而符合要求的活動。將 B→D 邊加入 Nonegraph 中，如圖 4-8(d)所示。接著執行 step6，因此時，J 值等於 I+1(而為 5)，所以讀取 Obverse_list 中第五個活動 E，但活動 E 並不滿足與活動 D 有交集的條件，所以直接結束 step6 迴圈。接著執行 step7，而回到 step3。

同樣的方法，step3 由 Reverse_list 中讀取活動 A，將節點 A 加入 Nonegraph 中。執行 step4，搜尋 Obverse_list，找到活動 A 在 Obverse_list 中所處的位置。因為活動 A 是 Obverse_list 的第一個活動，所以 I 值被設為 2。接下來的 step5，因此時 I 值為 2，所以讀取 Obverse_list 中第二個活動 B，因為活動 B 的開始時間比活動 A 的結束時間晚，所以活動 B 是找到最接近而符合要求的活動。將 A→B 邊加入 Nonegraph 中。接著執行 step6，因此時，J 值等於 I+1(而為 3)，所以讀取 Obverse_list 中第三個活動 C，活動 C 的開始時間早於活動 B 的結束時間(代表二活動有交集)，所以活動 C 也滿足條件，將 A→C 邊加入 Nonegraph 中，如圖 4-8(e)所示。重覆迴圈，讀取 Obverse_list 中第四個活動 D，活動 D 的執行時間與活動 B 沒有交集，所以結束 step6 迴圈，接著執行 step7，因 Reverse_list 中沒有其它活動而結束。因此，最後所傳回的邊是：{A→B、A

→C、B→D、C→E、D→E}。

綜合以上所述，演算法 1 的主要方法是先找出一個活動(設為 X)結束之後，第一個執行時間不交集的活動(設為 Y)，並考慮那些開始時間介於活動 Y 的執行時間的所有活動(設為 Z)，而認為活動 Y、活動 Z 是活動 X 符合前述條件且非遞移而產生的活動。

這個概念可以用圖 4-9 來說明，對活動 X 而言，所有比活動 X 晚開始的活動，可以分為二類，第一類執行時間與活動 X 有交集，如圖 4-9 的活動 B、C、D。這一類因為與活動 X 執行時間交集，所以不可能與活動 X 有執行順序關係。第二類則是執行時間與活動 X 沒有交集，如活動 Y、活動 E、活動 F、活動 G。這類才是可能與活動 X 有執行順序關係的活動，所以演算法 1 的 step5，目的就在跳過第一類的活動，直到第二類的第一個活動。



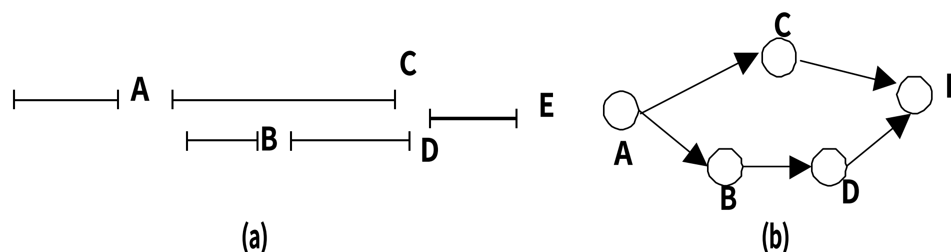
圖<4-9>

如圖 4-9，活動 Y 是第二類最早開始的活動，如果第二類活動中，有活動與活動 Y 的執行時間沒有交集，如活動 G，則這些活動的開始時間必晚於活動 Y 的結束時間，而且這些活動必因活動 Y 而遞移於活動 X，所以這些活動也不用列入考慮。因此真正值得考慮的是，第二類活動中與活動 Y 有交集的活動，即第二類活動中，開始時間介於活動 Y 的開始(T_1)、結束時間(T_2)點之間的活動，如活動 E、活動 F。所以當演算法 1 的 step5 完成時，會找到第二類活動中的第一個活動 Y(此時的 I 值代表 Obverse_list 中活動 Y 的位置)，接著執行的 step6，就準備由第二類活動中的第二個活動開始找起(所以一開始設 J 值為 I+1)，找出開始時間介於 T_1、T_2 之間的活動。

但有一點值得注意的是，如活動 E、活動 F 這些與活動 Y 執行時間有交集而被列入考慮的活動，其中如果有活動的執行時間可以被活動 Y 的執行時間所

包括，如活動 E，則考慮的時間點 T_2 應該被改成活動 E 的結束時間點(T_3)。這是因為，如果有活動的開始時間介於 T_3 與 T_2 之間，則這些活動雖然與活動 Y 有交集，但會因為活動 E 而遞移，如活動 F。所以在考慮這些與活動 Y 有交集的活動時，如果有活動的執行時間可以被活動 Y 所包括，則應將考慮區間的結束時間點往前移至這個活動的結束時間。因此，在演算法 1 的 step5、step6 中，變數 max 一開始被設為第一個活動(Y)的結束時間，如果與活動 Y 有交集的活動中，有比活動 Y 早結束的活動，則重新設定 max 值為這個活動的結束時間，而使得尋找時間區間縮小，開始時間超過 max 值的活動不再被列入考慮。

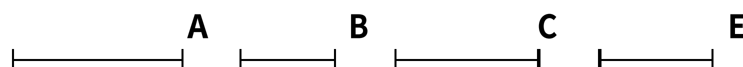
因此，如圖 4-10(a)的流程例，所找出的 Nonegraph 如圖 4-10(b)，對活動 A 而言，不會有 A→D 的邊。



圖<4-10>

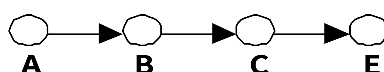
綜合以上的說明，演算法 1 可以用來找出各流程例中，每一個活動之後可能的下一個活動，如圖 4-10(b)中，傳回的邊是:{A→B、A→C、B→D、C→E、D→E}，即從圖 4-10(a)的流程例中，利用演算法所找出的結果，活動 A 之後可能的下個活動是活動 B、活動 C，而活動 B 之後下個活動可能是活動 D，活動 C 之後可能的活動是活動 E，活動 D 之後可能的活動是活動 E。因此，從這個流程例中，演算法 1 就可以找出每個活動可能的下個活動。如果能將所有的流程例依同樣的演算法，找出每一個活動執行後可能執行的下個活動，聯集起來，應該就可以得到每一個活動執行之後所有可能的下個活動了。

而在找到每一個活動執行之後所有可能的下個活動後，如果可以把其中不正確的部份刪去，當然就可以利用剩下的部份找出正確的有向圖。但是那些是不正確的部份呢？可以用圖 4-11 的例子來說明。



圖<4-11>

假設圖 4-11 是圖 4-2 這個流程的一次流程例，利用演算法 1 可以找出的 Nonegraph 如圖 4-12 所示，所以可以找到傳回的邊是:{A→B、B→C、C→E}。其中，可以很明顯的發現，其實有向圖中並沒有 B→C 邊，也就是說活動 C 並不是活動 B 的下個活動。這是因為在原來的圖 4-2 中，活動 B 與活動 C 是平行的，這兩個活動的執行時間可能不交集(none)，可能部份交集(overlap)，也可能是包含(contain)的關係，端視這兩活動的執行時間長短、先後而定。所以在其它的流程例中，這兩個活動之間也可能出現部份交集(overlap)、包含(contain)的關係。



圖<4-12>

因此，如果有兩個活動執行時間出現部分交集(overlap)或是包含(contain)的關係，則這兩個活動必是平行執行的，兩活動不可能是循序的執行關係。因為在第一節流程模式中，我們定義有向圖中兩活動之間的邊(edge)，代表第一個活動執行完成後，第二個活動才能開始執行，也就是說，這個邊代表這兩個活動執行時間的順序關係(dependence)。如果兩活動之間有這個邊、有這個順序關係，這兩個活動的執行時間是不會有交集的。因此，可以利用所有流程例的資料，找出每一個活動執行之後所有可能的下個活動，再將其中也出現於部分交集或包含的邊刪去，也就是刪去不正確的部份，接著只要利用剩下的部份建出有向圖就可以了。

將上述想法說明，整理成演算法 2 如下：

Algorithm 2

Step0. Let G=(V,E), with V is the set of activities of a process and E=ϕ.

Step1. For each instance

(a)call algorithm 1, let none_pair set = return value of algorithm 1

(b)if any two acitvities(u,v), activity u starts before activity v and their temporal duration have partial overlap, add u→v to overlap_pair set.

(c)if any two acitvities(u,v), activity u starts before activity v and activity u terminates after activity v, add u→v to contain_pair set.

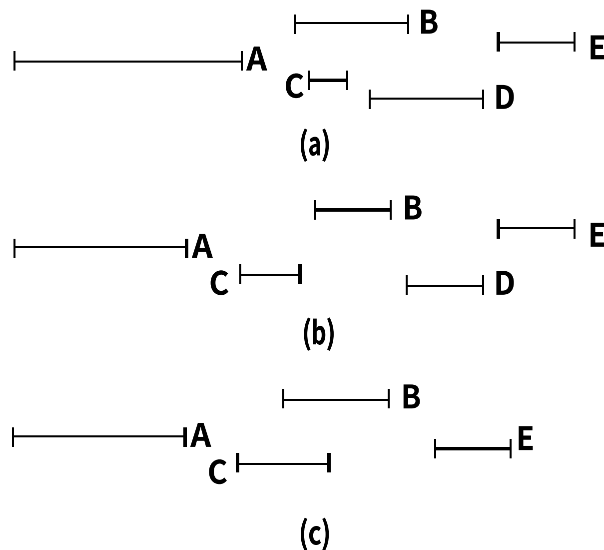
Step2. Union none_pair set, overlap_pair set and contain_pair set of all instances

Step3. If a none_pair or its reverse exist in overlap_pair set or contain_pair set, then delete this none_pair from none_pair set.

Step4. Add each none_pair to E

Step5. Return (V,E).

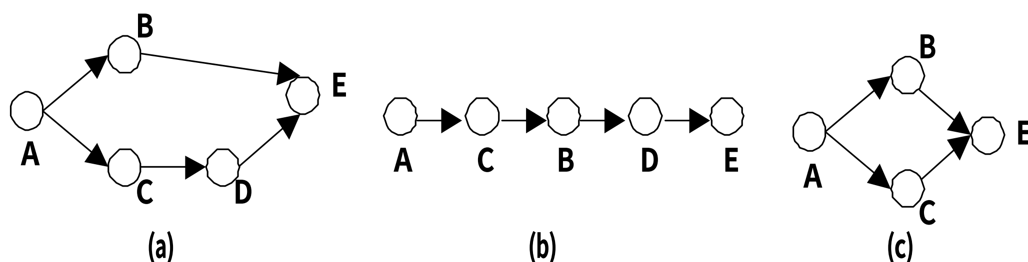
我們可以用圖 4-13 的例子來說明演算法 2。假設圖 4-13 中，(a)(b)(c)分別代表一個流程三次不同的執行記錄，則演算法 2 中，step0 首先令有向圖中的節點為所有活動的集合，邊為空集合。



圖<4-13>

接下來，step1 要將每一個流程例中，每一個活動的執行時間關係找出。所以圖 4-13 在 step1(a)中，利用演算法 1 所找出的 Nonegraph 如圖 4-14(a)所

示，傳回的邊，也就是 none_pair={A→B、A→C、B→E、C→D、D→E}。而 step1(b) 中所找到的 overlap_pair={B→D}(因為活動 B 與活動 D 執行時間部分交集，活動 B 執行開始時間較早，所以寫為 B→D)。step1(c) 中所找到的 contain_pair={B→C}(因為活動 B 與活動 C 執行時間完全包含，活動 B 執行開始時間較早，所以寫為 B→C)。



圖<4-14>

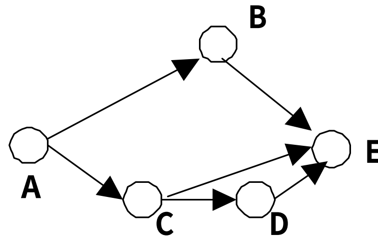
同樣的，圖 4-13(b)在 step1(a)中利用演算法 1 所找出的 Nonegraph 如圖 4-14(b)所示，傳回的邊，也就是 none_pair={A→C、C→B、B→D、D→E}。而 step1(b) 中所找到的 overlap_pair=φ。Step1(c)中所找到的 contain_pair=φ(因為在這個流程例中，沒有活動的執行時間是部份交集或包含的關係)。

最後，圖4-13(c)在step1(a)中利用演算法1所找出的Nonegraph如圖4-14(c)所示，傳回的邊，也就是 none_pair={A→B、A→C、B→E、C→E}。而 step1(b) 中所找到的 overlap_pair={C→B}。step1(c) 中所找到的 contain_pair=φ。

所以演算法 2 中，step3 所聯集起來的 none_pair 集合是：{A→B、A→C、B→D、B→E、C→B、C→D、C→E、D→E}，而 overlap_pair 集合是：{B→D、C→B}，contain_pair 是：{B→C}。

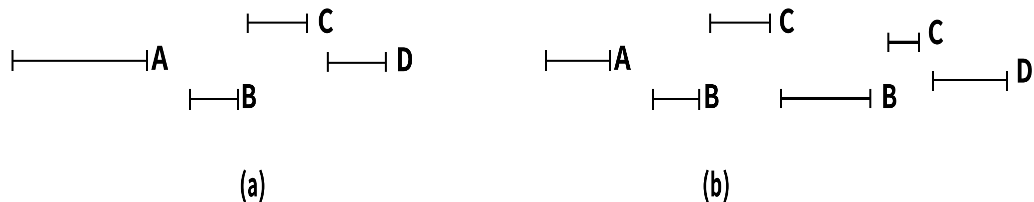
如果兩活動之間有執行順序關係，這兩個活動的執行時間是不會有交集的。利用這個想法，由所有流程例找出的 none_pair 集合中，刪去也存在 overlap_pair 集合或 contain_pair 集合的關係。這可以用一個例子來說明，如果活動 X 執行完成後，活動 Y 才能開始執行，則活動 X、活動 Y 不可能出現的時間關係是部份交集(overlap_pair X→Y，Y→X)、包含(contain_pair X→Y，Y→X)。所以應檢查在 overlap_pair 集合中是否出現 X→Y 或反向的 Y→X，當然也應該檢查在 contain_pair 集合中是否出現 X→Y 或反向的 Y→X，若有，則刪去這個 none_pair。因此，接下來的 step4，將 none_pair 集合中的 B→D、C→B 刪去，所以剩下的 none_pair 集合是{A→B、A→C、B→E、C→D、C→E、D→E}。

最後，利用剩下的 none_pair 所建立的有向圖就如圖 4-15 所示了。



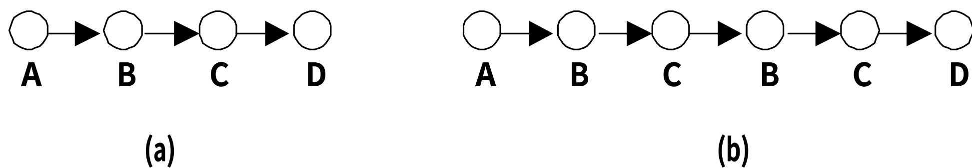
圖<4-15>

另外，必須注意的一點是，演算法是利用每一個實際記錄下來活動的開始時間、結束時間(一段執行時間)來代表一個活動，所以每個記錄下來的活動都被視為不同的活動。而為便於說明，前述各段以活動名稱來代替，實際上則可以給予各記錄下來的活動區間一個唯一的代號。當演算法執行完成，找出一個活動之後，所有可能的下個活動，並刪去不可能的活動後，將有向圖畫出。當然，如果活動是循序、平行執行的，從上述例子中，利用演算法很容易就可以把有向圖畫出來了。那麼，如果流程中有迴圈呢？依然可以用同樣的方法來找出有向圖。我們可以用圖 4-16 的例子來說明。



圖<4-16>

假設圖 4-16(a)(b)是圖 4-3 流程的二個流程例，則圖 4-16(a)利用演算法 1 找出的 Nonegraph 如圖 4-17(a)所示，所以得到的 none_pair 是：{A→B、B→C、C→D}，而 overlap_pair=φ、contain_pair=φ。



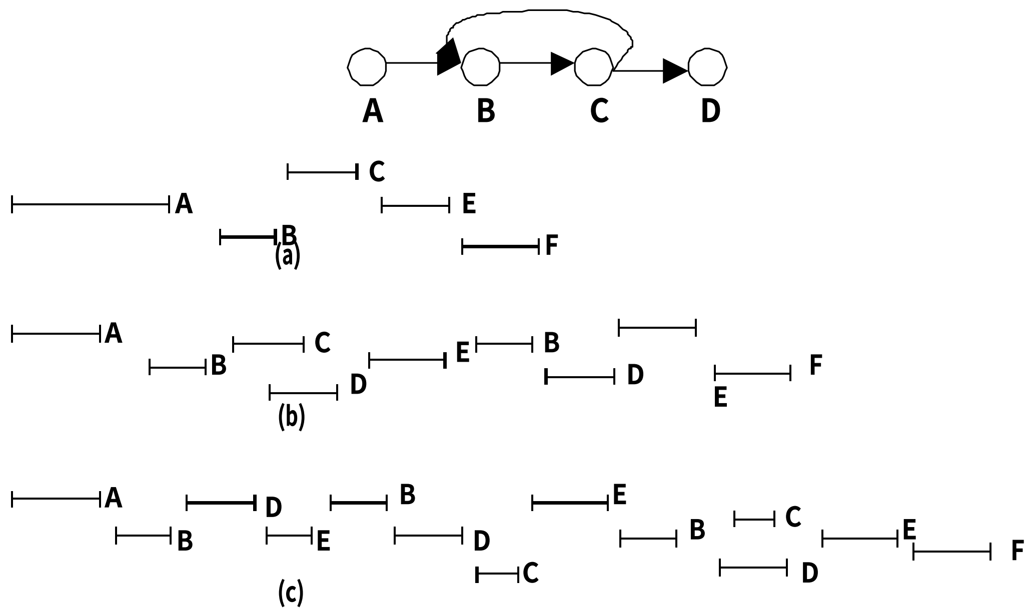
圖<4-17>

圖 4-16(b)利用演算法 1 找出的 Nonegraph 如圖 4-17(b)所示，所以得到的 none_pair 是：{A→B、B→C、C→B、C→D}，而 overlap_pair=φ、contain_pair=

φ。因此演算法 2 中找出的 none_pair 集合是：{A→B、B→C、C→B、C→D}，而 overlap_pair 集合是：{φ}、contain_pair 集合是：{φ}。所以 none_pair 集合刪去也出現在overlap_pair集合、contain_pair集合者後，剩下的none_pair 集合是：{A→B、B→C、C→B、C→D}。因此畫出的有向圖如圖 4-18 所示。

圖<4-18>

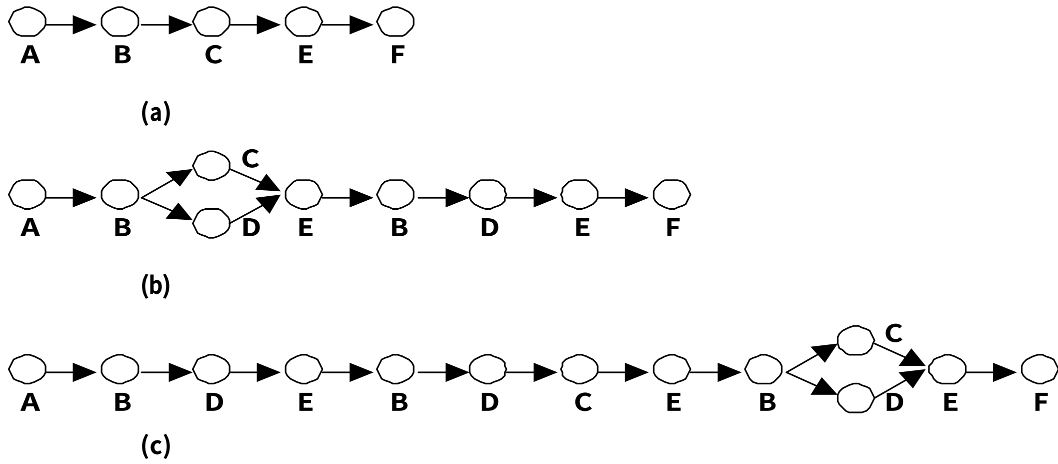
圖 4-3 的迴圈主體是循序執行的，較為簡單，如果類似圖 4-4，迴圈主體



也可以是平行，甚至是巢狀迴圈呢？我們可以分別用圖 4-19 和圖 4-22 的例子來說明：

圖<4-19>

假設圖 4-19(a)(b)(c)是圖 4-4(a)流程的三個流程例，則圖 4-19(a)利用演算法 1 找出的 NoneGraph 如圖 4-20(a)所示，所以得到的 none_pair 是：{A→B、B→C、C→E、E→F}，而 overlap_pair=φ、contain_pair=φ。

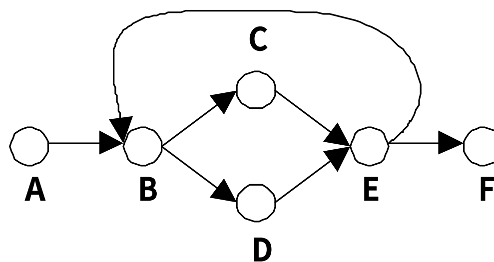


圖<4-20>

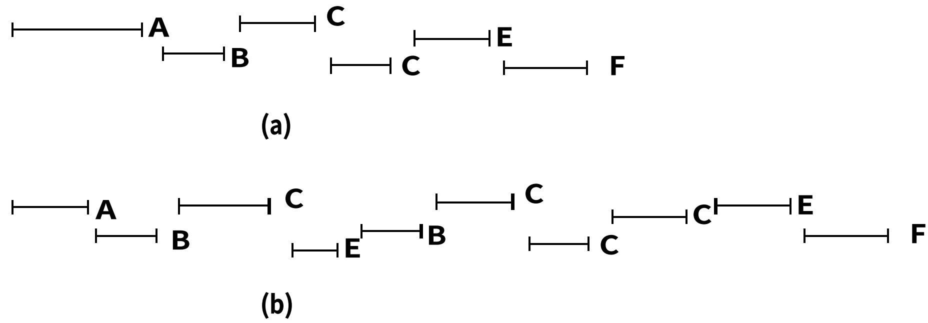
圖 4-19(b)利用演算法 1 找出的 Nonegraph 如圖 4-20(b)所示，所以得到的 none_pair 是：{A→B、B→C、B→D、C→E、D→E、E→B、E→F}，而 overlap_pair={ C→D}、contain_pair=ф。

圖 4-19(c)利用演算法 1 找出的 NoneGraph 如圖 4-20(c)所示，所以得到的 none_pair 是：{A→B、B→C、B→D、C→E、D→C、D→E、E→B、E→F}，而 overlap_pair=ф、contain_pair={D→C}。因此演算法 2 中找出的 none_pair 集合是：{ A→B、B→C、B→D、C→E、D→C、D→E、E→B、E→F}，而 overlap_pair 集合是：{C→D}、contain_pair 集合是：{ D→C}。所以 none_pair 集合刪去也出現在 overlap_pair 集合、contain_pair 集合者後，剩下的 none_pair 集合是：{A→B、B→C、B→D、C→E、D→E、E→B、E→F}。因此畫出的有向圖如圖 4-21 所示。

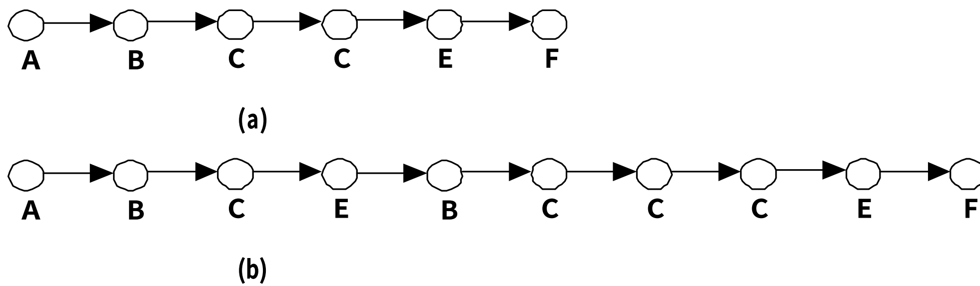
圖<4-21>



另外，假設圖 4-22(a)(b)是圖 4-4(b)流程的二個流程例，則圖 4-22(a)利用演算法 1 找出的 Nonegraph 如圖 4-23(a)所示，所以得到的 none_pair 是：{A→B、B→C、C→C、C→E、E→F}，而 overlap_pair=ф、contain_pair=ф。

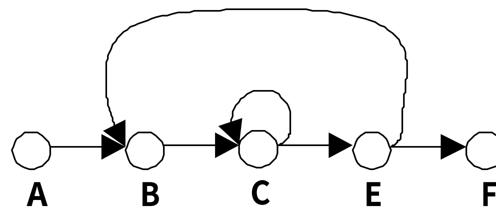


圖<4-22>



圖<4-23>

圖 4-22(b)利用演算法 1 找出的 Nonegraph 如圖 23(b)所示，所以得到的 none_pair 是：{A→B、B→C、C→C、C→E、E→B、E→F}，而 overlap_pair=ф、contain_pair=ф。因此演算法 2 中找出的 none_pair 集合是：{ A→B、B→C、C→C、C→E、E→B、E→F}，而 overlap_pair 集合是：{ф}、contain_pair 集合是：{ф}。所以 none_pair 集合刪去也出現在 overlap_pair 集合、contain_pair 集合者後，剩下的 none_pair 集合是：{ A→B、B→C、C→C、C→E、E→B、E→F}。因此最後畫出的有向圖如圖 4-24 所示。



圖<4-24>

由圖 4-19 與圖 4-22 的例子中，可以發現用同樣的演算法仍可以處理較為複雜的迴圈流程，這是因為演算法中，利用每一個實際記錄下來活動的開始時間、結束時間來代表一個活動(即用執行時間區段來代表一個個的活動)。並找出一個活動之後，所有緊臨的下個活動，代表這個活動所有可能的下個活動，刪去不可能的活動後，再將有向圖畫出，這樣我們應該就可以得到流程的控制

流定義，也就是流程中活動執行的先後順序關序了。

第三節 控制條件萃取演算法

利用演算法 1、2，可以得到一個有向圖，來表示流程中各活動執行的順序關係。除了這些執行順序關係，如第一節流程模式所說明的，有向圖中的每一個邊都附有一個布林函數，當一個活動執行完成後，這個活動所有連出去的邊上的布林函數都會被計算，只有當各邊的布林函數值為”真”時，這個邊所連的下一個活動才會被執行。所以流程的定義，除了要考慮上述的活動執行順序外，也須考慮這些布林函數。

當一個活動執行時，會參考、修改許多的資料，這些資料稱為參考資料(relevant data)，這些資料會被記錄在前述寫入變數事件(write event)中。當這個活動執行完成後，就會依這些參考資料的值，求算所有連出去邊上的布林函數值，來決定如何執行下一個活動。所以，可以選取流程例資料中，(1)一個活動的參考資料值和(2)該活動執行完成後，所有相連活動被執行的集合，組成分析資料，利用一些分類的方法，找出參考資料值落在那些值域時，各連出去邊上的布林函數值為”真”，也就是找出參考資料值落在那些值域時，下一個活動會被執行。

將上述想法說明，整理成演算法 3 如下：

Algorithm3

```
10. For each activity in G, which has more than one outgoing edge
20.     For each outgoing edge(u,v)
        {
30.         let edge_condition_data=Φ
40.         For each instance
            {
50.             read activity data from activity u to next activity u (or last activity of
                the instance)
                {
60.                 IF activity u has outgoing edge (u,v)
                    THEN insert a record (variable values, P) to edge_condition_data
                    Else insert a record (variable values, N) to edge_condition_data
                }
            }
70.         call any classification procedure, edge_condition_data as input, to decide
            outgoing rules of the outgoing edge(u,v)
        }
```

我們可以用圖 4-2 的例子來說明演算法 3。在圖 4-2 中，活動 A 執行完成後，有三個連出去的邊 A→B、A→C、A→D。設活動 A 有一個參考資料：V1(0<V1<100，V1 是整數)。設表 4-2 為 10 個流程例中，所收集活動 A 的參考資料(V1)值(如表 4-2 中"V1"欄所示)與各流程例中，活動 A 執行完成後，所執行相連活動的集合(如表 4-2 中"執行活動"欄所示)。

V1	執行活動	活動 B	活動 C	活動 D
95	B	P	N	N
85	B	P	N	N
75	B	P	N	N
65	B、C	P	P	N
55	C	N	P	N
45	C	N	P	N
35	C、D	N	P	P
25	D	N	N	P
15	D	N	N	P
5	D	N	N	P

表<4-2>

在表 4-2 中，有些 V1 值會導致二個以上的活動被執行，如當 V1 值為 65 時，活動 B、C 都會被執行，V1 值為 35 時，活動 C、D 都會被執行。所以，若利用分類方法來分類時，這兩筆資料無法直接判斷(這是因為一般分類方法，是利用資料的屬性與資料的類別作為輸入，判斷在那些屬性組合下，資料會屬於那一個類別。因此，同一屬性組合的資料，只能屬於同一類，否則就無法判斷了)。因此，我們將上述分類問題轉成多重決策(multi decision)的問題。因為在第一節流程模式中，本研究定義，當活動有數個連出去(outgoing)的邊時，每一個邊上的布林函數都必須被測試，以決定所連的下一個活動是否執行。而且，這些布林函數測試結果是獨立的，不受其它布林函數的成立與否所影響。

另外，在第一節流程模式中，我們也定義循序執行的活動，連出去邊上的布林函數值都是"真"(因為活動是依序執行的)，所以只要考慮那些有超過一個以上連出去邊的活動(代表這個活動執行完成後，進入平行的執行流程)就可以了。因此，圖 4-2 在演算法 3 中，行號 10 只考慮連出去邊在二個以上的活動 A。接著，行號 20，依序考慮活動 A 的每一個連出去邊 A→B、A→C、A→D。首先考慮 A→B 邊，在表 4-2 中(即行號 40，每一個流程例中)，當 V1 值為 95、85、75、65 時，活動 B 會執行，所以這些設為"P"類，都會加入(V1 值，P)到

edge_condition_data 中。而 V1 值為 55、45、35、25、15、5 時，活動 B 不會執行，所以設為”N”類，都會加入(V1 值，N)到 edge_condition_data 中，分類如表 4-2 中”活動 B”欄所示(這些加入資料到 edge_condition_data 的動作，在行號 60 執行)。所以此時的 edge_condition_data 如表 4-3”B”所示。

B	C	D
EXAMPLE FILE	**EXAMPLE FILE**	**EXAMPLE FILE**
%V1 Cond_B	%V1 Cond_B	%V1 Cond_B
95 P;	95 N;	95 N;
85 P;	85 N;	85 N;
75 P;	75 N;	75 N;
65 P;	65 P;	65 N;
55 N;	55 P;	55 N;
45 N;	45 P;	45 N;
35 N;	35 P;	35 P;
25 N;	25 N;	25 P;
15 N;	15 N;	15 P;
5 N;	5 N;	5 P;

表<4-3>

接下來，只要把表 4-3 的資料，透過一些分類(classification)方法來分類就可以了(行號 70)。例如 cn2[Clar89]，分類後的結果如表 4-4 中”Condition rule B”欄所示。由表 4-4 的”Condition rule B”欄，可以很容易地看出，當 V1 值大於 60 時，A→B 邊上的布林函數會為”真”。

Condition rule B	Condition rule C	Condition rule D
--------------------- |UN-ORDERED RULE LIST| *---------------------* IF v1 > 60.00 THEN condition = P [4 0] IF v1 < 60.00 THEN condition = N [0 6] (DEFAULT) condition = N [4 6]	*---------------------* |UN-ORDERED RULE LIST| *---------------------* IF 30.00 < v1 < 70.00 THEN condition = P [4 0] IF v1 > 70.00 THEN condition = N [0 3] IF v1 < 30.00 THEN condition = N [0 3] (DEFAULT) condition = N [4 6]	*---------------------* |UN-ORDERED RULE LIST| *---------------------* IF v1 < 40.00 THEN condition = P [4 0] IF v1 > 40.00 THEN condition = N [0 6] (DEFAULT) condition = N [4 6]

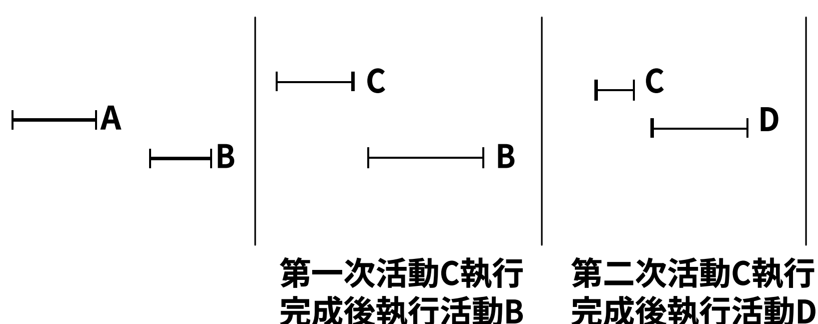
表<4-4>

同樣的方法，回到行號 20，對活動 A 的 A→C 邊來考慮。所產生的分類如表 4-2”活動 C”欄所示，而產生的 edge_condition_data 如表 4-3”C”欄所示，而分類後的結果如表 4-4”Condition rule C”欄所示。同樣的，再次回到行號 20，對活動 A 的 A→D 邊來考慮。所產生的分類如表 4-2”活動 D”欄所示，而產生的 edge_condition_data 如表 4-3”D”欄所示，而分類後的結果如表 4-4”Condition rule D”欄所示。透過上述方法，就可以知道 V1 值落在某個值域時，活動 B、C、D 是否會被執行了。

但是流程中可能會有迴圈，也就是說在每個流程例中，連出去邊數大於 1 的活動 u，可能執行一次以上。以圖 4-16、4-18 的例子來說明，圖 4-18 中連出

去邊數大於 1 的活動只有活動 C，因此以活動 C 來說明。活動 C 執行完成後可能執行活動 B(迴圈重覆執行)或執行活動 D(離開迴圈)。所以活動 C 可能執行一次以上，在圖 4-16(b)的流程例中，活動 C 即執行二次。第一次，活動 C 執行完成後 C→B 邊的條件為真，所以執行活動 B。而第二次，活動 C 執行完成後，C→D 邊上的條件為真，因此執行活動 D，也就是結束迴圈。所以，在同一個流程例中，活動 C 可能執行一次以上，而且每一次活動 C 執行完成後，會選擇連出去邊上條件為"真"的下一個活動來執行。因此，從同一個流程例中，活動 C 執行後究竟執行活動 B 還是活動 D，可能就可以得到數次的資料。

那麼怎麼辨別同一個流程例中各次的資料呢？以圖 4-16(b)為例，只要判斷活動 C 執行完成後至下次活動 C 執行之前(或這個流程例結束)，在這個區間內所執行的活動是否有活動 B 或活動 D 就可以了，概念如圖 4-25 所示。所以利用圖 4-25，可以知道在圖 4-16(b)的流程例中，第一次活動 C 執行之後，執行活動 B(若對 C→B 邊來考慮時，將[參考資料，P]加入 edge_condition_data 的檔案中，也就是說對 C→D 邊來考慮時，將[參考資料值，N]加入 edge_condition_data 的檔案中)。而第二次活動 C 執行之後，執行活動 D(若對 C→B 邊來考慮時，將[參考資料，N]加入 edge_condition_data 的檔案中，也就是說對 C→D 邊來考慮時，將[參考資料值，P]加入 edge_condition_data 的檔案中)。這個概念如行號 50 所表示。而利用這個方式，就可以將迴圈條件視為一般的平行分支條件了。也就是說，只要考慮迴圈重覆執行將使得某些活動執行一次以上，所以將同一個流程例分隔成數次活動執行的條件就可以了。



圖<4-25>

最後，如果針對一個有許多連出去邊的活動，所找出來各邊的布林函數為"真"的參考資料值域完全相同時，這個分支點是 AND(因為每一組參考資料值都會使所有連出去邊的布林函數為"真")。如果值域部份重疊時，這個分支點是

OR(因為某些參考資料值會使部份連出去邊的布林函數為”真”)，當然，如果值域完全不相同時，這個分支點是 XOR(因為每一組參考資料值都只會使一個連出去邊的布林函數為”真”)。

第四節　時間複雜度(time complexity)

在第二節的演算法裏(演算法 1、2)，演算法是利用多個流程例資料來萃取流程，所以流程例數目的多寡、流程中活動數目的多寡都將影響演算法的執行時間。

因此，假設流程中有 n 個活動，而收集了 m 個流程例。考慮演算法 2，在演算法 2 中，最關鍵費時的步驟是 step1，因為 step1 必須為每一個流程例，叫用演算法 1 建立 Nonegraph，而找出所有的 None_pair 集合。

而對演算法 1 而言，如果有一個節點欲加入 Nonegraph，則必須先搜尋 Obverse_list，找到這個活動在 Obverse_list 中的位置，設為 I。皆著以同方向，從位置 I 繼續往前找尋 Obverse_list，在最差的情形下，找尋至 Obverse_list 中最後一個活動。所以，當有一個節點欲加入 Nonegraph 時，最多搜尋 Obverse_list 中的 n 個活動。因此，處理每一個活動加入 Nonegraph 中的時間複雜度是 O(n)。當然，將一個流程例中 n 個活動都加入 Nonegraph，時間複雜度就是 $O(n^2)$了。

所以，在有 m 個流程例的情形下，整個演算法 2 的時間複雜度是 $O(mn^2)$。如果流程中有迴圈，設此迴圈重覆執行 k 次，則一個流程例最多可能的活動數就成為 kn，而時間複雜度就成為$(m(kn)^2)$了。

第五章. 錯誤資料處理

在這一章中，我們將考慮錯誤資料(noise)的問題，即演算法對錯誤資料的處理。演算法主要是由每一個流程例中，找出每一個活動緊臨的下一個活動，來代表這個活動可能的下個活動(演算法中認為兩活動間可能有執行順序，寫為none_pair)，收集了所有流程例的資料後，便可以找出這個活動所有可能的下個活動。再利用兩活動時間有交集(演算法中寫為 overlap_pair 或 contain_pair)，則不可能有執行順序關係的特性，自所有可能的下個活動中刪去，以找出最後的有向圖。這樣的演算法是利用多個流程例資料而來，但大量的資料中可能會有錯誤資料(noise)，演算法也必須考慮這個問題。

因此，可以考慮對所記錄下來的活動執行關係(包括 none_pair、overlap_pair 或 contain_pair)，計算其次數，然後在次數上加上一個門檻值(threshold)。如果記錄下來的活動執行關係次數太少，低於門檻值，則認為這個關係可能是由錯誤資料而來，而將這些關係刪去，避免影響最後的結果。

但是門檻值應如何設定呢？讓我們考慮一下，如果門檻值設得太低，自然無法過濾掉錯誤資料，但如果設得太高，則又可能將正確的活動執行關係去掉。因此必須設定一適合的門檻值，才能過濾不正確的錯誤資料而不影響正確的活動執行關係。那麼，適合的門檻值如何設定呢?

在考慮門檻值設定的高低前，讓我們先參考圖 3-7。圖 3-7 是兩活動間所有可能記錄的關係。演算法執行時，兩活動依其執行時間先後、長短，可記錄兩者之間有如圖 3-7(a)-(f)的關係(只有這六種關係)。另外，對於門檻值的設定，我們有一些基本假設：

(1)錯誤資料少於正確資料：門檻值的設定，目的在過濾錯誤資料，而剩下正確的資料。所以錯誤資料數應比門檻值小，而門檻值應比正確資料數小，才能藉由門檻值的設定達到過濾錯誤資料剩下正確資料的目的。如果錯誤資料數多於正確資料，則不可能設定一個門檻值，可以過濾次數多的錯誤資料卻可以留下次數少的正確資料。所以，考慮門檻值的設定，首要假設是錯誤資料數少於正確資料數。

(2)假設錯誤資料隨機(random)發生：錯誤資料的來源，可能是多記錄了活動、少記錄了活動或是記錄下來的活動時間不正確，而產生異樣的活動

執行關係。因為門檻值的設定，目的在處理記錄下來、不正確的資料，所以少記錄活動的錯誤，不可能由門檻值的設定來解決。因此，門檻值的設定目的在解決多記錄、錯誤記錄的活動執行。這些記錄下來的錯誤資料，假設其隨機發生，也就是圖 3-7(a)-(f)皆有可能發生，且發生機率相等。

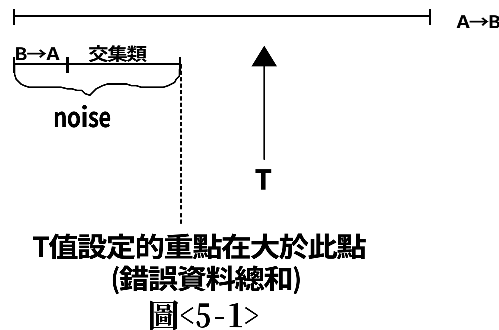
(3)若兩活動之間獨立，假設活動執行關係隨機(random)發生：如果有兩個活動(設為活動 A、活動 B)，它們是平行、獨立的，則前述圖 3-7(a)-(f)的執行關係都有可能發生。當然，如果有某些語意(semantic)資訊提供，如活動 A 執行時間很久，則活動 A、B 執行時間之間的關係，可能像圖 3-7(b)-(e)的機會較大，而像圖 3-7(a)、(f)的機會較小。但在沒有這些語意資訊的提供下，我們假設如果兩活動獨立，則這兩個活動執行時間關係隨機發生，也就是出現如圖 3-7 的六種關係的機率均相等(皆為 1/6)。當然，如果使用者可以有某些語意資訊，也可以用更準確的機率值來代替。

在這三個假設之下，考慮本研究的演算法 1、2。在演算法 2 的 step1 中，為每一種記錄下來的關係，加一個記數變數(counter)，以記錄這種執行關係的次數。另外，因為兩個活動被記錄下來的執行時間，如果有交集的關係(包括 contain_pair 和 overlap_pair)，也就是如圖 3-7(b)-(e)的四種關係，則會認為這兩個活動彼此平行、獨立，而刪去順序關係(如圖 3-7(a)、(f)的關係)；若沒有這四種關係，則承認如圖 3-7(a)、(f)的順序關係。所以，兩個活動之間究竟是有順序關係或獨立，端視如圖 3-7(b)-(e)的交集關係是否存在。

因此，我們將圖 3-7(b)-(e)的四種關係視為一類，圖 3-7(a)、(f)的順序關係，分別各視為一類，而將兩活動之間所有的執行關係分為三類，並分別計算各類關係的總次數。如果流程中有 n 個活動，收集了 m 個流程例，而錯誤資料的比率是 ε，並假設某兩個活動被記錄下來的次數有 k 次，則門檻值的設定必須考慮下列二種狀況：

(1)設有兩活動 A、B，這兩個活動之間有一定的執行順序關係(如活動 A 執行完成後，活動 B 才能開始執行，寫為 A→B)：則活動 A、B 在演算法中所記錄下來的關係，除了圖 3-7(a)以外，皆為錯誤資料。因此若出現圖 3-7(b)至圖 3-7(f)的記錄時，皆為錯誤資料(noise)，兩活動所記

錄如圖 3-7(b)至圖 3-7(f)的關係，次數加總應低於門檻值。此時理想的門檻值(T)設定如圖 5-1 所示。設定的重點在，T 值必須高於錯誤資料(noise)總數。



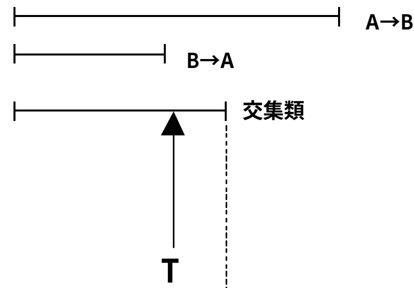
圖<5-1>

也就是說，如果門檻值的設定比圖 5-1 中錯誤資料總數和(包括如圖 3-7(b)至圖 3-7(f)的執行關係次數總和)低，則此門檻值的設定便是錯誤的。因此，假設門檻值設定為 T，而有至少 T 次以上的錯誤資料，這個錯誤機率是[Corm90]

$$P_1 = \Sigma^{k}_{x=T} \binom{k}{x} \varepsilon^{x} (1-\varepsilon)^{k-x} \leqq \binom{k}{T} \varepsilon^{T}$$

這個錯誤機率是門檻值 T 設定得太低，使得錯誤資料(noise)比門檻值高了。

(2)設有兩活動 A、B，這兩個活動之間是獨立的，沒有一定的執行順序關係：則活動 A、B 在演算法中所記錄下來的關係，圖 3-7(a)至圖 3-7(f)皆有可能。而且演算法是利用圖 3-7(b)至圖 3-7(e)的關係，以兩活動執行時若執行時間有交集，則兩活動不可能有執行順序關係的特性，而刪去活動 A、B 所記錄下來類似圖 3-7(a)、(f)的關係。因此，圖 3-7(b)至圖 3-7(e)的次數總和將必須高於門檻值(T)，才能利用這些關係將活動 A、B 之間的執行順序關係刪去。此時理想的門檻值(T)設定如圖 5-2 所示。設定的重點在，T 值必須低於類似圖 3-7(b)-(e)這四種交集關係的次數總和。



T值設定的重點在小於此點
(交集類關係總和)

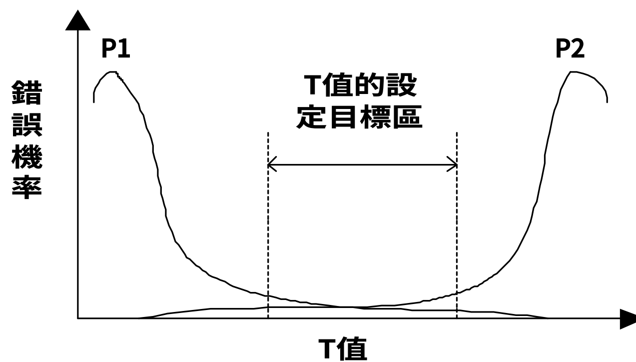
圖<5-2>

也就是說，所記錄下來類似圖 3-7(a)、(f)的關係，其次數總和必須小於 k-T 次(因為全部記錄次數為 k)，所以當類似圖 3-7(a)、(f)關係的次數總和大於 k-T 次時，門檻值的設定便是錯誤的。又因為活動 A、B 獨立，所以假設圖 3-7(a)至圖 3-7(f)發生的機率皆相等(為 1/6)，圖 3-7(a)、(f)的機率和是 1/3。所以，假設門檻值設定為 T，則發生錯誤的機率是

$$P_2=\sum_{x=k-T+1}^{k}\binom{k}{x}(1/3)^{x}(2/3)^{k-x}\leqq\binom{k}{k-T}(1/3)^{k-T}$$

這個錯誤機率是門檻值 T 設定得太高，使得正確的交集類執行順序關係比門檻值低了。

由上述說明可知，T 值的設定可能會發生兩種錯誤，圖 5-3 為這兩個錯誤機率的圖形，若 T 值增加，犯第一種錯誤 P_1 的機率會逐漸變小，但犯第二種錯誤 P_2 的機率會逐漸變大。反之，若 T 值減少，犯第一種錯誤 P_1 的機率會逐漸變大，但犯第二種錯誤 P_2 的機率會逐漸變小。而理想的 T 值便是設定在中間兩種錯誤機率皆接近 0 的區域。



圖<5-3>

由圖 5-3 可知，任意 T 值的設定，可能犯下最大的錯誤機率是 $\max(P_1, P_2)$。因此，為了取一個較為適當的門檻值，可以令這兩個錯誤機率相等，則可以得到 $\varepsilon^T=(1/3)^{k-T}$ 的關係。所以，如果可以知道 ε 和 k 值，我們就可以估計一個較適當的 T 值了。而其中 ε 值是所有資料中發生錯誤的機率，這個部份可以用一些統計抽樣的方法來估算這個值。至於 k 值，則是活動 A、B 在演算法執行時所計錄下來的關係次數，在第四章第二節的演算法 2 中，step2 完成後，就可以知道活動 A、B 各種關係的次數，自然就可以知道 k 值。所以只要在 step2 完成後，就可以利用 k 值與估計的 ε 值，來計算較適合的門檻值了。

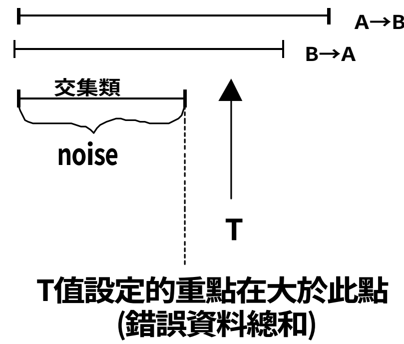
當門檻值計算出來，就可以作兩個檢查動作。第一個動作自然是檢查兩個活動之間是否獨立，也就是檢查如圖 3-7(b)-(e)四種關係次數總和是否大於門檻值 T。如果是，代表兩活動之間獨立，必須承認這四種關係，不能將這四種關係視為錯誤資料而刪去。如果不是，代表兩活動之間不是獨立的關係，這四種關係應被視為錯誤資料，而準備刪去。

準備刪去這四種關係，代表這兩個活動之間並非獨立，而是有某方向的順序執行關係。所以應作第二個檢查動作，將四種交集類的關係加上不同的順序關係，而得到五種關係的次數總和，以五種關係的次數總和作檢查，如果低於門檻值，則不僅刪去四種交集類的關係，也刪去該相加的順序關係。以圖 5-1 為例，當四種交集關係加上 B→A 的順序關係時，仍會低於適當設定的 T 值，因為這五種關係都是錯誤資料，所以將四種交集關係及 B→A 的順序關係一併刪去。當然，如果是四種交集關係加上 A→B 的順序關係時，會高於適當設定的 T 值(因為 A→B 的順序關係是正確的，會高於門檻值)，此時，自然只有四種交集的關係被刪去。

也就是說，當四種交集關係的次數總和小於門檻值 T，而認為兩活動並非獨立，準備刪去四種交集關係時，應一併檢查不同方向的順序關係與四種交集關係的次數加總(而成五種關係的次數總和)，如果次數和也小於門檻值，則將這五種關係皆刪去，如果大於，則只要刪去四種交集關係就可以了。

當然，如果我們考慮迴圈，則可能在兩個活動 A、B 之間，有 A→B 及 B→A 的順序關係，兩種順序關係都是正確的，則此時就只有四種交集關係是錯誤資料，而門檻值的設定應如圖 5-4 所示。在這種情形，其實門檻值設定的重點仍如圖 5-1，目的在高於錯誤資料的次數總和，所以公式並不變動。只是此時的

錯誤資料只有四種交集的關係，如果四種交集關係的次數總和小於門檻值，而認為兩活動獨立準備刪去四種交集關係時，不論是 A→B 的順序關係加上四種交集關係，或是 B→A 的順序關係加上四種交集關係，都會高於門檻值，而只能把四種交集關係刪去了。



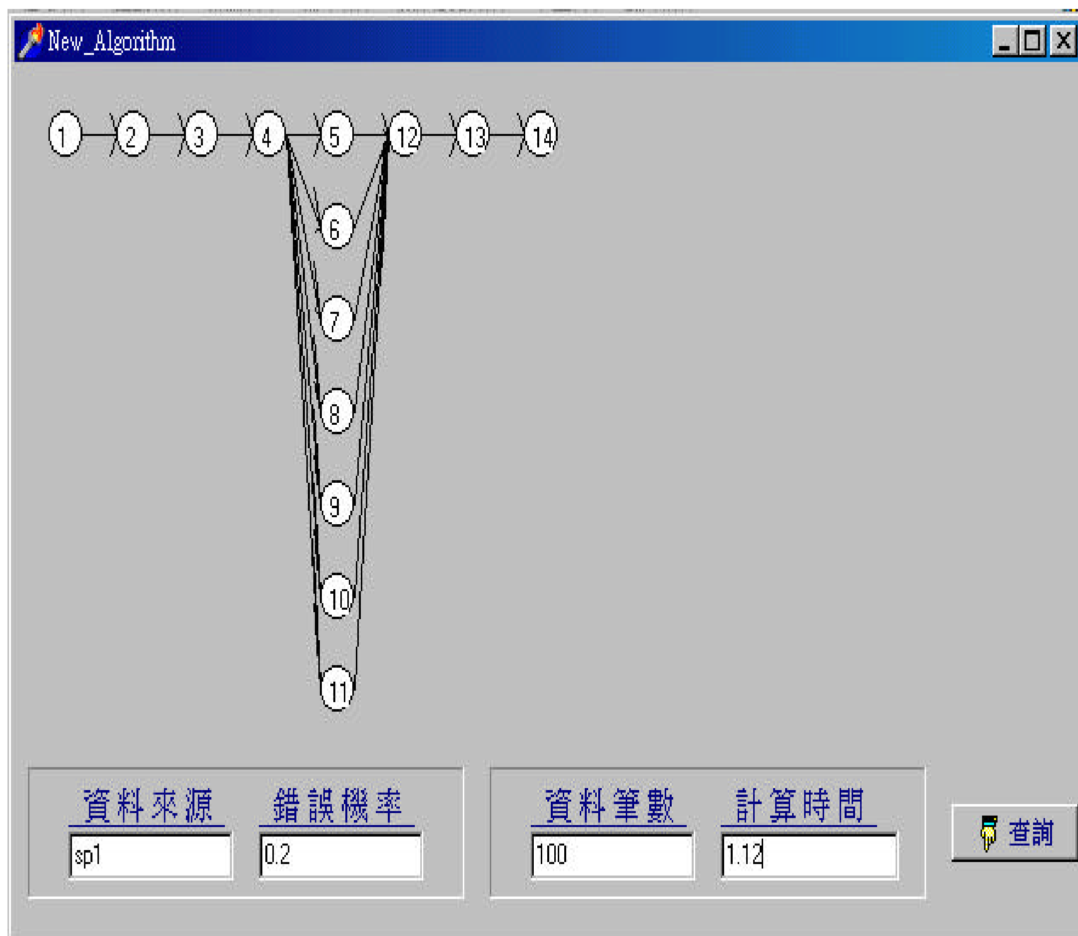
圖<5-4>

因此，我們可以修改原來的演算法 2，在 step1 中對各種執行關係加上一記數變數(counter)，計算各關係的次數，而在 step2 將各次流程例所找出的關係聯集起來時，將各種關係依前述分類加總次數，然後計算門檻值，利用門檻值刪去錯誤資料(noise)，再繼續進行原來演算法 2 的 step3，就可以過濾錯誤資料的影響了。將考慮錯誤資料而修改的演算 2，列示如附 3。

第六章. 演算法效能評估

將前述演算法說明，實作雛形系統。在 Win95/NT 下，採用主-從式架構，將流程執行所記錄的歷史資料存於後端資料庫內，資料庫採用 Paradox，而前端則是程式的執行，開發工具採用 Borland Delphi 3.0。

如圖 6-1，在畫面左下方，是資料輸入區，使用者可以輸入所記錄下來的流程例檔案，並給定估計的資料錯誤比率(ε)。然後選擇右下角的查詢鍵，雛型系統會讀入這些流程例資料，依演算法萃取出流程定義，以有向圖的方式表現於雛型系統的上半部。並統計流程例資料筆數及萃取流程所費的時間資料，顯示於右下角的資料輸出區。



圖<6-1>

因為第三章第二節的萃取演算法中，KTAIL 演算法與 Markov 演算法皆以 FSM 圖為輸出，與本研究演算法輸出模式不同，而且這兩個演算法需設定參數值，若參數值設定不同，所找出的流程定義會有相當程度的差異，難以比較。因此，在本節的效能評估中，未將 KTAIL 演算法與 Markov 演算法列入評估對象。

而由於 Agrawal 演算法也採用有向圖為流程模式，因此，為比較評估與 Agrawal 演算法的不同，也將 Agrawal 演算法以同樣系統實作。並以圖 6-3、圖 6-5、圖 6-7 三個實際流程，以模擬方式產生流程歷史資料，再分別以二個不同雛形系統實際測試。

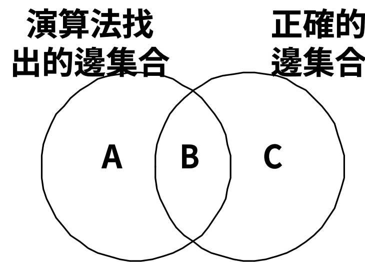
在演算法的績效(Performance)上，Agrawal 演算法的時間複雜度，如同第三章第二節的探討，是 $O(mn^3)$，而本研究所提出的演算法，時間複雜度如第四章第四節的探討，是 $O(mn^2)$，時間績效的優劣已很明顯。所以，在此效能評估實驗裡，不再針對時間績效進行比較，而著重於"質"的比較，也就是比較找出流程定義所需的流程例數目。另外，因為兩演算法對於錯誤資料的處理不同，因此本研究以正確的資料(positive example)進行評估，也就是說，利用模擬程式所產生的資料皆為正確資料，測試資料中並未包含錯誤資料(noise)。

因為流程中所有的活動，是一開始由使用者所定義的。演算法主要的目的是找出這些活動之間的執行順序。以有向圖來看，就是所有的點是已知的，目的在找出有向圖中的邊。因此，在實驗結果的比較上，我們定義了兩個效能指標來衡量實驗結果：

(1) precision = $\dfrac{|\text{該演算法找到而且正確的邊集合}|}{|\text{該演算法所有找到的邊集合}|}$

(2) recall = $\dfrac{|\text{該演算法找到而且正確的邊集合}|}{|\text{所有正確的邊集合}|}$

這兩個效能指標，可以用圖 6-2 來解釋。圖 6-2 中，集合 B 是演算法找出來的邊中正確的部份，而集合 A 是演算法所找出來的邊中錯誤的部份(當然，A 聯集 B 就是演算法所有找出來的邊集合)。集合 C 是演算法沒有找到但正確的邊集合(當然，B 聯集 C 就是所有正確的邊集合)。



圖<6-2>

因此，Precision = $\dfrac{|B|}{|A \cup B|}$

recall = $\dfrac{|B|}{|B \cup C|}$

我們分別利用這二個指標來衡量演算法的優劣。例如，有一流程，其中有活動 A、B、C，若流程中真正的邊是 A→B、B→C。當演算法找出流程定義，而認為有 A→B、B→C、A→C 邊時，recall 值因為找到的邊中，已找到所有正確的 A→B、B→C 邊，所以 recall 值為 1。而 precision 值因為所有找到的邊有三個，而其中正確的邊只有二個，所以 precision 值為 2/3。當然，如果有二個演算法 A、B，其中演算法 A 所得到的 recall 和 precision 值都比演算法 B 高，則我們認為演算法 A 優於演算法 B。

至於實驗資料，我們則利用模擬程式產生。在模擬程式中，對流程中的所有活動假設一個最長等待時間與最長執行時間。等待時間是可以執行某活動，到這個活動實際開始執行之間的時間間隔，在此時間間隔內隨機產生一時間值，作為這個活動的開始時間。相似地，最長執行時間是這個活動的最長執行時間，在 0 到最長執行時間內隨機任取一值，作為此活動實際執行時間長。當然，若加上這個活動的開始時間就是這個活動的結束時間了。

接下來，依各流程定義模擬活動執行先後，並記錄相關活動執行開始、結束事件及其時間點，作為雛形系統的輸入資料(因為 Agrawal 演算法假設活動執行為一瞬間，所以，以各活動的執行開始時間作為各活動的時間點，並以此產生 Agrawal 演算法所需的各流程例活動串列)，以利用雛形系統找出流程定義。因此，整理我們所考慮的模擬程式實驗參數，如表 6-1 所示：

參數符號	參數意義	參數設定
S	活動等待時間	∪(0, 10)
D	活動執行時間	∪(0, 20)
W	流程定義	順序、平行流程(圖 6-3) 簡單迴圈流程(圖 6-5) 複雜迴圈流程(圖 6-7)
N	流程例數目	介於 1 至 50 之間

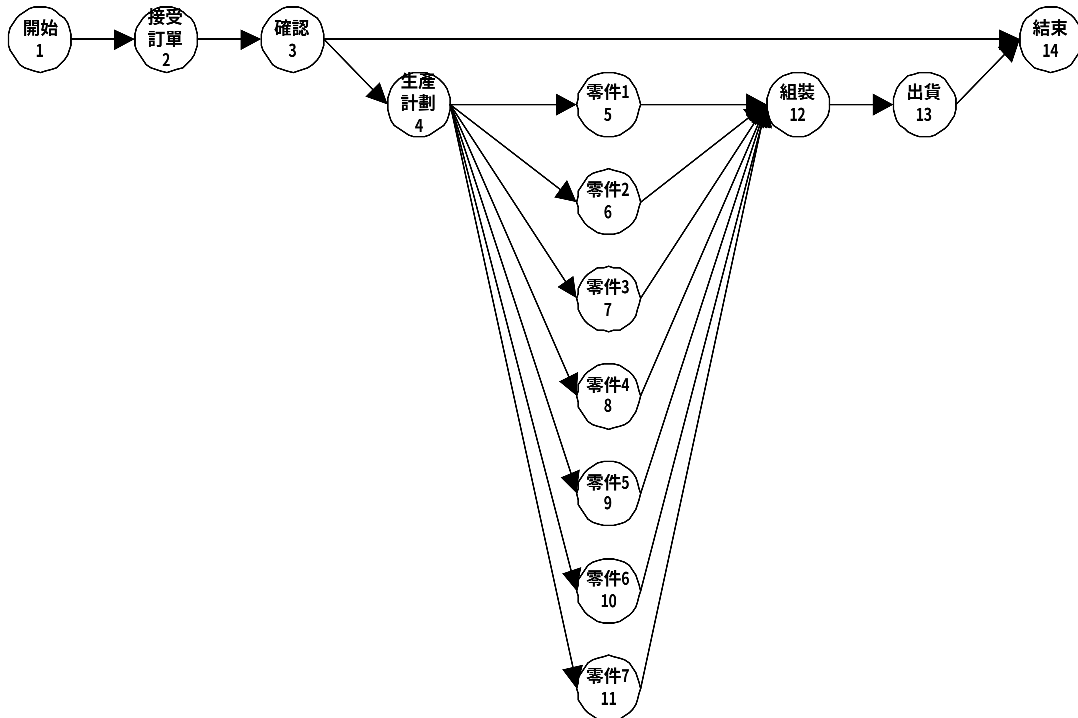
表<6-1>

表 6-1 中，參數 S 代表活動實際上開始的時間，當然，這個值會介於 0 和上述最長等待時間(設定為 10 個單位時間)之間，因此，變數值服從均等分配(∪(0, 10))。參數 D 代表活動實際上執行的時間，當然，這個值會介於 0 和上述最長執行時間(設定為 20 個單位時間)之間，因此，變數值服從均等分配(∪(0, 20))。

而參數 W 則是流程的定義，我們以圖 6-3、圖 6-5 和圖 6-7 三個實際流程，分別測試評估。圖 6-3 這個流程中，只有順序和平行的執行方式，沒有迴圈。而圖 6-5 的流程中，有一簡單迴圈(迴圈主體中所有活動在每一次迴圈執行時，皆會執行一次)。圖 6-7 則有複雜迴圈(也就是迴圈主體中所有活動，在每一次迴圈執行時，執行次數不定)。我們利用這三個性質不同、複雜程度不同的流程，來分別比較演算法在不同流程下，處理的優劣，這是參數 W 設定的意義，詳細流程定義如後各例所述。

最後一個參數 N，則代表流程例的數目，也就是模擬程式所需產生的流程例個數(雛型系統用以找出流程定義)。模擬程式會產生第一個流程例資料，再分別以二個雛型系統測試，若未找到正確流程，則以模擬程式產生第二個流程例資料，以二個流程例資料為輸入，再分別以二個雛型系統測試，重覆這個過程，直到找出正確、穩定的流程定義。由我們所使用的流程定義及以下的實驗結果，這個參數值約介於 1 到 50 之間。

我們以這樣的模擬程式產生流程歷史資料，以作為兩個雛型系統的輸入資料，進行實驗。並重覆實驗 30 次求平均值，以下為各實驗所使用之流程定義與實驗結果：



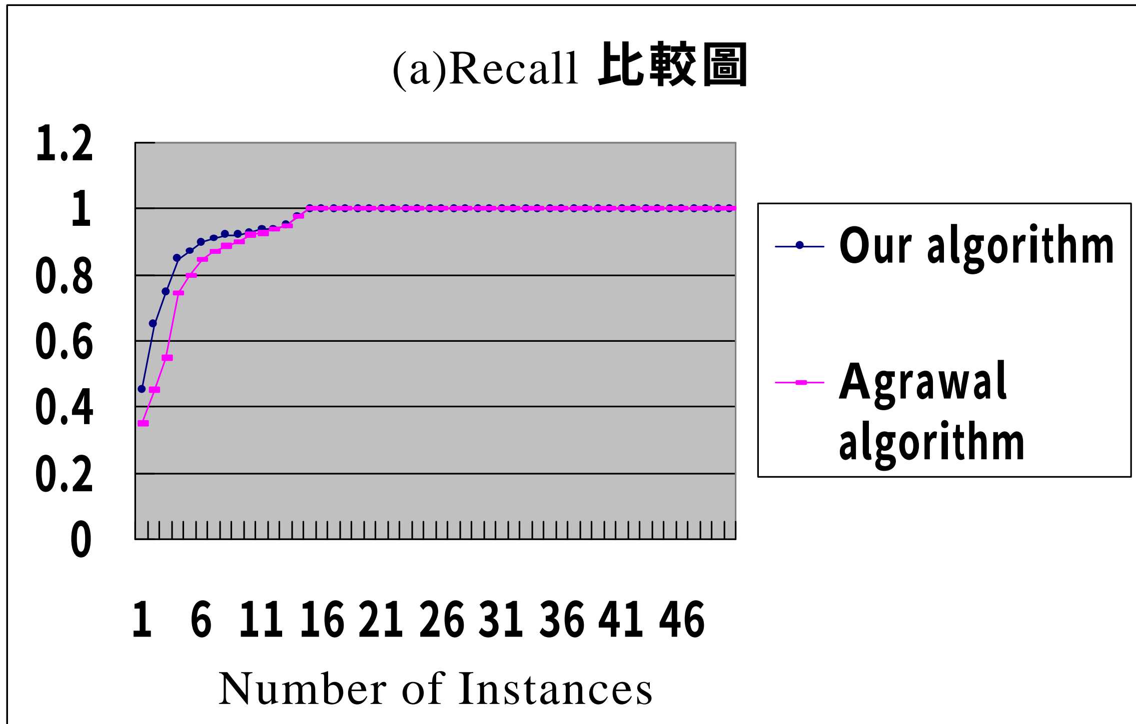
圖<6-3>

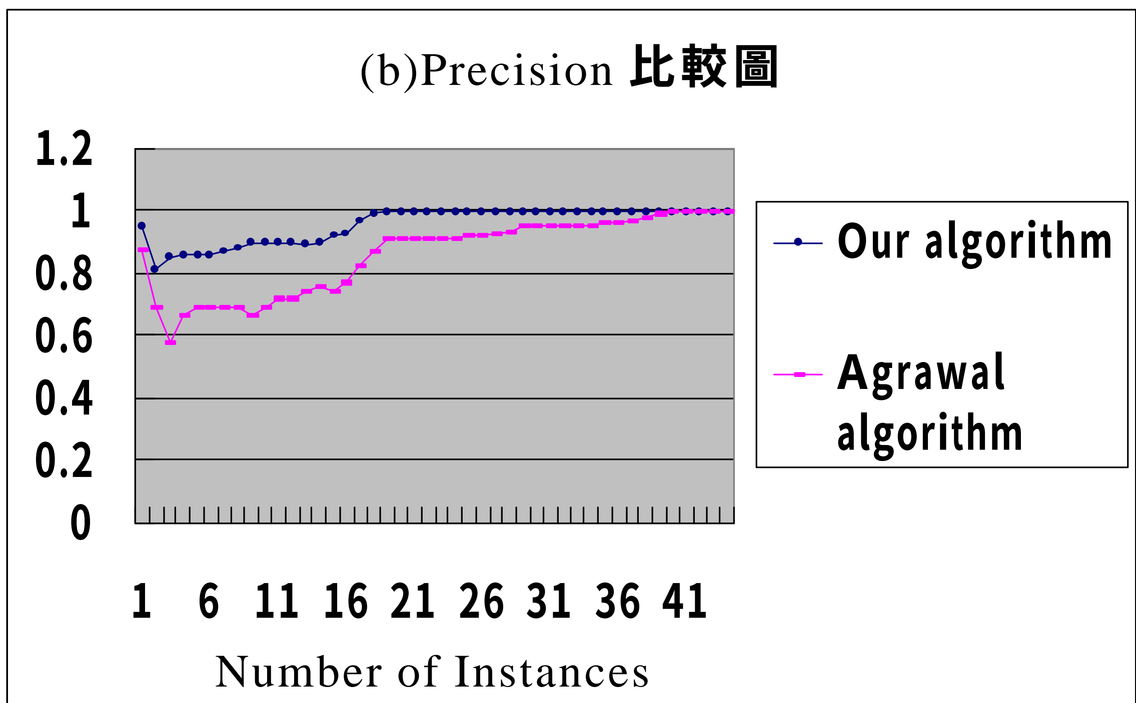
圖 6-3 是一零件製造公司的接單製造流程。該公司生產七種主要零件，當客戶訂貨時，先檢核客戶信用與公司產能，以決定是否接受該訂單。若接受該訂單則進入生產活動，生產該訂單所指定數種零件，生產完成後組裝、出貨，其流程以有向圖表示，如圖 6-3。

圖 6-4 為這些資料彙總整理結果。從圖 6-4(a)的 recall 圖可以發現，這兩個演算法在圖中 recall 值都逐漸上昇，而 recall 值的分母是固定的(所有正確的邊)，所以 recall 值逐漸上昇代表分子逐漸增加，也就是所找出正確的邊逐漸增加。兩個系統都大約在 15 個流程例時，可以把所有正確的邊找出來。

而在圖 6-4(b)的 precision 圖中，可以發現一開始的圖形上下起伏。這是因為圖 6-3 的流程中，訂單可能包含這家公司數種零件，所以在活動 4(生產計劃)之後是一個選擇(OR)的分歧，活動 5 至活動 11 並不一定在每一次的流程例中都會出現。而 precision 值的分子，是演算法所找到且正確的邊，會隨著流程例的增加而增加，因此分子值會逐漸增加直到所有正確的邊被找出(由 recall 圖，大約在 15 個流程例時)。但是，precision 值的分母，一方面隨著流程例的增加，而增加各活動之間可能的邊，但是一方面也會因為確認有些邊是不正確的，而在演算法中刪去，所以 precision 值的分母在一開始時並不穩定。直到流程例的增加使得所有的活動都出現後(即大約 15 個流程例後)，分母才會只因

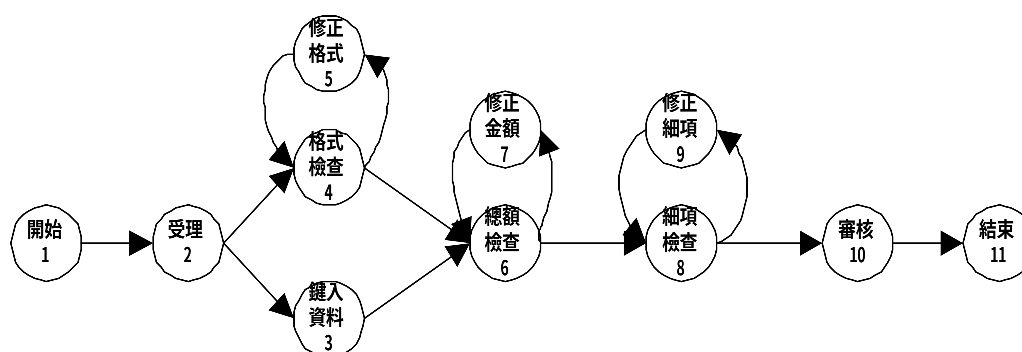
為確認有些邊是不正確而刪去的因素，逐漸縮小。所以，在 precision 圖中，在 15 個流程例後，precision 值因為分母的逐漸縮小(逐漸刪去不正確的邊)，而逐漸收斂於 1(只剩下正確的邊)。





圖<6-4>

圖 6-5 是一健康保險公司接受各醫院按月申報醫療費用的流程。當醫院申報醫療費用時，保險公司工作人員受理登記這些資料。若這些資料是以書面方式申報，則工作人員需將這些資料鍵入(key_in)資料庫，若這些申報資料已存成檔案，則需將這些資料讀入，轉成健康保險公司的資料格式。當然，若格式錯誤，則由工作人員詢問送件人員並修改直到格式正確。接下來，則由電腦系統檢查申報總金額是否與各細項金額加總相等，若不相等，同樣詢問修改直到資料一致。接下來更進一步，則針對各細項金額作檢查，若有錯誤，同樣檢查修改直到正確。當所有資料檢查完成後，就可以將所有相關資料送給費用審核人員，由審核人員進行審核了。其流程以有向圖表示，如圖 6-5 所示。

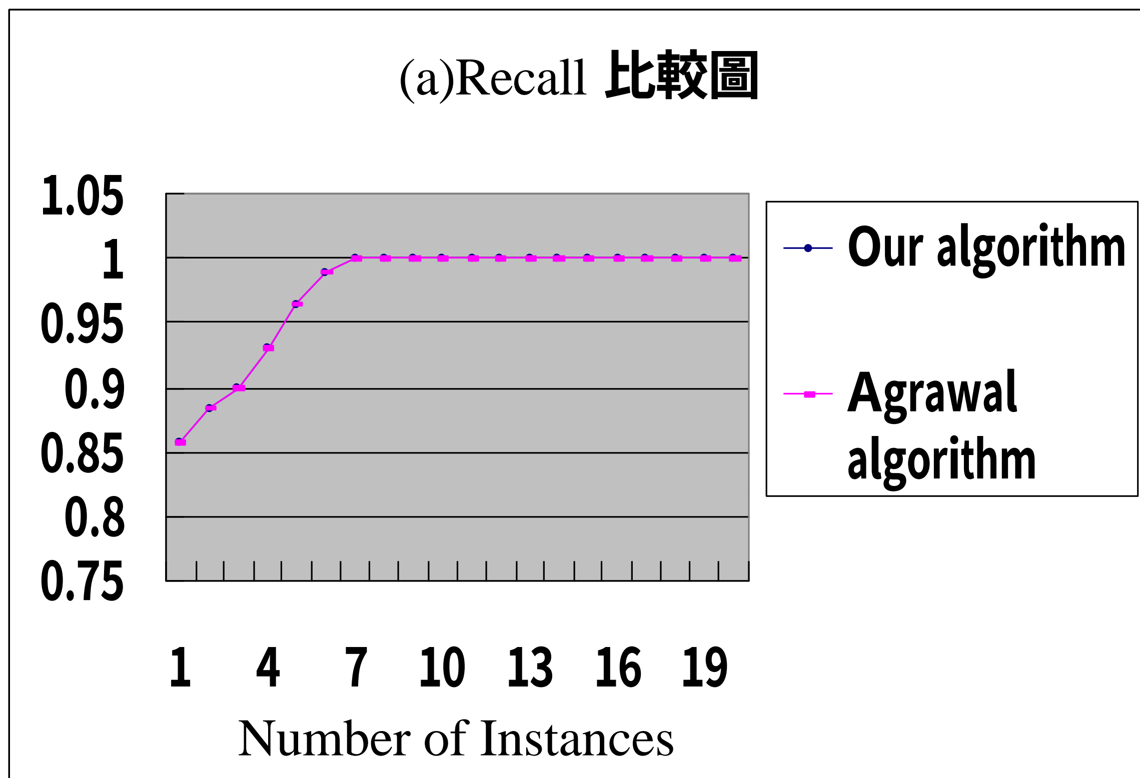


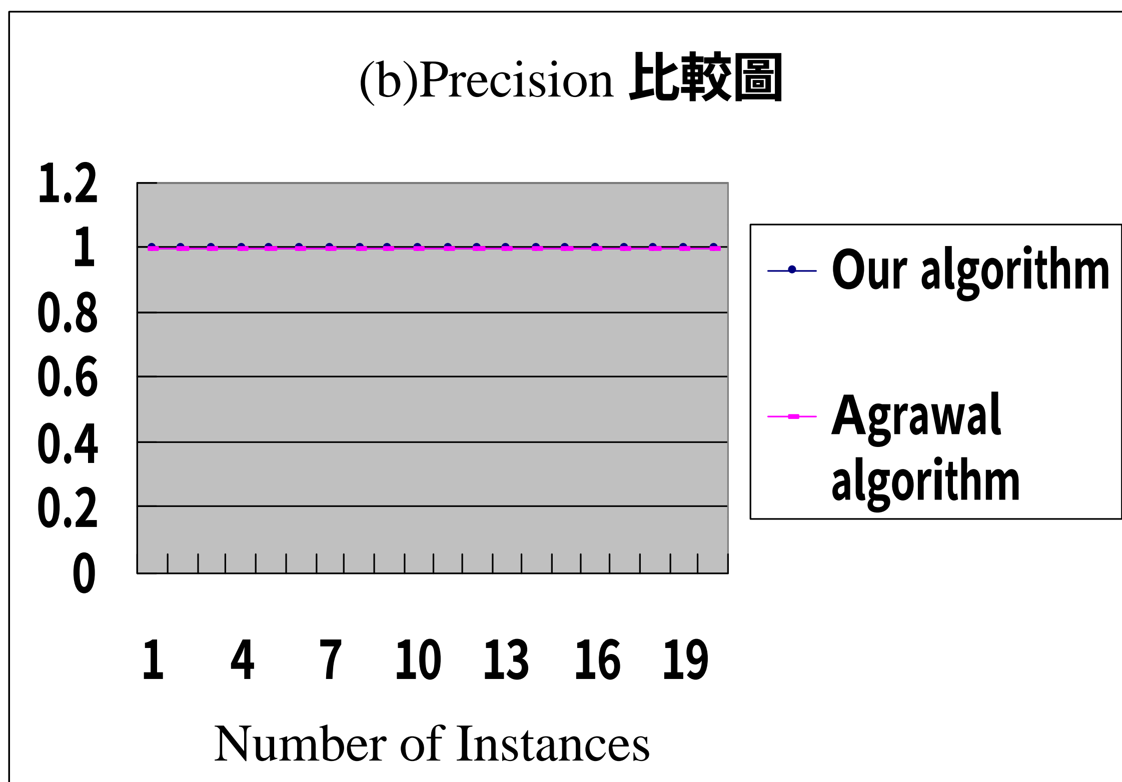
圖<6-5>

圖 6-6 為這些資料彙總整理結果。從圖 6-6 的 recall 圖，可以發現，圖中 recall 值逐漸上昇，而 recall 值的分母是固定的，所以 recall 值逐漸上昇代表分子逐漸增加，也就是所找出正確的邊逐漸增加。兩個系統都大約在 7 個流程例時，可以把所有正確的邊找出來。

而在 precision 圖，兩系統的 precision 值一直都是 1。Precision 值一直是 1，代表這兩個系統在這個流程都不會找出錯誤的邊。這是因為，在這個流程中，所有活動的執行都是順序的，沒有活動可能同時執行。活動 3(鍵入)與活動 4(格式檢查)雖然是平行的，但是這兩個活動不會在同一個流程例裡發生(因為醫院可能書面來申報或是磁片資料來申報，但不會同時有兩種資料，也就是說，活動 2(受理)之後為 XOR_Split)。而迴圈的部份，迴圈主體都是順序的簡單迴圈。因此，在這個流程裡，所有的活動將順序執行。在不會有活動同時執行，不會有多餘的執行順序的情形下，不會找出多餘的邊，因此，precision

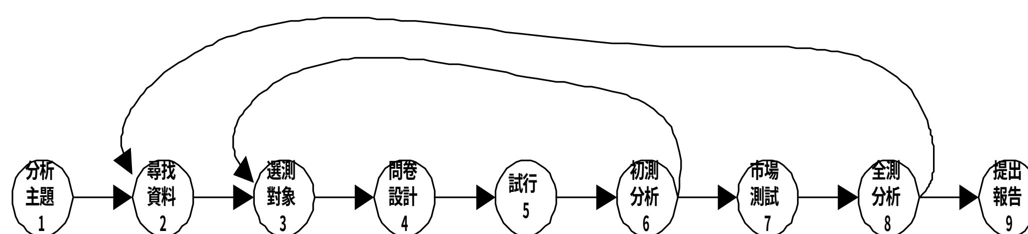
值一直為 1(分子與分母相同)。





圖<6-6>

圖 6-7 是一行銷公司進行市場調查的流程。當這個行銷公司接受委託進行市場調查時，首先開會分析調查主題，尋找相關資料，接下來則先進行小範圍的調查測試，包括選定小範圍測試對象、設計問卷、實際測試，以及初測結果分析。若初測結果分析不如預期，則重新開會修改問卷決定測試對象，再重新測試、分析，直到確定問卷內容、目標。接下來，則以此問卷進行大範圍測試，分析所得到結果，若大範圍測試結果不滿意，則回到起始的資料搜尋，重新進行測試、分析，直到得到正確市調資料。其流程以有向圖表示，如圖 6-7 所示。



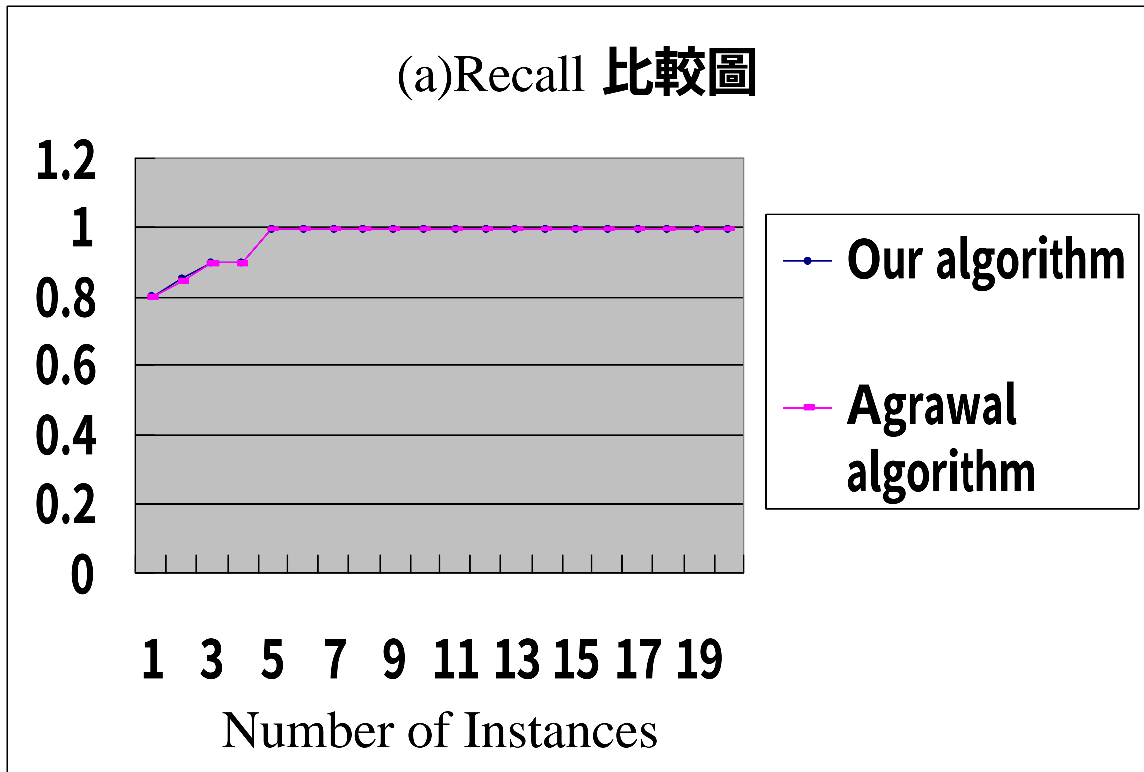
圖<6-7>

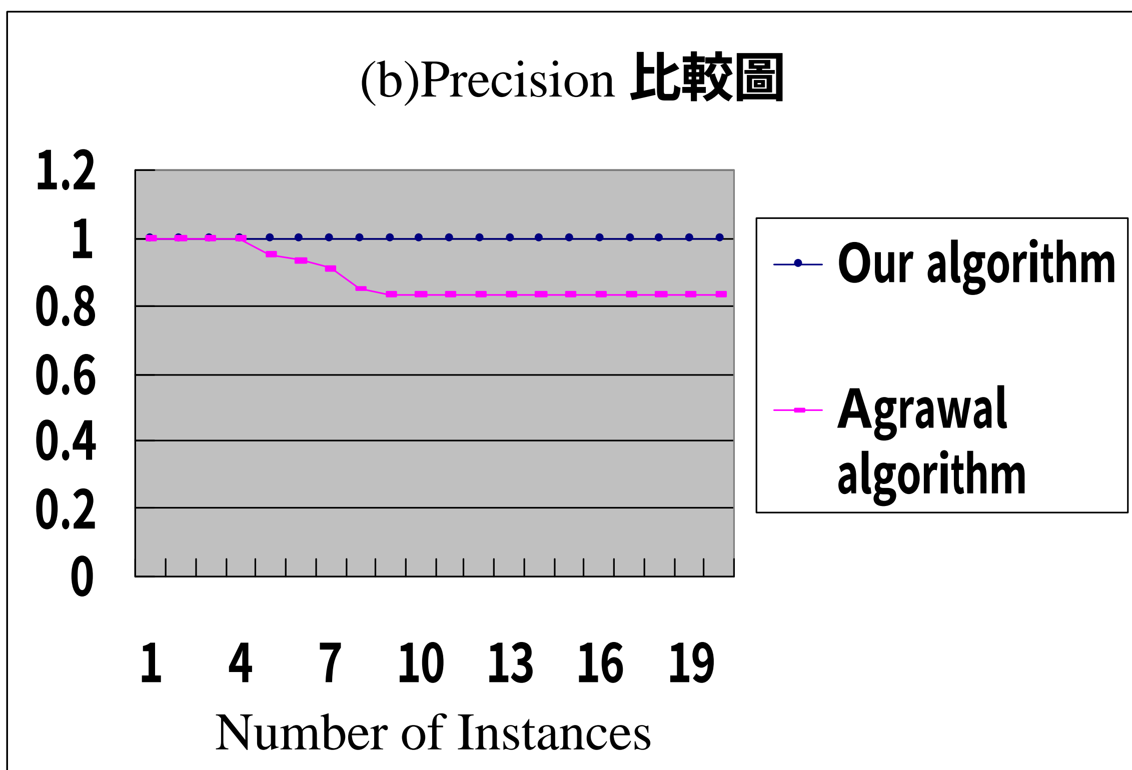
圖 6-8 為這些資料彙總整理結果。由圖 6-8(b)的 precision 圖中可以發現，Agrawal 演算法的 precision 值將收斂在 0.83。比較 recall 圖與 precision 圖，圖中大約在 5 個流程例時，recall 值為 1(代表已找出所有正確的邊，且之後 recall 值不再改變)，但 Agrawal 演算法的 precision 值卻開始下降，代表 precision 值的分母開始增加，當然此時增加的會是錯誤的邊。換句話說，Agrawal 演算法在此點之後找出錯誤的邊，而且即使流程例再增加，也不會去除這些錯誤的邊。

這是因為如同前述文獻探討中所討論的，迴圈主體中有部份活動執行次數並不固定時，會使得這些活動與迴圈中執行次數固定的活動產生雙向的關係，而使得正確的順序關係被刪去，在執行演算法遞移刪去時，就會產生多餘、錯誤的邊。

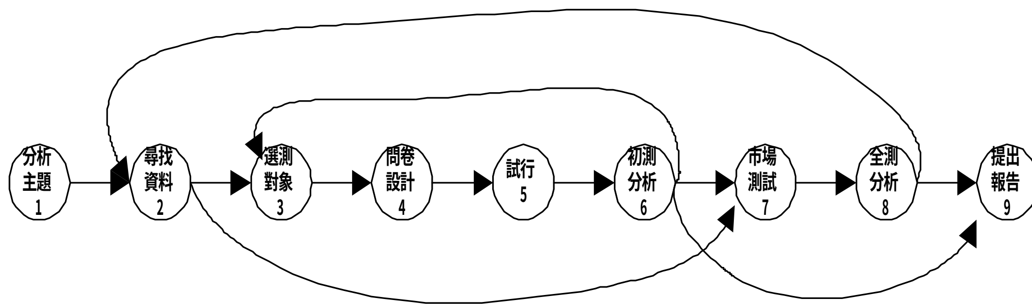
圖 6-9 是這個流程以 Agrawal 演算法所找出，即使流程例增加，也不再改變的流程定義。從圖 6-9 中可以明顯發現，活動 2(尋找資料)有二個連出去的邊至活動 3(選測對象)與活動 7(市場測試)，這是因為活動 3 與活動 7 出現了雙向的執行順序關係而被刪去，所以在執行 Agrawal 演算法刪去遞移邊的時候，活動 2 就會有連出去的邊分別連至活動 3 與活動 7 了。而即使流程例數目再增

加，這個圖形依然不會改變。





圖<6-8>



圖<6-9>

綜合上述三個實驗，將圖 6-3、圖 6-5 與圖 6-7 的實作結果整理，可以發現下列三個特點：

(1)在流程有部分活動可以同時執行時，我們的演算法比 Agrawal 演算法所需的流程例數目少：從圖 6-4(a)的 recall 圖可以發現，這兩個系統在 recall 圖上的差異並不明顯。但從圖 6-4(b)的 precision 圖，可以發現兩系統明顯不同。Precision 值的分子與 recall 值的分子相同，所以當 recall 值不再變動，約 15 個流程例時，precision 值的分子便不再變動。此時，因為 precision 值中分母代表演算法所有找出的邊，其中不正確的邊會逐漸被刪去，所以分母逐漸縮小而收斂於 1。因此，比較 15 個流程例以上的 precision 圖，我們的演算法大約 19-20 個流程例時收斂於 1，而 Agrawal 演算法的 precision 值在此區間一直比較低，且直到大約 39-40 個流程例時才收斂於 1。

因為 precision 值代表，所找出、正確的邊集合與所有找出的邊集合的比例，所以 precision 值一直比較低，在所找出、正確的邊集合相同的情形下(由 recall 圖所表示)，代表了多找出許多錯誤的邊。

而且，合併 recall 和 precision 圖來看，我們的演算法在有 19-20 個流程例時，找出所有正確的邊(recall=1)，而且沒有多餘的邊(precision=1)。因此，在這個流程中，平均只要 19-20 個流程例，我們的演算法就可以找出正確的流程。反觀 Agrawal 演算法，要直到 39-40 個流程例，才能找出所有正確的邊(recall=1)，而且沒有多餘的邊(precision=1)。因此，在這個流程中，Agrawal 演算法平均需要 39-40 個流程例，才能找出正確的流程。

這個差別的原因是，因為流程中活動 5-活動 11(各種零件製造活動)可

以同時執行，我們的演算法認為一個活動會執行一段時間，所以當活動執行時間有所交集時，就能夠確認這兩個活動是平行的，不是順序的關係。而在 Agrawal 演算法中，則假設每一個活動執行都是一瞬間，因此即使有兩個活動實際上是平行的，在記錄下來的資料裡還是先後的執行關係，所以有許多多出來的順序關係，必須有較多的流程例資料而使得這兩個活動出現雙向的關係時，才能刪去。所以，如果流程中有部份活動是可以同時執行的，我們的演算法所需的流程例數目將比 Agrawal 演算法少。

(2)若流程中所有活動不會同時執行、也沒有複雜迴圈時，兩個方法約需同樣的流程例數目：從圖 6-6 的 recall 圖，可以發現，兩個系統沒有明顯差別。而在 precision 圖，兩系統的 precision 值，因為不會找出錯誤的邊，所以一直都是 1。

這二個演算法在這個流程裡並沒有明顯分別，這是因為，在所有的活動皆順序執行且沒有複雜迴圈時，假設活動執行是一段時間或一瞬間，其實是相同的，沒有分顯分別。

(3)若流程中有複雜的迴圈時，Agrawal 演算法所找出的流程定義並非全部正確：由圖 6-8(b)的 precision 圖中可以發現，Agrawal 演算法的 precision 值將收斂在 0.83。這是因為如同前述文獻探討中所討論的，迴圈主體中有部份活動執行次數並不固定時，會使得這些活動與迴圈中執行次數固定的活動產生雙向的關係，而使得正確的順序關係被刪去，在執行演算法遞移刪去時，就會產生多餘、錯誤的邊。因此，在流程中有複雜迴圈時，我們的演算法可以找出正確的流程，而 Agrawal 演算法並不適合這種流程。

綜合以上三個流程實驗結果，本研究所提出的演算法，在時間上較有效率而且在流程中有部份活動可以同時執行的情形下，可以用比較少的流程例找出正確的流程。在複雜迴圈的處理上，也較 Agrawal 演算法適合。

另外，本研究對於控制條件的萃取(演算法 3)，因為演算法執行後是產生組織過的檔案，來叫用某種分類方法(如 cn2)分類。因而找出分類結果的正確性依所使用的分類方法而定，所以不再於此作演算法的效率、正確性評估。

第四章第三節控制條件的萃取，我們以 cn2 分類方法為例來找出控制條件。

但這種分類方法，只能找出類似”V1>30 and 20<V2<60”的變數範圍條件，不能找出如”V1+V2>100”之類函數型的條件。如果使用者可以確定變數的型態、函數的型態…等等更多的資訊，使用者可以利用更適合的分類方法來分類。例如，如果使用者可以確定所有的變數都是數值型態、產生的函數是線性函數，則可以利用統計學中區別分析(discriminatory analysis)的分類方法來分類，而可以直接得到控制條件的線性函數。類似的概念，使用者可以利用演算法 3 將所有資料整理後，利用適合的分類方法，以適合的型式表達控制條件。

第七章. 結論

找出企業既有流程，是企業進行流程分析、再造時，首先而必要的工作。然而，過去找出企業既有流程的工作，通常有賴於專家的幫助，而且必須經過許多個別訪談、團體會議來進行討論，而耗費眾多的時間、人力成本。

因此，本研究主要的貢獻在於，以一個系統化的方法，自動地萃取、找出企業中的既有流程，以利流程分析、改造的進行。在本研究中，我們提出了二個萃取演算法。一是控制流萃取演算法，透過這個演算法，可以由過去所記錄的活動執行時間，找出流程中活動的執行順序。另一個演算法是控制條件萃取演算法，透過這個演算法，可以由過去所記錄的相關寫入資料，找出流程中活動的控制條件。

我們也將演算法實作雛型系統，並與其它研究所提出的演算法作比較。本研究所提出的演算法在時間上有較佳的績效，而且在流程中有活動可以同時執行時，只須較少的流程例就可以找出正確的流程。當流程中有複雜迴圈時，找出流程的正確性亦較佳。

利用這個系統化方法的進行，可以由過去所記錄下來的歷史資料中，快速而正確的找出企業既有流程。不但節省了大量的時間、人力成本，也可以讓企業考量目前企業內部的狀況以及外在環境的變化，在不同的時期，進行流程的萃取、比較，以隨時檢視企業流程的品質及企業中資源的配置情形，掌握更精確的資訊。

另外，對於系統的評估，我們的研究以三種不同的流程、正確的模擬資料進行，在下一階段，我們將朝向二個方向作進一步的評估。首先是以模擬程式產生包含錯誤的資料進行評估。另一個方向則是以更具彈性、更一般化的模擬程式，作為流程定義的產生器(process generator)，來產生更多樣的流程定義及模擬資料，以驗証演算法在各種不同流程的處理優劣。

當然，我們也考慮以個案研究的方式，由實際個案所收集的資料進行萃取、評估系統，以進一步改善我們的系統。再者，我們也考慮與目前既有流程分析工具整合，使流程的萃取、分析能一貫而更有效率，以助於企業流程的改善。

第八章．參考文獻

[Adam98] N. R. Adam, V. Atluri and W. Huang, "Modeling and Analysis of Workflows Using Petri Nets," *Journal of Intelligent Information Systems*, Vol. 10, 1998

[Agra98] R. Agrawal, D. Gunopulos and F. Leymann, "Mining Process Models from Workflow Logs," *Research Report RJ 10100*, IBM Almaden Research Center, 1998 http://www.almaden.ibm.com/cs/quest

[Atti93] P. C. Attie, M. P. Singh, A. Sheth and M. Rusinkiewicz, "Specifying and Enforcing Intertask Dependencies," *VLDB*, 1993

[Bier72] A.W. Biermann and J.A. Feldman, "On the Synthesis of Finite State Machines from Samples of Their Behavior," *IEEE Transactions on Computers,* Vol. 21, No.6 , Jun., 1972

[Brad94] M. G. Bradac, D. E. Perry and L. G. Votta, "Prototyping a Process Monitoring Experiment," *IEEE Transactions on Software Engineering*, Vol. 20, No. 10, Oct., 1994

[Clar89] P. Clark and T. Niblett, "The CN2 Induction Algorithm," *Machine Learning Journal*, Vol. 3, No 4, 1989

[Cook95] J. Cook and A. Wolf, "Automating Process Discovery Through Event-Data Analysis," *Proc. 17th Intl. Conf. On software Engineering (ICSE17)*, Apr., 1995

[Corm90] T. H. Cormen, C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms," MIT Press, 1990

[Datt98] A. Datta, "Automating the Discovery of AS-IS Business Process Models: Probabilistic and Algorithmic Approaches," *Information Systems Research*, Vol. 9, No. 3, Sep., 1998.

[Dave93] T.H. Davenport and J.E. Short, "Process Innovation –Reengineering Work Through Information Technology," Boston: Harvard Business School Press, 1993

[Geor95] D. Georgakopoulos, M. Hornick and A. Sheth, "An Overview of Workflow Management: from Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases*, Vol.3, No.3, 1995

[Hamm93] M. Hammer and J. Champy, "Reengineering the Cooperation-A Manifesto for Business Revolution," A Division of Harper Colins Publishers, 1993

[Mana94] R. L. Managanelli and M. M. Klein, "The Reengineering Handbook," *American Management Association*, 1994

[Memo93] R. MeMo, K. T. Wong and P. Flores, "Action Workflow as The Exterprise Integration Technology," *IEEE Computer Society*, 1993

[WfMC94] *The Workflow Reference Model*, Workflow Management Coalition, No. TC-00-1003, Nov., 1994

[WfMC96_1] *Workflow Terminology & Glossary*, Workflow Management Coalition, No. WfMC-TC-1011, Jun., 1996

[WfMC96_2] *Audit Data Specification*, Workflow Management Coalition, No. WfMC-TC-1015, Nov., 1996

[WfMC98] *Interface 1: Process Definition Interchange*, Workflow Management Coalition, No. WfMC-TC-1016-P, Aug., 1998

附<1> Agrawal 演算法

1. Start with the graph G = (V,E), with V being the set of activities of the process and E =ψ.
2. Go through each execution in the log and uniquely identify each activity recorded in the log, thus create a new set of vertices V’ and graph G = (V’,E’).
3. For each process execution in L, and for each pair of activities u,v such that u terminates before v starts, add the edge (u,v) to E’. (In practice, steps 1-3 are executed together in one pass over the log.)
4. Remove from E’ the edges that appear in both directions.
5. For each strongly connected component of G’, remove from E’ all edges between vertices in the same strongly connected component.
6. For each process execution present in the log:
   (a) Find the induced subgraph of G’.
   (b) Compute the transitive reduction of the subgraph.
   (c) Mark those edges in E’ that are present in the transitive reduction.
7. Remove the unmarked edges in E’.
8. In the graph so obtained, merge the vertices that correspond to different instances of the same activity in the graph, thus reverting to the original set of vertices V.
9. Return the resulting graph.

附<2> [Datt98]修改後 Markov 演算法

1. Construct the nth order activity-sequence probability matrix by the given sample activity stream.
2. A threshold probability is fixed, as a parameter. A directed graph called the activity graph is constructed from the probability matrix
3. The activity graph AG created in the previous step includes repetitive (duplicate) edges between the same pairs of nodes. These are eliminated, all the remaining edges are uniquely labeled.
4. The activity graph AG is converted into its dual AG'.
5. Lastly, AG' from the above step is converted into the final Process Activity Graph for the BP in consideration, in the following manner: all the illegal activity sequences are removed from AG', by removing those edges in AG' that cause illegal sequences, and at the same time, taking care that , in doing so, none of the legal sequences are removed.

附<3> 演算法2 加入門檻值設定

Step0. Let G = (V,E), with V is the set of activities of a process and E =ψ.

Step1. For each instance

(a)call algorithm 1, let none_pair set = return value of algorithm1

(b)if any two activities(u,v), activity u starts before activity v and their temporal duration have partial overlap, add u→v to overlap_pair set.

(c)if any two activities(u,v), activity u starts before activity v and activity u terminates after activity v, add u→v to contain_pair set.

Step2. Union none_pair set, overlap_pair set and contain_pair set of all instances.

Then, for any two same node (A,B), summarlize their counter as following:

(a)Summarlize all counter of A→B none_pair as C1.

(b)Summarlize all counter of B→A none_pair as C2.

(c)Summarlize all counter of A→B、B→A overlap_pair and contain_pair as C3.

Step3. Caculate threshold T, and

(a)if C1+C3<T, then delete all C1 and C3 pair set.

(b)if C2+C3<T, then delete all C2 and C3 pair set.

(c)if C3<T, then delete all C3 pair set.

Step4. If a none_pair or its reverse exist in overlap_pair set or contain_pair set, then delete this none_pair from none_pair set.

Step5. Add each none_pair to E.

Step6. Return (V,E).