師大學報：數理與科技類
民國 89 年，45，21-36

完全圖上最大權重配對問題之自我穩定演算法的設計及分析

陳瑞宜　林順喜

國立臺灣師範大學資訊教育研究所

在 1974 年，Dijkstra 提出了自我穩定的概念。一個分散式系統不論其初始狀態爲何，最後都會收斂至正確的系統狀態稱之爲自我穩定系統。近年來，自我穩定演算法不用初始化的特性受到許多研究者的重視。Hsu 和 Huang 針對分散式網路中「最大配對」問題提出了自我穩定演算法，並利用變數函數分析法，證明了此演算法需耗用的時間複雜度爲 $O(n^3)$，然而 Tel 針對此一演算法提出不同的變數函數，證明最多需要 $O(n^3)$ 的時間複雜度。在本論文中，我們將自我穩定系統的理論應用在完全圖上的「最大權重配對」問題，設計出包含五個規則的自我穩定演算法，並針對此自我穩定演算法的正確性進行證明分析。最大權重問題是指當節點兩兩配對之後，其線段權重兩兩交換並不會找到更大的值，也就是除了希望在圖中找到最大配對之外，更進一步能夠使配對的權重達到最大。因此我們結合了 Hsu-Huang 最大配對自我穩定演算法，以及嶄新的交換配對規則，保留自我穩定系統容錯及自我穩定的特性，設計了時間複雜度爲 $O(n^2 + nk)$ 的一個最大權重配對問題之自我穩定演算法。

關鍵字：自我穩定演算法、系統容錯、最大配對問題、最大權重配對問題

緒論

一、研究背景

在 1974 年，Dijkstra【8】發表了自我穩定（self-stabilizing）的概念，以自我穩定環狀系統提出解決互斥問題（mutual exclusion）的演算法。一個分散式系統不論其初始狀態爲何，最後都會收斂至正確的系統狀態稱之爲自我穩定系統。因爲這種自我穩定的特性，對於不需初始化的系統有極大的用處。除此之外，分散式自我穩定演算法對於系統容錯（fault tolerance）方面也有相當大的貢獻。自我穩定演算法允許系統由任意可能的狀態開始，並保證在有限時間內會達到合法系統狀態，所以對於短暫及易變的錯誤具有容錯的能力【10】【14】【22】。

近年來由於分散式系統快速發展，自我穩定演算法不用初始化的特性受到許多研究者的重視，並將其應用在互斥問題（mutual exclusion）【2】、圖論（graph theory）【3】【5】【7】、時間同步問題（clock synchronization）【1】【21】、哲學家吃飯問題（dining philosophers problem）【15】、分散式資源分享與分配問題（distributed resource sharing and allocation）【24】、系統容錯（fault-tolerance）【13】以及通訊協定（communication protocols）【6】【9】等方面。其中圖論方面的研究更是不計其數，例如塗色問題（graph coloring）【11】、最大流量問題（maximum flow）【12】、深度優先搜尋（depth-first search）【4】、廣度優先搜尋（breadth-first search）【17】等問題皆有卓越的研究成果。

配對問題在圖論中同樣屬於基礎且重要的研究課題。目前為止，自我穩定演算法在最大配對問題（maximal matching problem）方面的研究已有相當成果【16】【23】。Hsu 和 Huang 在 1992 年針對分散式系統中最大配對問題提出了自我穩定演算法【16】。在這篇論文當中，Hsu 和 Huang 利用變數函數分析法，證明了此演算法需耗用的時間複雜度為 $O(n^3)$，此處 n 為無向圖 G 中節點的總數。1994 年 Tel 針對同一演算法提出不同的變數函數分析法證明最多只需要 $O(n^2)$ 的時間複雜度【23】。本論文將自我穩定演算法推廣至「最大權重配對」問題（maximal weighted matching problem）上。然而最大權重配對問題相較於最大配對問題有更多嚴苛的條件限制，在這裡我們已成功地克服這些困難。

二、研究目的

本研究之目的除了設計完全圖（complete graph）上最大權重配對問題的自我穩定演算法之外，並針對此自我穩定演算法的正確性進行驗證分析。在我們所設計的自我穩定演算法中，不論初始狀態為何，以及不論中央控制系統挑選了那個節點執行規則，系統保證能在 $O(n^2 + nk)$ 步驟內達到合法狀態。

三、研究之重要性

在分散式系統蓬勃發展的同時，如果系統能夠允許短暫性的錯誤，並花費相當少的資源將系統的整體行為回復到穩定的狀態，則是分散式系統領域中一個相當重要的研究重點。自我穩定演算法不但擁有系統容錯的特性，更能保證在有限的時間內將系統回復到穩定合法的狀態。

配對問題一直是圖論研究者相當關注的問題所在，生活上亦有許多重要的應用，例如：工作分配（task allocation）、工作排程(job scheduling)等方面。現今分散式系統發展迅速，許多研究者亦針對圖論上的各種問題研究出分散式系統上卓越的演算法。然而如果能夠允許突發性的短暫錯誤，並花費相當少的資源將系統的整體行為回復到穩定的狀態，則是分散式系統領域中一個相當重要的研究重點。因此如何能夠在解決這些配對問題的同時，融入系統容錯的特性以增進系統的可靠性，這就是本論文的重要成果所在。

四、自我穩定系統相關研究探討

為達成以上之研究目的，本研究先進行自我穩定演算法的相關文獻探討。在 1974 年由 Dijkstra 所提出的自我穩定演算法【8】在分散式系統中受到許多學者的注意，近年來相關研究也相當的多。在 Dijkstra 的定義裡，一個自我穩定系統不論在任何初始狀態下，皆能保證在有限步驟內達到特定的合法狀態。相反地，如果此系統可能永久停留在某不合法狀態，則不是個自我穩定系統。Dijkstra 觀察到分散式系統中節點的行為只能被與其相關的部分系統狀態所影響，因此自我穩定系統的困難性在於區域性的行為如何達到整體性的目的。也就是說，當節點只知道本身以及鄰居節點的資訊時，整體系統如何達到所需的合法穩定狀態是自我穩定系統中的研究重點。

自我穩定演算法通常由許多規則（rule）所組合而成，系統中每個處理器皆執行這些規則。每條規則包含了兩個部分，分別為特權部分（privilege part）和移動部分（move part），如圖 1-1。特權部分由一個布林函數所定義，如果某個處理器符合了其中一條規則中特權部分所定義的先決條件，則此處理器擁有特權執行此條規則的移動部分，而這個移動部分則使處理器的狀態做相關的改變。

規則：特權部分（先決條件）⇒移動部分（相關的移動）

圖 1-1 自我穩定演算法規則

在 Dijkstra 原始模型的定義中假設系統中存在一個中央控制系統（central demon），在每一個步驟（computation step）裡中央控制系統任意地挑選其

中一個符合特權部分的處理器執行所符合的規則，因此單位時間內只有一個處理器能夠改變狀態，也就是說系統中的處理器將循序地改變狀態。然而眞正運用在分散式系統上的模型通常爲分散式控制系統（distributed demon），也就是並沒有一個特定的中央控制單元控制各個處理器的行爲。在分散式控制系統模型中只要處理器各自符合自我穩定演算法中的規則且互不干擾狀態的情況下，系統中的處理器將各自獨立地改變狀態。雖然中央控制系統與分散式控制系統相比較，顯得功能效率上都比較薄弱，但是爲了單純上的考量，我們亦假設中央控制系統的存在。

既然自我穩定系統能夠在有限步驟內達到合法穩定狀態，因此在驗證系統穩定之正確性方面亦有一些必要需求條件如下：

系統穩定需求條件（i）：如果系統達到合法穩定狀態，則沒有任何節點能執行任何規則。

系統穩定需求條件（ii）：如果系統仍處於不合法狀態，則存在至少一個節點可以執行規則。

系統穩定需求條件（iii）：不論初始狀態爲何，以及不論中央控制系統挑選了那個節點執行規則，系統保證能在有限步驟內達到合法狀態。

除了利用上述三個系統穩定條件證明自我穩定系統的正確性之外，亦可利用變數函數分析法（variant function method）證明系統達到穩定合法狀態所需的時間複雜度。變數函數分析法最先由 Kessels 所提出【19】，在變數函數分析法中，首先定義一個有限函數（bounded function）F，藉由證明當演算法中的規則被執行時，此函數 F 會漸漸遞增或遞減的方式來證明自我穩定系統會在一定步驟內達到合法穩定狀態。這個傳統的方法在自我穩定系統的驗證上常常被研究者拿來使用。一般而言，在自我穩定演算法中的時間複雜度指的是需花多少的步驟（computation steps）達到合法穩定狀態。

在自我穩定系統中所允許的系統錯誤爲暫時性錯誤（transient faults）【18】。所謂的暫時性錯誤是由系統中處理器或是其他元件暫時性的失調所造成的有限時間錯誤。暫時性的錯誤發生時可能造成系統失誤（error）或故障（failure）。由於這些暫時性錯誤所造成的系統失誤只發生在瞬間，使得偵測錯誤方面相當的困難。若暫時性錯誤發生的情形相當的少見，所造成的損害可以人爲修正；然而如果這些暫時性錯誤是間歇性的，偵測錯誤則是極爲重要，即使發生錯誤的時間相當短暫。通常來說，偵測這種時斷時續的暫時性錯誤不但非常困難，甚至耗費昂貴。

因此，如果系統能夠允許這種短暫性的錯誤，並花費相當少的資源將系統的整體行爲回復到穩定的狀態，則是分散式系統領域中一個相當重要的研究重點。系統容錯的目的就是希望即使系統中部分錯誤已經發生，卻能避免影響整個系統而造成系統失敗。一般而言，備份（redundancy）是支援系統容錯的關鍵，備份可以是系統中的硬體、軟體、甚至於時間上的備份。當系統正常運作時，並不需這些系統備份執行運作；若是系統發生錯誤時，系統中備份的部分則維持了系統整體行爲的正確性。

1984 年 Lamport【20】認爲 Dijkstra 所提出的自我穩定系統雖然沒有提及系統容錯和可靠性等字眼，但自我穩定系統的確是容錯領域中重要的里程碑。當短暫性錯誤發生時，造成不一致的系統狀態，使得自我穩定系統摒除了錯誤的起源，不藉由任何形式外界的干預，仍能維持系統的正確性。自我穩定系統能在任何初始狀態下，保證在有限步驟內達到合法狀態。由系統不需初始化所延伸的特性就是系統容錯。當系統發生暫時性錯誤時，系統將整體行爲視爲初始狀態，並依照自我穩定的規則達到合法穩定狀態。這樣的容錯方式並不需要多餘的系統備份，所花費的只有系統重新達到合法狀態的時間。

我們所提出的最大權重配對演算法

一、最大權重配對問題

一個無向圖 $G(V,E,W)$ 中，V 代表節點（node）集合，E 代表線段（edge）集合，W 則代表各線段的權重（weight）集合。任一線段上皆有權重 w，w 爲介於 $1\sim k$ 之間的正整數，以 $w(i,j)$ 代表節點 i、j 之間的線段權重。最大權重配對問題（maximal weight matching）是要找出一種配對組合，使得兩兩交換配對之後並不會找到更大的總權重值。由於自我穩定系統模型中各節點只知道節點本身與鄰居節點資訊的限制，因此在本研究中我們針對完全圖（complete graph）設計第一個解決最大權重配對問題的自我穩定演算法。

根據最大權重配對問題的定義，首先我們構思出演算法的初步概念。如果目前有兩組配對分別爲 $[x,y]$ 與 $[i,j]$，其中四個節點之間線段的權重分別爲 $w(x,y)=a$、$w(x,i)=b$、$w(y,j)=c$、$w(i,j)=d$，如圖 2-1 所示，我們在線段旁用方框內的數字表示其權重。就節點 x 而言，若存在節點 i，使得 $b+c>a+d$，則將配對改爲 $[x,i]$ 與 $[y,j]$ 就會比原來的配對 $[x,y]$ 與 $[i,j]$ 的總權重值來得大。

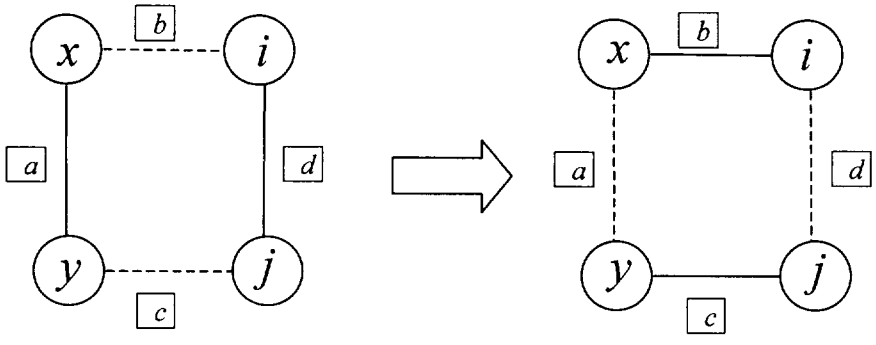


圖 2-1　最大權重配對演算法初步概念

以下範例說明了根據我們的初步構想，系統如何找出最大權重配對的自我穩定過程：

範例：

表 2-1　最大權重配對問題範例

	1	2	3	4	5	6	7	8
1		3	5	1	2	4	1	3
2			2	1	4	2	5	3
3				5	1	4	3	2
4					3	2	1	4
5						4	1	3
6							3	5
7								2
8								

表 2-1 爲一個含有 8 個節點的完全圖上線段權重表，其中橫座標、縱座標皆爲節點編號，而中間上三角形的內容即爲節點與節點之間線段的權重。舉例而言：$w(1,2)=3$ 代表了線段（1,2）上的權重爲 3、$w(4,7)=1$ 表示線段（4,7）的權重爲 1。最大權重配對問題則希望根據像這樣所給定的完全圖權重表找出最大權重配對。

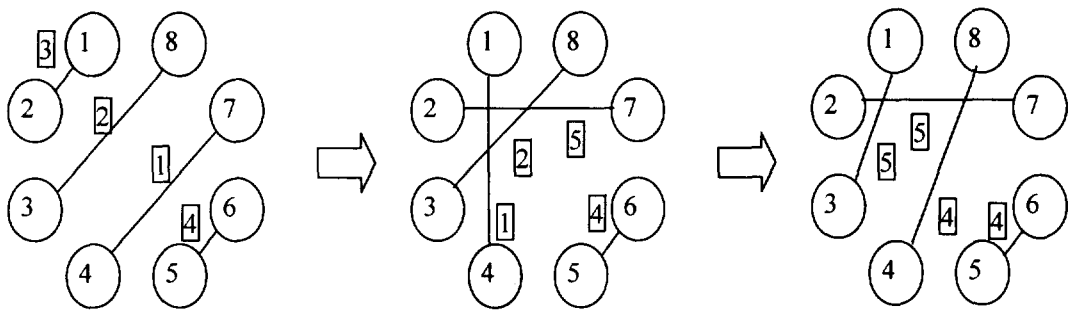


圖 2-2 最大權重配對問題初步解題過程

圖 2-2 說明了表 2-1 權重表中 8 個節點的分散式系統如何利用我們的初步構想，以自我穩定方式找出最大權重配對，其中灰色節點代表擁有特權執行演算法中的規則。假設當系統初始化時，配對組合爲[1,2]、[3,8]、[4,7]、[5,6]。當節點 1 發現與配對[4,7]中節點 4 配對，所形成的新配對[1,4]、[2,7]權重和 6 比目前的配對[1,2]、[4,7]權重和 4 來得大時，則將這兩組配對交換爲新的配對組合。然而當節點 4 發現配對組合[4,8]、[1,3]的權重和 9 大於原配對組合[1,4]、[3,8]權重和 3 時，則將配對組合交換爲[1,3]、[4,8]。當配對組合爲[1,3]、[2,7]、[4,8]、[5,6]時，系統中已經沒有任何節點可以找到更好的交換配對，所以系統達到合法穩定狀態，此時亦表示已找到一個「最大權重配對」的解，以上就是系統自我穩定的過程。

二、自我穩定演算法

㈠最大權重配對演算法

根據上節演算法的初步構想，我們設計出包含五個規則的自我穩定演算法。在上節的初步構想中，假設完全圖中所有節點已經兩兩配對才進行交換配對的動作；然而在現實的狀況中必須考慮系統發生錯誤時，節點指標可能隨時更動的現象，爲了符合自我穩定演算法中容錯的特性，因此必須在我們的初步構想中加入 Hsu-Huang 最大配對問題演算法的觀念。

除了 Hsu-Huang 演算法中所定義的節點指標（$i \to j$）、鄰居節點集合（$N(i)$）之外，最大權重配對自我穩定演算法中還定義了幾種資料結構。$r(i,j)$ 代表線段 (i,j) 上的暫存器，暫存器中的內容若爲節點編號 k，則表示節點 j 應在稍後準備將其指標指向 k。若 $r(i,j)=\phi$，則不需做此動作。雖然 $r(i,j) \neq r(j,i)$，但是節點 i、j 皆可以透過線段 (i,j) 讀取和寫入 $r(i,j)$ 與 $r(j,i)$ 暫存器的內容。系統模型中暫存器的使用在 Wang 所設計的系統負載平衡之自我穩定演算法論文【24】中有相同的設計。$w(i,j)$ 則爲上節所述之線段 (i,j) 的權重。圖 2-3 爲本章中之圖例表示方法，其中實線箭頭代表節點指標，例如 $i \to x$ 表示節點 i 將指標指向節點 x；$x \to i$ 表示節點 x 指標指向節點 i。另外虛線箭頭與編號代表暫存器內容，例如：$i \overset{y}{\cdots\cdots} > j$ 表示 $r(i,j)=y$；同樣地 $i \overset{j}{\cdots\cdots} > y$ 則表示暫存器 $r(i,y)=j$。線段上含有框線的數字則爲線段上的權重，如線段(i,x)上的權重爲 b、線段(i,j)上的權重爲 a。虛線代表節點 t 爲節點 i 的鄰居節點，也就是說節點 t 與節點 i 之間的線段以

虛線表示。在本論文中，我們針對完全圖（complete graph）設計的最大權重配對演算法，因此兩兩節點之間皆互爲鄰居。

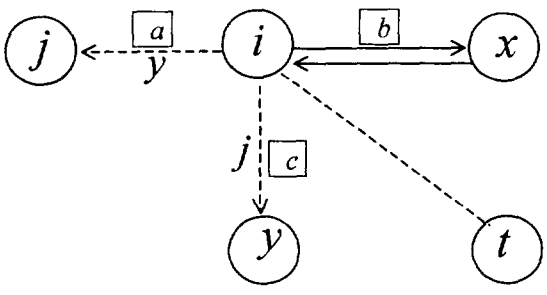


圖 2-3 圖例表示範例

我們所設計的最大權重配對問題之自我穩定演算法如下：

〈規則 1〉	$(\forall t \in N(i): r(i,t) = \phi) \wedge (\forall s \in N(i): r(s,i) = \phi) \wedge$ $(i \to null) \wedge (\forall k \in N(i): \neg(k \to i)) \wedge (\exists j \in N(i): (j \to null) \wedge$ $(\forall t' \in N(j): r(j,t') = \phi) \wedge (\forall s' \in N(j): r(s',j) = \phi))$ $\Rightarrow i \to j$
〈規則 2〉	$(\forall t \in N(i): r(i,t) = \phi) \wedge (\forall s \in N(i): r(s,i) = \phi) \wedge$ $(i \to j) \wedge (j \to k) \wedge (k \neq i) \wedge$ $(\forall t' \in N(j): r(j,t') = \phi) \wedge (\forall s' \in N(j): r(s',j) = \phi)$ $\Rightarrow i \to null$
〈規則 3〉	$(\forall t \in N(i): r(i,t) = \phi) \wedge (\forall s \in N(i): r(s,i) = \phi) \wedge$ $(i \to null) \wedge (\exists j \in N(i): j \to i) \wedge$ $(\forall t' \in N(j): r(j,t') = \phi) \wedge (\forall s' \in N(j): r(s',j) = \phi)$ $\Rightarrow i \to j$
〈規則 4〉	$(\forall t \in N(i): r(i,t) = \phi) \wedge (\forall s \in N(i): r(s,i) = \phi) \wedge$ $(i \to j) \wedge (j \to i)$ $(\forall t' \in N(j): r(j,t') = \phi) \wedge (\forall s' \in N(j): r(s',j) = \phi) \wedge$ $(\exists x,y: (x \to y) \wedge (y \to x)) \wedge$ $((\forall u \in N(x): r(x,u) = \phi) \wedge (\forall v \in N(x): r(v,x) = \phi) \wedge$ $(\forall u' \in N(y): r(y,u') = \phi) \wedge (\forall v' \in N(y): r(v',y) = \phi) \wedge$ $(w(i,x) + w(j,y) > w(i,j) + w(x,y)))$ $\Rightarrow i \to x; r(i,x) = i; r(i,y) = j; r(i,j) = y$
〈規則 5〉	$(\exists s \in N(i): r(s,i) \neq \phi)$ $\Rightarrow i \to r(s,i); r(s,i) = \phi$

首先，當節點 i 和鄰居節點之間的暫存器尚有內容沒有清除時，表示節點 i 或其中的鄰居節點眞正的指標位置尚未正確，也就是必須依照其他規則利用暫存器的內容修正節點指標，因此不能有更改狀態的動作。在〈規則 1〉、〈規則 2〉、〈規則 3〉、〈規則 4〉中皆必須判斷節點 i 以及相關節點暫存器的內容是否有尚未完成的工作，來決定是否可以執行這些規則。

〈規則 1〉、〈規則 2〉、〈規則 3〉是由 Hsu-Huang 最大配對自我穩定演算法【16】中的三條規則修改而來。這三條規則與 Hsu-Huang 最大配對演算法中的規則最主要不同在於加入了暫存器內容的控制。當節點 i 與鄰居節點之間的暫存器並沒有任何內容時，表示節點 i 可以修正自己的指標。如果沒有任何鄰居節點指標指向節點 i 的情形時，節點 i 可以根據〈規則 1〉找到一個指標同樣接地而且沒有尚未完成工作的鄰居節點 j，節點 i 將自己的指標指向節點 j，如圖 2-4 所示。

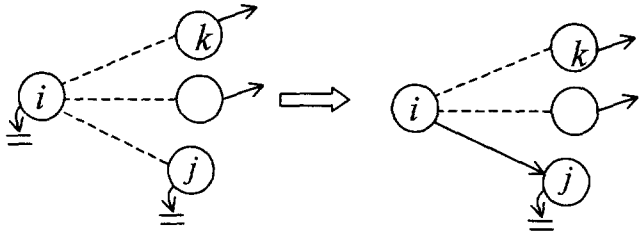


圖 2-4 最大權重配對自我穩定演算法〈規則 1〉

〈規則 2〉則是指當節點 i 指標所指的鄰居節點 j 指向別的節點 k，而且節點 i 和節點 j 都沒有尚未完成的工作，則如圖 2-5 所示將節點 i 的指標接地。

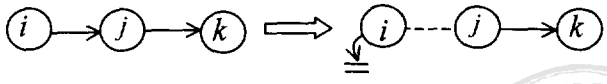


圖 2-5 最大權重配對自我穩定演算法〈規則 2〉

圖 2-6 爲最大權重配對自我穩定演算法的〈規則 3〉，當節點 i 找到一個鄰居節點 j，節點 i 和節點 j 都沒有尚未完成的工作，而且節點 j 的指標亦指向節點 i 時，則節點 i 可利用〈規則 3〉將指標指向鄰居節點 j。

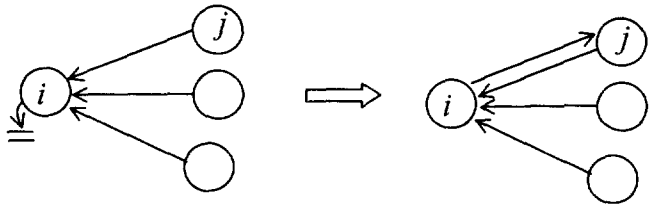


圖 2-6 最大權重配對自我穩定演算法〈規則 3〉

〈規則 4〉、〈規則 5〉實爲本演算法中的重心，也就是上節中所述初步構想的實際規則。〈規則 4〉中節點 i 雖然已經與節點 j 配對成功，但是由於系統中存在另一組配對 $[x,y]$，誠如上節初步構想中所述交換配對後的權重值會更大，則執行交換動作。同樣地，不論是節點 i、節點 j、節點 x 或節點 y 都要符合沒有尚未完成工作的條件，以免更動到尚未完成工作的節點。在圖 2-7 中，a、b、c、d 爲四個節點之間線段的權重值，由於 $b+c>a+d$，因此節點 i 根據〈規則 4〉將指標改指向節點 x，並且將其他該改變的資訊登錄在相關暫存器中。

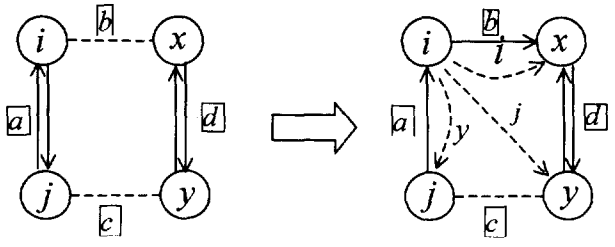


圖 2-7 最大權重配對自我穩定演算法〈規則 4〉

節點 i 在線段（i,x）上的暫存器中填入自己的節點編號 i，表示節點 x 的指標必須更改指標指向自己；節點 i 在線段（i,j）上的暫存器中填入節點編號 y，表示節點 j 的指標必須更改指標指向節點 y；節點 i 在線段（i,y）上的暫存器中填入節點編號 j，表示節點 y 的指標必須更改指標指向節點 j。也就是說在〈規則 4〉中節點 i 共有四件工作需要執行，如表 2-2 所示。

表 2-2 〈規則 4〉中節點 i 的四項工作

〈規則 4〉中節點 i 的四件工作	更改自己的指標，指向節點 x
	在線段（i,x）上的暫存器中填入自己的節點編號 i
	在線段（i,j）上的暫存器中填入節點編號 y
	在線段（i,y）上的暫存器中填入節點編號 j

倘若節點 i 與鄰居節點的暫存器尚有未清除的內容時，表示當節點 i 擁有特權執行演算法的規則時，節點 i 必須先將指標指向暫存器的內容，也就是節點 i 實際上該配對的節點。因此〈規則 5〉中節點 i 根據暫存器內容將自己的指標指向正確的節點，並將暫存器的內容清除。圖 2-8 中，節點 j 與節點 i 線段暫存器 $r(i,j)$ 上的內容爲節點編號 y，於是根據〈規則 5〉節點 j 將指標指向節點 y，並將線段 $r(i,j)$ 暫存器的內容清除。

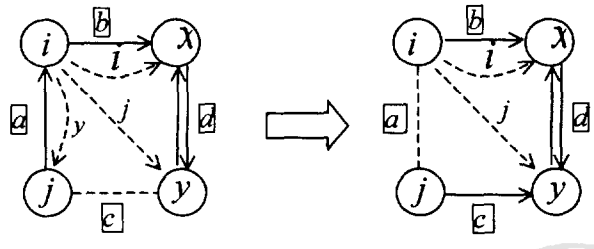


圖 2-8 最大權重配對自我穩定演算法〈規則 4〉

雖然在初步構想中，我們將交換配對視爲一完整動作，然而在實際演算法規則中，交換配對部分則分爲〈規則 4〉及〈規則 5〉兩個部分，因此需要防止尚未完成交換配對工作的節點隨意更動指標，這樣的設計更能符合實際上分散式自我穩定系統模型的定義。

二、最大權重配對演算法範例

接著，我們以上節表 2-1 的範例說明最大權重配對演算法逐步達成自我穩定的一個可能過程。圖 2-9 與範例圖 2-2 最大不相同的地方在系統於初始狀況時，各個節點的指標爲任意指向其他節點或是接地的，而非兩兩節點已經配對。在圖 2-9 中灰色節點代表擁有特權執行演算法中的規則，而灰色節點旁亦註明了此節點執行了哪條規則。

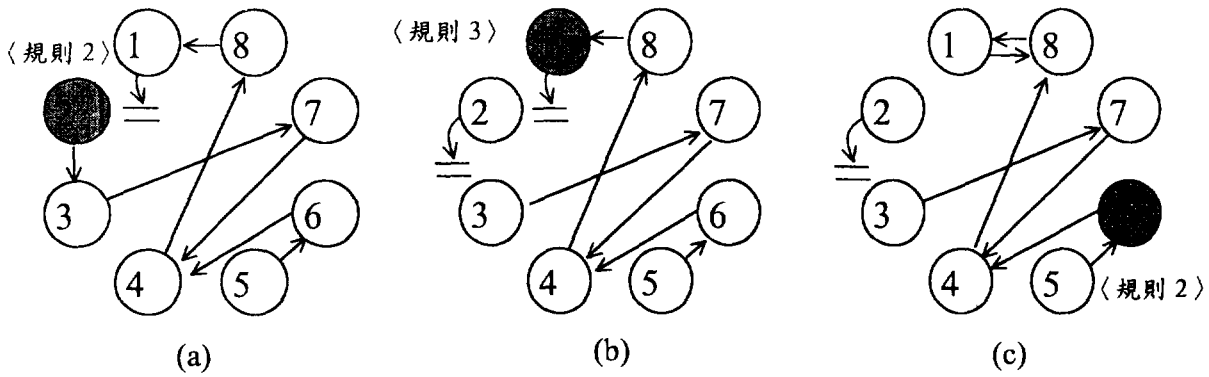
範例：

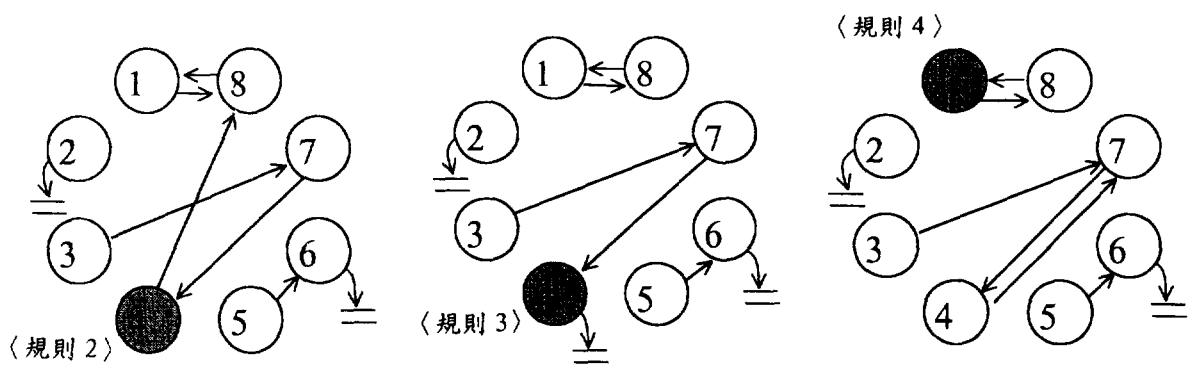
	1	2	3	4	5	6	7	8
1		3	5	1	2	4	1	3
2			2	1	4	2	5	3
3				5	1	4	3	2
4					3	2	1	4
5						4	1	3
6							3	5
7								2
8								

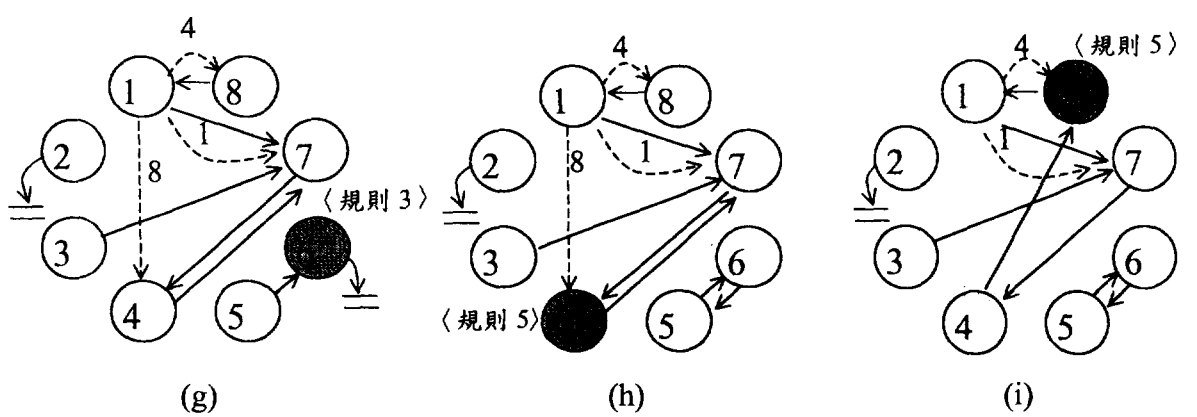
在圖 2-9(a)系統初始狀態時，節點 2 的指標指向節點 3，但是因爲節點 3 的指標卻指向節點 7，因此節點 2 符合〈規則 2〉的先決條件，而將自己的指標接地。接著圖 2-9(b)中由於節點 1 本身指標接地，因此希望與將指標指向自己的節點 8 配對，故執行〈規則 3〉將指標指向節點 8。同樣地，圖 2-9(c)和(d)中節點 6 和節點 4 亦執行〈規則 2〉將自己的指標接地，讓自己有其他配對的機會。在圖 2-9(e)中由於節點 4 將自己的指標接地，因此節點 4 可以根據〈規則 3〉的先決條件，與指向它的節點 7 配對。以上節點轉變狀態皆符合暫存器內容的條件。

在圖 2-9(f)中，因爲系統中已經產生兩對配對，當節點 1 發現若將配對組合[1,8]、[4,7]轉變爲[1,7]、[4,8]時，權重和會由 4 增加爲 5，因此節點 1 執行〈規則 4〉，將自己的指標指向節點 7，並在線段（1,8）上暫存器填入『4』、線段（1,7）上暫存器填入『1』、線段（1,4）上暫存器填入『8』等節點資訊。接下來圖 2-9(g)中因爲節點 1、4、7、8 的相關暫存器都有尚未完成的節點資訊，因此這些節點都有尚未完成的工作。此時，節點 6 因爲節點 5 將指標指向自己，而且節點 5、6 相關的暫存器並沒有內容，故節點 6 執行〈規則 3〉，將自己的指標指向節點 5 形成配對。接著圖 2-9(h)由節點 4 執行〈規則 5〉繼續未完成的交換配對工作，將自己的指標依照暫存器 $r(1,4)=8$ 之指示指向節點 8，並將暫存器內容清除。同理，圖 2-9(i)中節點 8 亦根據暫存器 $r(1,8)=4$ 執行〈規則 5〉將指標指向節點 4，與節點 4 形成新的配對。圖 2-9(j)中節點 7 也根據暫存器 $r(1,7)=1$ 的節點資訊將指標指向節點 1，與節點 1 形成新的配對，完成整個交換配對動作。

接著，圖 2-9(k)中由於節點 3 指標指向已經配對的節點 7，因此執行〈規則 2〉，將自己的指標接地，並在下一個步驟時（圖 2-9(l)），執行〈規則 1〉將自己的指標指向指標同樣接地的節點2。在圖2-9(m)中節點 2 因此可執行〈規則 3〉，與節點 3 配對。在圖 2-9(n)中，在[1,7]、[2,3]兩組配對中，節點 1 發現若將配對交換爲[1,3]、[2,7]時，權重和 10 大於原來配對的權重和 3，因此執行〈規則 4〉將指標指向節點 3，並將線段（1,3）上暫存器填入『1』、線段（1,2）上暫存器填入『7』、線段（1,7）上暫存器填入『2』。節點 2、3、7 則根據這些暫存器上的內容執行〈規則 5〉完成配對交換工作（如圖 2-9(o)至(g)所示），而整個系統也達到合法穩定狀態。在系統自我穩定過程中，各個節點均依照最大權重配對演算法的規則，除了希望找到最大配對之外，更在兩兩配對之後試圖找出更大的權重配對。







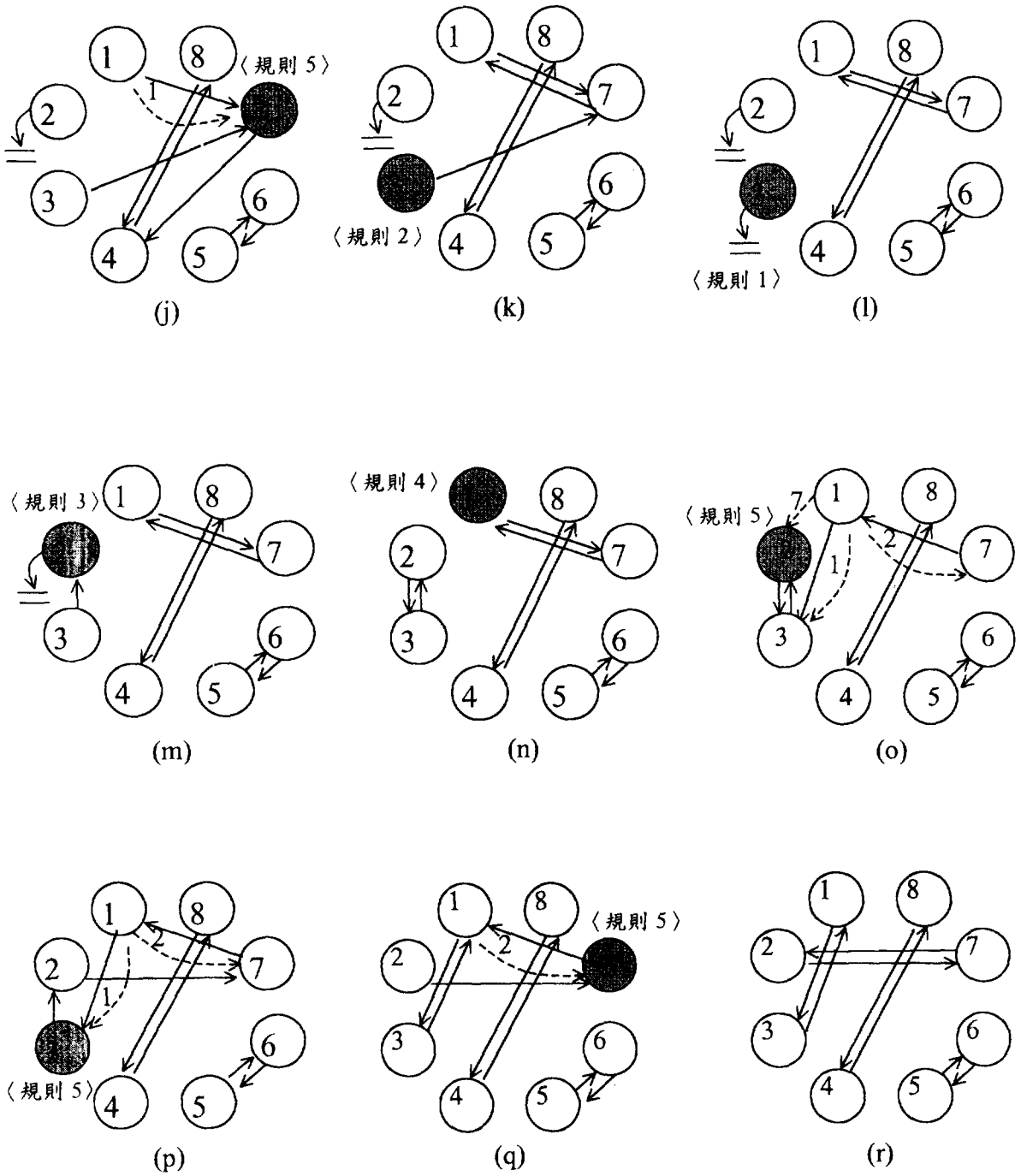


圖 2-9 最大權重配對演算法範例

正確性證明與時間複雜度討論

在本論文緒言所述的驗證系統自我穩定之正確性有三個必要需求條件 ：

系統穩定需求條件（i）：如果系統達到合法穩定狀態，則沒有任何節點能執行任何規則。

系統穩定需求條件（ii）：如果系統仍處於不合法狀態，則存在至少一個節點可以執行規則。

系統穩定需求條件（iii）：不論初始狀態爲何，以及不論中央控制系統挑選了那個節點執行規則，系統保證能在有限步驟內達到合法狀態。

接下來在這節中將證明我們的演算法符合這三條系統穩定條件。由最大權重配對定義中，我們可以知道當所有節點皆已配對，而且找不到任何配對可以交換使得權重和更大時稱之爲合法穩定狀態。在這個定義之下，很輕易地可以證明我們的演算法符合系統穩定需求條件（i）。當系統達到合法穩定狀態時，表示所有的節點都已配對，因此不會執行演算法配對部分的〈規則 1〉、〈規則 2〉、〈規則 3〉；而且所有的節點也都已找到最大權重的配對，因此也不會執行演算法中交換配對及修正指標的〈規則 4〉、〈規則 5〉。

相反地如果系統尚未達到合法穩定狀態，表示系統中尚有未配對的節點或是還有可以改善權重的交換配對，因此至少都還存在一個節點可以執行演算法中的規則，證明了我們的演算法也符合系統穩定需求條件（ii）。

爲了證明我們的演算法符合系統穩定需求條件（iii），首先證明最大權重配對自我穩定演算法會在有限時間內達到合法穩定狀態。

定理一：當節點兩兩配對之後，在穩定的過程中配對組合並不會重複。

證明：根據〈規則 4〉的定義，只有權重和會更大時才交換配對，所以系統的總權重和會越來越大。假設在尋找最大權重配對穩定過程中，整個系統配對組合分別爲 $S_1, S_2, S_3, ..., S_k$ ，其相關的總權重和分別爲 $w_1, w_2, w_3, ..., w_k$ ，如圖 3-1 所示。

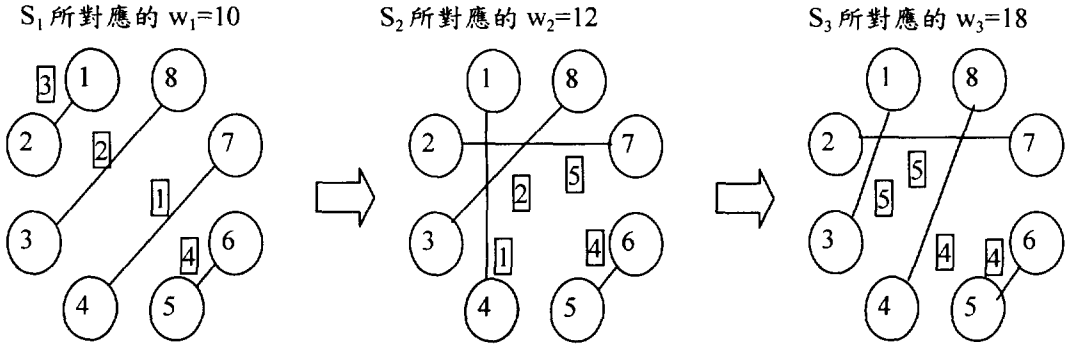


圖 3-1　系統配對組合及相關總權重和說明

這些組合配對經過執行一連串的〈規則 4〉及〈規則 5〉之後回到S_1配對組合。也就是以下的情形出現：

$$S_1 \Rightarrow S_2 \Rightarrow S_3 \Rightarrow ... \Rightarrow S_k \Rightarrow S_1$$

然而我們知道由定義中 $w_1 < w_2 < w_3 < ... < w_k < w_1$ 產生矛盾。由此證明了當節點兩兩配對之後，系統在穩定的過程中配對組合並不會重複。

因爲系統的配對組合個數有限，所以由定理一可以得知當節點已經兩兩配對之後配對組合不會重複，因此在有限的時間內系統保證會達到穩定合法狀態。然而當系統中節點尚爲兩兩配對之前，由 Hsu-Huang 演算法證明了系統保證會在有限時間內將節點兩兩配對。因此證明我們的演算法符合系統穩定需求條件（iii），不論初始狀態爲何，以及不論中央控制系統挑選了那個節點執行規則，系統保證能在有限步驟內達到合法狀態。

定理二：最大權重配對之自我穩定演算法的時間複雜度爲$O(n^2+nk)$。

證明：以下我們針對最大權重配對問題定義了一個函數：

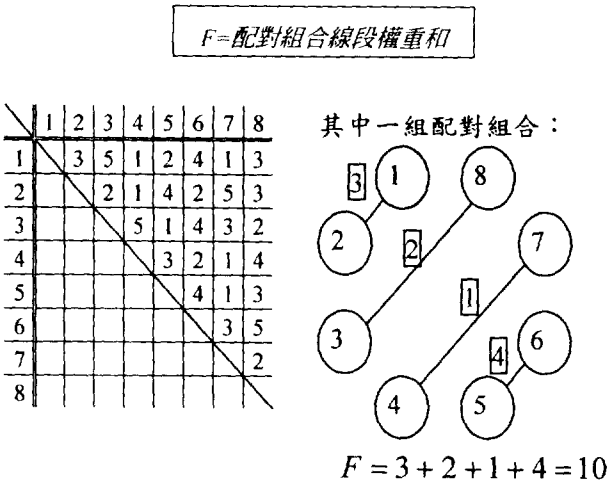


圖 3-2 F 函數範例

圖 3-2 爲 F 函數範例，藉由證明函數 F 會慢慢遞增來分析時間複雜度。當線段上的權重介於 $1 \sim k$ 之間，則我們可以算出當一個節點個數爲 n 的系統達到合法穩定狀態時，最大的權重和爲 $\frac{n}{2} * k$，也就是說函數 F 的最大值爲 $\frac{n*k}{2}$。同理可知，當節點兩兩配對成功但是每個線段上的權重皆爲 1 時，此時系統可能的最小權重和爲 $\frac{n}{2}$，也是函數 F 的最小值應爲 $\frac{n}{2}$。由此可知函數 F 的值介於 $\frac{n}{2} \sim \frac{nk}{2}$ 之間。當節點執行〈規則 4〉時，配對交換之後權重和會增加，因此函數 F 也會遞增。若在初始狀況時，節點兩兩配對的線段權重皆爲 1，也就是最糟狀況時，系統必須花費 $\frac{nk}{2} - \frac{n}{2} = \frac{n(k-1)}{2} = O(nk)$ 的時間達到函數 F 的最大值。換句話說，就是當節點兩兩任意配對之後，要達到最大權重配對需花 $O(nk)$ 的時間。

以上的證明只考慮了任意兩兩節點配對之後達到合法穩定狀態的時間，接下來我們尚須考慮到任意初始狀態到節點兩兩配對的穩定時間，也就是 Hsu-Huang 最大配對演算法穩定的時間。由於 Hsu-Huang 最大配對演算法時間複雜度爲 $O(n^2)$，所以最大權重配對之自我穩定演算法的時間複雜度爲 $O(n^2+nk)$。

在最大權重配對之自我穩定演算法中，系統不但保證在有限的時間內達到合法穩定狀態，並且符合自我穩定演算法的容錯特性。首先，當節點指標因系統暫時性失誤造成錯誤時，系統將視爲初始狀況並根據本演算法中的〈規則 1〉、〈規則 2〉、〈規則 3〉自行更正節點指標，使節點兩兩配對。這三條規則係由 Hsu-Huang 最大配對自我穩定演算法修正而來，仍保存其自我穩定容錯特性。

若因爲系統的暫時性失誤造成暫存器內容更動產生錯誤時，最大權重配對自我穩定系統亦符合容錯功能。當暫存器內容更動而造成節點指標錯誤時，雖然〈規則 5〉會讓節點指標根據暫存器錯誤內容

指向錯誤的節點，但是同樣地系統會依照演算法中的〈規則 1〉、〈規則 2〉、〈規則 3〉自行更正節點指標，使節點重新兩兩配對，如圖 3-3 所示。其中，圖 3-3(a)爲原來正確之系統狀態，圖 3-3(b)則爲系統發生暫時性錯誤時造成暫存器發生錯誤，使得 $r(i,x)=y$ 、 $r(i,y)=i$ 。圖 3-4 即爲圖 3-3 (b)重新達到初始狀態的一種可能過程。首先，圖 3-4(a)中節點 x 執行〈規則 5〉依照暫存器 $r(i,x)=y$ 的資訊將指標指向節點 y，而圖 3-4(b)中節點 y 則根據暫存器 $r(i,y)=i$ ，將指標指向節點 i；接著圖 3-4(c)中節點 j 執行〈規則 5〉將指標指向節點 y，使得系統達到初始狀態，接著系統即可根據〈規則 1〉、〈規則 2〉、〈規則 3〉達到節點兩兩配對，如圖 3-4(e)、(f)、(g)、(h)所示，並根據〈規則 4〉、〈規則 5〉交換配對使得權重和更大，達到自我穩定狀態。

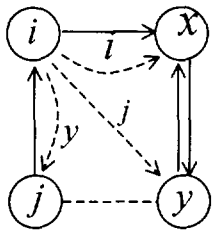


圖 3-3(a) 正確系統狀態

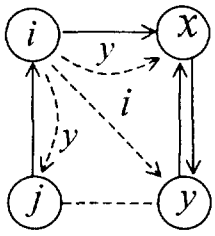


圖 3-3(b) 系統暫存器內容產生錯誤

圖 3-3 最大權重配對系統錯誤範例

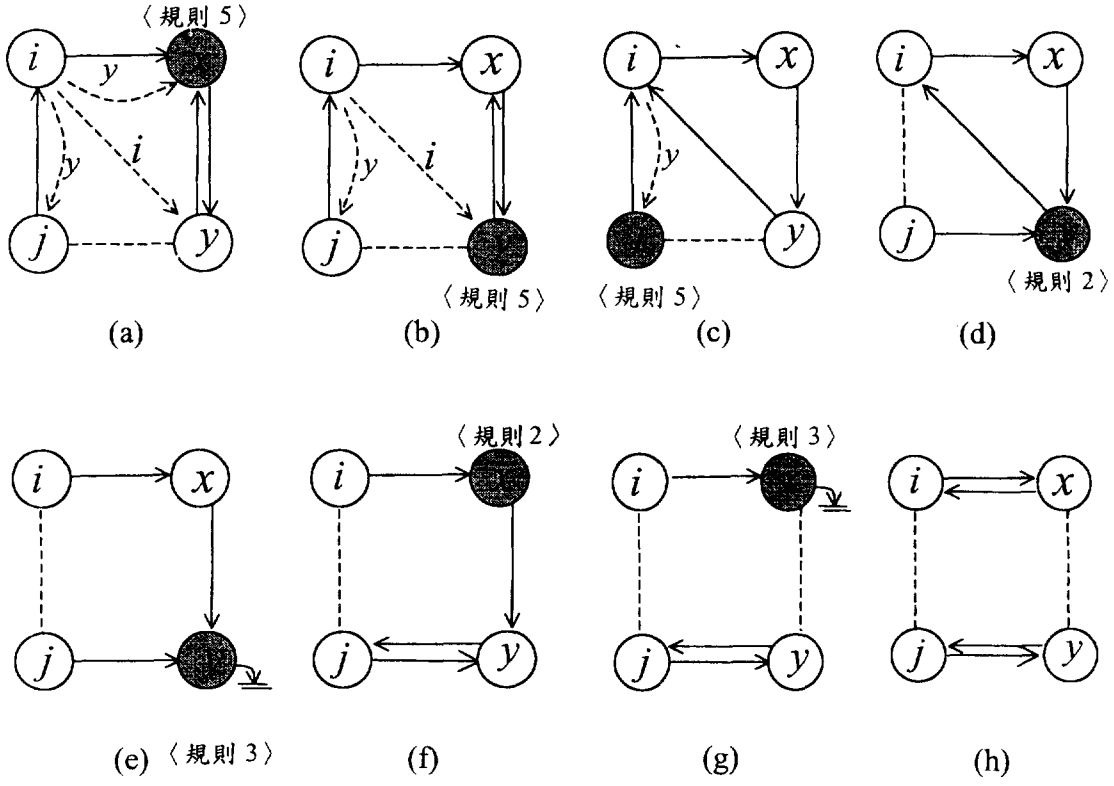


圖 3-4 最大權重配對系統容錯範例

結論與未來研究方向

一、結論

自我穩定系統不論其初始狀態爲何，最後都會在有限的時間內收斂至正確的系統狀態。因爲這種自我穩定的特性，對於不需初始化的系統有極大的用處。除此之外，分散式自我穩定演算法對於系統容錯方面也有相當大的貢獻。在本研究中，我們將自我穩定系統的理論應用在完全圖上的最大權重配對問題，設計出包含五個規則的自我穩定演算法，並針對此自我穩定演算法的正確性進行驗證分析。最大權重問題是指當節點兩兩配對之後，其線段權重兩兩交換並不會找到更大的值，也就是除了希望在圖中找到最大配對之外，更進一步能夠使配對的權重達到最大。因此我們結合了 Hsu-Huang 最大配對自我穩定演算法，以及嶄新的交換配對規則，保留自我穩定系統容錯及自我穩定的特性，設計了時間複雜度爲 $O(n^2+nk)$ 的一個最大權重配對問題之自我穩定演算法，然而此演算法亦達到局部最佳化（local optimum）。

二、未來研究方向

儘管本研究在最大權重配對問題上探討了自我穩定系統的設計與應用，但大體而言僅能算是初步的雛形設計，未來仍有需要改進之處與值得進一步研究的方向。

目前，我們僅針對完全圖設計解決最大權重配對問題的自我穩定演算法，是否可以突破自我穩定系統模型中各節點只知道節點本身與鄰居節點資訊的限制，推廣至一般圖形（general graph）的最大權重配對問題，即是未來的研究方向。此外，在本研究中最大權重配對問題之自我穩定演算法的時間複雜度爲 $O(n^2+nk)$，是否有可能再進一步降低時間複雜度加快系統穩定時間亦是值得繼續研究的重點。

參考文獻

B. Awerbuch, S. Kutten, Y. Mansour, B.P. Ddhamir, G. Varghesse, *Time optimal self-stabilizing synchronizationm*, Proceedings of the 25th Annual ACM Symposium on Theory of Computing, (1993) 652-661.

J. Beauquier, S. Delaët, *Probabilistic self-stabilizing mutual exclusion in uniform rings*, Principles of Distributed Computing, 8 (1994) 378.

N. S. Chen, F. P. Yu and S. T. Huang, *A self-stabilizing algorithm for constructing spanning trees*, Inform. Process. Lett., 39 (1991) 147-151.

Z. Collin, S. Dolev, *Self-stabilizing depth–first search*, Inform Process. Lett., 49 (1994) 297-301.

A. K. Datta, T. F. Gonzalez, V. Thiagarajan, *Self-stabilizing algorithms for tree metrics*, Parallel Process. Lett., 8(1) (1998) 121-133.

J. Desel, E. Kindler, T. Vesper, R. Walter, *A simplified poof for a self-stabilizing protocil: a game of cards*, Inform. Process. Lett., 54 (1995) 327-328.

S. Dolev, A. Israeli, S. Moran, *Uniform dynamic self-stabilizing leader election*, IEEE Transactions on parallel and distributed systems, 8(4) (1997).

E. W. Dijsktra, *Self-stabilizing systems in spite of distributed control*, Comm. ACM, 17 (1974) 643-644.

O. Flauzac, V. Villain, *An implementable dynamic automatic self-stabilizing protocol*, International Symposium on Parallel Architectures, Algorithms and Networks, IEEE, (1997) 91-97.

S. Ghosh, *Binary self-stabilization in distributed Systems*, Inform Process. Lett., 40 (1991) 153-159.

S. Ghosh and M. H. Karaata, *A self-stabilizing algorithm for coloring planar graphs*, Distributed Computing, 7(1) (1993) 55-59.

S. Ghosh, A. Gupta and S. V. Pemmaraju, *A self-stabilizing algorithm for the maximum flow problem*, Distributed Computing , 10(4) (1997)167-180.

S. Ghosh, A. Gupta, T. Herman and S. V. Pemmaraju, *Fault-containing self-stabilizing algorithms*, Proceedings of the 15th ACM Principle of Distributed Computing, (1996) 45-54.

T. Herman, *Probabilistic self-stabilization*, Inform Process. Lett., 35(29) (1990)63-67.

D. Hoover, J. Poole, *A distributed self-stabilizing solution to the dining philosophers problem*, Inform. Process. Lett., 41 (1992) 209-213.

S. C. Hsu and S. T. Huang, *A self-stabilizing algorithm for maximal matching*, Inform. Process. Lett., 43(2) (1992) 77-81.

S. T. Huang and N. S Chen, *A self-stabilizing algorithm for constructing breadth-first trees*, Inform Process. Lett., 41 (1992) 109-117.

P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice-Hall (1994).

J. L. W. Kessels, *An exercise in proving self-stabilization with a variant function*, Inform. Process. Lett., 29 (1988) 39-42.

L. Lamport, *Solved problems, unsolved problems and non-problems in concurrency*, In Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing, (1984) 1-11.

M. Papatriantafilou, P. Tsigas, *On self-stabilizing wait-free clock synchronization*, Parallel Processing Letters, 7(3) (1997) 321-328.

M. Schneider, *Self-stabilization*, ACM Computing Surveys, 25(1) (1993) 45-67.

G. Tel, *Maximal matching stabilizes in quadratic time*, Inform. Process. Lett., 49 (1994) 271-272.

C. A. Wang, *Self-stabilizing algorithms for Distributed systems*, MS.D Thesis, Department of Computer Science and Information Engineering, National Taiwan University, Taiwan (1995).

收稿日期：88 年 9 月 14 日
修正日期：89 年 4 月 10 日
接受日期：89 年 6 月 2 日

The Designs and Analyses of Self-Stabilizing Algorithm for Maximal Weight Matching Problem on Complete Graphs

Rue-Yi Chen Shun-Shii Lin

Department of Information and Computer Education
National Taiwan Normal University
Taipei, Taiwan, Republic of China

Abstract

In 1974, Dijsktra defined a self-stabilizing system as a system which is guaranteed to arrive at a legitimate state in a finite number of steps regardless of its initial state. Since his introduction, self-stabilizing algorithms gained wide-spread research interest. The objectives of this research are to design and analyze self-stabilizing algorithms for maximal weight matching problem. Firstly, Hsu and Huang proved that the time complexity of their self-stabilizing algorithm for finding a maximal matching in distributed networks is $O(n^3)$, where n is the number of nodes in the graph. In 1994, Tel introduced a variant function to show that the time complexity of Hsu-Huang's algorithm is $O(n^2)$. In this paper, we design a self-stabilizing algorithm for maximal weight matching of the complete graph and prove its correctness. The maximal weight matching problem is defined not only to find the maximal matching of the complete graph, but also to let the total weight of the matching edges be maximal. We combine Hsu-Huang's maximal matching algorithm and new swapping rules. This system possesses the properties of fault tolerance and self-stabilization and has a time complexity $O(n^2 + nk)$, where k is the largest weight over all edges in the graph.

Keywords: self-stabilizing algorithm, fault tolerance, maximal matching problem, maximal weight matching problem.