第二章 平行 Alpha-Beta 搜尋演算法

第一節 前言與 Alpha-Beta 搜尋演算法

在棋類演算法中，著名的 Alpha-Beta 演算法作者 Brudno 在 1963 年發表了第一篇的論文【2】，其後相關演算法應用在棋類的論文接二連三的產生，如：NegaScout 演算法【9】、同形表（transposition table）【5,11】、逐層加深（iterative deepening）搜尋【5】、寧靜搜尋（quiescence search）【3,8】、空著裁剪（null move pruning）【1,3,4】等，其它應用在棋類的論文在此便不花費篇幅介紹，僅列出參考資料，但在介紹平行 Alpha-Beta 搜尋演算法前，讓我們先對 Alpha-Beta 搜尋演算法有著基本的瞭解。

Alpha-Beta 演算法是透過對遊戲樹中每一個節點都有一個分數，然後照演算法制訂所謂的 Alpha 下邊界（lower bound）及 Beta 上邊界（upper bound），對最後搜尋所得到的結果，限制其在 Alpha 與 Beta 範圍裡。而 Alpha 及 Beta 之間的範圍，稱為“window”。由於 Alpha-Beta 是深度優先搜尋，如果在搜尋過程中發現某些後來搜尋的子樹其結果不會落在“window”之內，則代表該子樹不會影響最後搜尋結果，就不必搜尋該子樹，於是進行所謂的切捨(cut off)，發現能大量減少原本整體需要搜尋的節點數，節省搜尋的時間。

而在平行化 Alpha-Beta 搜尋演算法時，我們主要考量三個負擔（overhead）：

- 額外搜尋負擔（Extra search overhead）
- 同步負擔（Synchronization overhead）

- 溝通負擔（Communication overhead）

首先，在額外搜尋負擔（Extra search overhead）裡，我們暫不區分每個節點的型態是否有差別，假定每個節點在程式執行中所使用的指令數是一樣的，則我們可以直觀的判定：搜尋的節點數越多，則額外搜尋負擔（Extra search overhead）越大，反之亦然。

而同步負擔（Synchronization overhead）則是表示在多 CPU 的環境裡，某個 CPU 正處於閒置（idle）的狀態，它必須等到某些事件觸發後或是受到其它 CPU 呼叫，才去啟動，這樣就會造成時間的浪費。例如說：假定有 4 個 CPU：P1、P2、P3、P4，正在處理同一個節點的子代，則 P2、P3、P4 可能必須等到 P1 傳回一個值或是呼叫它們啟動後，才各自往不同的分支去進行搜尋。

最後的溝通負擔（Communication overhead），基本上就是在多 CPU 的環境中，CPU彼此必須傳遞訊息做動態溝通的成本。例如：在平行化 Alpha-Beta 搜尋演算法裡，我們必須動態及時性的調整 Alpha、Beta 的值，當有某一個 CPU 發現有變更 Alpha 或 Beta 的值時，它必須立刻通知其它 CPU 一起改變 Alpha 或 Beta 的值。

在本章後面的小節裡，分別介紹 PVS（Principal Variation Splitting）搜尋演算法【10】、EPVS（Enhanced Principal Variation Splitting）搜尋演算法【6】、及 DTS（Dynamic Tree Splitting）搜尋演算法【7】，這三個演算法都是架構在平行化 Alpha-Beta 搜尋演算法裡。PVS（Principal Variation Splitting）搜尋演算法

【10】算是在平行化 Alpha-Beta 搜尋演算法裡最早期的成果，而 EPVS（Enhanced Principal Variation Splitting）搜尋演算法【6】則是將 PVS（Principal Variation Splitting）搜尋演算法【10】更進一步的改良，為目前最好實作且能將平行化 Alpha-Beta 搜尋演算法獲得很好效能的演算法。而 DTS（Dynamic Tree Splitting）搜尋演算法【7】是目前平行化 Alpha-Beta 搜尋演算法裡有最好的效能。但是實作上的複雜度，較 PVS 演算法、EPVS 演算法複雜許多。

第二節　三種節點分類

在介紹平行化 Alpha-Beta 搜尋演算法之前，我們必須先瞭解一下遊戲樹裡面的節點型態。Knuth 和 Moore 兩位大師清楚地將所有節點分成三類【9】，分別是 PV（Principal Variation）節點、CUT 節點、與 ALL 節點。

PV 節點： 根節點（root）的位置是 PV 節點，而 PV 節點底下的第一個子節點也是 PV 節點，然後其它的子節點皆歸為 CUT 節點。所以在 Alpha-Beta 演算法的遊戲樹中，最左邊的節點都是 PV 節點。

CUT 節點：CUT 節點為 PV 節點或是 ALL 節點的子節點，而 CUT 節點的子節點則是 ALL 節點。在 Knuth 和 Moore 的論文中【9】提到，CUT 節點底下只需要檢查第一個節點，其它的節點會被切捨掉（cut off）。

ALL 節點：ALL 節點為 CUT 節點的子節點，所以 ALL 節點與 CUT 節點是互相交替的，但是 ALL 節點底下的所有子節點（CUT 節點）都必須要被搜尋檢

查的，所以在做平行搜尋時，這是一個很好的分割點（Splitting point）。

此三種節點可以圖 2-1 來表示。

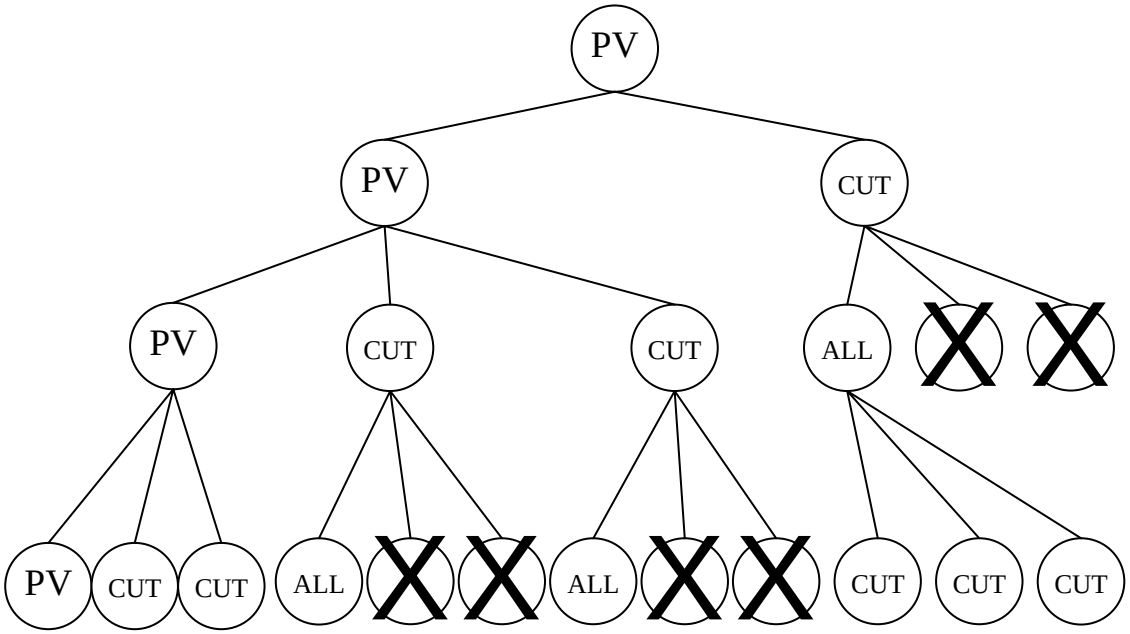

圖 2-1：三種節點型態圖示

第三節　PVS 搜尋演算法

PVS（Principal Variation Splitting）搜尋演算法是依照深度優先搜尋的一種演算法。首先，PVS 搜尋演算法會先從根節點（root）開始，從最左邊展開 PV 節點到深度為 N（預設搜尋深度）的地方，之後，傳回值至 N-1 層的 PV 節點，由該 PV 節點做分派 CPU 的平行搜尋，將各棵子樹各自交由一個 CPU 處理，而當該 CPU 做完該子樹的搜尋後，則進入閒置（idle），等待下一次的分派。

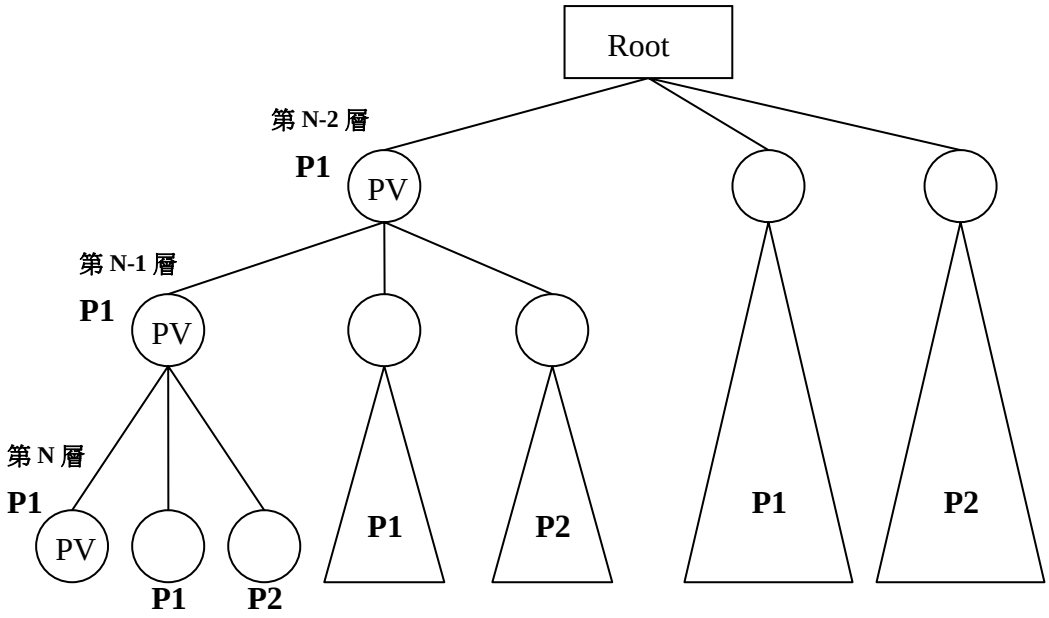

圖 2-2：PVS 搜尋演算法處理一棵小遊戲樹

圖 2-2 是使用兩個 CPU：P1、P2 做 PVS 搜尋演算法在處理一棵小遊戲樹的示意圖，首先由主 CPU P1 從 Root 開始，一路搜尋最左邊的 PV 節點到第 N 層，當一路到第 N 層的 PV 節點搜尋完後，傳回值至第 N-1 層的 PV 節點時，便將剩下的子樹分別交給所有的 CPU 去處理。所以 P1 被安排搜尋一個子樹，P2 被安排搜尋另一個子樹，當該層的所有子樹都搜尋完後，再繼續往第 N-2 層處理，依樣分派一個子樹交給一個 CPU 去處理，遞迴地處理直到 Root 底下所有子樹都被搜尋完畢後則停止。

然而在棋類的遊戲樹使用 PVS 搜尋演算法，往往由於各子樹所需要的搜尋節點數量不一，所以分配到工作量小的 CPU 做完後要等待分配到工作量大的 CPU 也做完後，才能往上一層去做另一層分派工作，所以整體 CPU 的閒置時間較長，造成效能提升有限。表 2-1 是 PVS 搜尋演算法的效能表現。

Processors	1	2	4	8	16
Speed-Up	1.0	1.8	3.0	4.1	4.6

表 2-1：PVS 搜尋演算法處理棋類遊戲樹的效能【10】

第四節　EPVS 搜尋演算法

EPVS（Enhanced Principal Variation Splitting）搜尋演算法【6】顧名思義就是將 PVS 搜尋演算法做更進一步的改良。EPVS 搜尋演算法的精神與 PVS 搜尋演算法是一樣的，分派 CPU 的方法一樣是一棵子樹交給一棵 CPU 去處理，但是在處理 CPU 閒置時，有著更好的作法。

PVS 搜尋演算法是同一層裡的子樹已經被分配光後，當有 CPU 已經處理完該子樹的工作時，就進入閒置狀態做等待。而 EPVS 搜尋演算法不會進入閒置狀態做等待，當有 CPU 已經處理完該子樹的工作時，它會請所有其它正在工作的 CPU 暫時停止，然後加入其它 CPU 當中的第一個（最左邊的）CPU 去一起處理它的子樹，然而一起處理子樹的方法就是延用 PVS 搜尋演算法的方法去做。表 2-2 是 EPVS 搜尋演算法的效能。

Processors	1	2	4	8	16
Speed-Up	1.0	1.9	3.4	5.4	6.0

表 2-2：EPVS 搜尋演算法處理棋類遊戲樹的效能【6】

EPVS 搜尋演算法已經大大提升了 PVS 搜尋演算法的效能，當 CPU 完成了該子樹的搜尋，它雖然要多花一點溝通負擔（Communication overhead）來請其

它 CPU 暫時停止工作，避免加入其它 CPU 工作時造成取得資料有錯誤，但已經大幅降低了 PVS 搜尋演算法所造成 CPU 閒置的情形，提升了整體效能。但是在 8 個 CPU、16 個 CPU，或是更多個 CPU 的環境下，它的表現就會被溝通負擔（Communication overhead）給拖累，造成所表現的效能還不是很令人滿意。

第五節　DTS 搜尋演算法

DTS（Dynamic Tree Splitting）搜尋演算法【7】與 PVS 搜尋演算法及 EPVS 搜尋演算法也是很相似，但是 PVS 搜尋演算法及 EPVS 搜尋演算法是屬於需要由一個主 CPU（Master）去安排工作給其它 CPU（Slave），是一種主從式架構（Master-Slave）。而 DTS 搜尋演算法的每個 CPU 的地位是相同的，不必經過特別指定的 CPU 才可安排工作給其它 CPU，各個 CPU 會將本身處理子樹的資訊放在共享記憶體（shared memory）中，方便讓其它來幫忙的 CPU 使用。

在 DTS 搜尋演算法的論文【7】中提到，作者使用了以下函式來表達他的設計概念，在此就其精神簡述一下。

Help 指令：當某一個 CPU 完成了其負責的子樹的工作後，它會對其它所有 CPU 廣播一個 Help 指令，表示詢問其它 CPU 是否需要幫助，然後等待其它 CPU 的回應，若是收到某一 CPU 的回應確定需要幫忙，便加入該 CPU 的工作，不像是 EPVS 搜尋演算法必須將其它 CPU 都暫時停止工作。

Split()：當某一 CPU 允許其它 CPU 來幫忙，該 CPU 必須將子樹資訊複製一

份到共享記憶體中讓其它來幫忙的 CPU 存取。而在這邊由於要避免兩個以上的 CPU 同樣存取相同子樹資訊，所以作者採用一個 SPLIT BLOCK（分割區塊）的作法，控制 CPU 固定存取共享記憶體中指定區塊的資料，避免錯誤發生。

其它像是 Select()是用在有在做平行搜尋的節點（current node），必須在共享記憶體中取得下一步的走法（move list）時使用的；Merge()是處理結合平行搜尋時，不同子樹搜尋後出來的資訊，譬如說子樹 1 搜尋後的 Alpha-Beta 是（-5, 20），而子樹 2 搜尋後的 Alpha-Beta 是（-10, 10），則必須做 Alpha 取最大值、Beta 取最小值等處理，使 Alpha-Beta 變成（-5, 10）再傳回上一層的分割點（Splitting point）。

這些函式都是作者為了提升平行搜尋的效能而特別設計的，由於在多 CPU 的環境下做平行搜尋，如果是 CPU 們各做各的，則程式設計就比較簡單，但是要做到在共享記憶體中彼此分享資訊，則程式設計的複雜度就較困難許多，但是相對的效能提升也能大幅增加，表 2-3 便是 DTS 搜尋演算法的效能。

Processors	1	2	4	8	16
Speed-Up	1.0	2.0	3.7	6.6	11.1

表 2-3：DTS 搜尋演算法處理棋類遊戲樹的效能【7】

另外，DTS 搜尋演算法不像 PVS 搜尋演算法固定只能在 PV 節點上做分割（Splitting），而是在各個節點都有機會發生，所以要是在 PV 節點或是 ALL 節點做分割的話，對整體效率都是好的表現，但是要是在 CUT 節點做分割的話，

反而會造成 CPU 做了白工。這是由於目前電腦棋類的搜尋技術對 CUT 節點的第一個子節點發生切捨率都高達 80%以上，所以對 CUT 節點做分割只會造成 CPU 的資源浪費。挑選好的分割點（Splitting point）的方法至今仍然是一個很值得研究的議題，DTS 搜尋演算法作者 Robert M. Hyatt 也是將此議題放在他未來的研究工作上。