第三章 DTS 搜尋演算法效能改良

第一節　過去的效能與實驗數據發現

在研究 DTS（Dynamic Tree Splitting）搜尋演算法時，在作者 Robert M. Hyatt 的研究報告【7】上顯示，他的程式在 16 個 CPU 的電腦上，可提昇效能達 11 倍之多，但這似乎還有進步的空間。我們發現在他的演算法上，若是所使用的 CPU 數量越多，所造成額外搜尋的節點數會有暴增的情形，如表 3-1 所示。

	processors				
pos	1	2	4	8	16
1	87,735,974	89,052,012	105,025,123	109,467,495	155,514,410
2	88,954,757	90,289,077	100,568,301	110,988,161	137,965,406
3	101,302,792	102,822,332	111,433,074	117,366,515	119,271,093
4	71,726,853	72,802,754	74,853,409	88,137,085	104,230,094
5	49,386,616	50,127,414	55,834,316	61,619,298	89,506,306
6	133,238,718	135,237,296	146,562,594	168,838,428	226,225,307
7	58,593,747	62,602,792	66,243,490	68,868,878	93,575,946
8	225,906,282	229,294,872	248,496,917	261,728,552	340,548,431
9	122,264,617	124,098,584	138,226,951	159,930,005	199,204,874
10	75,301,353	76,430,872	80,651,716	83,656,702	93,431,597
11	95,321,494	96,751,315	104,853,646	107,369,070	123,994,812
12	79,975,416	85,447,418	85,657,884	94,000,085	112,174,209

表3-1：作者Robert M. Hyatt實驗12個盤面的總共搜尋的節點數【7】

表 3-1 為論文【7】的實驗數據。其中 POS 為：position of search 。即表示作者 Robert M. Hyatt 給了該程式 12 個盤面做測試，分別在 1、2、4、8、16 個 Processor 下做搜尋。我們發現，雖然使用多個 CPU 能增加整體效能，但是在整體搜尋的過程中，所需搜尋的總節點數也增加了許多。

在研究的過程中，我們首先探討了為何總節點數會有暴增的情形，我們發現原因如下所述。

在遇到分割點（Split point）時，若是我們將所有 CPU 都拿來同時平行搜尋下面的子樹時，（假定現在有 8 個 CPU），狀況將如圖 3-1 所示。反之，若是我們遇到 Split point 時，只使用 4 個 CPU 的情形，將會如圖 3-2 所示。

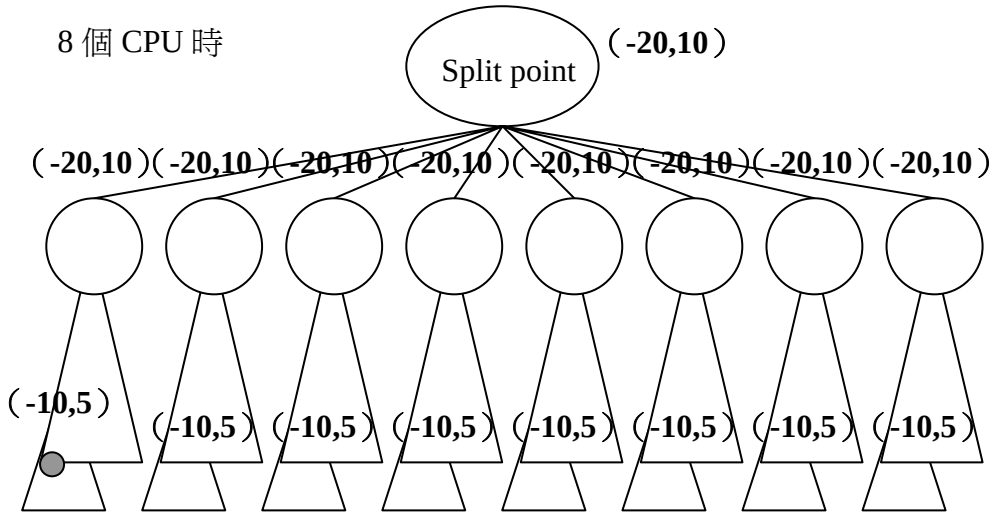

圖 3-1：8 個 CPU 時的示意圖

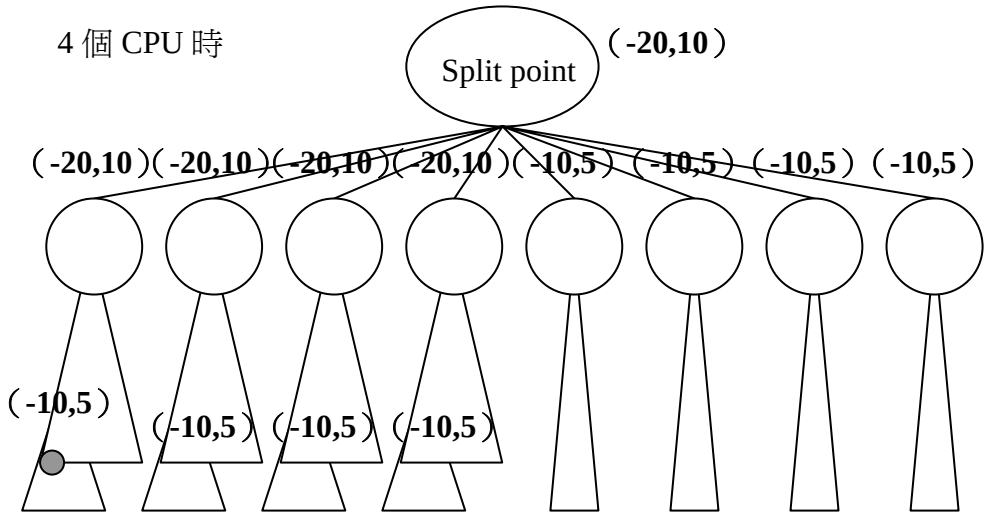

圖 3-2：4 個 CPU 時的示意圖

一開始，我們使用（-20, 10）這個期望搜尋範圍（Window）去做搜尋，在搜尋途中，我們會碰到降低期望搜尋範圍的情形，若是降低期望搜尋範圍的值改為（-10, 5），則後面的搜尋將都會使用這個較小的範圍（-10, 5）來做搜尋，所造成需要搜尋的節點數比用大範圍的（-20, 20）變少很多。

很明顯的我們發現，在一個分割點（Split point）的底下有 8 棵子樹需要做搜尋，若是一口氣就使用 8 個 CPU 去做搜尋，這樣 8 個 CPU 一開始都會使用大範圍去做搜尋；若只先用 4 個 CPU 去做搜尋，在搜尋時先降低其範圍，這樣後面的子樹就會使用較小的範圍去搜尋，則整體所需要搜尋的總節點數就會減少。

直觀的，要做平行搜尋，CPU 數量自然是越多越好，整體速度效能自然越高。但若是 DTS（Dynamic Tree Splitting）搜尋演算法是應用在平行 Alpha-Beta 搜尋演算法下，適當的安排 CPU 的分配工作，便可獲得更高的效能。而如何針對整棵遊戲樹予以適當的切割，使每個 CPU 的負荷量都能平均分配，便是本章下面提出的改良方法。

第二節　改良方法一：橫向思考

當我們遇到分割點（Split point）時，我們不一口氣將所有的 CPU 都分配下去做子樹的搜尋，而是先保留若干個 CPU 來幫忙其它的 CPU 工作。譬如說：在 8 個 CPU 的環境下，當我們遇到分割點（Split point）時，原本的 DTS 搜尋演算法是將 8 個 CPU 都各別安排一棵子樹去做搜尋，但我們在 8 個 CPU 裡保留了 2

個 CPU 先不安排子樹去工作，而是將這保留的 2 個 CPU 先去幫忙那 6 個 CPU 完成各自的子樹搜尋，如圖 3-3 所示。

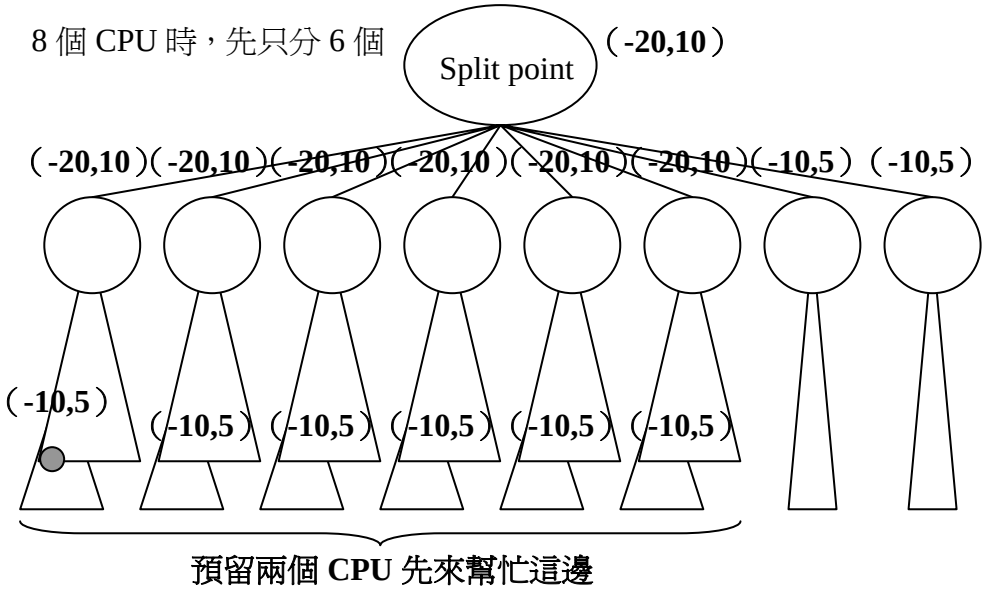

圖 3-3：有 8 個 CPU 時，先只分 6 個 CPU 的示意圖

如此一來，先提升原先 6 個子樹的搜尋效率，加快找到降低期望搜尋範圍，使得後面的第 7 個子樹與第 8 個子樹可以減少整體搜尋的節點數，降低搜尋負擔（Search overhead）。

但是，由於我們保留 2 個 CPU 去幫忙前面的 6 個 CPU 的子樹搜尋，而當它們要加入幫忙時，一樣是使用 DTS 搜尋演算法做運算，這樣會造成在前 6 個子樹裡，我們的分割點（Split point）又增加了，這樣相對的也就增加了同步負擔（Synchronization overhead）與溝通負擔（Communication overhead）。

我們賺到的是減少了整體搜尋的節點數，降低了搜尋負擔；但是相對的分割點增加，額外增加了整體的同步負擔與溝通負擔。在這兩者取捨之下，很幸運的

我們可以找到較好的保留 CPU 數量，使得我們的改良方法一成功的增加了原本 DTS 搜尋演算法的效率。

實作時，在 Crafty 的程式碼部分，主要是在 Thread.c 裡面，控制一個 max_thread_group 的參數，當有 CPU 要執行 DTS 搜尋演算法時，它會去判斷一個 For 迴圈：

```
for (proc = 0; proc < shared->max_threads && nblocks < shared->max_thread_group;
        proc++) {    ……
                執行 DTS 搜尋演算法;
                nblocks++;
                …… }
```

其中 nblocks 是紀錄在共享記憶體中，我們能取得的區塊數，區塊中記錄了該分割點（Split point）所分出的子樹資訊，所以若是將 max_thread_group 設為 6，則該分割點最多只能取出 6 個子樹的資訊給空閒的 CPU 使用，而在 DTS 搜尋演算法中，每一個節點都有可能為分割點，所以剩下 2 個空閒的 CPU，會去在其它分割點做 DTS 搜尋演算法。

DTS 搜尋演算法的作者 Robert M. Hyatt 在論文中曾提到希望分割點的總數要越少越好，而在我們的改良方法一中，則是改變了這個觀念，我們是增加了分割點的總數，卻將整體遊戲樹做橫向的瘦身，降低了整體需要搜尋的節點數，來

換得提升整體的效能。

第三節　改良方法二：縱向思考

在透過研究改良方法一的過程中，我們發現分割點增加得越多，確實造成整體負擔越大。所以我們開始尋找一種降低整體分割點數量的方法，這便是改良方法二的由來。

改良方法二主要是做一個層數的控制，控制允許使用 DTS 搜尋演算法是在哪一個層數之上，請看圖 3-4。

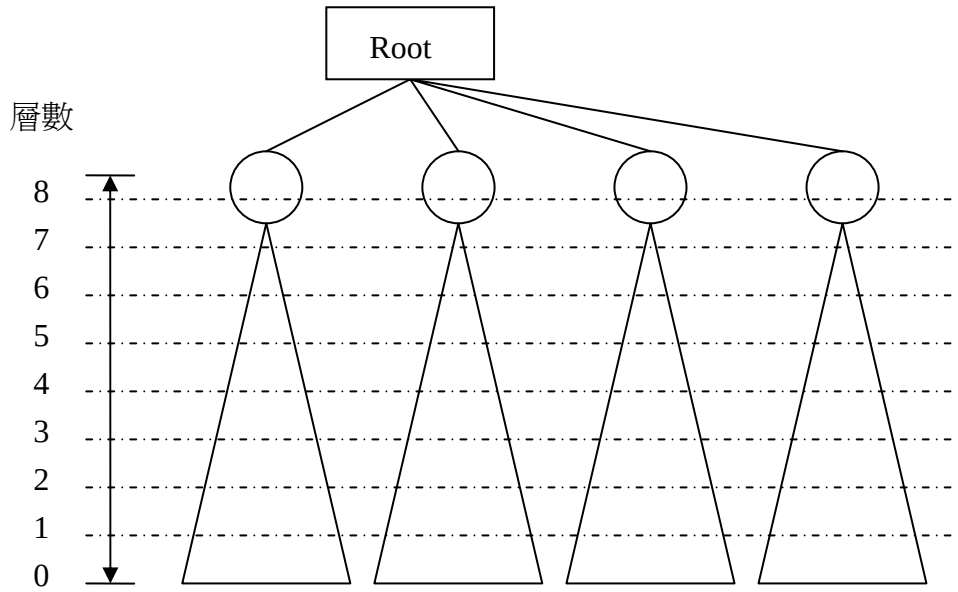

圖 3.4：遊戲樹的層數示意圖

如果我們限制只有在第 8 層的層數以上才可以做 DTS 搜尋演算法的分割（Split），那麼整個遊戲樹的分割點只會有 1 個，就是在根節點（root）；但要是沒有做層數的控制，則第 0 層以上每個節點都可以做 DTS 搜尋演算法的分割，那整體的分割點就會變成很多很多；而譬如說我們控制一定要在第 4 層以上才可

以做 DTS 搜尋演算法的分割，那這樣第 0、1、2、3 層都不會發生分割的動作，所以相較於沒有控制的情形，我們的整體分割點就會比較少。由此觀之，我們控制的層數越高，分割點的數量就會越少；但是若高過於某一個層數後，則又會將 DTS 搜尋演算法的效能打折扣，像是若如圖 3-4 的情形，我們限制在層數 8 以上才可以做 DTS 搜尋演算法的分割，雖然整體分割點會變成只有一個，但是每個子樹則退化變成只有一個 CPU 在工作，則完全失去了 DTS 搜尋演算法的效果。

DTS 搜尋演算法的作者 Robert M. Hyatt 也曾在論文中提到希望能找到“理想”的分割點，才去做分派 CPU 工作的動作，避免有 CPU 是做白工。可惜的是，“理想”的分割點難以判別，故實際上尚無法避免 CPU 做白工的情形。我們則是統一限定一定要到某個層數以上，才可以做 DTS 搜尋演算法的分割。這樣的作法，也是因為考量一個 CPU 不應該在整個遊戲樹底下較低的層數做分割，因為這樣只取得到一棵小子樹的工作量，然後很快的做完，就馬上又要與其它 CPU 做溝通來取得工作，有大材小用之感。因此安排各 CPU 合理的工作量，避免浪費與其它 CPU 溝通的負擔，這是改良方法二能提升整體效能的地方。

而在 Crafty 程式碼中，主要是在 Search.c 裡面，控制一個 min_thread_depth 的參數，當要進入 DTS 搜尋演算法之前，我們要先做一個判斷：

```
if (shared->smp_idle && moves_searched && shared->min_thread_depth <= depth)
    {……
      進入 DTS 搜尋演算法;
```

……}

depth 是指我們現在正在搜尋的節點的層數，若沒有大於等於我們設定的 min_thread_depth，則就不使用 DTS 搜尋演算法，也就不會發生 Split 的狀況。