第四章 效能分析

第一節　改良方法一的實驗結果

在本章中，我們的測試環境如下：

作業系統：FreeBSD 6.1-BETA4

CPU：AMD Opteron（tm） Processor 875（2205.01-MHz K8-class CPU）

記憶體：8192 MB

為了驗證改良方法一，首先我們使用 Crafty 20.14 版將之改寫以做實驗，它是將 Thread 事先產生好，然後等待使用。我們分別測試了在 8 個 Thread 裡，一開始只先分派使用 7、6、5 個 Thread 做實驗，而如果是一次就分 8 個 Thread，則便是原來 Crafty 的設計方法。我們共測試了 12 個盤面（如附錄 A 所示），每個盤面都是做了 10 次測試取得平均值，實驗結果如表 4-1 所示。

盤面 1	8 個分 8 個	8 個分 7 個	8 個分 6 個	8 個分 5 個
節點數	217827431	185959745	153123536	178346550
執行時間	25.13	21.961	18.503	21.651
nps	8.7	8.48	8.25	8.2
分割點數	109892	123941	136068	157293
盤面 2				
節點數	506352440	393177325	433282129	497209830
執行時間	52.864	42.428	47.592	55.342
nps	9.55	9.25	9.12	8.94
分割點數	167864	193837	267838	365379
盤面 3				
節點數	329814373	286023191	276036510	273031123
執行時間	34.805	31.222	30.753	30.869
nps	9.48	9.15	8.99	8.74
分割點數	131587	156978	182943	218792
盤面 4				

節點數	452997154	379309441	390566679	429491373
執行時間	43.924	37.493	39.501	44.438
nps	10.34	10.13	9.89	9.67
分割點數	132364	163278	218394	289579
盤面 5	8 個分 8 個	8 個分 7 個	8 個分 6 個	8 個分 5 個
節點數	621819672	546141203	537063924	571519127
執行時間	60.42	54.164	54.119	58.533
nps	10.23	10.06	9.9	9.72
分割點數	256002	266460	296661	353365
盤面 6				
節點數	184536312	154100345	150038657	167050199
執行時間	19.077	16.549	16.421	18.604
nps	9.69	9.27	9.12	8.88
分割點數	57594	72661	91960	119514
盤面 7				
節點數	466223366	420704237	394591142	396289940
執行時間	44.674	41.391	39.728	40.736
nps	10.43	10.14	9.93	9.72
分割點數	233705	264680	298095	337305
盤面 8				
節點數	124865972	104556795	83713019	88489349
執行時間	12.248	9.933	9.435	10.149
nps	10.17	9.55	8.86	8.69
分割點數	48935	70219	81880	97362
盤面 9	8 個分 8 個	8 個分 7 個	8 個分 6 個	8 個分 5 個
節點數	150312497	119661654	113392313	114222004
執行時間	16.928	13.954	13.666	14.024
nps	8.87	8.55	8.28	8.09
分割點數	68498	74866	92889	109577
盤面 10				
節點數	131627473	100331129	93760317	108496435
執行時間	14.652	12.323	11.139	13.227
nps	8.97	8.7	8.39	8.18
分割點數	67887	70924	80333	107641
盤面 11				

節點數	463911754	415212983	372429271	448475335
執行時間	44.696	41.594	38.505	46.904
nps	10.38	9.96	9.66	9.57
分割點數	181262	223639	243041	334047
盤面 12	8 個分 8 個	8 個分 7 個	8 個分 6 個	8 個分 5 個
節點數	243505821	198563038	168337433	204779837
執行時間	25.839	21.971	19.29	23.647
nps	9.41	9.02	8.69	8.62
分割點數	105914	118849	125093	175541

表 4-1：改良方法一的實驗結果

在表 4-1 中節點數代表該盤面我們搜尋到 15 層的總搜尋節點數；執行時間為程式做完 15 層後總共所花費的時間（秒）；nps 為 node per second，在此其單位為百萬，即平均每秒跑了多少百萬個節點數；分割點數是指搜尋到 15 層，總共有多少個節點做了 DTS 搜尋演算法的分割運算。而 8 個分 8 個就是代表我們有 8 個 Thread，而遇到分割點時，我們就 8 個 Thread 全用；8 個分 7 個就是代表我們有 8 個 Thread，在遇到分割點時，只使用了 7 個 Thread 去對該分割點做 DTS 搜尋演算法的運算，保留了 1 個 Thread 在另外的分割點做 DTS 搜尋演算法的運算，以此類推。

不失一般性的，我們也有做 8 個分 4 個，但由於這樣的分法會造成總搜尋的節點數會比上面的 4 種分法都要多很多，分割點數也是相對多很多，執行時間也相對提高，造成效能更差，在此就不予列出。從實驗結果我們得知，總搜尋節點數在“8 個先分 6 個 Thread”為最少，而分割點數則是節節上升。如圖 4-1、4-2 所示。

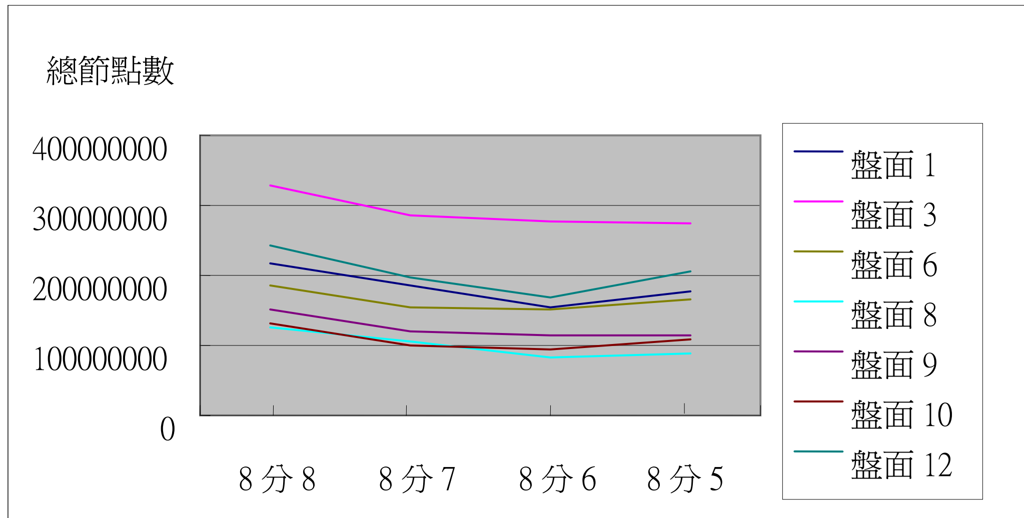

圖 4-1：改良方法一其中盤面的總節點數示意圖

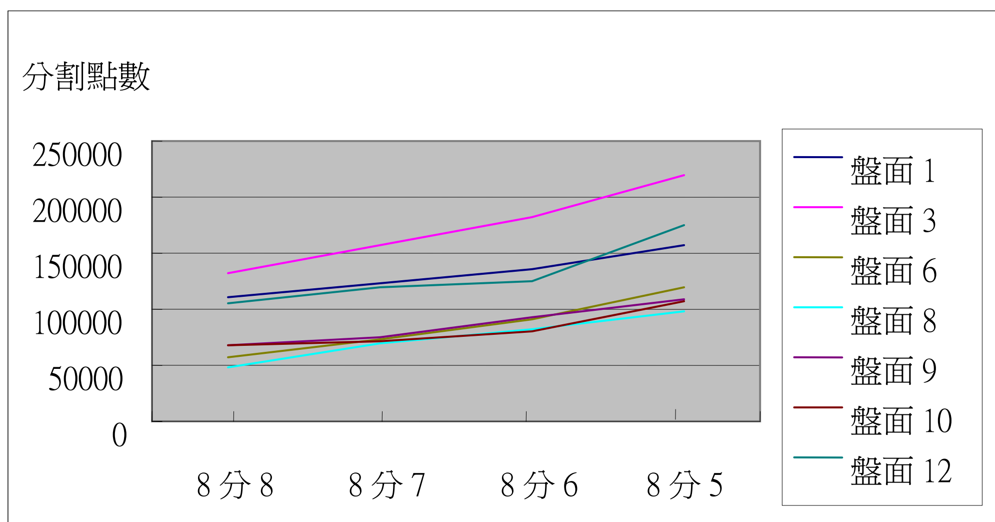

圖 4-2：改良方法一其中盤面的分割點數示意圖

圖 4-1 與 4-2 我們只顯示了 12 個盤面中的其中 6 個，因為若是 12 個盤面的數據都顯示的話，左邊的節點數範圍變廣，則圖中的線條的斜率會比較不明顯，會使得曲線看起來較平坦，所以在此只取 6 個總節點數較接近的 6 個盤面。實驗結果顯示，我們一開始只分 6 個 Thread 去工作，然後留下 2 個 Thread 來幫忙前

面 6 個 Thread 所得到的效能是最好的。而若是 8 個 Thread 只先分 5 個，卻造成比 8 個 Thread 先分 6 個的整體搜尋節點數變多，這是由於留下的 3 個 Thread 可能會有做白工的情形。這個做白工的情形，我們留到第二節再做討論。而由實驗數據我們看到，8 個分 6 個，可以降低原本需要搜尋的總節點數，雖然多增加了一些分割點造成的額外負擔，但是整體搜尋時間上是有明顯的進步，所提升的效能平均約為 20%，如表 4-2 所示。

	8 個分 8 個的 執行時間（T1）	8 個分 6 個的 執行時間（T2）	提升效能 （T1 / T2 %）
盤面 1	**25.13**	**18.503**	**135.82%**
盤面 2	**52.864**	**47.592**	**111.08%**
盤面 3	**34.805**	**30.753**	**113.18%**
盤面 4	**43.924**	**39.501**	**111.20%**
盤面 5	**60.42**	**54.119**	**111.64%**
盤面 6	**19.077**	**16.421**	**116.18%**
盤面 7	**44.674**	**39.728**	**112.45%**
盤面 8	**12.248**	**9.435**	**129.81%**
盤面 9	**16.928**	**13.666**	**123.87%**
盤面 10	**14.652**	**11.139**	**131.54%**
盤面 11	**44.696**	**38.505**	**116.08%**
盤面 12	**25.839**	**19.29**	**133.95%**
			平均為 120.57%

表 4-2：改良方法一的提升效能

第二節　改良方法一簡易的分析及驗證方法

之前在第一節所提到做白工的情形，主要是因為在某一節點做單 CPU 的展開搜尋，假定原本可能有 35 個子節點要搜尋，但是它可能在第 2 個子節點就發

生 fail high。所謂 fail high 即是該節點的傳回值已經大於原本期望搜尋範圍（Window）的 Beta 值，由於展開的子節點是有經過走法排序後的結果，如果第 2 個子節點已經發生 fail high 了，代表後面的子節點就算去進行搜尋，其結果也不會影響 Beta 值，所以後面的子節點全部可以切捨掉，也稱為 Beta 切捨（Beta cut）。在此狀況下，35 個分支就只有 2 個分支被完整展開，其它 33 個分支被切捨掉了。而要是使用平行搜尋，若是分了 6 個 CPU 在此節點，則當第 2 個 CPU 做完後發現了 fail high，代表後面子節點的 4 個 CPU 先前所做的工都是白費的。

Fail high 發生的機率與棋類程式裡面的走法排序與審局函數有著相當大的關聯。而在改良方法一的實作下我們發現，要是程式的 fail high 發生機率越高，則 CPU 的分配方法是一開始就全部給，只留下少量的 CPU 去幫忙。反之，要是程式的 fail high 發生機率越低，則 CPU 的分配方法是一開始分配出去的比較少，額外留下幫忙的 CPU 要留較多。

我們使用 Crafty 20.14 版，根據大略的實驗統計，程式中每個盤面發生 fail high 的機率是 87%～93%左右，我們便取平均，計算出平均發生 fail high 的機率約為 90%。而原始每一個節點的子節點數平均為 35 個，透過 fail high 發生的機率，我們估計實際上平均每個節點會展開的子節點數為 2.12 個，故原先 8 個 CPU 我們只先分 6 個 CPU 工作，剩下的 2 個 CPU 去幫忙某一 CPU 時，這樣最多為 2+1 個 CPU 在一起工作，由於平均會展開的子節點數約為 2.12，所以要是再多一個 CPU 的話，則又提升了做白工的情形。

透過 fail high 發生的機率，我們尋找會發生 fail high 在哪一個分支的情形。如圖 4-3 所示。

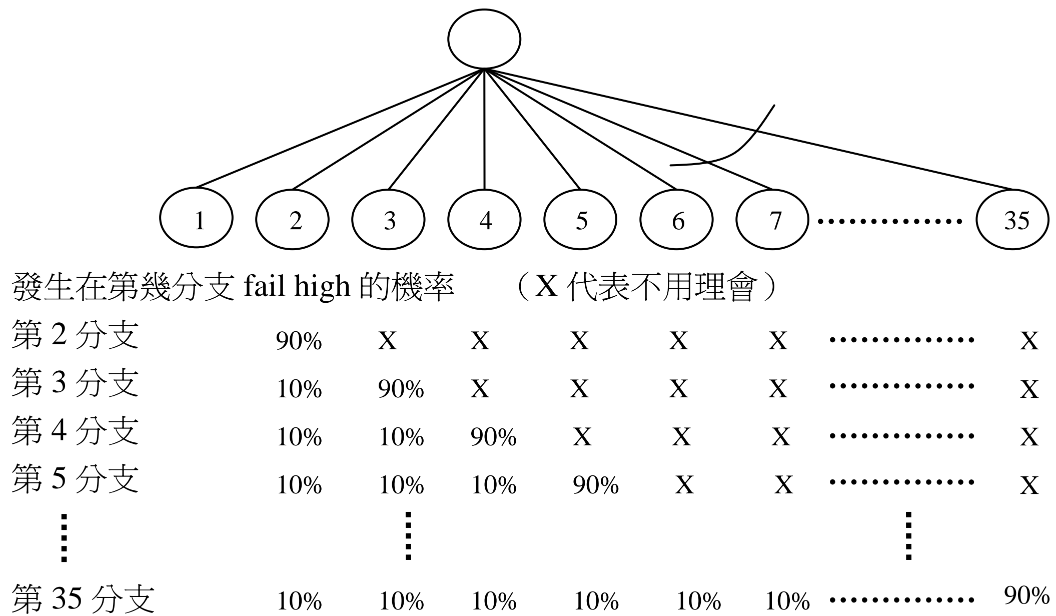

圖 4-3：節點會發生在第幾分支 fail high 的機率示意圖

所以在第 2 分支就發生 fail high 的情形為 90%，此時共完整展開 2 個分支。而在第 3 分支發生 fail high 的情形為 0.1*0.9＝9%，此時共完整展開 3 個分支。在第 4 分支發生 fail high 的情形為 0.1*0.1*0.9＝0.9%，此時共完整展開 4 個分支，其它以此類推。透過機率的詳細計算，其完整展開的分支數量的平均值＝ $2*0.9+3*0.1*0.9+4*(0.1)^2*0.9+5*(0.1)^3*0.9+\ldots\ldots+35*(0.1)^{33}*0.9 \fallingdotseq 2.12$。其結論也是大約等同於直接取（$1+\dfrac{1}{\text{fail high機率}}$）。例如：若是 fail high 為 10%，則代表大約在第 11 個分支才發生 fail high，所以共有 11 個分支必須展開。而若 fail high 為 100%，則代表第 2 分支一定就會發生 fail high，所以後面所有的分支都不需展開，也就代表不需要任何 CPU 的支援。藉由上述的情形，我們推導出一

種簡易的分析及驗證方法，可以事先推估出一套棋類程式的最佳 CPU 分配法則：

保留的 CPU 量 ≦ $\left(1+\dfrac{1}{\text{fail high機率}}\right)$

在我們的實驗裡，原始節點的平均分支度是 35，fail high 機率是 0.9，所以是

$\left(1+\dfrac{1}{0.9}\right)=2.1$，故我們保留了 2 個 CPU，再加上原本的 1 個 CPU，這樣總共就為 3 個 CPU，在 Crafty 20.14 版估計會得到最好的效能。

相同的，若是將 Crafty 20.14 版使用在擁有 16 個 CPU 的機器去執行，假定節點的平均分支度仍然是 35，fail high 機率也是 0.9，我們可以判定其效能表現最好的情形，應該發生在 16 個分 14 個，仍然保留 2 個 CPU。同理推廣到使用在有 N 個 CPU 的機器，我們判定最好的效能應該也是在 N 個分 N-2 個，保留 2 個 CPU，會得到最好的效能。

承上，假定 Crafty 20.14 版的 fail high 機率只有 0.5，而節點的平均分支度仍然是 35，在擁有 16 個 CPU 的機器去執行的情形下，則我們可以判定其效能表現最好的情形就是不必要做任何的保留了，因為 $\left(1+\dfrac{1}{0.5}\right)=3$，保留 3 個 CPU。若是在 N 個分 N-3 個，保留 3 個 CPU 會得到最好的效能。

第三節　改良方法二的實驗結果

同樣的測試環境，同樣是使用 Crafty 20.14 版，我們分別測試了控制在第 2、3、4、5、6 層才允許使用 DTS 做分割的實驗，實驗數據如表 4-3 所示：

盤面 1	第 2 層	第 3 層	第 4 層	第 5 層	第 6 層
節點數	262596288	240068172	180681646	225264456	235690478
執行時間	43.99	26.64	19.13	23.77	24.96
nps	6	9	9.4	9.5	9.4
分割點數	597554	112764	32618	18112	7851
盤面 2					
節點數	437185194	420784697	393902630	479855313	540510027
執行時間	78	45.58	38.97	47	52.31
nps	5.6	9.2	10.1	10.2	10.3
分割點數	1250696	258592	63302	34121	16748
盤面 3					
節點數	379279737	313906554	251095171	273982127	311146229
執行時間	69	34.69	25.16	27.47	31.29
nps	5.4	9	10	10	9.9
分割點數	1119902	210878	47812	20329	9459
盤面 4					
節點數	466477295	440656358	382751580	398780270	442716925
執行時間	77	45.08	34.52	39.42	43.9
nps	6	9.8	10.9	11	10.8
分割點數	1231656	264648	72312	22172	10891
盤面 5					
節點數	616139568	586957297	523619016	561946935	584758085
執行時間	104	67	49.03	53.02	55.04
nps	5.9	9.9	11	10.7	10.8
分割點數	1657446	366055	87450	28223	12780
盤面 6					
節點數	192834951	144645434	101503855	143510411	151123151
執行時間	23.24	15.63	10.37	14.37	15.57
nps	6.3	9.3	9.8	10.2	10
分割點數	324896	90772	39649	18101	7375
盤面 7					
節點數	513890170	424441131	405183267	440291666	468528951
執行時間	66	40.76	34.69	41.07	45.3
nps	6.2	10.5	11.7	11.3	11.3
分割點數	997753	171437	58994	22961	10589
盤面 8	第 2 層	第 3 層	第 4 層	第 5 層	第 6 層
節點數	131541349	106503437	76425908	96976752	105027149

執行時間	17.62	10.29	7.11	8.96	9.56
nps	7.5	10.4	10.7	10.8	10.8
分割點數	188762	37727	18609	7408	3496
盤面 9					
節點數	159703579	132364481	104904198	114205469	123478923
執行時間	24.55	14.71	11.75	12.11	13.01
nps	6.5	9	9.4	9.5	9.5
分割點數	282816	58806	18132	9424	4582
盤面 10					
節點數	112483825	109064171	92993014	109777853	142410504
執行時間	16.04	12.99	9.59	11.01	14.24
nps	7	9	9.7	10	10
分割點數	160661	56515	18313	9861	5105
盤面 11					
節點數	505168972	483662903	443294447	475168127	587996044
執行時間	71	46.32	38.36	41.32	53.07
nps	7.2	10.4	11.6	11	11.1
分割點數	880033	183941	40933	20200	14908
盤面 12					
節點數	297505156	271661381	230044707	262752061	290607194
執行時間	39.08	29.02	22.83	25.85	27.47
nps	7.6	9.4	10.1	10.2	10.2
分割點數	372383	118937	37285	15716	9963

表 4-3：改良方法二的實驗結果

由實驗結果得知，將其控制在“第 4 層”才允許做 DTS 搜尋演算法的分割所得到的總搜尋節點數是最少的，而分割點數則是從第 2 層到第 7 層一路下降，如圖 4-3、4-4 所示。

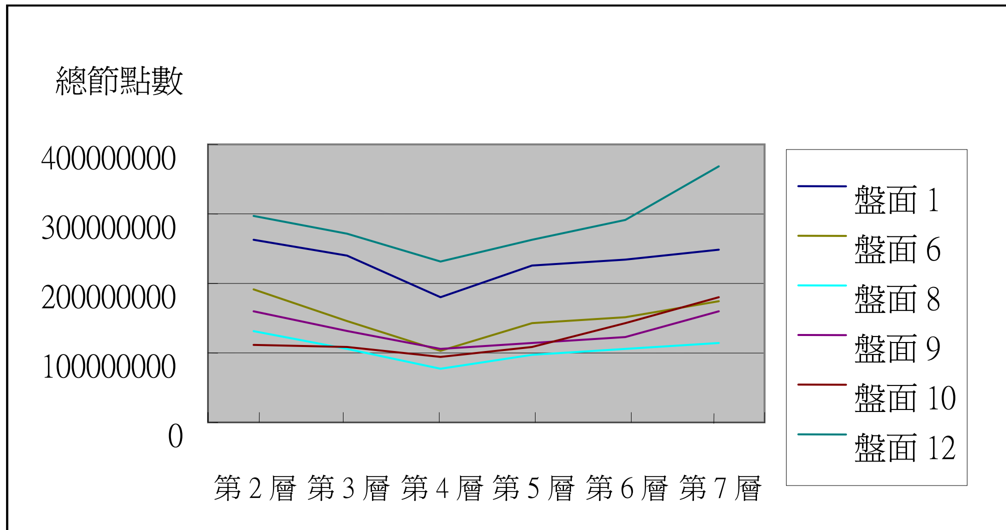

圖 4-4：改良方法二其中盤面的總節點數示意圖

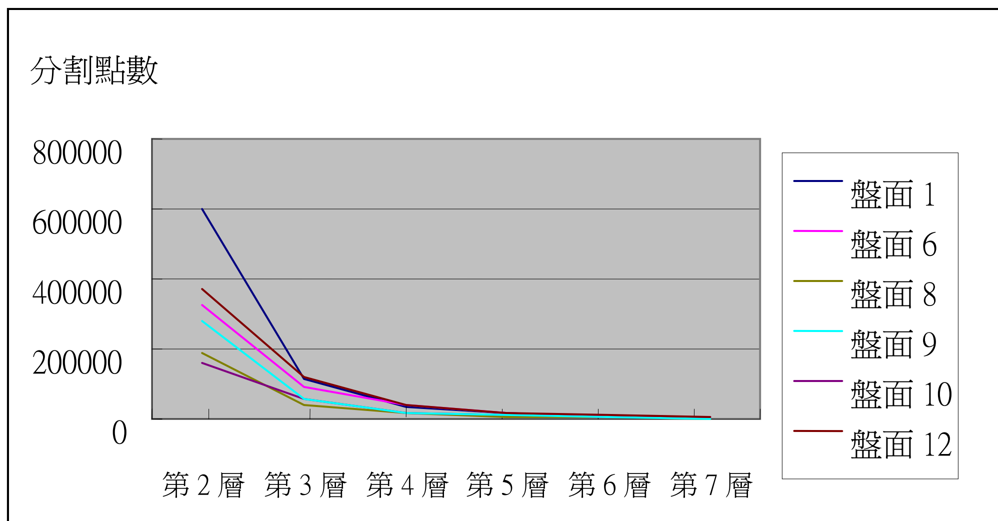

圖 4-5：改良方法二其中盤面的分割點數示意圖

同第四章第一節，為了使圖 4-4、4-5 的曲線起來較明顯，我們依然從 12 個盤面中，只取總節點數相近的 6 個盤面在圖 4-4、4-5 中表示。

雖然要是拉高到第 5、6 層，分割點數會減少更多，但是整體搜尋節點數又開始額外增加。這邊的整體搜尋節點數額外增加原因是因為層數拉高，則在越高的層數我們的期望搜尋範圍（Window）還沒達到縮小的情形，就像是第三章的

圖 3-1 與圖 3-2 所描述的情形，我們一開始都是使用大範圍的期望搜尋範圍（Window）去做搜尋，所以相對的整體搜尋節點就會增加。

我們將程式控制在第 4 層才允許做 DTS 搜尋演算法，與先前 Crafty 20.14 預設是在第 3 層做比較，發現我們的平均效能又提升 31%左右。

	控制在第 3 層的執行時間（T1）	控制在第 4 層的執行時間（T2）	提升效能（T1 / T2 %）
盤面 1	**26.64**	**19.13**	**139.258%**
盤面 2	**45.58**	**38.97**	**116.962%**
盤面 3	**34.69**	**25.16**	**137.878%**
盤面 4	**45.08**	**34.52**	**130.591%**
盤面 5	**67**	**49.03**	**136.663%**
盤面 6	**15.63**	**10.37**	**150.723%**
盤面 7	**40.76**	**34.69**	**117.498%**
盤面 8	**10.29**	**7.11**	**144.726%**
盤面 9	**14.71**	**11.75**	**125.191%**
盤面 10	**12.99**	**9.59**	**135.454%**
盤面 11	**46.32**	**38.36**	**120.751%**
盤面 12	**29.02**	**22.83**	**127.113%**
		平均為 131.817%	

表 4-4：改良方法二的提升效能