國立交通大學

多媒體工程研究所

碩士論文

適用於象棋開局庫之工作層級極小極大化搜尋

Job-Level Minmax Search for Chinese Chess Opening Book

研究生：甘崇緯

指導教授：吳毅成　教授

中華民國一百年九月

適用於象棋開局庫之工作層級極小極大化搜尋

Job-Level Minmax Search for Chinese Chess Opening Book

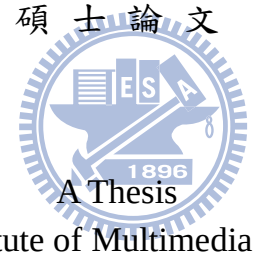
研 究 生：甘崇緯　　Student：Chung-Wei Kan

指導教授：吳毅成　　Advisor：I-Chen Wu

國 立 交 通 大 學

多 媒 體 工 程 研 究 所

碩 士 論 文



A Thesis

Submitted to Institute of Multimedia and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

September 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年九月

適用於象棋開局庫之工作層級極小極大化搜尋

研究生：甘崇緯　　指導教授：吳毅成

國立交通大學 多媒體工程研究所

摘要

本論文的目的在於找到一個快速驗證開局庫是否適合 AI 程式的方式，使得程式編寫者在改變開局庫資料及 AI 程式時，不用擔心需要花許多時間在銜接開局庫與 AI 程式的搭配上。我們進一步提出幾種可增進效能的策略，並且實驗比較何種策略在驗證開局庫上能增進較多的效能。

根據我們的實驗顯示，本論文提出的 job-level SSS* opening 確實能更快速驗證象棋開局庫，且更新所有父節點跟有限度的中止工作能更進一步增進效能，達到本論文的目的。

Job-Level Minmax Search for Chinese Chess Opening Book

Student: Chung-Wei Kan Advisor: I-Chen Wu

Institute of Multimedia and Engineering

National Chiao Tung University

Abstract

The main goal of this thesis is to find a quick way to verify if an opening is suitable for an AI program or not. The programmer would spend less time on changing the opening data and the AI program while merging opening into the AI program. Furthermore, we propose some strategies that could enhance the effectiveness and find the one of which benefits much. According to our experiment, this thesis showed that the job-level SSS* opening method performs more quickly than other methods.

誌謝

首先感謝指導教授吳毅成老師這兩年的諄諄教誨，在學業方面給我指點迷津，並在我遇到困惑時，不厭其煩地與我共同討論。因為有老師的教導與督促，導引我走向正確的研究方向，我才能順利完成論文。同時也感謝許舜欽教授、林順喜教授以及陳志昌教授在百忙之中抽空給我指導和討論，以完整我的研究。對於您們的教誨，學生在此表達萬分的感謝。

在論文研究期間，非常感謝鈊象電子股份有限公司贊助獎學金，使我在這段研究期間，不用分心擔心零用金，讓我能更加專心地投入研究，積極努力地完成論文。

再來要感謝實驗室的同學，同甘共苦楊景元、新好男人林正宏、智慧導師蔡心迪、強大支援陳昱維、有求必應韓尚餘、林修全，還有無私提攜的學長姐孫德中、林秉宏、林宏軒、林沂珊、曾汶傑、陳柏廷與象棋組賴隆億、陳建宇、郭青樺、林柏翰以及各位強者學弟妹，陳干越、康皓華、劉浩雲、左存道、胡嘉芸、鄭吉閎等等，在課業上遇到問題時，與我一起討論切磋，大家互相砥礪加油。

最後要感謝我摯愛的家人及愛人，讓我沒有後顧之憂的完成學業，總是支持著我，我愛你們。

民國一百年九月 於 新竹市交通大學工程三館 IAT 實驗室

目錄

圖目錄

表目錄

第一章　緒論

中國象棋是華人世界中最熱門的棋類遊戲之一，從中國宋朝開始確立發展至今已 800 多年[24]。可謂博大精深的棋類遊戲，本論文針對快速驗證象棋開局庫做相關研究探討。

首先在 1.1 節中將簡單介紹象棋及規則；於 1.2 節說明電腦象棋的發展；1.3 節為本文的研究動機與目標；1.4 節講述本文貢獻；1.5 節本章節闡述本論文大綱。

1.1　象棋介紹及規則

象棋，是中國傳統的二人對弈棋類遊戲。現今象棋規則確定於南宋時期，至今發展已 800 多年。其他類似的有朝鮮將棋及日本將棋。為與國際象棋（西洋棋）等區別，又稱中國象棋。

象棋分紅方及黑方，紅方先手。雙方共有 16 隻棋子，分別為一隻帥(將)、兩隻仕（士)、兩隻相（象)、兩隻俥（車)、兩隻傌（馬)、兩隻炮（包)、五隻兵（卒)，各兵種走法不一，遊戲的勝負在於先捕捉到對方帥（將）者為勝。

1.2　電腦象棋的發展

電腦象棋的研究從 1980 年左右起步，即開始陸續舉行人腦對電腦象棋比賽和電腦對電腦的象棋比賽，象棋程式也開始參加升段賽[25]，蓬勃發展至今，現已有多項定期舉辦的電腦象棋比賽，部分列舉如下：

1. 電腦奧林匹亞(1989-至今)
2. 全球電腦象棋爭霸賽(2004、2007、2010-至今)
3. TAAI 人工智慧技術與應用會議(2010 年與全球電腦象棋爭霸賽合辦)
4. TCGA 電腦對局比賽(2011-至今)
5. 中國機器博弈錦標賽(2006-至今)

而人腦與電腦的比賽也有從 1999 年開始舉辦的人腦對電腦象棋大賽，至 2010 年在交通大學舉辦時，已邁向第十一屆。

1.3 研究動機

本實驗室也積極參加各種電腦象棋競賽，在比賽中一直在累積經驗，繼續加強我們的程式和開局庫。尤其在 2010 年及 2011 年的各項比賽中，發現我們雖然將開局庫資料擴大增加到 68 萬多筆來作統計分析，卻仍然出現一脫離開局及陷入不利棋局，表示我們的電腦程式在開局庫的選點上，不能只憑藉統計資料，最主要的是必須配合使用的電腦程式。

在以往對這方面的研究並沒有太多，而在 2005 年陳志昌老師的博士論文「電腦象棋知識庫系統之研製」中有提出改善的方式[1]，效果良好但計算量龐大，於是本篇論文主要在探討如何快速驗證適合電腦程式的開局庫。

1.4 主要貢獻

此篇論文的主要貢獻為提出適用於驗證開局庫的平行化 SSS*演算法[20][21]，可快速驗證開局庫。並比較 SSS*與 Alpha-Beta[12]何種較適合驗

證開局庫。同時探討加速本演算法的各項策略，實驗分析優劣。

1.5 論文大綱

本論文第一章緒論介紹象棋及電腦象棋的發展，並且說明研究動機。第二章講述研究背景，包含開局庫的製作、開局統計與中局審局一致性策略、SSS*演算法、本實驗室開發的 CGDG 系統[2][3][5][6]，而開局統計與中局審局一致性策略是陳志昌老師提出的避免走至不利開局的方式，本篇可視為此方式的改進；第三章闡述本論文提出的演算法的設計，說明如何將 SSS*演算法使用在開局庫上，並說明如何進行平行化，討論 Abort Job 的時機，以及提出因應開局樹特性的增進效能的策略；在第四章將本論文提出的方法進行相關實驗，獲得實驗數據並作探討。最後，第五章提出總結以及未來可發展、研究的方向。

第二章　研究背景

2.1　開局庫

開局，為棋類遊戲剛開始的幾回合，佈署棋子的過程。而優良的開局，會被專家編寫成開局譜，或為隨著經驗的累積，由高段棋士所熟用。而這些優良的開局譜整理起來，即為開局庫，常為電腦 AI 程式使用。

2.1.1　象棋開局

象棋的開局，主要目的為調動主力棋子，組成陣形來互相掩護或搭配攻擊。而開局時對於部份對手的布局，會有特別幾種固定的應著，若不下這些應著，雖然可能暫時沒有危機，但容易在步入中局時遇上對方安排的陷阱，導致於陷入不利棋局。因此，掌握開局的知識是象棋的重要課題。

而象棋歷史淵遠流長，至今已 800 多年[24]，不論是專家研究出的開局譜，還是前人的經驗累積的心得，都讓象棋開局博大精深。

而電腦象棋若能具備開局庫，不但有助於避免陷入開局陷阱，更可以吸取 800 多年的象棋經驗，壯大電腦象棋的開局能力。

2.1.2　電腦象棋開局庫

有賴現今網路的發展，帶動了網路棋類遊戲的普及，也因此在網路上留下了大量的棋譜。若從中擷取高段棋手的棋譜，將有助於快速建立開局

庫。自今本實驗室已收集了 68 萬多局網路象棋高手對戰棋譜，並且加以統計分析各盤面的勝敗和等資料。加上陳志昌教授在「電腦象棋知識庫系統之研製」提出的強弱連結的觀念[1]，大大提升了本開局庫的實力。

2.2 開局統計與中局審局一致性策略

為了解決一脫離開局即陷入不利盤面，陳老師提出了要以中局審局分數作為開局庫選點考量的重要依據[1]。因為陳老師發現，AI 程式並未被告知此開局譜布局的「計畫」，所以當一脫譜，必須由 AI 進行中局搜尋引擎審局計算時，AI 有可能會走偏離於原本的「計畫」，而調動棋子成 AI 適合的盤面。而如此雖然對 AI 較好，但也失去了開局安排好的先機，容易讓對手有機可趁。

而開局統計與中局審局一致性策略的作法是，會將開局庫的每個節點預留一個存記中局分數的欄位；再使用門檻值挑出葉節點(Leaf Node)，將每個葉節點做搜尋審局評分；若是評分分數與統計資料一致，就進行反向的 Minmax 更新，如此，就可以避免開局時走到 AI 不擅長的盤面，達到目的。

但是如此會有個缺點，就是當我們的開局庫資料作更新，造成葉節點與原來不同、統計資料也與原本不一；或是 AI 程式作更新，造成同一個節點進行搜尋審局評分時回傳分數與原本不一致，都使得要將部分或全部葉節點的重新進行搜尋審局評分，將是個很大的計算量。

2.3 AB-SSS*

AB-SSS*演算法[15][16][17]是 SSS*演算法[20][21]的改寫，也改進了一

些 SSS*的缺點，並且把架構建立在簡單易懂的 Alpha-Beta Search 演算法上[12]，而 Alpha-Beta Search 為 Minmax Search 的改良[14]，因此在介紹 AB-SSS*前，要先來介紹 Minmax Search、Alpha-Beta、SSS*。

2.3.1 Minmax Search

圖 1 是一個 Mini-Max Search Tree 的例子，在 Mini-Max Search Tree 中有兩種節點(Node)，一種是方型的 Max Node，一種是圓形的 Min Node。每個節點上的數字為我方的分數，分數越高對我方越有利，分數的更新是從葉節點（Leaf node）往上更新，而葉節點的分數可由評估函數得到。至於更新的方式，Max Node 可以當作是我方，我方一定會選對我方最有利的走法，因此會選分數最大的子節點（Children Node），更新成自己的分數，而 Min Node 則可以當作是對方，對方一定會選對我方最不利的走法，因此會選分數最小的子節點，更新成自己的分數。最後更新完，每個節點的分數，就代表若進行到這個節點，雙方都是最強時結果會是多少分數，而圖 1 中的粗紅線，代表雙方都走最好時形成的路線。

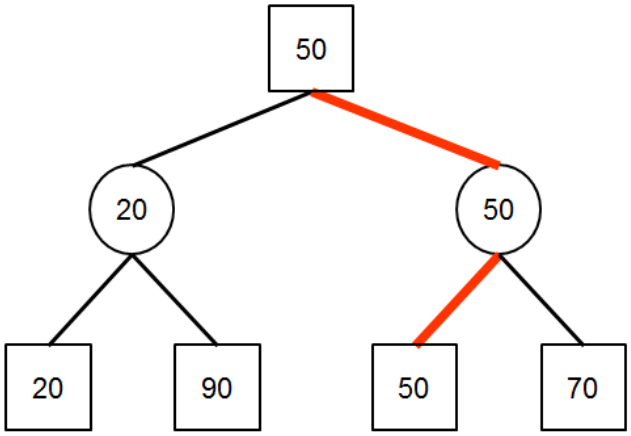


圖 1. Minmax Search Tree

2.3.2 Alpha-Beta Search

Alpha-Beta Search 是 Mini-Max Search 的一種改良，我們的目的是為了得到根節點（Root Node）的分數，才能知道從根節點開始，要選擇何種路徑走。而原本的 Minmax Search，有許多地方可以不需要搜尋就得到根節點的分數。舉圖 2 為例，節點 B 是一個 Min Node，已經找完第一個子節點得知其分數是 10 分，這時候可以得知 B 的分數一定是小於等於 10 分。而節點 A 是一個 Max Node，已經找完第一個子節點得知其分數是 15 分。則在第一個子節點 15 分和第二個子節點 （節點 B）小於等於 10 分之間，A 一定會選擇第一個子節點，因此節點 B 確切的分數是多少已經不重要，B 剩下的子節點就可以修剪（Prune）不做搜尋。

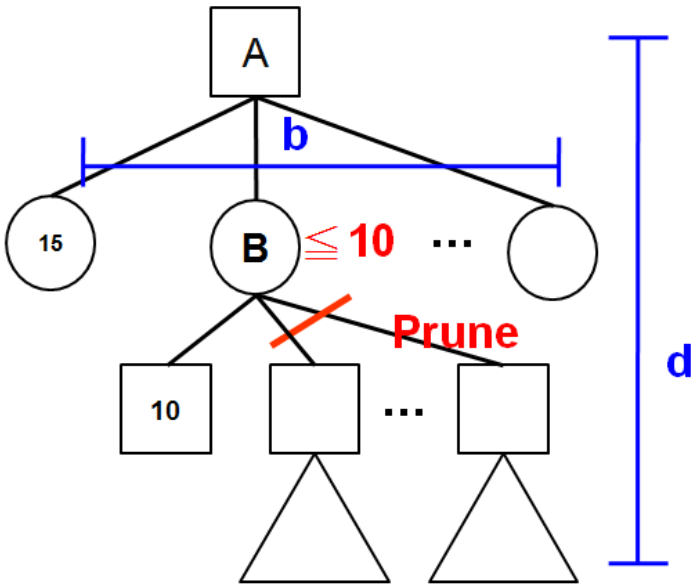


圖 2. Alpha-Beta Search

類似的方式可以砍掉許多子樹（Subtree），節省許多搜尋時間。若 b 為搜尋樹寬度，d 為搜尋樹高度，在 best case 時，甚至可以將原本的時間

複雜度從 $O(b^d)$改良至 $O(b^{d/2})$[22]。因此跟原本的 Minmax Search 比，效率大幅提升。而本論文使用的 Alpha-Beta Search 為效率較傳統 Fail-Hard 版本高的 Fail-Soft 版本[9]。

2.3.3 SSS*

SSS*是一個 Best-First Search 演算法，理論上來說比起 Alpha-Beta Search，SSS*可以搜尋較少的 node 數即可獲得 Alpha-Beta Search 所獲得的 Minmax 值。但是他有許多的缺點，演算法複雜又難懂，在使用上效能比起 Alpha-Beta Search 低很多，也耗費相當多記憶體，還必須維護一個有序的 OPEN 串列，總結 SSS*的話，有這五點：

1. 演算法複雜難懂。
2. 實際運用上需要大量的記憶體。
3. 效能”低”，因為要維護一個有序的 OPEN 串列。
4. 被證明需要搜尋的葉節點量小於等於 Alpha-Beta。
5. 在模擬實驗中在也顯示需要搜尋的葉節點量比 Alpha-Beta 少。

也因為效能低的這個主因，讓 SSS*演算法並不常見，被提出後一直未受重視，而下一節提出了改善 SSS*的方法，AB-SSS*。

2.3.4 AB-SSS*

AB-SSS*是一個 SSS*演算法的改寫，使用一連續的呼叫零窗口（Zero

Window）Alpha-Beta Search 搭配換位表（Transposition Table）來達成。

AB-SSS* 演算法（C-Like）如下：

```
function AB-SSS*()
{
    dG = + ∞ ;
    do{
        dR = dG;
        dG = AlphaBeta(dR-1, dR);
    } while (dR != dG);
    return dG;
}
```

根據 Fail-Soft 的概念，當在 Fail-High 發生時，回傳值為該節點的下界；當在 Fail-Low 發生時，回傳值為該節點的上界[7][8]。有了這個概念後，我們在看以下的圖解，來了解 AB-SSS*的運作。

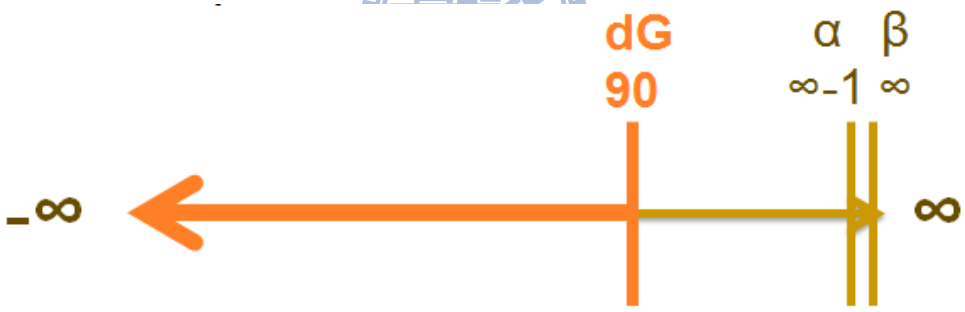


圖 3. AB-SSS*演算法示意圖-(Pass1)

當在 Pass1 時，呼叫 Alpha-Beta Search 時使用的零窗口為 (∞-1, ∞)，因為不可能有值大於∞，所以只可能發生 Fail-Low，而 Fail-Low 代表回傳值為該節點的上界，當該節點為根節點時，即為此根節點分數的上界。如圖 3，回傳值 dG 為 90，代表根節點的分數上界為 90，換句話說，之後出現的分數只可能≦90。

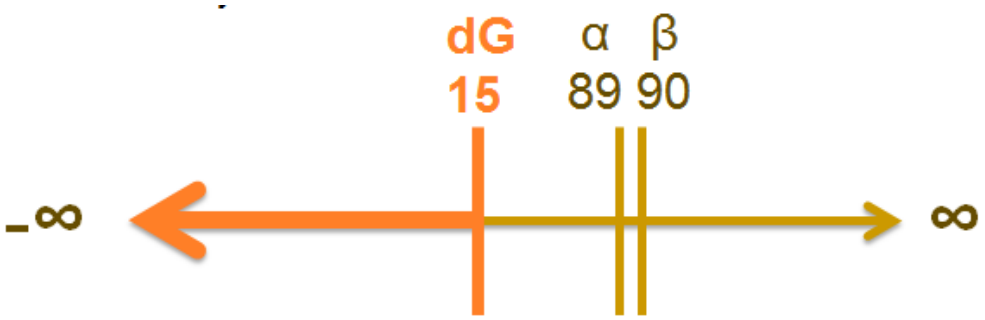


圖 4. AB-SSS*演算法示意圖-(Pass2)

在 Pass2 時，就根據之前的 dG 值，再形成零窗口呼叫 Alpha-Beta Search，如此得到下一個 dG 值，就可再度更新根節點分數的上界。如圖 4，根節點上界被更新為 15。就一直如此迭代下去，當回傳分數 dG 跟之前的 dG 值一樣時，此時發生 Fail-High（15 ≧ β），回傳值為下界，此時上界等於下界，即為 Minmax 值，如圖 5。

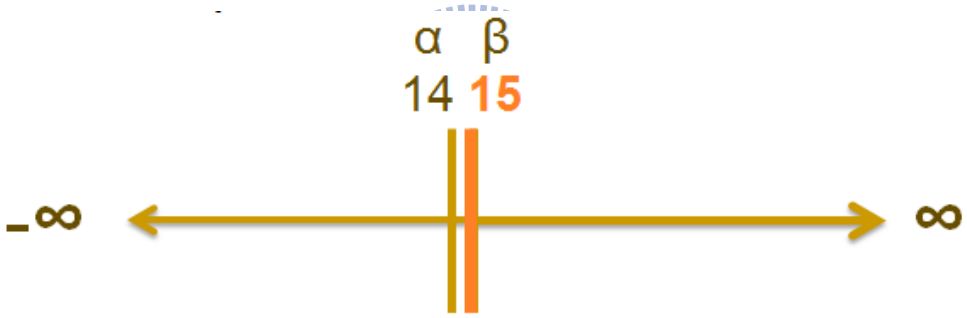


圖 5. AB-SSS*演算法示意圖-(Pass3)

而從−∞方向開始搜尋的，稱為 AB-DUAL*，演算法（C-Like）如下：

```
function AB-DUAL*()
{
    dG = − ∞ ;
    do{
        dR = dG;
        dG = AlphaBeta(dR, dR+1);
    } while (dR != dG);
    return dG;
}
```

其執行過程就和 SSS*類似，只是 dG 值為隨著每次的迭代越來越大，直到找到 Minmax 值，在這就不詳述，僅以圖 6～圖 8 表示過程。

圖 6. AB-DUAL*演算法示意圖-(Pass1)

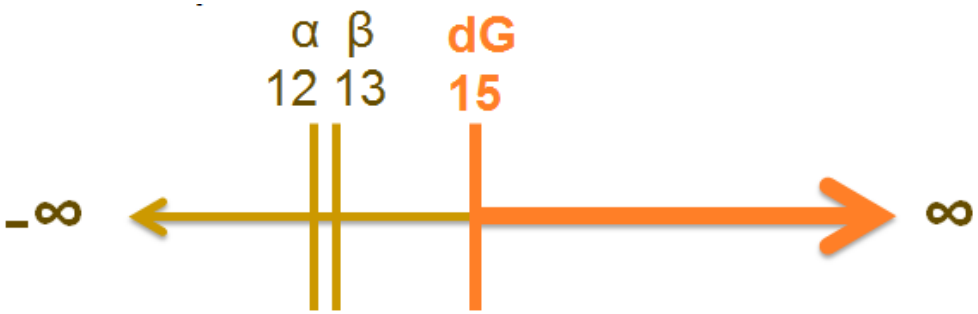


圖 7. AB-DUAL*演算法示意圖-(Pass2)

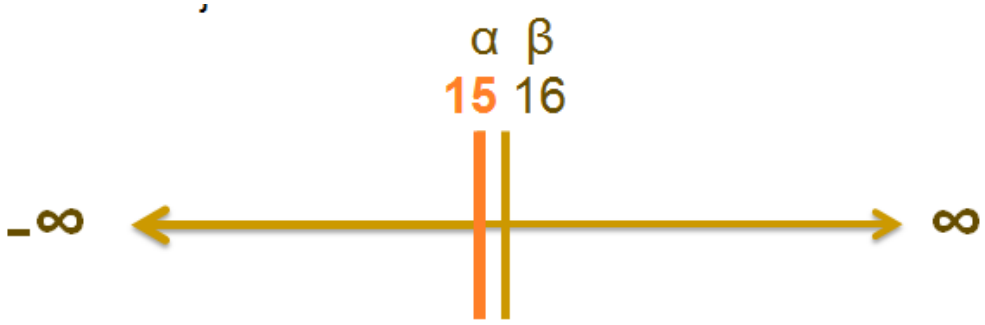


圖 8. AB-DUAL*演算法示意圖-(Pass3)

2.4 CGDG

全名為 Computer Game Desktop Grid[2][3][5][6]，中文為電腦遊戲桌機網格系統。為本實驗室開發的志願型計算系統，透過網路整合許多閒置的計算資源，來為電腦遊戲做運算。

2.4.1 系統架構

CGDG 系統有三個角色：

1. 使用者（User）。
2. 仲介者（Broker）。
3. 工作者（Worker）。

CGDG 系統的使用者（User），主要功能為發送工作（Jobs），例如將 AI 運算的工作告知給系統的中樞，仲介者（Broker），仲介者會分配工作至後端的運算資源，也就是工作者（Worker），工作者運算結束後，將結果回傳給仲介者，仲介者再將結果傳回給發出此工作的使用者。如圖 9。

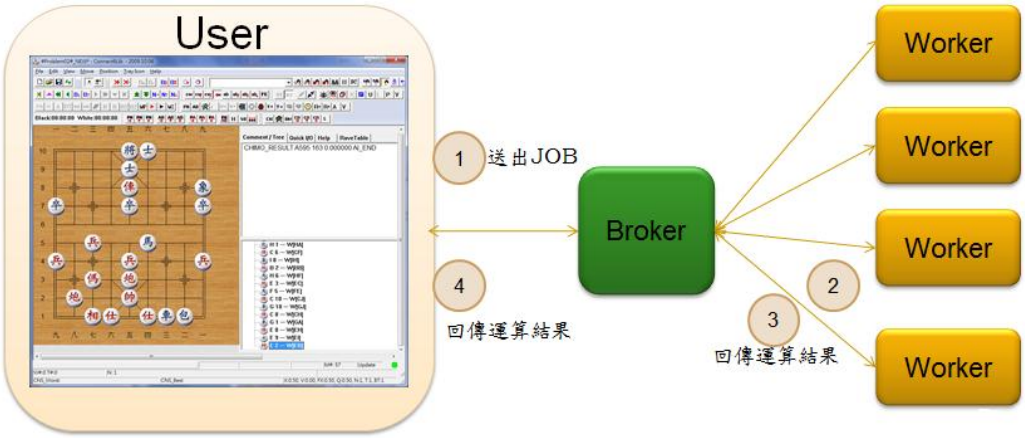


圖 9. CGDG 系統組織架構-

CGDG 系統也同時支援多個使用者，像是圍棋、象棋、六子棋等運算同時進行，仲介者將會以他們的身分，適當地為他們分配運算資源。

2.4.2 工作層級

工作層級（Job-Level）的概念在於將程式中會用到的運算作為一個工作（Job）[23]，並將該運算抽離原程式，單獨成為另外一個運算程式；當原程式需要用到該運算時，將會呼叫該運算程式起來執行，並回收該程式運算的結果。由於將常用到的運算部份獨立出來，因此當程式需要同時執行該運算時，就可以平行化執行。

第三章　設計方法

在這章節，將會說明為了達到快速驗證開局庫，我的設計方法。將各個部分的設計方法在各小節做介紹。3.1 節將簡介我實作的一個配合 CGDG 系統[2][3][5][6]的平行化驗證開局庫演算法－適用於開局庫之工作層級 SSS*[20][21]搜尋（job-level SSS* opening）（簡稱 JL-SSS* opening）。3.2 節將說明我為何採用其設計方法，包括如何使 SSS*達到平行化送出工作（Job）的選點策略、遇到相同盤面會出現的問題等等。特別一提的是，在本章節提及的 SSS*，其在實作上內容皆為 AB-DUAL*[15][16][17]，為了減少冗長詞句和代表性問題，在命名上以 SSS*代表 AB-DUAL*的概念。

3.1　JL-SSS* Opening 介紹

為了要達到快速驗證象棋開局庫的目的，我們提出了 job-level SSS* opening 演算法，他有以下的特性：

1. 一種平行化的 SSS*。
2. 將開局庫製作成的開局樹當作 SSS*的搜尋樹。
3. 可同時選定多個檢驗節點。
4. 可較 Alpha-Beta[12]更快得到該開局庫 Minmax 的分數。
5. 實作上以 AB-DUAL*代替複雜的 SSS*。

而之後的章節，將會針對這些特性，說明原因。

3.2 使用 SSS*的優勢

本節將在 3.2.1 介紹我們為何使用 SSS*，並且在 3.2.2 進一步說明為何實作上我們以 AB-DUAL*代替 SSS*。

3.2.1 使用 SSS*作為選點演算法

使用 SSS*演算法作為選點演算法，原因如下：

1. SSS*比起 Alpha-Beta，需要拜訪的節點較少。
2. 開局庫驗證必要儲存所有數據（Data），SSS*的缺點消失。

但即便如此，要實作 SSS*依然存在著 SSS*演算法繁瑣複雜的缺點。因此，我們將採舉 AB-DUAL*。

3.2.2 實作上使用 AB-DUAL*

先說明為何使用 AB-DUAL*之前，先解釋 DUAL*的特性，再說明為了使用 AB-DUAL*。

3.2.2.1 DUAL*的特性

因為 DUAL*是從$(-\infty, -\infty+1)$開始做 Alpha-Beta 搜尋，因此在根節點只可能發生 Fail-High（在根節點找到 Minmax 值之前）；換句話說，每次在根節點獲得的分數，皆為根節點分數的下界，也就是根節點至少的分數。

而本篇論文的目的是要快速驗證該開局庫對該 AI 的配合度，換句話說，也就是 Minmax 的分數，決定了該開局庫對該 AI 配合度的高低，若在幾個 Pass 之後，根節點的分數已經達到我們認可的配合程度，也就是達到某個門檻值之後，若我們時間有限，即可將對該開局庫的搜尋暫停，因為我們可以認可該開局庫夠好了，並不一定要真的獲得 Minmax 值。

3.2.2.2 AB-DUAL*的好處

使用 AB-DUAL*的原因就是，因為它是 DUAL*的改寫，程式簡單易懂又好寫，於是在實作上，都採用 AB-DUAL*。

3.3 平行化的選點方式

JL-SSS* opening 是將開局樹當作搜尋樹，來進行選取檢驗節點的動作。而 JL-SSS* opening 是使用 AB-DUAL*，在 AB-DUAL*中，是使用零窗口（Zero Window）的 Alpha-Beta Search 來進行，所以選點理所當然是根據 Alpha-Beta Search 時的 Alpha-Beta 值來選點，如圖 10。

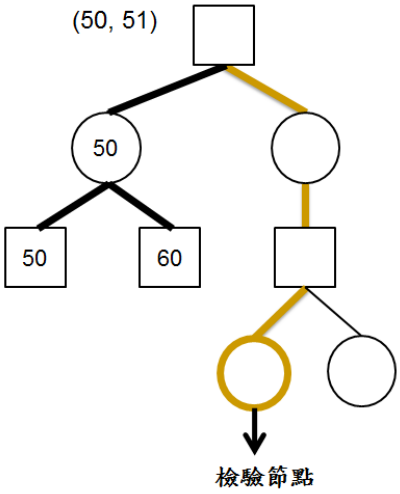


圖 10. 平行化選點方式-1

但若光是如此選點，會出現一個問題，那就是當我們想要再選一個檢驗節點，根據一樣的窗口（Window），會選到同一個點。

為了解決上述的現象，我在 3.3.1 提出一個 Virtual Fail 的概念，來達成平行化選點方式。而我在使用 SSS*演算法時，觀察到 Pass 1 有個特性，可以幫助我們在 Pass 1 時達到很好的平行化效率，所以 3.3.2 將介紹 SSS*演算法 Pass 1 的特性。

3.3.1 Virtual Fail

Virtual Fail 是指對於我們選到的檢驗節點，再送出工作（Job）時，假設此檢驗節點回傳值不會造成修剪（Prune），於是下次選點時，可以拜訪該節點周邊的節點。圖 11 為設定此檢驗節點為 Virtual Fail 後，可順利選到下一點的範例。

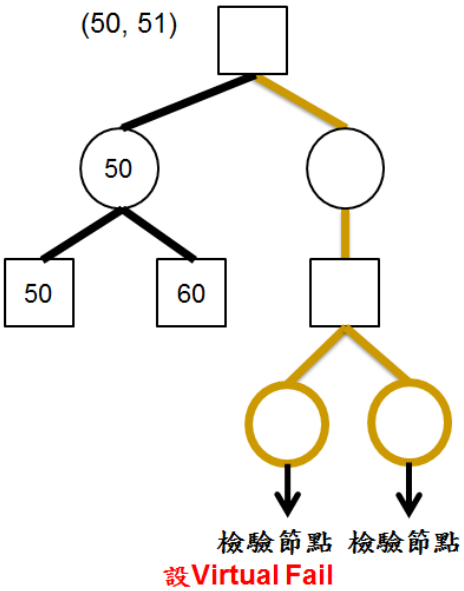


圖 11. 平行化選點方式-2

3.3.2 SSS* Pass 1 的特性

在 SSS*中，Pass 1 有個特性，就是 Pass 1 時，走點策略永遠是我方全走，但對方只走第一步。而 DUAL*剛好相反，Pass 1 時，走點策略永遠是我方走第一步，對方全走。如圖 12 所示。

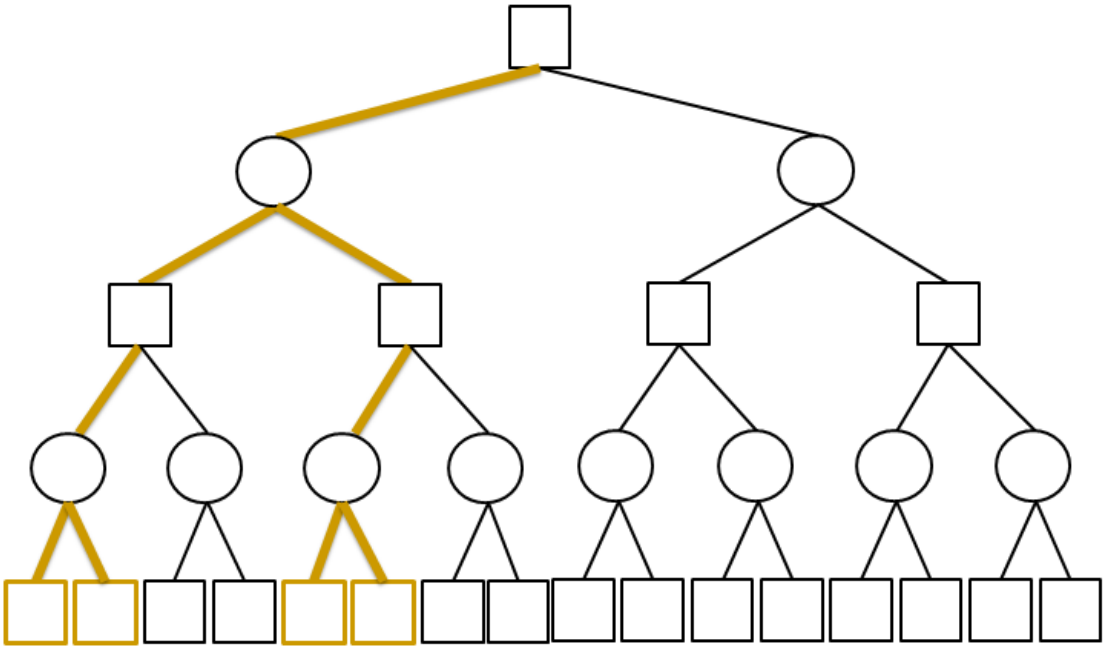


圖 12. AB-DUAL*在 Pass 1 所選擇的走點

所以在 Pass 1 的選點，是不會因為檢驗節點回傳的分數而做任何改變，因此在 Pass 1 時，我們是先 Pass 1 需要被計算的節點全記錄下來，再去做運算，等到 Pass 1 所有的節點被運算結束，Minmax 反向更新好根節點的分數後，再開始使用新的零窗口（Zero Window），搭配 Virtual Fail 進行 Pass 2 的搜尋。

3.4 開局樹獨立盤面的問題

本篇論文所使用的開局樹，每個節點，皆為獨立的盤面，也就是說，所有節點，不會出現重複盤面的狀況。換句話說，這不是一顆真正的樹，而是一顆子節點可能擁有多個父節點的樹。而一個盤面，會有多種不同的走法所形成，如圖 13(A)，為經過不同路徑，而選到同一個檢驗節點。所以會產生的問題，就是當經由一條路徑，對一個節點進行處理時，可能會影響另一條路徑。

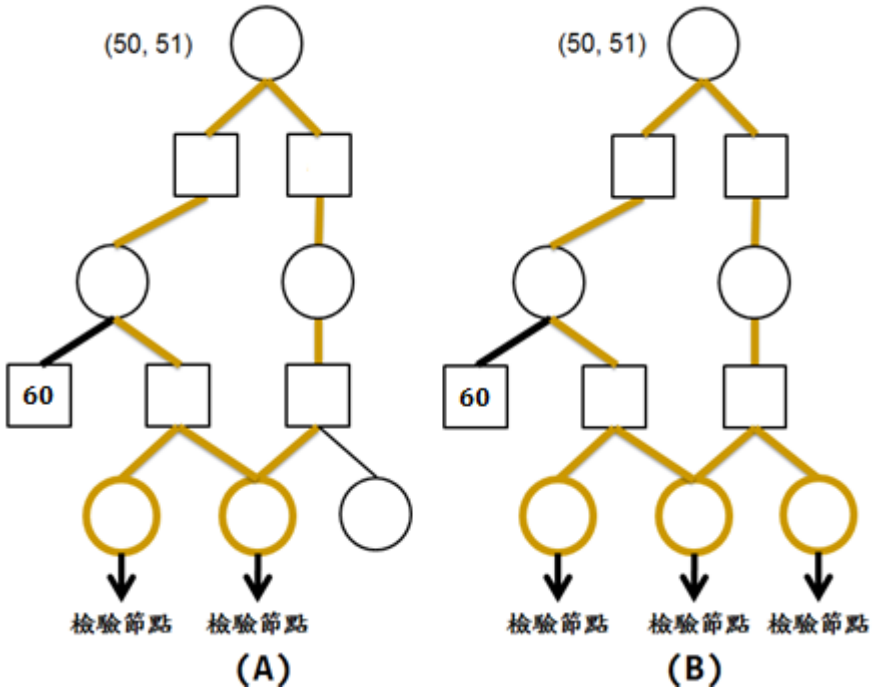


圖 13. 開局樹獨立盤面問題

當選到同一個檢驗節點時，很直接地採取策略就是，因為有設 Virtual Fail，所以自然會去改選其他的檢驗節點，如圖 13(B)。若是選到的檢驗節點，是已經驗證過回傳評估分數了，就直接使用該評估分數即可。關於其他問題和改進法，將在 3.6 節和 3.7 節提及。

特別一提的，是本開局樹並沒有迴圈問題，我們使用的方式是：我們會記下每個盤面從初始盤面到達的最少步數 m，若 A 盤面可走至 B 盤面，則 A 盤面 m 必小於 B 盤面 m。如此避免迴圈問題。

那也因為本論文使用的開局樹沒有迴圈，所以並沒有使用開局庫即可利用棋規求和的能力。

3.5 修剪的特性

在討論中止工作（Abort Job）時，我們要先了解修剪（Prune），特別是修剪在 AB-DUAL*裡的現象。於是 3.5.1 節將先介紹一般 Alpha-Beta Search 裡的修剪，3.5.2 節將介紹在 AB-DUAL*中，修剪的意義。而了解修剪之後，我們在 3.6 節討論在甚麼修剪條件下，我們應該執行 Abort Job。

3.5.1 Alpha-Beta Search 的修剪

一般的 Alpha-Beta Search，執行修剪的時候代表被修剪節點不需要再被計算，所以再也不會被拜訪。

而如果搜尋的是本論文探討的開局樹，將會從不同路徑拜訪到同一個節點，但每次拜訪時關心的窗口（Window）（也就是 α、β 值)可能不一樣，若是不一樣，那勢必要重新往下搜尋，以更新使用該窗口所得的分數及會被修剪的節點。

3.5.2 AB-DUAL*的修剪

AB-DUAL*會以不同的零窗口（Zero Window）多次從根節點（Root）開始搜尋樹。從 Pass1，窗口(−∞, −∞+1)開始，零窗口值只會越來越高，而此一特性，不論是一般的搜尋樹，還是搜尋的是本論文探討的開局樹，都有搜尋時，修剪過的節點，還是可能再被拜訪到的現象，以下將舉例子說明。

假設在 AB-DUAL*的某次 Pass n，零窗口為(50, 51)，這時在 Min Node 若是 Fail Low[7][8]，將會造成修剪，如圖 14(A1)，Min Node A 有一步獲得 20 分，20 ≦ 50，造成修剪；在 Max Node 若是 Fail High，將會造成修剪，如圖 14(B1)，Max Node B 有一步獲得 60 分，60 ≧ 51，造成修剪。而在下次 Pass n+1，零窗口升高到(80, 81)，這時在同一個 Min Node 必定依然 Fail Low，維持修剪的狀態，如圖 14(A2) Min Node A 有一步獲得 20 分，20 ≦ 80，維持修剪的狀態；而這時在同一個 Max Node，則不一定 Fail High，所以不一定造成修剪，如圖 14(B2) ，Max Node B 有一步可走到 60 分，60 不≧ 81，不造成修剪，改變了修剪的狀態。

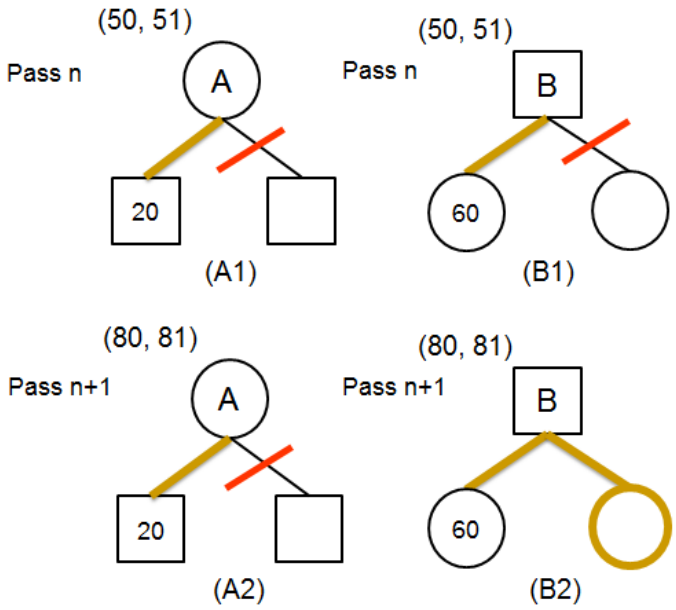


圖 14. AB-DUAL* 修剪的例子

由此可以看出，在 AB-DUAL*之下，Min Node 修剪的狀態，是不會再被改變的，因為零窗口只會越來越高，也就是說每次遇到的 α 值只會越來越大，若是 Min Node 曾經有一步可以獲得 $\leqq \alpha$ 的分數，造成 Prune，則在此次 AB-DUAL*搜尋裡，此 Min Node 對其子節點修剪的狀態不會改變。相反的，Max Node 對其子節點的狀態就可能被改變，即使 Max Node 曾經有一步可以獲得 $\geqq \beta$ 的分數，隨著 β 值只會越來越大，可能產生此次並不會 $\geqq \beta$，而不造成修剪。而 AB-SSS*情況剛好反過來，Max Node 對其子節點修剪的狀態不會改變；而 Min Node 則不一定。

為了此簡單說明此現象，我們定義了兩個新名詞：

1. **Soft-Prune**: 對其子節點修剪的狀態**可能**再被改變的修剪。
2. **Hard-Prune**: 對其子節點修剪的狀態**不會**再被改變的修剪。

而 Soft-Prune 發生在 AB-SSS*的 Min Node 及 AB-DUAL*的 Max Node；Hard-Prune 發生在 AB-SSS*的 Max Node 及 AB-DUAL*的 Min Node。

3.6 執行中止工作的策略探討

中止工作（Abort Job）的動作，是在使用 CGDG 系統時，將送出的工作（Job）刪除，若該工作已被運算，則可使負責的運算資源空出來。在 3.4 節時我們了解到我們使用的是一個可經由不同路徑走到同一節點的開局樹。因此當我們透過某條路徑對檢驗節點執行中止工作時，可能會影響到另一條路徑。

在送出的檢驗節點中有一點回傳造成樹的更新時，便會判斷已送出的工作是否有可被中止的，以節省運算資源，更快驗證出該開局庫的適合度。而在 3.5 節中已經定義了 Soft-Prune 與 Hard-Prune，將簡化我們訂定各種中止條件時的陳述。我們訂定的 4 個策略如下：

1. 策略 A：發生修剪就中止工作。
2. 策略 B：一律不中止工作。
3. 策略 C：確定不會再被拜訪才中止工作。
4. 策略 D：幾乎不會再被拜訪才中止工作。

在 3.6.1～3.6.4 節裡將會詳述各個策略所使用的條件細節。

3.6.1 策略 A：發生修剪就中止工作

我們訂定的第一個中止工作的策略是，只要發生修剪，就中止工作。仔細講，就是只要當初送出工作的路徑，在某個檢驗節點回傳後，狀態改成被修剪，就中止此工作。這是比較直覺的判斷，但是因為使用的是本論文探討的開局樹，可能會中止的工作會再次被拜訪且送出工作。下面將舉

例子說明。

假設 ABC 是三個已送出工作的檢驗節點，如圖 15(A)所示。檢驗節點 A 先回傳了，為 70 分，經過檢查後，發現 70 會讓 B 節點當初送出運算的路徑被修剪，所以中止工作 B，如圖 15(B)。但當下一次從根節點選點時，卻再次選到節點 B，於是再次送出工作，重新運算，如圖 15(C)(D)。

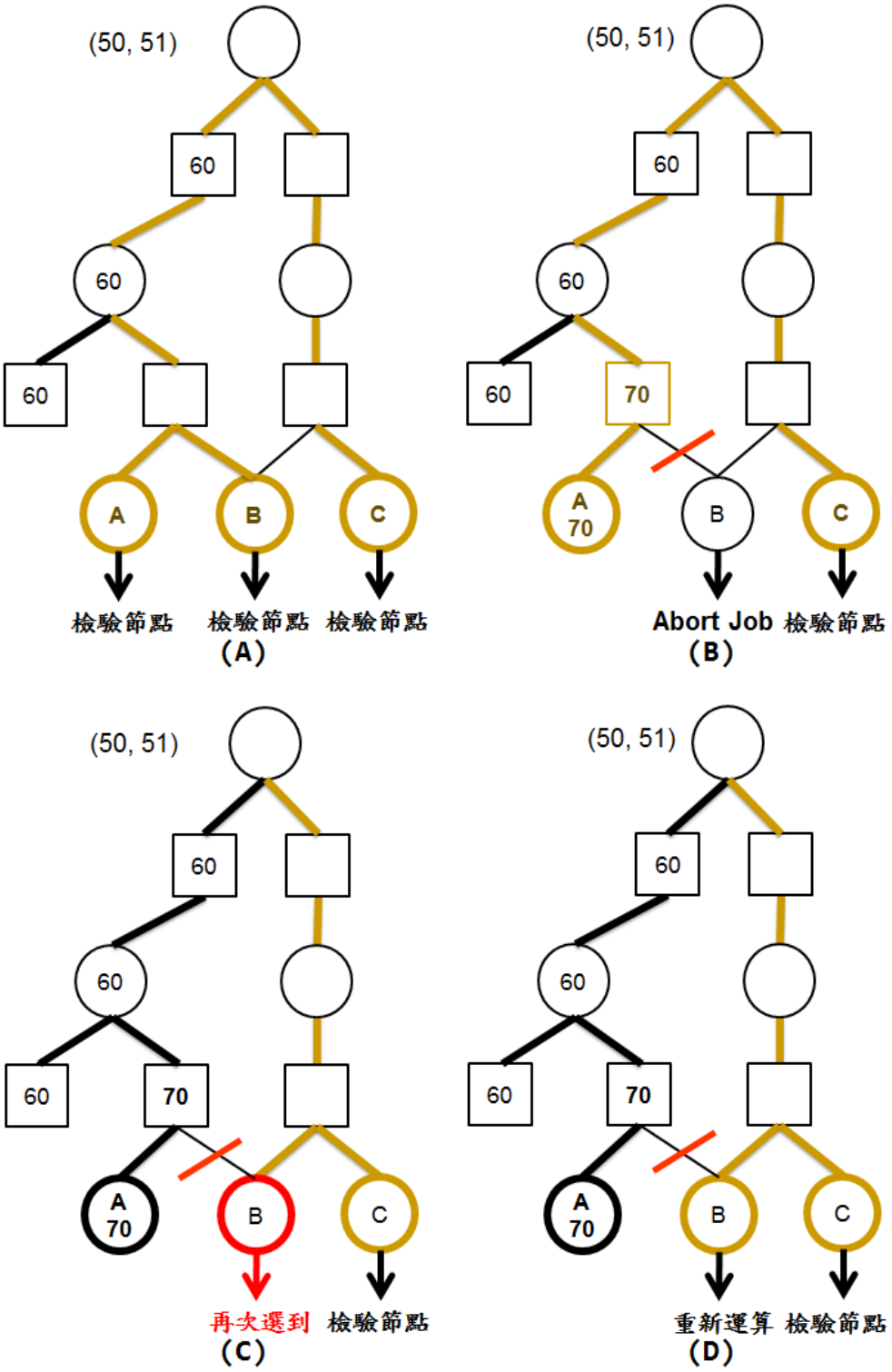


圖 15. 策略 A 發生修剪就中止工作之例子

明顯的，雖然這是一個很直覺的作法，但可能會遇到，已經中止的工作，卻再次被選到而送出檢驗。

3.6.2 策略 B：一律不中止工作

為了解決策略 A 會重複計算到同一個工作（Job）的問題，另一個很直覺的作法，就是一律不中止工作，這樣在設計上方便，也可以解決問題。但是會產生另一個問題，若是檢驗節點的時間很長，如此可能會拖延時間在不可能再被拜訪的檢驗工作上。圖 16 為不中止工作的例子。

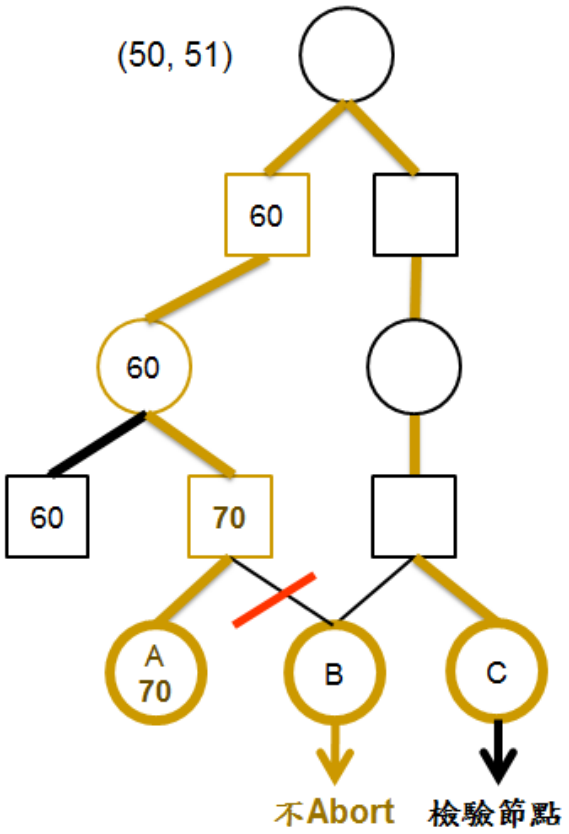


圖 16. 策略 B 不中止工作之例子

3.6.3 策略 C：確定不會再被拜訪才中止工作

為了同時解決策略 A 和策略 B 的問題，於是策略 C 的目標為，想要中止再也不會被拜訪的檢驗節點工作。在 3.5 節已經定義了 Hard-Prune，為確定被修剪的分支不會再被拜訪。於是我們利用 Hard-Prune，來設計策略 C。我們檢查所有從檢驗節點到根節點的路徑，看是否全部的路徑上皆有一個分支是被 Hard-Prune 的，若是全部都有，我們才執行中止工作。以 AB-DUAL*來說，Hard-Prune 就是 Min Node 對於其子節點（Children Node）的修剪。這樣算是解決了被中止的工作再次被送出檢驗的問題，以及會拖延時間在不可能再被拜訪的檢驗工作上這兩大問題。但是如此還是有一個小問題，那就是因為條件太過嚴苛，在效能上，可能無法有太大的提升。圖 17 為在策略 C 下中止工作的例子，此例子特地與之前的例子調換 Min Node 和 Max Node，窗口改以(70, 71)，強調 Min Node 與 Max Node 在修剪上意義的不同。

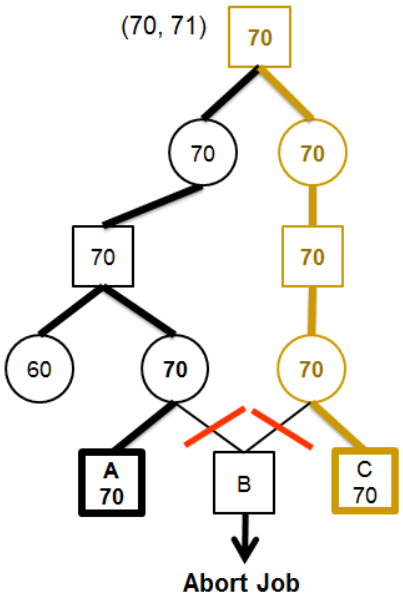


圖 17. 策略 C 確定不會再拜訪才中止工作之例子

3.6.4 策略 D：幾乎不會再被拜訪才中止工作

為了解決策略 C 所提到的小小問題，我們設計了一個幾乎不會再被拜訪才中止工作的策略 D，我們檢查所有從檢驗節點到根節點的路徑，看是否全部的路徑上皆有一個分支是被修剪的，若是全部都有，我們才執行中止工作。與策略 C 非常類似，差別只在於，我們不再強調一定要 Hard-Prune，而是只要修剪就好。如此雖然與策略 A 一樣有可能會遇到，已經中止的工作，卻再次被選到而送出檢驗，但策略已嚴苛許多，重複計算的程度將獲得改善。圖 18 為在策略 D 下中止工作 B 的例子。

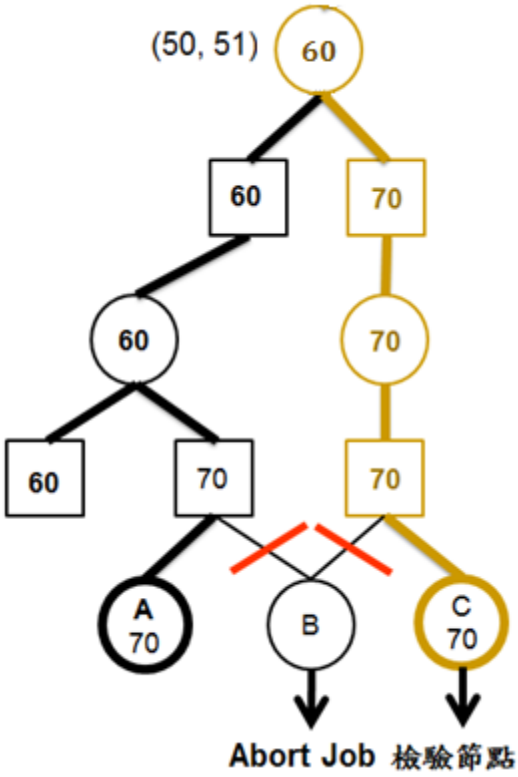


圖 18. 策略 D 幾乎不會再拜訪就中止工作之例子

3.7 更新所有父節點

有鑒於本論文探討的開局樹，裡面的節點是可以有多個父節點的，因此當我們運算完一個檢驗節點，應該將其資訊更新到每個父節點，而不是跟傳統的 Alpha-Beta 搜尋一樣，只有更新搜尋下來的路徑。我們將此策略稱為更新所有父節點（Update All Parents）。如此一來開局樹資訊量的獲取速度加快，也更加快驗證的所需的時間。圖 19 為更新所有父節點的例子。

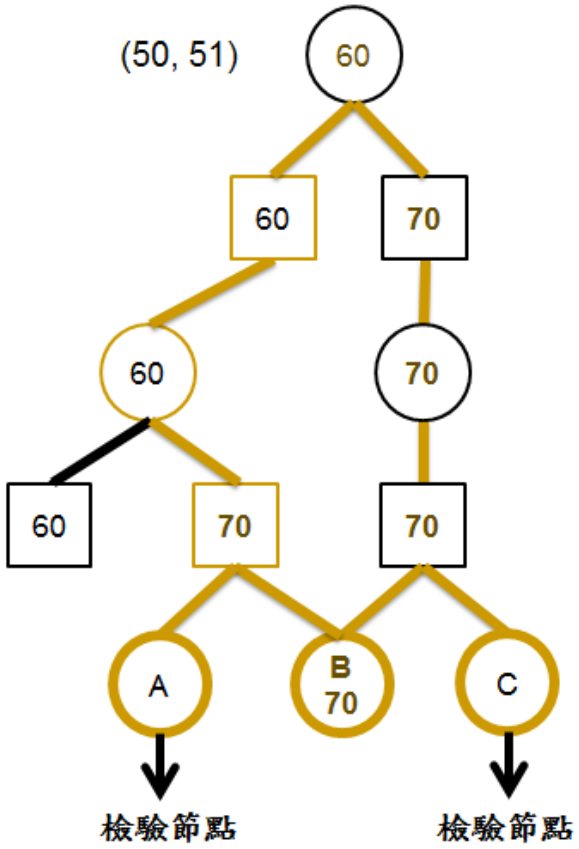


圖 19. 更新所有父節點的例子

第四章　實驗分析與討論

本章透過模擬的方式實驗看使用 DUAL*有比 Alpha-Beta 增進多少效能，以及 DUAL*在各種增進效能的策略上的表現。4.1 節介紹實驗環境，4.2 節介紹所使用的開局樹以及對於可信步的判斷，4.3 節將先為大家解說在圖表使用的各演算法的縮寫，4.4 節將比較全算、Alpha-Beta、AB-DUAL*基本版的效能，4.5 節比較 DUAL*與增進策略的搭配上的表現，4.6 節將列出在各核心數上在 DUAL*各策略增進的效能。

4.1　實驗環境

我們實驗的環境如下：

- 使用機器：AMD Phenom(tm)II X6 1090T Processor 3.20 GHz
- 開局庫：使用網路上 68 萬多盤高手對局的棋譜整理
- 檢驗節點程式：棋謀(2010 年 ICGA 銅牌)
- 回傳分數：棋謀運算 1 億個節點後回傳之分數

  (平均運算時間 7.96min)

4.2　開局樹

將開局庫裡走此手順的步大於 10 盤的，且走此手順比例佔兄弟節點 1%以上的步，擷取出來建立開局樹。此開局樹共有 113565 個節點，其中有 21745 個葉節點。

建完樹之後，每個節點再根據勝率、和率、敗率和與兄弟節點盤面比

例計算出 Rank 值，依此 Rank 值做子節點的排序，而 Rank 值的計算公式，紅黑方並不相同：對於紅方，首重勝率；而黑方則是勝率、和率一樣重視。且子節點 Rank 值與其兄弟節點裡最高的 Rank 值相近，則為信任步；反之，有一定的差距，則設不信任。在該方觀點時，只走信任步，不走不信任步。

4.3 實驗結果圖表簡寫說明

因為本論文採用的策略繁多，且名稱較長，故在此先定義簡稱，以方便以下圖表的說明。

表 1. 實驗結果圖表簡寫說明

簡寫名稱	說明
Alpha-Beta	Job-Level Alpha-Beta Search Opening。
DUAL*	Job-Level SSS* Opening，因實際上是使用 AB-DUAL*，而 AB-DUAL*只是 DUAL*的改寫，於是由此代稱。
Basic DUAL*	採用 B 策略（一率不中止工作） 的 DUAL*，因為實作最簡單，效率也是最低，所以當作 Basic 版本的 DUAL*。
AbortC DUAL*	採用 C 策略(確定不會再拜訪才中止工作）的 DUAL*。
AbortD DUAL*	採用 D 策略(幾乎不會再拜訪就中止工作）的 DUAL*。
Update DUAL*	更新所有父節點的 DUAL*。
AD + U DUAL*	AbortD + Update DUAL*。

4.4 DUAL*與 Alpha-Beta 的比較

為了比較 Alpha-Beta Search 與使用 AB-DUAL*在開局庫的表現，我也實作了 JL-Alpha-Beta Search Opening，採用的 Alpha-Beta 一樣是 Fail Soft 的版本[9]。而表 2 是在紅方觀點下，比較全算、使用 JL-Alpha-Beta Search Opening 和 JL-SSS* opening 的表格。

表 2. 全算、Alpha-Beta 和 SSS*的比較表

紅方	全算		Alpha-Beta		Basic DUAL*	
	運算 Job 數	完成時間 (Hr.)	運算 Job 數	完成時間 (Hr.)	運算 Job 數	完成時間 (Hr.)
1 core	8926	**1197.8**	1307	**267.9**	1061	**183.0**
2 cores	8926	**598.9**	1682	**168.5**	1343	**116.4**
4 cores	8926	**299.4**	2215	**103.5**	1665	**73.1**
8 cores	8926	**149.7**	2825	**62.8**	2069	**45.6**
16 cores	8926	**74.9**	3611	**38.0**	2573	**28.4**

可以看到在 1 核的狀況，全算需要 1197.8 個小時，是非常大的運算量，而使用 Alpha-Beta，可以將運算完成時間縮減到 267.9 的小時，已經減少相當多的計算量了，但 Basic DUAL*更是進一步的將完成時間縮減到 183 個小時，而且在各核的狀況，都比 Alpha-Beta 來的好，在 16 核時，更是讓完成時間進步到了一天又 4.4 個小時。以上數據都在在佐證了我們使用 DUAL*演算法的正確選擇。所以之後我們將只討論使用 DUAL*以及所搭配的各種策略增進的情況。

4.5 DUAL*與增進策略的搭配比較

我們實驗了各個策略搭配 DUAL*的結果，紅方觀點實驗結果請見圖 20、21；黑方觀點實驗結果請見圖 22、23。紅方在 1 核的表現上，Basic 版需要 183 小時，而 Update 版減少到了 150.5 小時；在 16 核的表現上，Basic 版需要 28.4 小時，AbortD 版要 26 小時，Update 版要 22.3 小時，AD + U 版要 21.1 小時，將效能增加到在一天內即可驗證完。

正如預測的，若光看 Abort Job 的策略，AbortD DUAL*表現的最好，相較於 AbortC DUAL*在 2 核～16 核最多只增進了近 4%，AbortD DUAL*都增進了 4%以上。特別要注意的是在紅方 Update DUAL*表現相當良好，在 1 核時就將效能增進了 21%，在 16 核的表現上更是將效能增進了 34%；而黑方雖然 Update DUAL*表現得沒那麼出色，但也有 9%～18%的提升。這也顯示更新所有父節點（Update All Parents）是個需要採取的策略。

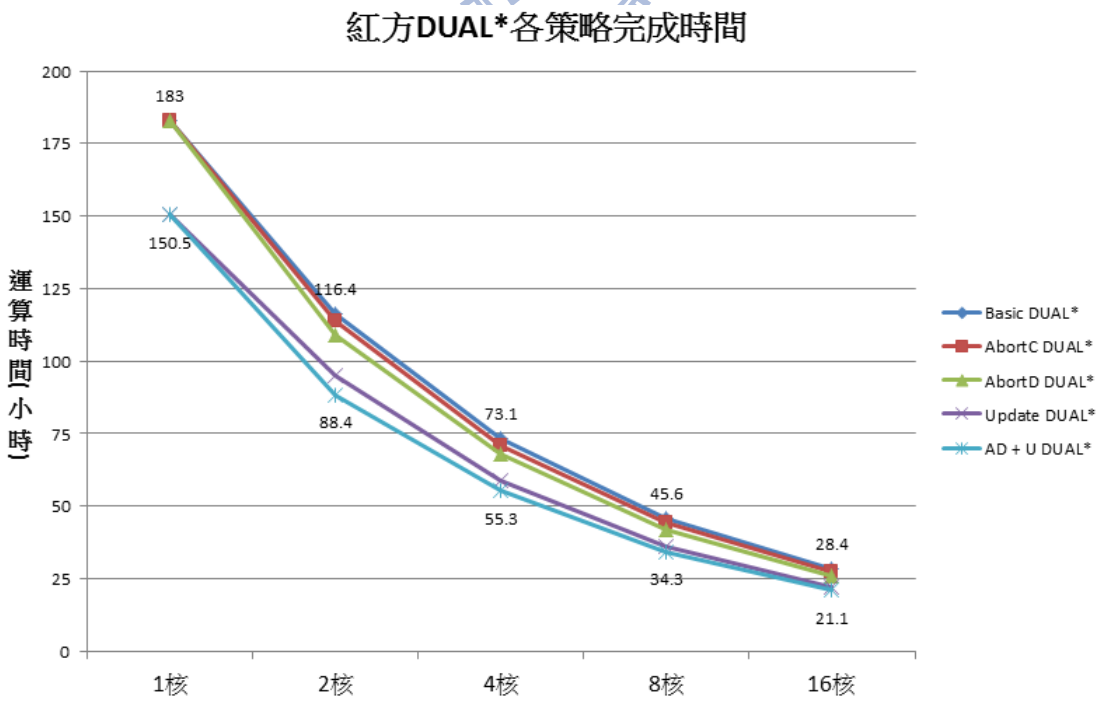


圖 20. 紅方 DUAL*各策略完成時間圖

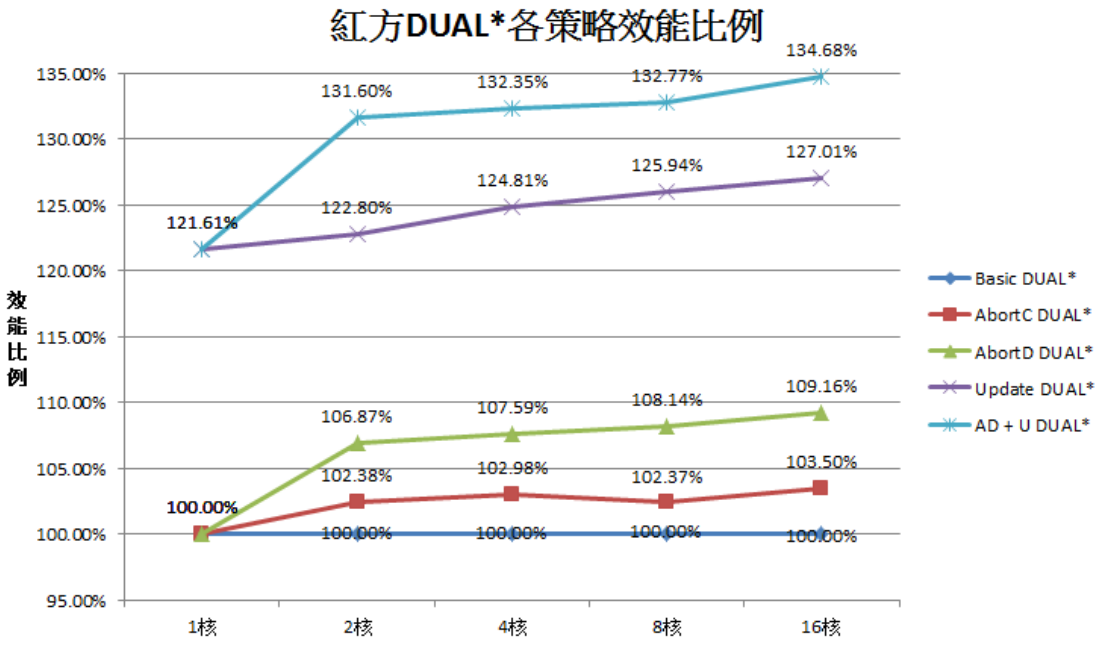


圖 21. 紅方 DUAL*各策略效能比例圖

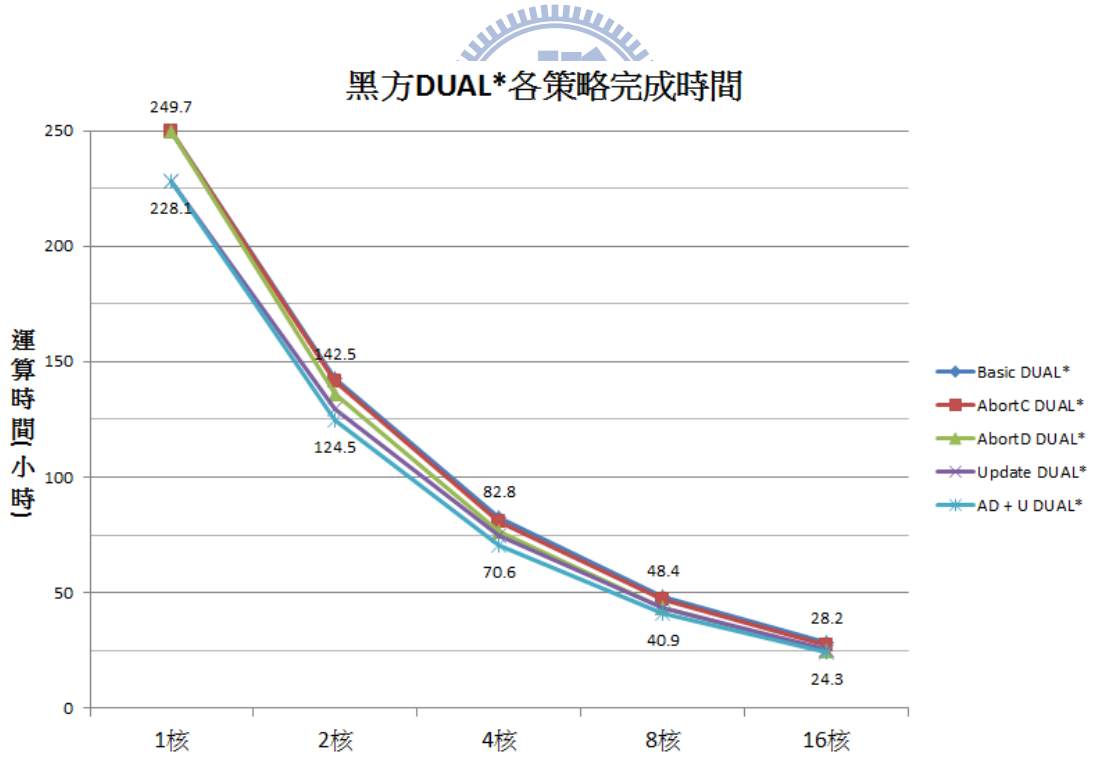


圖 22. 黑方 DUAL*各策略完成時間圖

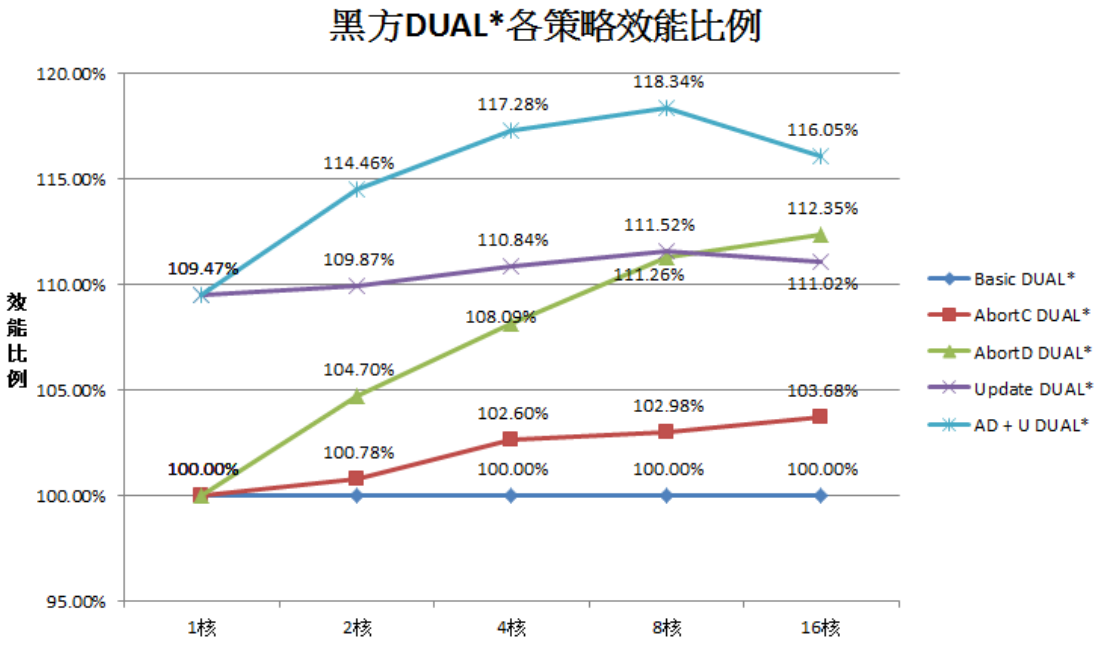


圖 23. 黑方 DUAL*各策略效能比例圖

而黑方更新所有父節點上較紅方沒有效果的原因是，因為黑方在 Rank 值的計算上，相較於紅方傾向求和；所以在可信步範圍裡，往均勢盤面上走的路較多，而這種盤面分數大致都在 0 附近，使得這種分數在向上更新所有父節點時，相較於紅方，在更新後產生的效果上較差。

為了佐證我們的說法，我們把紅方與黑方 Rank 值的計算方式調換，重新做實驗，如圖 24。改版後在 Update 上的表現， 確實提升許多。

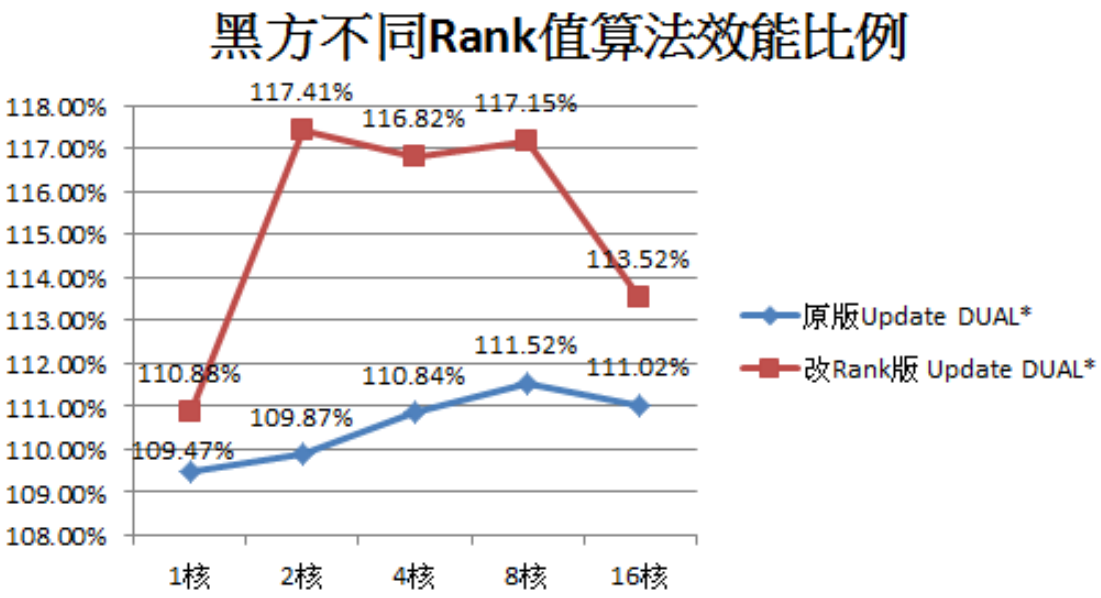


圖 24. 黑方不同 Rank 值算法 Update 效能比例圖

4.6 各核平行化增加效率

我們對於各核平行化增進效率上，也做了統計分析，如圖 25、圖 26，在各核平行化增進效能上來說，各策略上，紅方 2 核比 1 核增進至少 1.57 倍，16 核比 1 核增進至少 6.45 倍;黑方 2 核比 1 核增進至少 1.75 倍，16 核比 1 核增進至少 8.85 倍。再仔細觀察，紅方在每增加兩倍的運算資源，紅方效能增加至少 1.5 倍，黑方效能增加至少 1.6 倍。

在各核平行化單核增進效率中，可以看到即使在 16 核，每核也都至少保持 4 成的有效運算，平行化效能不差。

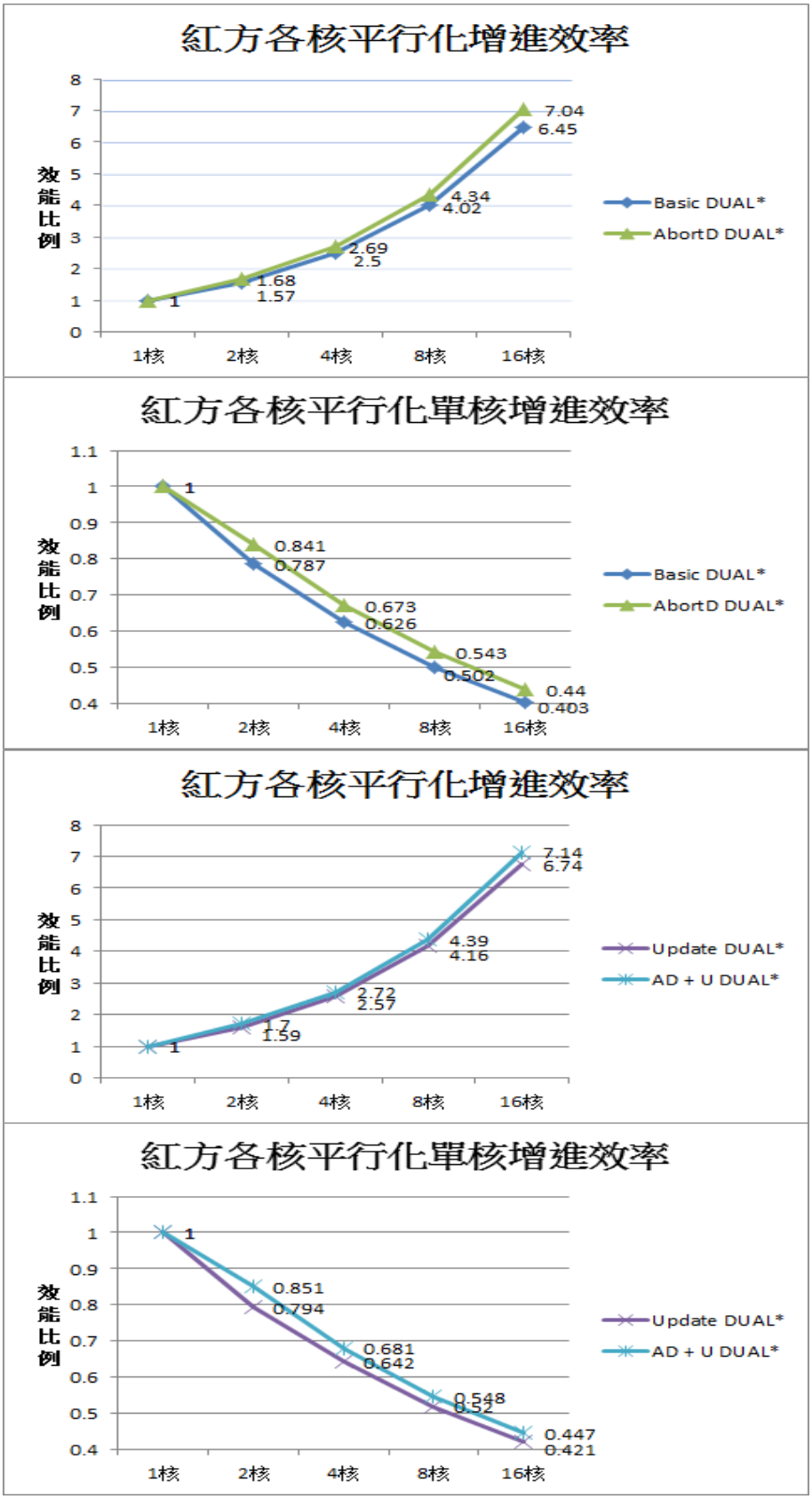


圖 25. 紅方各核平行化增進效率圖

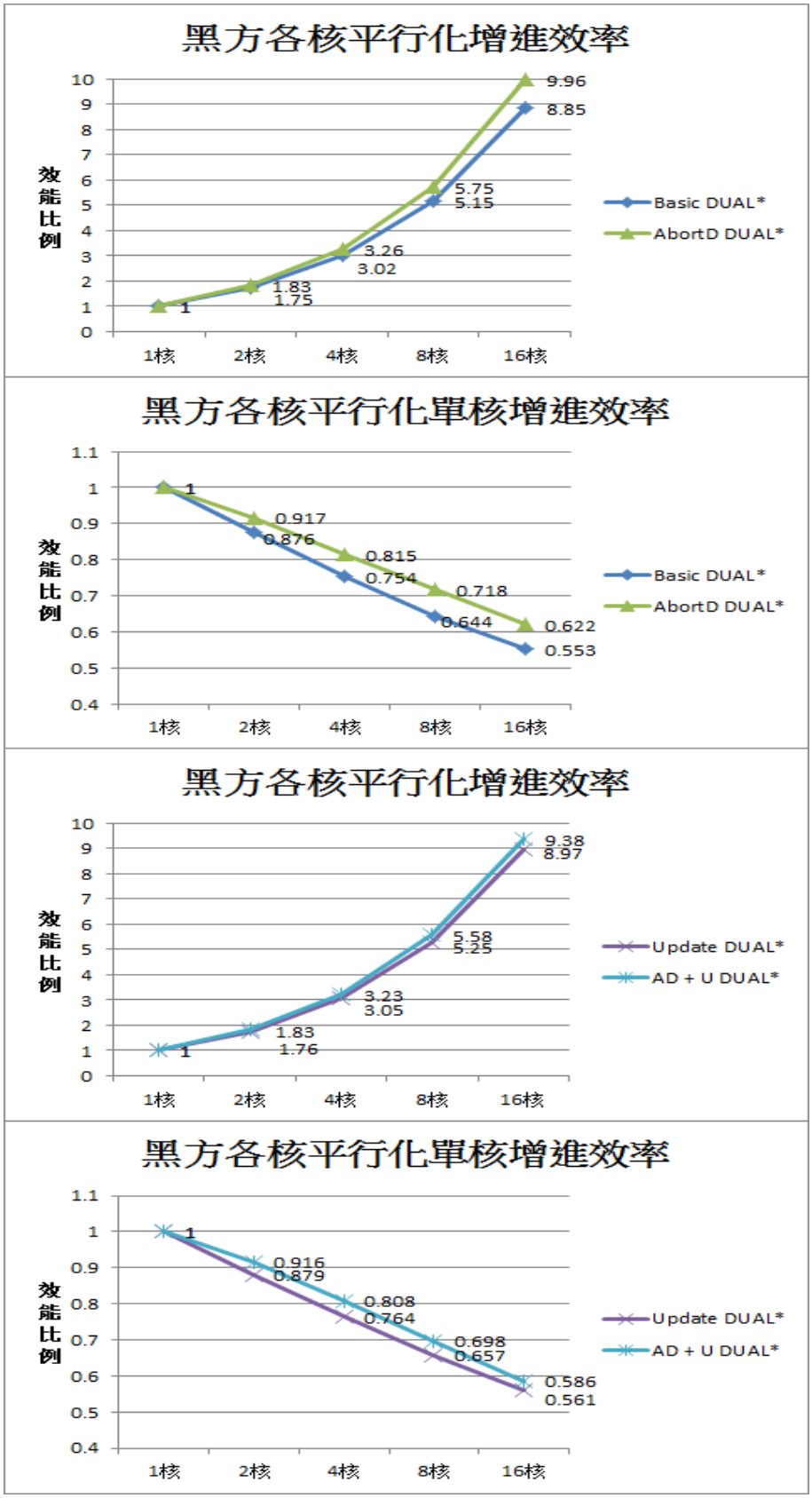


圖 26. 黑方各核平行化增進效率圖

第五章　結論與未來展望

本論文提出的 Job-Level SSS* Opening 確實能快速的驗證象棋開局庫的適合度，因此使用此方法後，將不用擔心需要很長的時間或是很多的運算資源，才能確認該開局庫是否適合此 AI。經過實驗也顯示，AB-DUAL*比起 Alpha-Beta 更能快速驗證開局庫。本實驗的例子，AB-DAUL*在 1 核運算資源上，比全算快了 6 倍多，比 Alpha-Beta 快了近 1.5 倍；AB-DAUL*在 16 核運算資源上，比全算快了 2.6 倍，比 Alpha-Beta 快了近 1.4 倍。

在更新所有父節點策略上，實驗結果顯示，確實能幫助增進效能，是一個可信賴的增進效能的策略。本實驗的例子，比起純用 AB-DUAL*，加上更新所有父節點策略後，紅方至少增進 21%，黑方至少增進 9%。

在中止工作策略上，而經過實驗的比較，策略 D（幾乎不會再被拜訪才中止工作）的效果確實最好，提升的效能比策略 B（一律不中止工作）和 C（確定不會再被拜訪才中止工作）好許多，也是個值得信賴的增進效能的策略。本實驗的例子，策略 D 比策略 B 和策略 C 至少好上 3.8%。

簡而言之，本篇論文主要的貢獻如下：

1. JL-SSS* opening 確實能快速的驗證開局庫的適合度。
2. AB-DUAL*比起 Alpha-Beta 更能快速驗證開局庫。
3. 更新所有父節點是一個可信賴的增進效能策略。
4. 中止工作的各策略中，策略 D 是實驗結果最好的策略。

而本篇論文仍有許多可以進一步加強、改善的地方如下：

1. 進一步調整中止工作的策略，也許在放寬一些更能增加效能，但同時也要考慮到穩定度。

2. 可再試驗多一些的選點方式，也許可以再增進效能。

3. 進一步驗證非葉節點的節點，加強開局庫的可信度。

希望藉由這些努力方向，在之後可以有更快驗證開局庫的方法，協助 AI 提升能力，拉近 AI 與人類高階棋士的距離。至於是否真的能幫助 AI 的能力提升，還須待驗證。

參考資料

[1] 陳志昌（2005）。 電腦象棋知識庫系統之研製。

[2] 陳靖平、吳毅成（1999）。適用於六子棋應用的網格計算系統。

[3] 陳昱維、吳毅成（2011）。適用於電腦遊戲之志願型計算系統仲介者。

[4] 許舜欽, 曹國明 （1991）。電腦象棋開局知識庫系統之研製，台灣大學工程學刊，第五十三期，頁 75—86。.

[5] 鄒忻芸、吳毅成（2000）。CGDG 桌機格網之應用層框架一般化與資源分配管理。

[6] 韓尚餘、吳毅成（2011）。適用於電腦對局遊戲之志願型計算系統工作端。

[7] Bruce Moreland's Programming Topics Site, available at http://brucemo.com/compchess/programming/index.htm

[8] J.P. Fishburn (1980). An optimization of alpha-beta search, SIGART Bulletin, Issue 72.

[9] J.P. Fishburn (1983). Another optimization of alpha-beta search, SIGART Bulletin, Issue 84.

[10] ICGA Tournaments, available at http://www.grappa.univ-lille3.fr/icga/

[11] International Computer Games Association, available at http://ilk.uvt.nl/icga/

[12] D.E. Knuth and R.W. Moore (1975). An analysis of alpha-beta pruning, Artificial Intelligence, 6:293–326.

[13] T. Marsland, A. Reinefeld, and J. Schaeffer (1987). Low Overhead Alternatives to SSS*. Artificial Intelligence, Vol. 31, No. 2, pp. 185-199. ISSN 0004-3702.

[14] J. von Neumann and O. Morgenstern (1944). Theory of Games and Economic Behavior. Princeton University Press, Princeton, NJ.

[15] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin(1994). SSS* = a-b + TT. Technical Report TR-CS-94-17, Department of Computing Science, University of Alberta, Edmonton, AB, Canada.

[16] A. Plaat, J. Schaeffer, W. Pijls and A. de Bruin (1995). A Minimax Algorithm Better than Alpha-Beta? No and Yes. TR-95-15, Department of Computing Science, University of Alberta, Edmonton, AB, Canada.

[17] A. Plaat, J. Schaeffer, W. Pijls and A. de Bruin (1996). Best-First Fixed-Depth Minimax Algorithms. Artificial Intelligence, Vol. 87, Nos.

1-2, pp. 255-293. ISSN 0004-3702.
[18] A. Plaat, J. Schaeffer, W. Pijls and A. de Bruin (1999). A Minimax Algorithm better than SSS*. Artificial Intelligence, Vol. 87, pp. 255–293. ISSN 0004-3702.
[19] J. Pearl (1980). Scout: A Simple Game-Searching Algorithm with Proven Optimal Properties. Proceedings of the First Annual National Conference on Artificial Intelligence, pp. 143-145.
[20] I. Roizen and J. Pearl (1983). A Minimax Algorithm Better than Alpha-Beta? Yes and No. Artificial Intelligence, Vol. 21, pp. 199-230.
[21] G.C. Stockman (1979). A minimax algorithm better than alpha-beta? Artificial Intelligence, 12(2):179–196.
[22] J.R. Slagle and J.K. Dixon (1969). Experiments with some programs that search game trees, JACM 16, 2 189-207.
[23] I.C. Wu, H.H. Lin, P.H. Lin, D.J. Sun, Y.C. Chan, B.T. Chen. (2010). Job-Level Proof Number Search For Connect6. Computers and Games.
[24] WIKIPEDIA, Chinese Chess, available at http://en.wikipedia.org/wiki/Chinese_Chess
[25] S.J. Yen, J. C. Chen, T. N. Yang and S.C. Hsu (2004). Computer Chinese Chess. ICGA Journal, Vol. 27, No. 1. pp. 3-18.