國立台灣師範大學
資訊工程研究所碩士論文

指導教授：林順喜博士

電腦暗棋程式 Darkness 的設計與實作
The Design and Implementation of the Dark Chess Program Darkness

研究生：詹凱翔 撰

中華民國 一百零二 年 七 月

目錄

圖目錄

摘要

電腦暗棋於 2010 年正式加入電腦奧林匹亞競賽，從此開始有許多團隊加入研究的行列。其共通課題皆為：走步與翻子的時機如何掌握？這個問題是相當困難的，想要完滿的解決並不容易。

我們的目的是開發出一款能夠處理翻走問題的暗棋程式，目前來說顯然這是不實際的。在折衷處理之下，本程式以速度為優先考量，同時也希望能夠具有一定水準的棋力。

暗棋程式的棋力強弱，與搜尋的深度息息相關，而搜尋的深度又受電腦硬體以及演算法限制，所以我們的核心目標就是使用各種手段為程式加速，來加強棋力。

本程式使用 Alpha-beta pruning 以及 Bitboard，並搭配 Transposition table 加速，擁有高速運算的能力，能夠在大部分的局面下走出較好的走步，並在搜尋中避開大量的重複盤面來節省時間。

目前本程式已於 TAAI 2012 以及 TCGA 2013 分別取得銅牌，比賽中也發現仍然有可修正的部分，接下來將全力發展後續改良設計，期望能於未來的比賽取得更好的成績。

關鍵字: 電腦暗棋、人工智慧

ABSTRACT

Computer Dark Chess officially became a competition item in the computer Olympiad in 2010. From then on, there are many teams joined this research area. Its common topics are: what's the strategy to choose to "Move" or "Reveal"? This topic is quite difficult; hence perfect solution is not easy to be derived.

Our aim is to develop a program which can solve this topic. But, it is clearly not practical at present time. In a compromise deal, speed is the first priority in our consideration. We hope that our program has a stronger ability than others.

The ability of Dark Chess program is decided by its searching depth. The searching depth is limited by computer hardware and the algorithms used in the program. Here our core goal is to use a variety of means to speed up and improve the ability of our program.

Our program uses alpha-beta pruning algorithm, bitboard structure, and transposition table to achieve a high performance objective. And it can output better moves in most cases, and avoids a huge amount of repeated boards during the search to save time.

So far our program has won the bronze medals in TAAI 2012 and TGCA 2013. We still found our many deficiencies during the competitions. We hope that we can continuously improve our program and get good scores in the future competitions.

Keywords: Chinese Dark Chess, artificial intelligence

第一章 緒論

1.1 研究背景

在人工智慧中，電腦對局是一個相當重要的部分，人類在進行競爭的思考行為是極端複雜的，而電腦對局的目的即是讓電腦能夠像真人一樣，在各種遊戲中進行競賽，從而貼近人類的思考模式。

目前電腦對局的現況可以說是蓬勃發展，在各種競技遊戲都有良好的進展，且部分遊戲的人工智慧已經追上人類棋手的水準，甚至有些已經遠遠超過，在人機對局上可以輕鬆獲勝，這證明電腦對局能夠有效的貼近甚至超越人類思考的模式。

電腦對局對遊戲的分類可粗分為兩類（在此只討論兩人遊戲），分別是完全資訊遊戲（perfect information）和不完全資訊遊戲（imperfect information）。完全資訊遊戲中，玩家所作的所有動作都可被另一方所得知，換言之，也就是雙方都能夠掌握盤面上所有的資訊，沒有任何機率成份來影響遊戲的勝負，西洋棋、圍棋以及象棋都是完全資訊遊戲；不完全資訊遊戲則是玩家的某些動作另一方無法得知，無法得到所有的資訊，僅能憑盤面上以及自己持有的資訊作出判斷，如此就必須對無法得知的資訊作機率計算，樸克類的各種遊戲即是此類遊戲最好的代表。電腦暗棋則在近幾年於電腦對局中異軍突出，成為一個具有良好發展性的不完全資訊遊戲，該遊戲在 2010 年第十五屆電腦奧林匹亞電腦對局競賽中（The 15th Computer Olympiad, 2010），正式列為競賽項目之一。

本研究中實作出一個暗棋程式 Darkness，於 2012 年 11 月獲得由中華民國人工智慧學會（Taiwanese Association for Artificial Intelligence, TAAI）於台南成功大學舉辦的電腦對局比賽銅牌、並且也在 2013 年 6 月獲得臺灣電腦對局學會(Taiwan Computer Game Association , TCGA)於新竹交通大學舉辦的電腦對局比賽銅牌。

1.2 研究目的

象棋是最富有中國特色的棋類，而暗棋是脫胎於象棋的一種新玩法，利用一面有不同文字、另一面圖案相同的特徵，剛好象棋的棋盤一半為 32 格，就發展出了這樣一個特別的玩法。此玩法較普通全盤的象棋簡單，被發明出來後迅速傳播於各級學校之間，大部分的台灣人都有玩過，是許多人兒時的回憶，也能代表台灣的特色。

而暗棋與普通棋類(如西洋棋)的區別在於三個地方，其一是具有暗子，也就是具有機率性，在這裡玩家就無法掌握全盤的資訊，即使當下優勢，很有可能因為一次翻子而喪失原有的優勢；相反的也有可能劣勢方翻了一個大子在好位置上，而反敗為勝，所以謹慎的翻棋策略是暗棋很重要的一環。其二則是棋子的移動規則單純，除炮之外的所有子都只能走相鄰一格，這點與其它棋類的差異非常大，如西洋棋與中國象棋，都具有數種長距離移動的子，如此就會產生各種互相掩護的特殊棋型，必須加入審局的考慮之中；而暗棋因為除砲之外的所有子移動方式都相同，所以只需考慮子力大小，相對單純很多。其三是獲勝條件的大差異，傳統棋類的獲勝條件是擒王，即是有 sudden death 的類型，迫著搜尋顯得十分重要，只要能夠搜尋出吃掉對方王的走步，其它非王的棋子無論損失多少都沒關係；而暗棋則不同，獲勝條件是吃掉對方所有子，所以並沒有特別需要保護的子存在，

其搜尋方針就有所不同。

電腦暗棋於 2010 年正式加入電腦奧林匹亞競賽，從此開始有許多團隊加入研究的行列，其共通課題皆為：走步與翻子的時機如何掌握？許多盤面走步在淺層乍看之下有所獲利，事實上卻是全盤皆殺的死局，唯有翻子才有一線生機；又有些盤面翻子風險極高卻無明顯利益，走閒步不翻子較翻子好，這其中的分寸不易拿捏，導致許多盤面由人類來看是必贏的，可以輕鬆拿下，由電腦來比時卻大都以和局收場；或者己方劣勢有機會求和，卻亂翻棋導致輸掉的情形。這些問題的產生都是因為暗棋屬於不完全資訊遊戲，程式難以判斷局勢的好壞，容易做出錯誤的決定，即使是正確的決定由於運氣因素也有可能出現輸棋的結果。

由於以上原因，我們的目的當然是開發出一款能夠處理上述問題的暗棋程式，這是不容易的，所以希望能夠折衷處理，製作出一款速度與棋力皆有一定水準的程式，並且寫出易於理解的研究教學文獻，希望能夠讓對暗棋有興趣的人可以快速學會演算法以及實作技巧，進而寫出擁有一定棋力的暗棋程式。

1.3 研究意義

暗棋與其它熱門棋類最大的不同點，就是暗棋具有運氣成份，但棋力又對勝負影響相當大，在計算機率上，電腦可以非常精準的去計算翻子的期望值，輕鬆得出理論上的最佳解，電腦的計算和記憶能力本身就是一個遠強於人類的優勢；在盤面評估上，只要教給電腦足夠清晰的邏輯，就很難說電腦與人類誰比較好。

但是當兩者結合在一起時，有趣的事情發生了，走步與翻棋究竟哪個比較好，人類可以得出較為正確的結論，電腦卻面臨了困難的抉擇，平常的搜尋樹是只有

考慮走子的，加入翻棋後，程式沒有一個好的平衡點來決定要翻或是不翻，這也正是暗棋的醍醐味。

在參加過的比賽以及實驗中都發現，有大於五成的盤面最後都以和局作收，這顯示目前的程式對於局勢並沒有辦法很好的掌握，程式會無法做出有效的攻擊或防守，而互相走閒步導致和棋。

在殘局的解決上，是否能讓必贏的盤面真正走到贏不要和，必輸的盤面想辦法拖到和？事實上殘局問題也能跟翻棋問題合併，在能翻的棋子數量較少時，盤面大勢已定，如何進行一個完善且平衡的搜尋就是一個非常深入的研究課題。

第二章 文獻探討

2.1 暗棋棋規介紹

暗棋是一個兩人的不完全資訊遊戲，使用的棋盤大小為中國象棋的一半，為4×8 之方格，與中國象棋不同，棋子置於格內而非線上，使用的棋子與中國象棋相同，為正面有字、反面無字的棋子。

按紅黑方由大到小分別為「帥、仕、相、俥、傌、炮、兵」和「將、士、象、車、馬、包、卒」，其中兵最小，但可以吃將，而將不能吃兵；卒亦同理。所有棋子除炮以外移動範圍皆為相鄰一格，炮能夠隔一子吃掉任何敵方的子（普通移動則不能吃掉任何一子）。

遊戲的初始布置為棋子將反面朝上，隨機分配至每個棋格上。反面朝上的棋子稱為暗子，正面朝上則稱為明子。

開始遊戲時，雙方分別選擇先後手後，必須做以下兩種動作之一。

「翻子」：將一枚子翻面變為明子。先行方第一手翻出棋子之顏色，該色即為他的己方色；反之則為敵方色。

「走子」：將一枚明子移動至相鄰空格上，或是與自身同級或較小之敵方明子上，並移除目標位置之子，若為炮則可隔一子移動至敵方明子上，不可移動至暗子上。

如此往復循環，消滅對方所有棋子者為勝，若雙方 40 步無吃子及翻子，則以和局論。

2.2 相關論文及程式介紹

首篇電腦暗棋的論文為「電腦暗棋之設計及實作」[1]，由國立台灣師範大學資訊工程研究所謝曜安撰於 2008 年 6 月。在這之前都少有人對電腦暗棋程式做深入的研究，此篇論文為首篇提出暗棋演算法且對其有深入研究的論文，在當時作者所製作的電腦暗棋程式已能擊敗市面上大部分的商業暗棋程式了。

該篇論文主要使用棋盤棋子映射結構，以一維陣列紀錄哪些位置上有棋子，再紀錄每個棋子分別在哪個位置，來作資料的交叉查詢。圖 2.1.1 顯示其棋盤資料結構，圖 2.1.2 則顯示其棋子編號。

未使用較為直觀的二維陣列表示法的原因是一維陣列表示法處理上較為迅速，且高階語言的二維陣列在經過編譯程式優化之後仍然會轉變為一維陣列。

24	16	8	0
25	17	9	1
26	18	10	2
27	19	11	3
28	20	12	4
29	21	13	5
30	22	14	6
31	23	15	7

圖 2.1.1 棋盤資料結構

空格	將	士		象		車	
0	1	2	3	4	5	6	7
馬		包		卒			
8	9	10	11	12	13	14	15
卒	帥	仕		相		俥	
16	17	18	19	20	21	22	23
傌		炮		兵			
24	25	26	27	28	29	30	31
兵	空白						
32	33						

圖 2.1.2 棋子編號

該程式的演算法分為幾個部分：

Alpha-Beta 演算法：該演算法類似於普遍使用於遊戲樹搜尋的 Nega-max 演算法，差別是多出了α和β兩個參數，α是記錄 max 層節點目前的最大值，而β是記錄 min 層節點的最小值，兩個參數以 call by value 的方式傳遞給更深層的子樹。若於取 max 值時，發現一個大於等於β的值，即不需再對該層其它分支進行搜尋，此稱β截斷；同理，取 min 值時若發現該值小於等於α，也一樣不需對該層的其它分支進行搜尋，稱為α截斷。

Transposition Table：是一個可將已搜尋過的節點資訊儲存起來的 hash table。節點資訊內含：盤面情形，搜尋深度以及分數，節點類型（min or max）。盤面情形一般使用 Zobrist Hash 產生的 key 值來標記，可以快速查詢該盤面是否已搜尋過。

允許空步：暗棋遊戲規則是不允許空步的，在程式實作中我們賦予空步另一個意義，在還有棋子可翻的盤面時，若每一個合法走步都會使情況變得更糟糕，此時不應該走子，而去翻棋，而程式中就以走空步代表這個情況。

寧靜搜尋:搜尋至葉節點時不立刻回傳分數,若仍有吃子步則必須繼續展開，直到呈現寧靜盤面為止，也就是盤面無法產生任何吃子走步，此時回傳的審局分數會較原本的方式更有參考性，可以克服部分因為水平效應產生的搜尋錯誤，提升搜尋準度。

Iterative Deepening：於 Alpha-Beta 搜尋時，先對淺層作搜尋，再將最大深度逐步加深,淺層得到的分數做為較深層走步排序的參考,以利於 Alpha-Beta 截斷，提升搜尋速度。

翻子與走子的選擇：對可見的棋子先做走子搜尋，之後再對每一個暗子賦予可能出現的兵種後作較淺層的走子搜尋，比較後取出最有利的位置。若翻棋的分數較高則翻棋，否則仍然走子。

循環盤面剪裁：以一個具有 128 格的 Hash Table，記錄當前搜尋中走過盤面的 Zobrist Key 值，每進入一層就會記錄，退回一層時則刪除。預備對當下盤面的合法走步子樹進行搜尋時，會先進行 Hash Table 內容的檢查，若 Hash Table 內已有相同的盤面存在，則直接呼叫審局函數並返回該層的審局分數，不需要再對子樹搜尋，可以提高搜尋的效率。

第二篇論文為「電腦暗棋程式與經驗法則之配合與實作」[2]，由國立東華大學資訊工程研究所賴學誠撰於 2008 年 7 月，概念上使用大量經驗法則，在翻棋系統上有特別的作法，內容是隔一子期待可以翻出包或炮來吃掉對方的棋子，反之則是翻出被對方包/炮攻擊較小的位置。另一個特別的創新則是靈氣系統，統整各子之間互相影響的行為給予分數，給分的基本原則是影響力由近而遠分數遞減，最多影響到距離為 5 的棋子，影響的行為分為正負兩種，正分是由我方大子給予小子保護，分數增加；負分則為對手子力的威脅，分數減少。靈氣系統的影響權重表如圖 2.1.3 所示。

棋種	將	士	象	車	馬	包	卒
帥	120	0	0	0	0	0	108
仕	108	106	0	0	0	0	0
相	60	60	58	0	0	0	0
俥	36	36	36	34	0	0	0
傌	24	24	24	24	22	0	0
炮	48	48	48	48	48	0	0
兵	0	12	12	12	12	0	10

圖 2.1.3 靈氣系統權重表

該程式的搜尋與[1]的程式一樣採用 Alpha-Beta 搭配 Iterative Deepening，但加上以經驗法則取得的位置分數表(如圖 2.1.4)，以求得更正確的審局分。

-10	-5	-5	-10
-5			-5
-5			-5
-5			-5
-5			-5
-5			-5
-5			-5
-10	-5	-5	-10

圖 2.1.4 位置扣分權重圖

第三篇為「暗棋中棋種間食物鏈關係之探討與實作」[3]，由國立台灣師範大學資訊工程研究所謝政孝撰於 2010 年 6 月，本篇論文主要為改善先前[1]的作法，改善走步產生，並使用食物鏈的概念動態改變棋子分數。食物鏈的基本概念是在棋子計算分數時，先計算棋子還能夠吃下哪些子（包括已翻和未翻），加總後再乘上該棋子自身的權重，最後配合所有場上的子加上共同基本分數得出棋子的權重分。

另一個增加準確度的方式是採用 Flood Fill 的方式偵測棋子之間的相連程度，若相連則計算威脅與被威脅的程度，來得出盤面棋子相互的關係並給出距離分，如圖 2.1.5 與圖 2.1.6 所示。

傌	包	■	車
仕	■	■	車
■	兵	象	■
■	卒	炮	■
■	相	卒	相
■	■	■	■
馬	兵	兵	俥
■	■	■	■

圖 2.1.5 Flood Fill 示範盤面

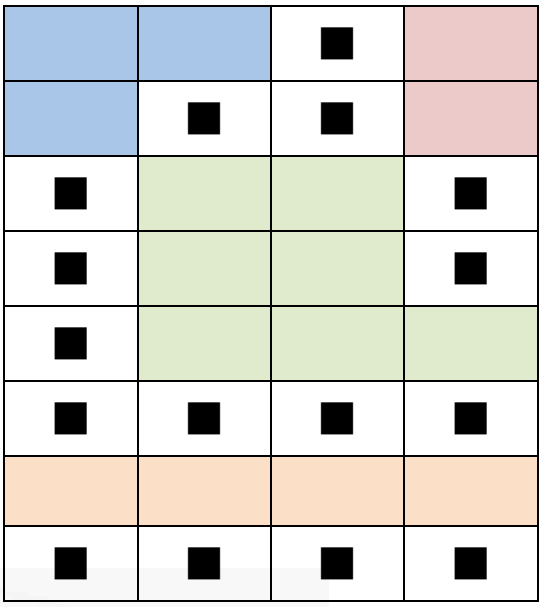

圖 2.1.6 Flood Fill 結果

另棋子的距離採用 Manhattan Distance 來計算，其計算方式為 2^{d+1}，其中 d 為兩子間的距離，如圖 2.1.7 與圖 2.1.8 所示。

兵			
			相
	將		
			馬
士			

圖 2.1.7 距離分示意盤面

距離	相	兵
將	2^{5+1}	2^{5+1}
士	2^{7+1}	2^{5+1}
馬	2^{4+1}	2^{8+1}

圖 2.1.8 距離計算結果

最後再給予盤面的固定位置權重分，如圖 2.1.9 所示。

2	3	3	2
3	4	4	3
3	4	4	3
3	5	5	3
3	5	5	3
3	4	4	3
3	4	4	3
2	3	3	2

圖 2.1.9 固定位置權重分

將以上的棋子權重分、距離分以及固定位置權重分相加即為審局函數輸出的最終分數。

該程式的翻棋部分採取相對保守的翻棋策略，優先選擇無損失的位置翻棋，將風險留給對方，例如隔一子翻棋以期待翻到炮或包，翻斜角期待可以牽制對方棋子。

該程式命名為 Dark Chess Beta，於 2010 年獲得 Computer Olympiad 亞軍。

第四篇為「電腦暗棋之人工智慧改良」[4]，由國立台灣師範大學資訊工程研究所勞永祥撰於 2011 年 6 月，本篇論文也使用 Alpha-Beta 配合 Iterative Deepening 作為搜尋演算法，在距離分的計算上提出了較為嚴謹且詳細的作法。

計算距離分的目的：

1.了解走子步是否有意義，盤面狀況是否連通，是否可以吃或者壓迫到子。

2.讓棋子確立目標，壓迫最有價值的子，選擇最好的走步。

3.若對方子力較大，可以避開有威脅的大子。

根據經驗上的判斷，可歸納出以下的規則：

1. 未翻子當然可視為障礙物。

2. 若到達目標棋子的路徑上有對方子對自身有威脅，則視為障礙物。

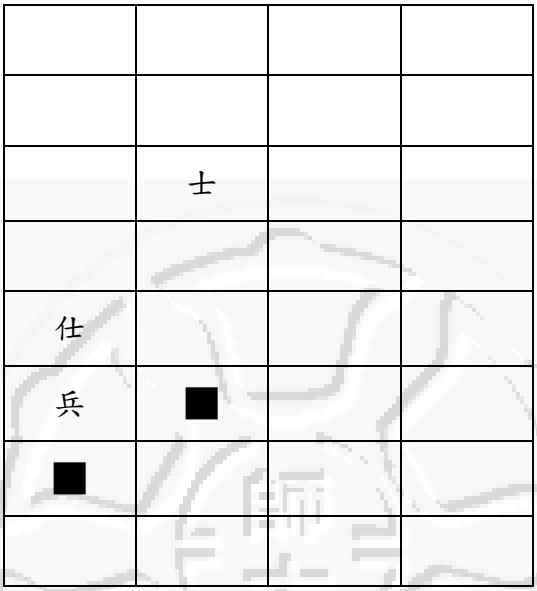

圖 2.1.10 距離威脅盤面示意

在此情況中黑士必須通過紅仕才可吃到紅兵，通過時就會被紅仕吃掉，所以黑仕不會對紅兵產生影響力。

3. 目標棋子子力與自身相等時,具有影響力。如上圖 2.1.10,黑士可以靠近紅仕，限制紅仕的走步，故仍然對紅仕有影響力。

4. 控制域的產生：棋子的攻擊範圍，棋手在下棋時會知道若黑士想吃紅兵，必須繞過暗棋才有辦法吃到。

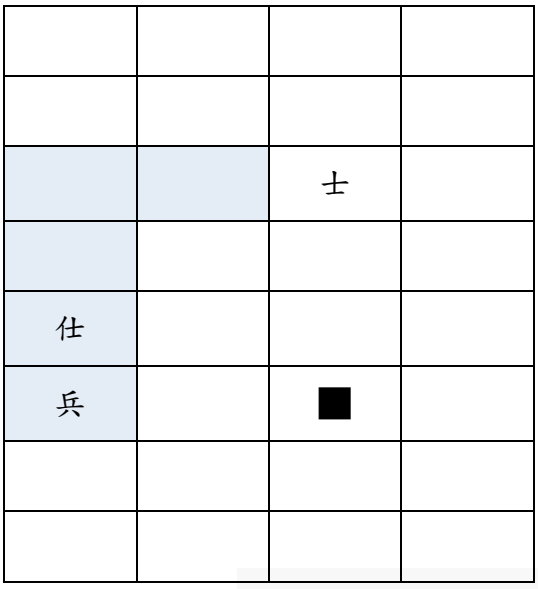

圖 2.1.11 Manhattan Distance

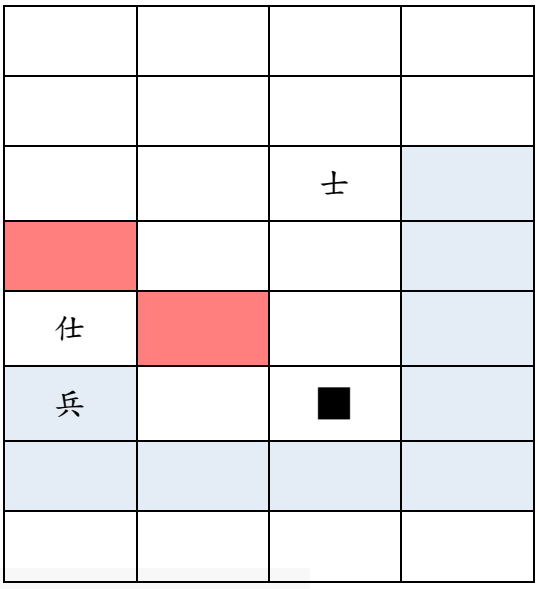

圖 2.1.12 Best Fit Search

5. 距離計算使用 Best Fit Search 來替代 Manhattan Distance。

使用 Manhattan Distance 計算出黑士與紅兵的距離為 5，在人類的認知上這完全錯誤，一般而言並不會認為黑士可以安全通過紅仕的控制域去壓迫紅兵，故黑士到紅兵的距離，認知上不會只有 5 步。使用控制域以及 Best Fit Search 所得到的距離則為 9，與人類的認知相合，影響力計算較以往精確許多。

第五篇為「電腦暗棋程式 DarkCraft 的設計與實作」[5]，由國立台灣師範大學資訊工程研究所施宣丞撰於 2012 年 6 月，該篇論文認為走步產生與審局函數是影響速度的重中之重，剛好當下西洋棋已發展出 BitBoard 的技術，於是嘗試與暗棋作出結合，但暗棋中的兵種「炮」能夠跳過一個棋子攻擊對手，這在 BitBoard 的運用上是較困難的，在西洋棋中也有多重移動選擇的棋子如皇后以及城堡，其處理方式是使用 Magic Number 搭配建好的 Table 來迅速查詢每個位置的移動產生。產生方式如圖 2.1.13 與圖 2.1.14 所示。

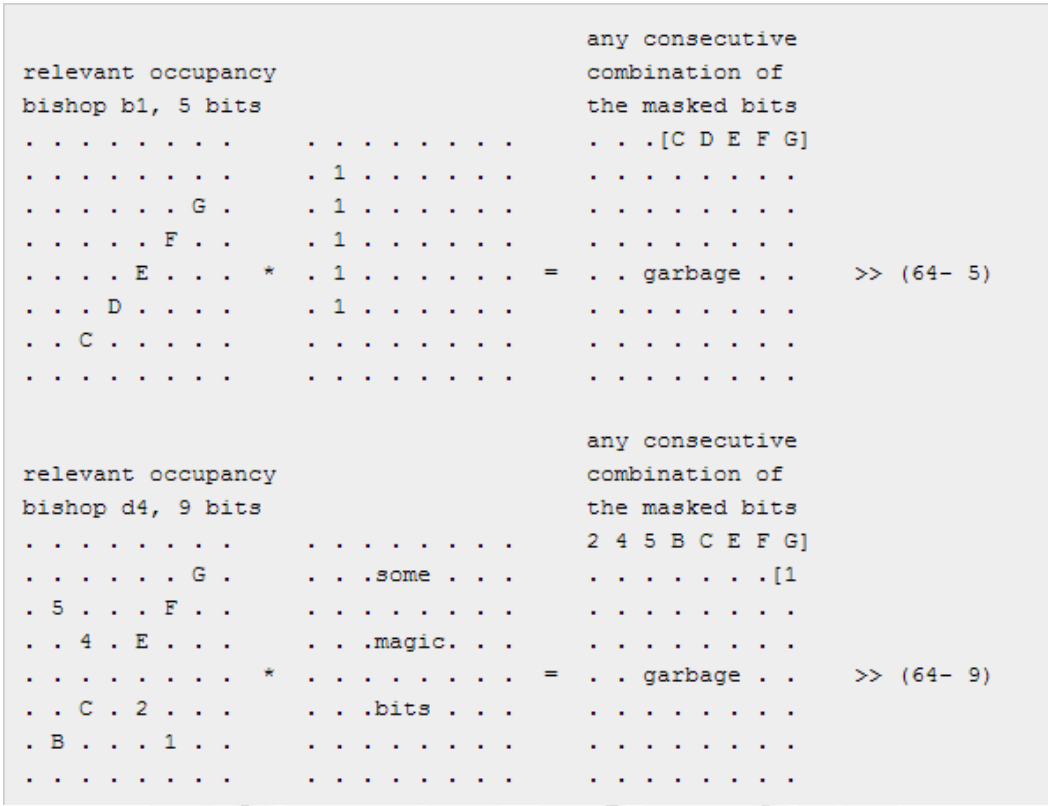

圖 2.1.13 Magic Number 使用方式 1

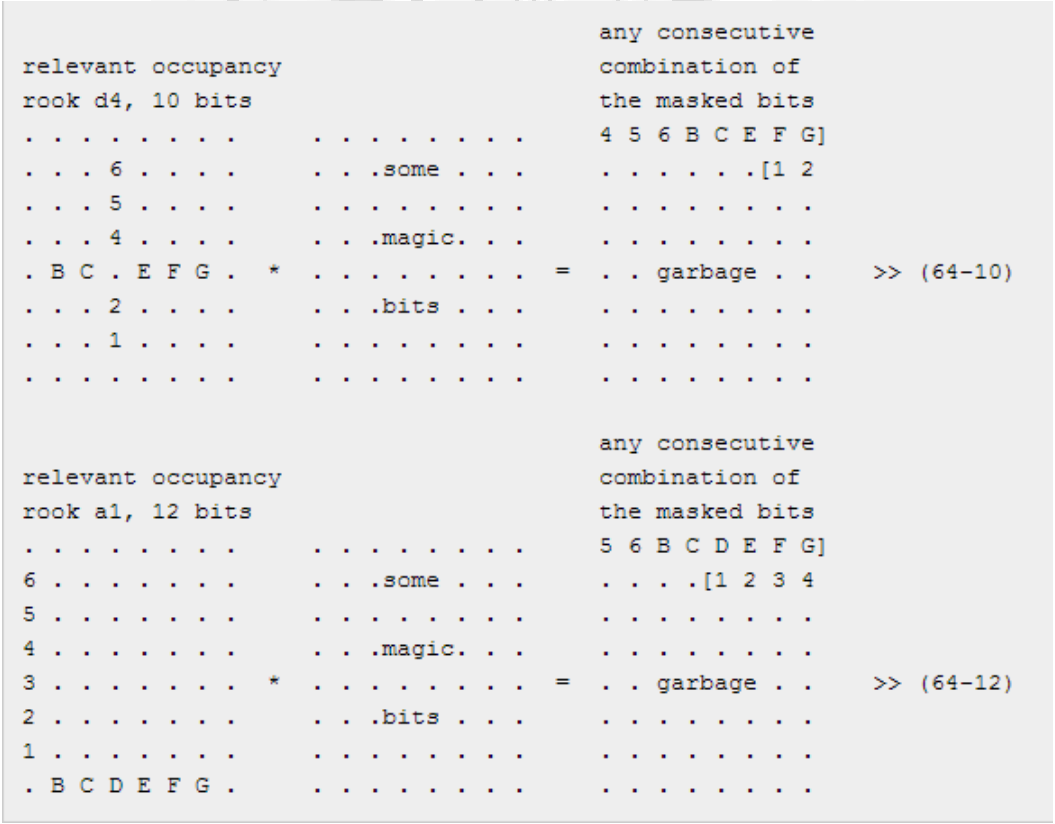

圖 2.1.14 Magic Number 使用方式 2

將表示盤面的 64 個 bit 無號整數乘上一個 Magic Number，可以將對應的 Bit 轉換到最高的幾個位元上，如此只要做移位就可以得到資訊，查表即可得到相對應的走步。表格的大小取決於走步的多樣性，每多一個可能性就會大一倍。

該篇論文作者認為這樣的方式佔用了太多的記憶體且太慢，且 Perfect Hashing 的 Magic Number 並不容易找到，故研發出了一種在暗棋中特有的方式來取得炮的走步，其方法如下。

基本概念：行與列皆可以透過左移或右移簡單的做出轉換。(如圖 2.1.15 及圖 2.1.16 所示)

			X
			X
			X
			X
			X
			X
			X
			X

圖 2.1.15 右方第一行

0x11111111

		X	
		X	
		X	
		X	
		X	
		X	
		X	
		X	

圖 2.1.16 右方第二行

0x22222222

在橫列的部分，只有數種可能性，故全部列出，以 mask 的方式取出即可。左右是對稱的，依樣製表即可，如圖 2.1.17 所示。

			包
		■	包
	■		包
	○	■	包
■			包
○		■	包
○	■		包
■	○	■	包
		包	

圖 2.1.17 橫列炮位的處理方式

而直行的部分，由於數量較多，作者採取切割分開處理的方式，以炮所在的位置作為中間點來切割，分為上下兩部分，再分別對上部取兩次最大位元；下部分取兩次最小位元即可得到所要的炮位走步，複雜度就相較於 Magic Number 減少了許多，如圖 2.1.18 所示。。

			■
■	○	包	
	■		包
包		■	■
■			○
○	包	○	
	■		■
■	○	■	

圖 2.1.18 直行炮位的處理方式

施宣丞所開發的暗棋程式「DarkCraft」榮獲在荷蘭舉辦之 2011 年電腦奧林匹亞對局程式競賽（16th Computer Olympiad）暗棋金牌。相關比賽的結果如圖 2.1.19 和 2.1.20 所示。

Rank	Program	Country	Title
1	DarkCraft	TWN	Gold medal
2	Dark_chesser	TWN	Silver medal
3	Diablo	TWN	Bronze medal
4	TopGun	TWN	

圖 2.1.19 2011 年 Computer Olympiad 比賽結果

Rank	Program	Country	Title
1	Modark	TWN	Gold medal
2	Dark Chess Beta	TWN	Silver medal
3	Leave-or-Lose	TWN	Bronze medal
4	Flipper	TWN	
4	Dark_chesser	TWN	
4	Sparrow	TWN	

圖 2.1.20 2010 年 Computer Olympiad 比賽結果

圖 2.1.19 和 2.1.20 為 2010 與 2011Computer Olympiad 比賽結果

註:2012 年未舉辦 Computer Olympiad

此外，也有一些論文探討暗棋的其它各種技術，請見[8][9][10][11]。

第三章 資料結構與演算法

暗棋的資料結構通常使用一維陣列，於[1]、[5]兩篇論文中都是，不使用二維陣列是為了提升處理速度，同時使用兩個一維陣列來儲存 32 格棋盤的盤面狀態以及 32 顆棋子分別所在的位置(以供反向查表之用)，如此一來無論是要用棋子查位置，或用位置查棋子都可以快速獲得答案，非常實用。另一個更有效率的資料結構則為 BitBoard，於「電腦暗棋程式 DarkCraft 的設計與實作」[5]中所使用。

棋類遊戲的電腦程式有一個關鍵步驟叫做走步產生，當電腦要開始作搜尋的動作時，必須先知道有哪些走步可以走，知道以後才能順利進行搜尋，搜尋中盤面會不斷的變化，只要有變化就必須重新呼叫走步產生。由此可以看出走步產生占搜尋行為本身的比例相當高，程式的速度要快就必須讓走步產生迅速，而走步產生的速度關鍵在於資料結構，能夠越快產生出走步的資料結構就是越好的結構。

而每個走步產生的順序也是一個關鍵，在特定演算法中（例如 Alpha-Beta）走步的排序將與搜尋的速度有極大相關性，若能將越好的走步越早做搜尋，則能夠更有效的排除其它較差走步以節省時間。

3.1 BitBoard

由[5]這篇論文我們知道，BitBoard 是一個非常有效率的資料結構，由於暗棋的棋盤是 32 格，BitBoard 以整數儲存資訊也是 32 Bits，這個資料結構在暗棋上意外的適用，所以我們決定仍採用這個資料結構來加強我們的程式，以期望能夠達到搜尋加速的效果。圖 3.1.1 及 3.1.2 為我們所使用的表示法。

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12
19	18	17	16
23	22	21	20
27	26	25	24
31	30	29	28

圖 3.1.1 32 位元對應位置圖

X	X	X	X
		X	X
	X	X	
X			
			X

圖 3.1.2 16 位元表示法 0x1086030F

暗棋有帥、仕、相、硨、傌、炮、兵、將、士、象、車、馬、包、卒共十四種兵種以及空格、未翻子，需要十六個整數來儲存它們，為了方便撰寫程式，可以先定義整數 U32，然後宣告十六個整數跟幾個必要的資訊，如下所示。

■	■		■
卒		兵	■
■		包	
相	仕	帥	
■		仕	
■	■		
包	卒		卒
■		將	■

圖 3.1.3 範例盤面

X	X		X
			X
X			
X			
X	X		
X			X

圖 3.1.4 piece[15] 未翻開的暗子 0x90C8081D

圖 3.1.3 為一個範例盤面，■代表未翻開的暗子，假設目前盤面是這個樣子，圖 3.1.4 展示變數 piece[15]所儲存的值。

		X	
	X		X
		X	

圖 3.1.5 帥所在位置的可能移動範圍

0x25200

X			
		X	
X	X		X
		X	

圖 3.1.6 黑子的分布

0x2D000280

X			
	X		X

圖 3.1.7 卒的位置

0x5000080

X	X	X	X
	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X
X	X	X	X
X		X	
X	X	X	X

圖 3.1.8 NOT 卒

0xFAFFFF7F

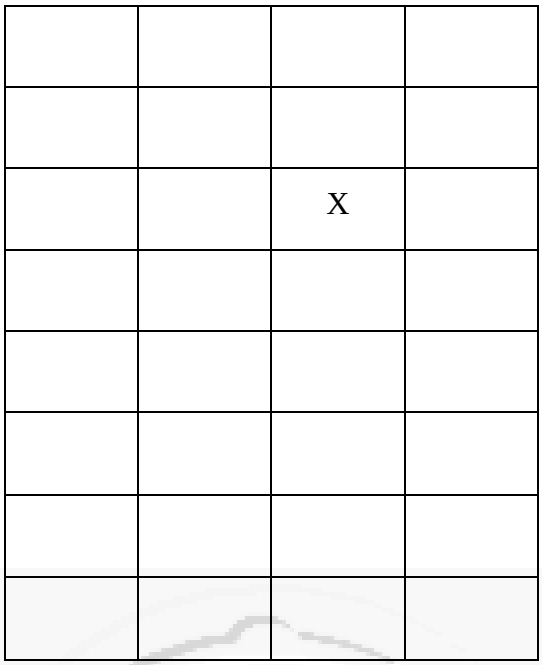

圖 3.1.9 最終得到的帥吃子步

0x200

若想要得到一些複合資訊，可將數個變數作各種邏輯運算來得到想要的結果，以上圖 3.1.5~3.1.9 即表示了走步產生時，帥吃子步產生的方式，將帥所在位置的遮罩與黑子作 AND 運算並把卒排除(對卒作 NOT 以後再 AND)的結果。至於炮走步的產生已於前文說明，故不在此贅述。

如此在走步產生這個環節上就能以極快的速度取得目標走步，其它在搜尋中所需的資訊也能以類似的方式取得。以下附上 BitBoard 結構的宣告方式。

```
typedef unsigned int U32; // 可以表示2^0~2^32-1 = 0x00000000 ~ 0xFFFFFFFF
U32 piece[16]; // 各個索引分別代表:0 空格, 1 帥, 2 仕, 3 相, 4 俥, 5 傌, 6 炮, 7 兵,8 將, 9 士, 10 象, 11 車, 12 馬, 13 包, 14 卒,15 未翻子。
U32 red, black, occupied; // 所有紅子位置，所有黑子位置，所有有棋子的位置。
```

3.2 程式的搜尋演算法

傳統的棋類搜尋演算法都是使用最大最小搜尋樹。最大最小搜尋樹是依據人類下棋觀點所研發出的一種演算法，棋手思考下一步棋要下哪時，當然會想要下在看起來最好的一個位置上，而對方同樣也會下在對他最有利的位置來作反制，若盤面對我方有利則給予高分，反之則給低分。以電腦觀點來看就是一個搜尋樹，輪我方走時會選擇對自己最好的，輪敵方走時就選擇對自己最差的，直到底層以後回傳，最後根據傳回的分數選擇最好的走步。

搜尋樹的成長是呈指數成長的，若每方每步都有 10 種選擇可走，雙方各走 10 步，則總共就有10^{20}個節點。這種成長速度使得深層的搜尋完全無法在有限時間內作完，所以通常會將雙方可走的步數作個限制，達到限制的節點稱為葉節點，達到葉節點時我們必須賦予該盤面一個分數，最後會根據這個分數來決定這個盤面是好或壞，賦予分數的動作即是審局函數。

棋手下棋時的基本思考原則是：我走了 A 走步，對方可能會用 B 作為回應，若我再走 C 走步，則對方走 D 走步回應。如此反覆繼續深入，直到棋手認為推演的覆蓋度足夠，則會開始評價下這手棋造成的局面對現況的影響是否良好，若覺得不夠好，則會退回去初始盤面不走 A 而走另一步，對方的回應又有所不同。如此繼續推演直到將所有走法嘗試一次之後取出最好的出步。

利用樹狀結構來將人類這樣的思考行為表示出來，稱為遊戲樹。

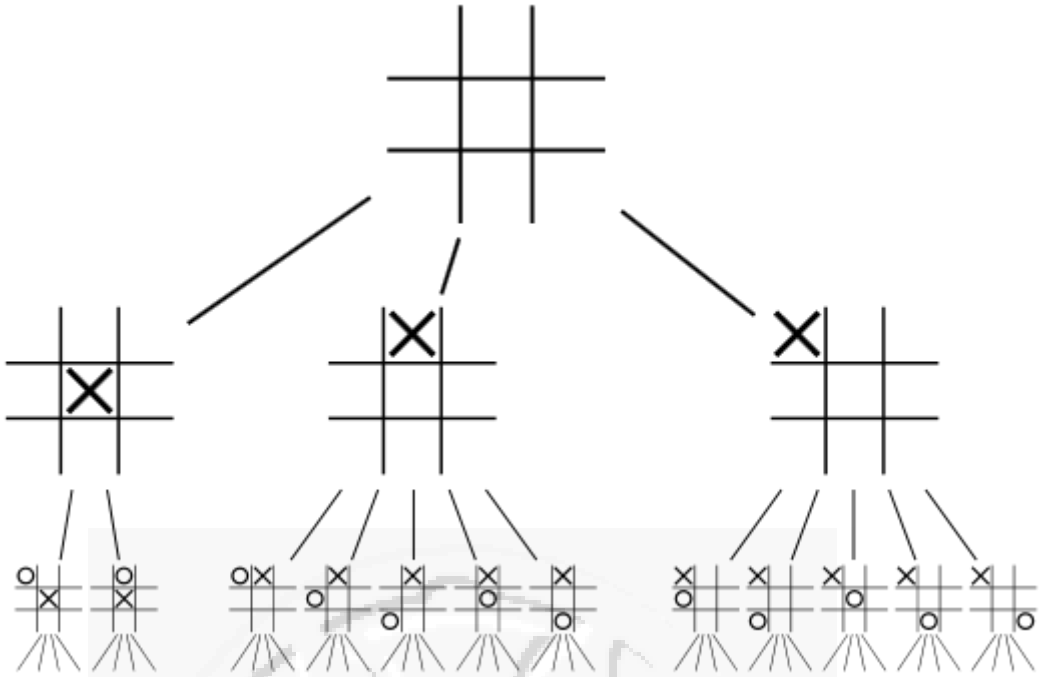

圖 3.2.1 井字遊戲的遊戲樹

以井字遊戲為例，由圖 3.2.1 可以發現搜尋層數越深，其節點數量呈現指數成長，並且井字遊戲的分支數量是隨層數逐漸縮小的，因為棋子放入以後不會再取出，能下的著手會越來越少，所以能夠在較短的時間之內搜到底層完成搜尋，得出最好的答案。但西洋棋或是中國象棋並不會因為棋子數量變動而使走步減少，甚至可能在棋子數量減少時，由於盤面更空曠導致走步選擇更多。雙方皆有 n 種走步可走的情形下，若雙方各推 k 步，這棵遊戲樹就具有 n^{2k} 個節點需要搜尋，這種巨大的搜尋量通常沒有辦法在短時間內搜完，所以必須要避開部分無用的分支或者限制搜尋的深度。

搜尋至所限制的深度時，我們會呼叫審局函數，對現在的盤面情形給予一個對應的分數，然後往父節點層層向上回傳至根節點，再搜下一個走步，重複至整棵樹搜尋完畢，根節點即可得知哪一個走步最高分。

搜尋演算法我們採用 Alpha-Beta Pruning，該演算法於 1956 年由 McCarthy

John 提出概念，在 1963 年由 Alexander Brudno 發表。

Alpha-Beta Pruning 脫胎於 Min-Max 演算法，大幅提升搜尋效率。

Min-Max Search 演算法設定敵我雙方皆會選擇對自己最有利的走步執行，採取深度優先的方式由根節點逐層往下搜尋，到了限制深度時，於葉節點呼叫審局函數給予分數以確定盤面好壞。

該演算法分為取 Min 以及取 Max 兩部分，分別在我方層以及敵方層使用，在我方層時會取最大值作為最佳走步，而輪敵方走步時則會取最小值作為最佳走步，如此交互進行至指定深度即回傳結果輸出。

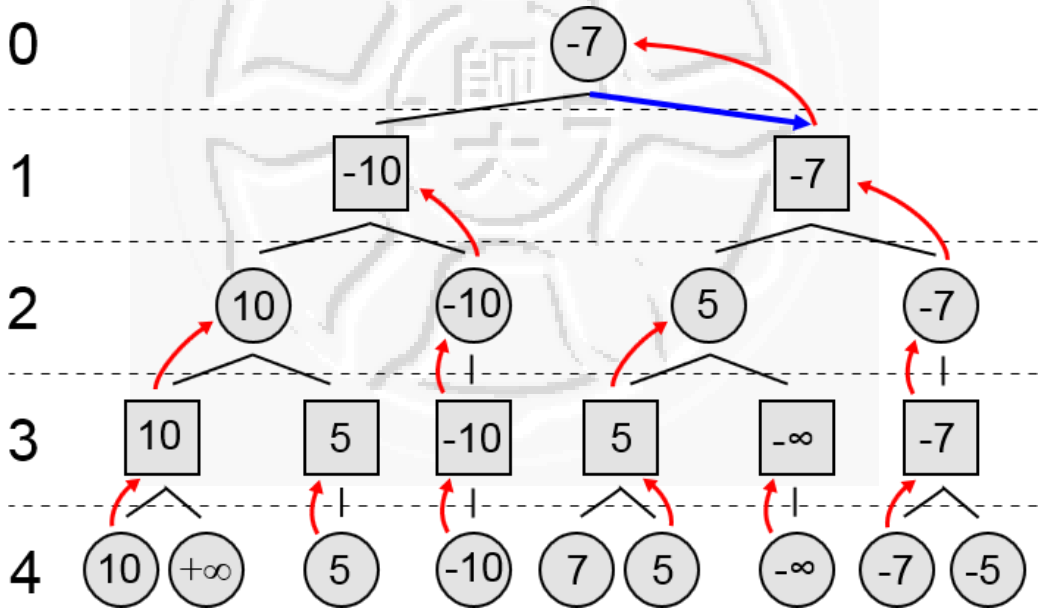

圖 3.2.2 Min-Max 搜尋的例子

圖 3.2.2 即為 Min-Max 搜尋的例子，以深度優先的方式在根節點我們得到第一次的審局分後，即會開始進行與其他走步的比較，並依照所屬層數進行取最大值或最小值的作法，往復循環最終就能在根節點得到對我方最有利的走步。

Min-Max Search 演算法對整棵樹進行搜尋，所有節點一個不漏，能夠解決問題沒有錯誤，但效率低落。其虛擬碼如圖 3.2.3 所示。

```
int maxi( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves) {
        score = mini( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}

int mini( int depth ) {
    if ( depth == 0 ) return -evaluate();
    int min = +oo;
    for ( all moves) {
        score = maxi( depth - 1 );
        if( score < min )
            min = score;
    }
    return min;
}
```

圖 3.2.3 Min-Max 演算法的虛擬碼

Alpha-Beta 演算法對於 Min-Max 演算法的改進就是發現無法改變上層選擇時即停止搜尋所在節點的子樹。

圖 3.2.4 即為一個較單純的例子，單數層為最大層使用方型節點，會選擇對子樹取最大值，最小層則使用圓形節點，對子樹取最小值。三個截斷的時間點都是餘下子樹已經無法對上層的選擇做出影響，左方截斷時第四層的 4 已經過小，沒有任何機會取代上層的 5；中央第四層截斷的 6 也相同，雖是同分，但也代表結果不會再改變，因為第四層是最小層，該層的分數只會往更小變化，最多是持平；右方的截斷則發生在第二層，5 已經過小無法影響根節點的 6。

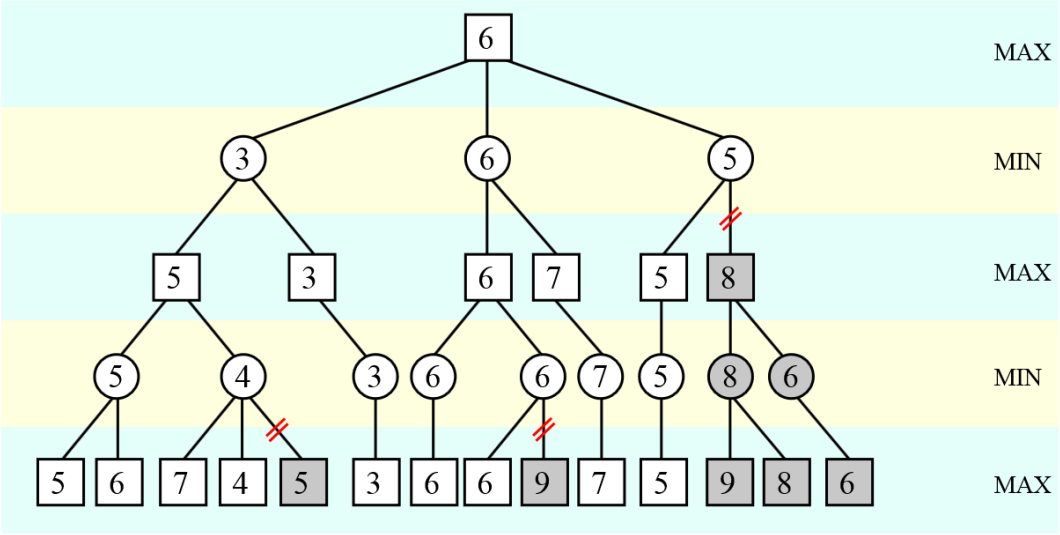

圖 3.2.4 Alpha-Beta 搜尋演算法例子

Alpha-Beta Search 使用了 α 和 β 兩個參數，α 代表所有最大層目前的最大值，β 則代表最小層目前的最小值，兩者的作用是做為界限使用，在 Alpha-Beta 搜尋過程中，兩者會交錯傳遞至下層。當在最大層取最大值發現分數大於 β 時，即代表上層取最小時不會取到目前節點，故可忽略餘下其它分支，不需搜尋，此稱 β 截斷；同理，於最小層取最小值時若發現其值小於 α，就可以放棄其它分支的搜尋，此稱 α 截斷。

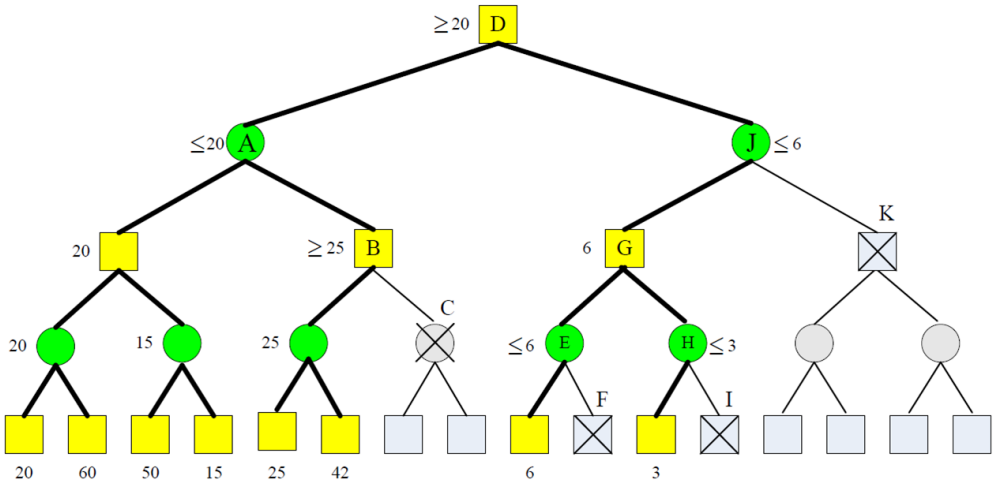

圖 3.2.5 具有 α 、 β 值的 Alpha-Beta Search Tree

圖 3.2.5 是一個具有 α 、β 值的遊戲樹，裡面分別有四次截斷，順序如下：

1. C 點因為 B 取值 25 ≧ 20，造成 β 截斷。

2. F 點因為 E 取值 6 ≦ 6，造成 α 截斷。

3. I 點因為 H 取值 3 ≦ 6，造成 α 截斷。

4. K 點因為 J 取值 6 ≦ 20，造成 α 截斷。

在此我們可以發現，α 與 β 值的收斂速度影響了截斷的發生時機，若收斂的越快則能夠截斷更多的子樹，增強搜尋速度。而 α 與 β 值的收斂速度是由走步的產生排序決定的，若越好的步能夠越早搜到，能夠截斷的節點就越多；相反的若是由最差的走步開始逐個嘗試，則不會發生任何截斷，其整體效果比 Min-Max 還差（因為還要傳遞 α 和 β ）。

圖 3.2.5 總共截斷了 12 個節點，相較於原本的 31 個節點，使用 Alpha-Beta 只需搜尋 19 個節點，節省了約三分之一的時間，顯示了對走法有良好的排序時，能夠節省更多的計算資源。

Knuth 和 Moore 提出證明 Alpha-Beta 搜尋最少所需拜訪的節點數[6]如下：

$n = 2b^{d/2} - 1$ (d 為偶數)

$n = b^{(d+1)/2} + b^{(d-1)/2} - 1$ (d 為奇數)

上式中，n 為搜尋總節點量，d 為搜尋深度，b 為每層分支數量。

節點被分為三類：

Type 1：PV(Principle Variation)，每個分支第一次搜尋的的節點(圖上為最左子節點)皆為 Type 1。根節點為 Type 1。

Type 2：Cut Nodes。Type 1 的子節點中，除最左子節點外，皆為 Cut Nodes。

Type 3：All Nodes。Type 2 的子節點為 Type 3，Type 3 的子節點為 Type 2。

節點的示意圖如圖 3.2.6：

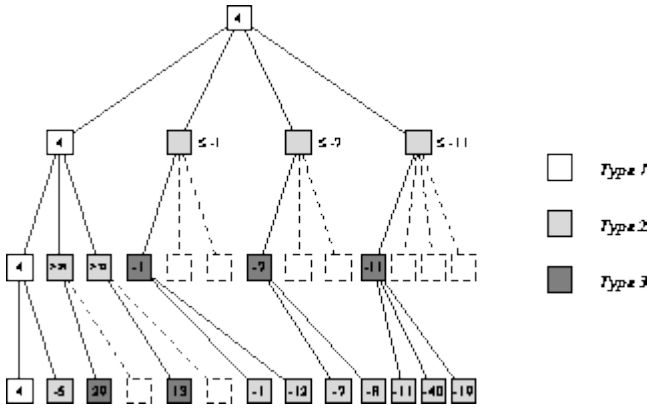

圖 3.2.6 Nodes Type

走步的排序只對 Type 1 和 Type 2 生效，能夠節省搜尋時間，Type 3 則無幫助。

```
int alphaBetaMax( int alpha, int beta, int depthleft ) {
   if ( depthleft == 0 ) return evaluate();
   for ( all moves) {
      score = alphaBetaMin( alpha, beta, depthleft - 1 );
      if( score >= beta )
         return beta;   // fail hard beta-cutoff
      if( score > alpha )
         alpha = score; // alpha acts like max in MiniMax
   }
   return alpha;
}

int alphaBetaMin( int alpha, int beta, int depthleft ) {
   if ( depthleft == 0 ) return -evaluate();
   for ( all moves) {
      score = alphaBetaMax( alpha, beta, depthleft - 1 );
      if( score <= alpha )
         return alpha; // fail hard alpha-cutoff
      if( score < beta )
         beta = score; // beta acts like min in MiniMax
   }
   return beta;
}
```

圖 3.2.7 Alpha-Beta 的虛擬碼

Alpha-Beta 搜尋虛擬碼如圖 3.2.7 所示。透過圖 3.2.7 我們可以發現，無論是 Min-Max 或是 Alpha-Beta，程式碼都分成兩個部分，分別掌握最小值端與最大值端，除了取值行為本身不同以外，α 與 β 截斷判斷也完全不同。

Knuth 和 Moore 在 1975 年提出了 Nega-Max 搜尋演算法，將最小值與最大值整合在一起，其核心概念為：在一個零和遊戲中，對我方有利的走步，同時也是對敵方不利的走步。

Nega-Max 虛擬碼如圖 3.2.8 所示。

```
int negaMax( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves)  {
        score = -negaMax( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
```

圖 3.2.8 Nega-Max 搜尋的虛擬碼

更進一步，NegaMax 形式的 Alpha-Beta Search 虛擬碼不只將最大最小兩端作整合，α 與 β 在往下層傳遞時也能以相同的方式改變。下層分數取負值時，同時將 α 與 β 順序對調並且加上負號，即能保持相同的作用，在這種情況下，α 截斷的行為變得與 β 截斷的類似，消除了於 Min-Max 基底下 Alpha-Beta Search 演算法 α 截斷與 β 截斷行為的不同，讓程式碼更簡潔也更易維護。

其虛擬碼如下。

```
int alphaBeta( int alpha, int beta, int depthleft ) {
   if( depthleft == 0 ) return quiesce( alpha, beta );
   for ( all moves)  {
      score = -alphaBeta( -beta, -alpha, depthleft - 1 );
      if( score >= beta )
         return beta;   //  fail hard beta-cutoff
      if( score > alpha )
         alpha = score; // alpha acts like max in MiniMax
   }
   return alpha;
}
```

圖 3.2.9 NegaMax 形式的 Alpha-Beta Search 虛擬碼

與傳統棋類不同，暗棋多出了一個翻子的動作，也就是可以選擇不走子去翻暗子，而傳統的 Alpha-Beta 必須選擇一個走步以讓搜尋繼續進行下去，無論走步的結果多差。所以我們在搜尋裡面加入了空步，即讓對方直接進行下一次動作，以代表翻棋動作的執行。在執行空步的結果比走步還好的時候，就是代表該翻棋了，此時我們就會讓程式去執行翻棋的動作。

搜尋中我們不考慮直接加入翻棋的行動而選擇以空步取代的原因是，翻棋的期望值計算會導致分支度過大，並且準度有限， 同時α與β值的設定也是一個問題，種種因素下我們決定將翻棋的計算獨立在 Alpha-Beta 之外。

3.2 審局函數

審局函數在搜尋中也占據非常重要的地位，由於每次進入到葉節點都需要打分數，所以審局函數一定要快速，分數又扮演著辨別好壞的腳色，所以正確性也要有一定的要求。審局分的正確與否的重點在於是否有鑑別度，能不能把好步與

壞步分開，這與設計者的棋力相關，在子力上的設定是否符合比例原則，或者在某些情況時給予額外的加分，都會影響到其正確性。而快速的要求則是因為棋類對弈都有時間限制，逾時則會被判負，若審局函數造成搜尋過慢，則只能降低深度來減少搜尋時間，但同時棋力則會相對降低。

審局函數的種類一般分為動態審局函數以及靜態審局函數，其分別就是分數是否隨著盤面變化而改動，動態審局函數會根據不同的盤面給予棋子不同的子力，一般而言越強大的子分數越高。以暗棋而言，若對方帥未被吃，我方的卒的分數就會相對較高，隨著卒越來越稀少，其價值越高；若已成功吃到帥，則卒的價值就會瞬間降至最低點。但實際上對局時的情形變化很大，目前仍很難精確去掌握，若調整錯誤導致下了錯手反而有可能得不償失。相反的靜態審局函數就是依照基本的兌子原則去設計子力分，讓該分數在大部分的情形下都對，不需要在對局中調整，程式好寫且執行速度快。

目前的審局函數分為兩種，一種是對整體盤面依照所剩子力計分，可以大致上評判目前的局勢孰優孰劣；而另一種方式則是漸進式更新，單一盤面起始都以 0 分計算，場上發生吃子步時，針對吃或被吃的子加減分，這種評分方式只有在不同的盤面間相互比較才有效。

本程式目前採用漸進式更新審局函數，也就是從零分開始，吃子加分，被吃子減分的方式。因為傳統的方式是每一次都結算盤面上的子力分布，我方為正分、敵方為負分，計算後分數越高則表示局面對我方越有利，傳統的方法在每個葉節點都需要做大量的結算以獲得分數，這樣的計算量太大，且暗棋有很多的盤面最後沒有吃子，也就是分數毫無變化，這樣計算就是做白工了。

所以目前本程式的作法是有吃子或被吃子的狀況發生時才調整分數，如此一來未吃子時盤面就是 0 分，不需要任何額外計算，非常節省計算時間。

3.3 後續發展

電腦對局已經過長時間的發展，也演化出許多可用在各種棋類的演算法。但這些棋類都有一個共通點，他們大都是完全資訊遊戲，而暗棋卻是不完全資訊遊戲，多了一個翻棋的動作，使得適用於其他棋類的演算法放到暗棋上時顯得格格不入，要使用的話必須將翻棋考慮進去。翻棋時想知道哪個位置最好，必須計算期望值，需要強大的計算與記憶能力，這是電腦的強項，在演算法足夠好的情形下，只論翻子的話電腦應可以翻得比人類更精準，這是暗棋未來大有可為的一個部份。

目前暗棋距離被列為正式比賽項目已過兩年了，期間有各種新的作法出現，但仍沒有很好的解決翻棋與走子抉擇的理想方案，許多可走可翻的盤面仍然無法歸納出一個較好的解答模式；殘局的處理也還處於瓶頸，稍微複雜的殘局即無法走出正確解，可能反勝為敗，從這兩個未解決的難題，可看出暗棋的發展空間仍無限寬廣。

第四章改良方法與步驟

4.1 基本概念

暗棋程式的棋力強弱，與搜尋的深度息息相關，而搜尋的深度又受電腦的運算速度限制，即使無法搜到真正的葉節點來得到正確結果（即分出勝負），整體趨勢而言仍是越深層的搜尋能夠得到越正確的結果，所以我們核心的目標就是使用各種手段為程式加速，來加強棋力。

4.2 重複盤面處理

遊戲樹的搜尋過程中，會發生許多選擇不同走步，但卻進入到相同的最終盤面的情況，如以下所示：

■			
卒	俥	兵	■
■			
相			
		■	
■			
包		兵	
■	■	將	■

圖 4.2.1 起始盤面

■			
俥		兵	■
■			
包			
		■	
■			
■	■	兵	■

圖 4.2.2 結果盤面

圖 4.2.1 與 4.2.2 的盤面由紅方先行，走 c2-c1(兵吃將)，a2-a5(包吃相)，b7-a7(俥吃卒)，與 b7-a7(俥吃卒)，a2-a5(包吃相)，c2-c1(兵吃將)之走步順序相異但結果相同。

在這種情形下，如果能夠將之前已經拜訪過的盤面結果存下來，到了再次遇到相同盤面時，就可以將之前存的結果取出使用，不需再往下搜該盤面的子樹，節省大量計算資源。

我們使用 Transposition Table 來儲存搜尋中所得到的資訊，但是一個節點的資訊量其實並不小，其中包含了盤面 32 顆棋子狀態、盤面分數、目前行動方、節點所在深度……等等資訊，若要全部儲存的話，除了耗用大量空間以外，存取速度也是一個必須考慮的部分，儲存的資訊越多，比對的成本也會相對提高。

4.3 Zobrist Hashing

因此我們決定使用 Zobrist Hashing 的技術來取代平常的儲存方式，這個方式可以節省大量的儲存空間以及存取時間。

Zobrist Hashing[7]是由 Albert Lindsey Zobrist 在 1969 年提出的一個雜湊函數，適合使用於各種棋類遊戲的 AI 上，其原理是將大量的隨機亂數以邏輯 XOR 運算相互組合，來產生新的亂數以表示各種盤面。

作法是在進入搜尋之前，預先設定好 32 Bits 的無號整數，依照遊戲類型給予數量不等的亂數，基本必須要有(棋子種類 × 棋盤格數)個亂數，而其它的資訊則視使用者需求而定。

以暗棋舉例，雙方共有 15 種棋子狀態(紅黑方各七種加上暗子，空格為 0)，而位置有 32 個，故我們需要產生共 480(=15×32) 個亂數來作為基底。

該亂數表以一個二維陣列儲存，表示方法為 Array[棋種(0~14)][位置(0~31)]。如果要把包放置到 27 號位置，我們會把代表現在盤面的亂數與 Array[包][27]作 XOR 運算，如此則得到我們要的結果。

範例:(如圖 4.3.1)

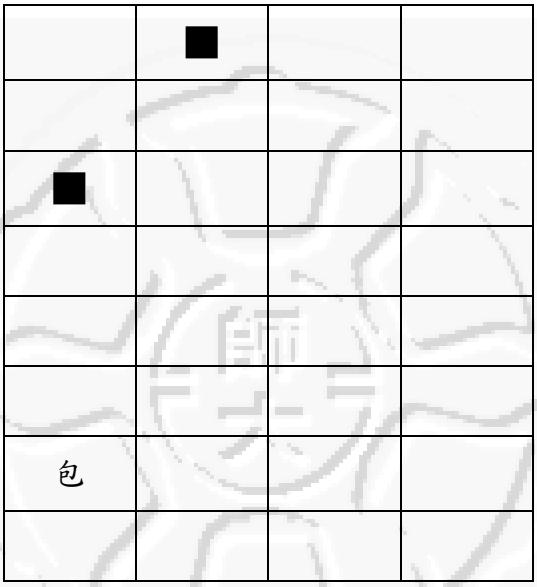

圖 4.3.1 範例盤面

圖 4.3.1 該盤面的 Key 值即為 Array[暗子][2] ⊕ Array[暗子][11] ⊕ Array[包][27]。

有了上述的資料結構後，我們在開局時會預先設定好棋盤上所有位置皆為暗子的盤面，採取漸進式更新，雙方的每一手棋都需要 2~3 個步驟來對 Zobrist Key 作更新，其步驟如下：

1. ZKey ^= Array[棋子種類][棋子原始位置]。

2. ZKey ^= Array[棋子種類][棋子目標位置]。

3. 若該走步為吃子步，則還需要做 ZKey ^= Array[被吃子種類][被吃子位置]。

4. 翻棋的話則比照 1、2 辦理，在相同位置消去暗子，將翻出子放入。

在我們的作法中，只儲存 Zobrist Key 是不夠的，我們還需要一些變數代表目前的行動方，該盤面的搜尋深度，該次搜尋的最大深度.....等資訊。如此最後產生 Zobrist Key 才能夠在 Transposition Table 上使用。

4.4 Transposition Table

實際建立 Transposition Table 時，我們會發現搜尋的節點數非常多，從 Transposition Table 的使用效率出發，其大小越大當然是越好，能夠儲存更多的資訊以供查詢，但電腦的記憶體空間大小是有限的，不可能提供一個無限大的空間供 Transposition Table 使用。我們運用 Zobrist Hashing 技術來提升 Transposition Table 的存取速度，但由於 32 Bits 的變數完全不足以表示所有的棋盤狀況，並且記憶體大小也不足以支撐這麼大的 table，於是我們將會面臨 Collision 的問題。

所謂的 Collision，就是同一組 Hash Key 卻代表了不只一種意義，從暗棋上來說，即是代表這組 Hash Key 可能代表不只一個盤面，但是 Transposition Table 的 Entry 裡面只會存一組資料，此種情形下仍然存取 Hash Table 內的資料會讓搜尋時的分數產生錯誤，最終則會導致錯誤的決策，選擇較差的走步。

以下是 Collision 的一個簡單例子：

- 00010100 ⊕ 10001001 = 10011101
- 10010010 ⊕ 00001111 = 10011101

此例中我們可以發現，同一組 Key 由兩組不同的變數來組成，這就意味著兩種不同的盤面有相同的 Key，會造成嚴重的誤判。

4.5 Collision 避免

目前製作 Transposition Table 大多都會面臨 Collision 的問題，而現行的改善方法有下列幾種：

1. 增加 Hash Table 的大小
2. 增加儲存資訊量，最完整則是儲存完整盤面加上其他資訊
3. 使用多層的 Table，即發現有資料存在時會導入第二層再作檢查。

其中增加 Table 的大小當然是簡單又有效的方式，不過同上一段說的，電腦的記憶體大小是有限的，並且 Table 越大時其成本越高，但每擴大一倍則可以降低一倍的 Collision Rate。

而第二種方法，若增加資訊儲存量同樣也能夠確保 Collision Rate 的下降，但同時儲存成本以及檢查成本則會大幅上升，這在要求搜尋速度的棋類遊戲搜尋顯然是相當不適用。

雙層或多層的 Hash Table，若不增加儲存額外資訊的話，即使都有將值存入，但發生 Collision 時，我們沒有辦法得知哪一筆資料才是正確的，所以仍需存入額外資訊判別哪一個才是正確的資料，Hash Key 只是檢查的第一道關卡。

綜合目前許多程式的作法，普遍傾向於承擔 Collision 的風險，因為 Collision Rate 通常在數千到數萬分之一，看起來相當低，並且發生 Collision 的節點還有不一定會出現在主要分支上，劇烈影響搜尋結果導致錯誤。

通過 Transposition Table，我們得以進行更深層的搜尋，但在同時由於搜尋節點數量的急遽增加，Collision 所累積的風險會導致每次的搜尋都有更多的機會出錯，而一盤棋只要出了數次錯誤通常就大勢已去，這是不允許的。

經過多方測試，我們發現一個方式可以有效的針對 Zobrist Hash 來做 Collision 的檢測，並且實作簡單效果好。其敘述如下。

這個方式的核心概念是針對所存的資料做二次檢查，但與前面所提方法不同的是，我們所額外儲存的東西簡單而且可以有效降低 Collision Rate。

這個方式就是使用 64 Bits 的 Zobrist Key 來取代原本的 32 Bits，將 64 Bits 拆為兩份 32 Bits，將其中一組作為檢測碼，另一組則為原本的 Zobrist Key，雖然會占用較原本多一倍的空間，但可大幅降低 Collision Rate，較單純擴大一倍 Table 大小的效果好很多。

舉一個 64 Bits 的 Zobrist Key 的例子如下：

0x42f0aae1469c31d，其中 0x42f0aae 為 Key，0x1469c31d 為檢查碼。

本程式的實作上只使用 16 Bits 作為檢查碼，但這樣最佳情形下就可以將錯誤率降至原本的 $1/2^{16}$，約為原本的六萬分之一。實驗證實這個改善方式非常有效，降低後的 Collision Rate 在極端複雜盤面下仍不會超過三十萬分之一。

以下是使用一個複雜盤面搜尋 12 層的結果，其中除了 Alpha-Beta 外也加入了其它必須的搜尋，Transposition Table 總大小為 4G bytes (不含監控結構)，其盤面與結果如下：

			兵
		相	
	帥		俥
	兵		
仕		仕	
車	士	卒	卒
	象		士

圖 4.5.1 實驗測試盤面

真實節點數	87532393706
拜訪節點數	309120799
節省搜尋量	0.9964
Transposition Table 使用率	0.123865
第一層碰撞總次數	1193176
真實碰撞總次數	202
Hit Count	166958205
第一層碰撞率	0.003859902
Table Hit Rate	0.536227
真實碰撞率	0.0000007
改善率	5906.81
使用時間(秒)	76.7

圖 4.5.2 Collision Avoidance 實驗結果

圖 4.5.2 中，第一層碰撞代表初次檢查時通過但未經過二次檢測的存取行為，這意味著在沒有加上第二次檢查前，這些錯誤的資料會直接被存取，而將近千分之四這麼高的錯誤率將會很容易導致遊戲樹的許多節點被省略不搜索，導致錯誤的結果。

真實碰撞則是在正式程式執行中不會發現的錯誤，經過二次檢測後，這樣的錯誤降到只剩下 202 個，改善率約 5900 倍左右，雖與最佳情形下的六萬倍有些許差距，但這樣的結果仍然是相當良好，並且表中也顯示未使用 Table 時真正必須拜訪節點與使用 Table 後的節點數量的差距，我們節省了超過 99.5%的搜尋時間，這證明這個技術是非常實用的。

接下來說明我們選擇使用 4G bytes 大小的 Table 原因，以下是另一個實驗即是針對 Table 大小的實驗，同樣使用上述的盤面進行 12 層的搜尋，但針對各種不同大小的 Table Size 作實驗：

Table Size	空間使用率	碰撞率	時間（秒）
4G	0.123865	0.0000007	76.7
2G	0.233030	0.0000016	76
1G	0.410903	0.0000030	76.4
512M	0.653651	0.0000059	88.5
256M	0.879369	0.0000117	121.8
128M	0.986821	0.0000112	172.2
64M	0.999819	0.0000104	245.3
32M	0.999985	0.0000109	388.9

圖 4.5.3 Table Size 實驗結果

圖 4.5.3 中可以很明顯看到,隨著 Table 空間使用率的上升,不但 Collision Rate 上升，搜尋時間也隨之大幅成長，這說明了 Table 大小對程式棋力也有相當影響力，但同時我們在這個實驗也能看到，即使在只有 32M 大小的 Table 上，碰撞率也只有約十萬分之一，整體而言非常成功。

而最終決定 4G bytes 這個大小是由於在暗棋目前的比賽中，雙方每走一步就必須重新呼叫搜尋程式，故 Table 的空間就必須重新配置，4G bytes 這個大小是我們覺得在現行程式下平衡性最好的，不只在 Collision Rate 上表現良好，且 Initialize 成本也相對沒那麼高，在許多實驗中我們也嘗試過使用更大的 Table，但高昂的 Initialize 成本導致每一次開啟程式都需數秒來 Initial，而比賽中搜尋時間是有限的，若每一手都會浪費數秒的話，累積的成本相當可觀。

4.6 同分步情形處理

同分步的情形分為兩種，一種是所有盤面都是 0 分，什麼子也沒有吃到，這時我們就需要找出一個趨勢，希望我方大子可以去壓迫對方較小的子，所以我們採取單方搜尋的方式，希望能給程式一個趨勢來做出行動，但是單方搜尋會有一些普通 min-max 搜尋不會產生的問題，所以本程式在針對走步產生時就引入了前文中控制域的概念。

單方搜尋只產生我方走步,目的是讓我方棋子可以看到未來的趨勢,搜尋中，由於只產生我方走步，故可能會經過對方所掌控的位置。在走步產生時程式即會檢查周圍是否存在對我方具有威脅的棋子，若存在，則此走步為非法步不產生。需要檢查的部分除了周圍以外，還有炮的威脅問題，如圖 4.6.1 依照盤面將所有炮可能出現的位置逐一嘗試，這樣速度太慢，我們採取反向的作法，將目前欲產

生之子當作炮來產生走步,如此最多只有三個走步出現,若出現走步的目標是炮，則代表選擇此步會被炮打到，即為非法步不產生。

■	■	車	■
卒		○	
■		傌	
		帥	
■		仕	
■	■	○	
包	卒	包	卒
■		○	■

圖 4.6.1 炮的威脅產生示意圖

另一種則是有吃到子或是被吃子，是非 0 分的同分步，此時我們就可以針對吃到子的深度給予額外的調分以判別高下，越早吃到子的走步應該要比較晚吃到子的多分。例如：第三層吃到就-3 分，第五層就-5 分，如此一來第三層吃到子的走步就會比第五層吃到子的走步高 2 分，雖然兩步都能夠吃到子，但如此做以後程式就會選擇較快吃到子的那一步。

4.7 翻棋

翻棋是暗棋中很重要的一個動作，目標就是在所有的暗子中選擇一個最好的點去翻。透過程式來做，就是選取一個期望值最高的子去翻，但若僅僅是翻出來

的話，很有可能發生翻出來後馬上被吃的情況。

為了能夠求得更精確的答案，我們決定將所有暗子代入所有可能翻出的子去搜尋六層走步，其分數累加做為該位置的分數，如此就可以得到這個位置的期望值，而最後選擇分數最高的一子來翻。

4.8 循環盤面處理

暗棋中，陷入循環盤面（通常為某一方長追）時，通常意味著這盤棋有很大的機會會和棋，但是在許多的和棋盤面中，由人類來看應該要是某一方獲勝的，所以本程式在偵測到此情形時，對即將進入循環盤面的走步由漸進式更新審局分數改為雙方子力差的分數，如此可以評斷出進入循環走步時，對我方是否有利：若評斷出來我方為大負分，代表這盤棋我方的贏面很小，應該要讓它進入循環盤面繼續循環以求和；反之若為大正分，則代表我方優勢極大，可以考慮放棄被長追的子，以利打開局面。

帥			
	卒		
		相	仕
	象		仕
		車	

圖 4.8.1 循環示意圖（輪黑）

由圖 4.8.1 的情況是紅方已佔極大的優勢，黑先行時動卒左移抓帥，紅方為了不損失帥而選擇逃帥往右，黑方會再動卒右移抓帥，緊追不捨，如此就會形成循環，而造成和局，當然對紅方來說該贏而未贏就是一個錯誤的決策。在本程式偵測到循環時，我方執紅則會放棄逃帥，進而上仕逼象，之後便可一個個吃掉獲得勝利；若是執黑，則計算出來會是我方不利，會選擇繼續進入循環以求和局。

第五章 結論與未來研究方向

5.1 結論

本研究中已實作出一個暗棋程式 Darkness，完成本論文中所述的各項技術，並於 TAAI 2012 與 TCGA 2013 舉辦的比賽中獲得良好的表現，利用 Transposition Table 大幅加強運算速度顯然對於棋力的上升非常有幫助，在能夠看得比別人深的情況下，所掌握的情報足以讓某些可能會和的局面走到勝利，並且其他加強的手段（如循環判別）也確實有起到作用。

5.2 未來研究方向

在殘局的處理上仍然有許多盤面無法得知正確的解，這些盤面由人類棋士來下的話非常容易，但對本程式來說，處於子太多的情況下就無法搜尋至足夠深而得到正確結果，期望未來能夠開發出讓關鍵子做出壓迫行為的作法。

另外在少子殘局的部分，雖然無法解出的盤面較多子殘局少，但仍然存在且為數眾多，個人認為在少子殘局上，只要有恰當的動態配分的演算法就能解決很多盤面。並且一個良好的動態配分的演算法不只是使用在殘局上，即使在中盤的審局上也非常有幫助。

參考文獻

[1]謝曜安，電腦暗棋之設計及實作，國立台灣師範大學資工所碩士論文，2008。

[2]賴學誠，電腦暗棋程式與經驗法則之配合與實作，國立東華大學資工所碩士論文，2008。

[3]謝政孝，暗棋中棋種間食物鏈關係之探討與實作，國立臺灣師範大學資工所碩士論文，2010。

[4]勞永祥，電腦暗棋之人工智慧改良，國立臺灣師範大學資工所碩士論文，2011。

[5]施宣丞，電腦暗棋程式 DarkCraft 的設計與實作，國立臺灣師範大學資工所碩士論文，2012。

[6] D. E. Knuth and R. W. Moore, "An Analysis of Alpha–Beta Pruning", Artificial Intelligence 6(4), pp. 293–326, 1975.

[7] Albert Lindsey Zobrist, "A New Hashing Method with Application for Game Playing", Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1969.

[8] J.-C. Chen, T.-Y. Lin, S.-C. Hsu and T.-S. Hsu, "Design and Implementation of Computer Chinese Dark Chess Endgame Database", Proceeding of TCGA Workshop 2012, Hualien, Taiwan, pp. 5–9, 2012.

[9] Shi-Jim Yen, Cheng-Wei Chou, Jr-Chang Chen, I-Chen Wu and Kuo-Yuan Kao, "The Art of the Chinese Dark Chess Program DIABLE", Proceeding of TAAI Workshop 2012, Tainan, 2012.

[10] Shi-Jim Yen, Shih-Yuan Chiu and I-Chen Wu, "MODARK Wins Chinese Dark Chess Tournament," ICGA Journal, vol. 33, no. 4, 2010, pp. 230-231, 2010.

[11] B.N. Chen, B.-J. Shen and T.S. Hsu, "Chinese Dark Chess", ICGA Journal 33(2), pp. 93–106, 2010.