行政院國家科學委員會補助專題研究計畫 ■ 成 果 報 告
□期中進度報告

六子棋與K子棋之研究

The Study of Connect6 and K-in-a-row games

計畫類別：■ 個別型計畫　　□ 整合型計畫
計畫編號： NSC 95-2221-E-009 -122 -MY2
執行期間：2006 年　08 月　01 日至　2008 年　7 月　31 日

計畫主持人：吳毅成
共同主持人：
計畫參與人員：林秉宏、林宏軒、曾汶傑、吳冠翬、陳靖平

成果報告類型(依經費核定清單規定繳交)：□精簡報告　■完整報告

本成果報告包括以下應繳交之附件：
□赴國外出差或研習心得報告一份
□赴大陸地區出差或研習心得報告一份
□出席國際學術會議心得報告及發表之論文各一份
□國際合作研究計畫國外研究報告書一份

處理方式：除產學合作研究計畫、提升產業技術及人才培育研究計畫、列管計畫及下列情形者外，得立即公開查詢
□涉及專利或其他智慧財產權，□一年□二年後可公開查詢

執行單位：國立交通大學資訊工程學系

中　華　民　國　九十七　年　　十　月　　二十 日

中文摘要

本計畫主持人在 2005 年第十一屆的國際電腦賽局發展研討會發表一篇論文提出一系列新的 k 子棋遊戲。此遊戲被一般化為 Connect(m,n,k,p,q)：黑白兩方玩家各持黑白子，黑方先下 q 子，然後輪流各下 p 子在 mxn 棋盤上；先連成 k 子者獲勝。這一系列新的 k 子棋中，最重要的就是六子棋：Connect(19,19,6,2,1)，並正式命名為連六棋，英文名稱為 Connect6。六子棋的重要特性在於雙方每次下玩一子後，其盤面子數必然比對方多一子，這平衡優勢的特性有助於其公平性；傳統五子棋（一般規則）不具有此特性，因而有明顯的不公平現象。

由於這是全世界第一篇論文提出六子棋，對研究人員來說，這新遊戲尚有相當多未解的研究問題(open problems)，是一個相當值得研究的新問題。本計畫兩年的研究工作項目中，第一年的研究工作項目如下：

1. 收集及分析相關資料。
2. 研究設計基本的六子棋人工智慧程式。
3. 研究提出適用於六子棋的新 null-move heuristics 方法，並以此證明六子棋一些棋型為必勝棋型。
4. 提出並設計適用於六子棋的新 dependency-based search 技術。

第二年

5. 研究設計適用於六子棋的 threat proof search 技術。
6. 研究利用程式之大量搜尋，來產生新的六子棋詰棋或開局定石。
7. 利用這些技術發展棋力強大的六子棋程式。

從上述的研究中，我們的研究獲得一些重大成果，就是**用這技術證明了不少高手下過的 TX-H9 開局（參見 Figure 13(a)）是黑必勝**。這對六子棋界棋士而言，是一個震撼的結果。同時在本報告撰寫完成前（民國 97 年 10 月 3 日），本計畫主持人亦利用此技術改良的**「交大六號」六子棋程式獲得第十三屆國際奧林匹亞賽局競賽冠軍，為我國獲得唯一的一面金牌及獎牌。**

對這些新遊戲尤其是六子棋，由於其規則簡單、遊戲公平、及玩法複雜，有機會成為一項由台灣研究發展出，並與五子棋一樣普及全世界的遊戲。這對提升國家形象及知名度亦有相當的助益。

關鍵字詞：人工智慧、五子棋、六子棋（連六棋）、詰棋、開局定石、搜尋技術、迫著策略、null-move heuristics、dependency-based search、proof-number search。

Abstract

In the 11th Advances in Computer Games Conference (ACG'11), the principle investigator (PI) of this project presented a paper that proposed a new family of k-in-a-row games, called Connect(m,n,k,p,q). Two players, say B and W, alternately put p stones, black and white respectively, on unoccupied intersections of an mxn board, except for that B initially places q stones. The one who gets k consecutive stones first (horizontally, vertically or diagonally) of her own wins. Among these games, Connect(19,19,6,2,1) is most interesting to this paper and is named Connect6. For Connect6, since one player always has one more stone than the other after making each move. This balanced advantage makes it promising as a fair game, unlike the traditional Gomoku (five-in-a-row) apparently favoring B.

Since the game was newly introduced, there have been many open problems. This project is to investigate these problems. In this project, the research items of the first year are as follows.

1. Collect and analyze the related data.
2. Study and design a primitive artificial intelligence program for Connect6.
3. Study and propose a new null-move heuristics technique for Connect6, and prove some winning patterns based on the technique.
4. Propose and design a new dependency-based search technique for Connect6.

The research items of the second year are as follows.

5. Study and design a proof-number search technique for Connect6.
6. Generate new Tsumego and Joseki for Connect6 by running programs as described above.
7. Study and design a strong artificial intelligence program based on the above search techniques.

In this project, we proved that a popular Joseki, TX-H9 Joseki, won by black (the first player). And, we also wrote a program, named NCTU6, and won the gold of the 13th Computer Olympiad in 2008.

For Connect6 as well as some potentially fair connect games, since they have the features of simple rules, fairness and high complexity in playing, they has high chances to be popular as Gomoku all over the world. This will be of great help to promote the image and name of our country.

Keywords：Artificial intelligence, Gomoku, Connect6, Connect games, Tsumego, Joseki, Search techniques, threat-based search strategy, null-move heuristics, dependency-based search, proof-number search。

1. INTRODUCTION

A new family of k-in-a-row games were introduced and presented by Wu and Huang (Wu and Huang, 2005). A *k*-in-a-row game is called *Connect(m,n,k,p,q)* as follows. Two players, say *B* and *W*, alternately place *p* stones on an *m*×*n* board in each turn except for that the first player places *q* stones for the first move. The player who gets *k* consecutive stones of his own first wins. Games in the family are also called *Connect* games. Obviously, Go-Moku in the free style is *Connect(15,15,5,1,1)*. For simplicity, *Connect(k,p,q)* denotes the game *Connect(∞,∞,k,p,q)*, played on infinite boards.

Among these Connect games, Connect(6,2,1), also named *Connect6*, is most interesting due to fairness and game complexity. Since Connect6 was presented (Wu and Huang, 2005), tens of thousands of players, including many Renju dan players, have played this game in a Taiwan game site (ThinkNewIdea Inc., 2005). So far, these dan players still have not been able to identify who the game favors.

Like Go-Moku, threats are the key to win for Connect6. Since Connect6 was presented, many players have learned how to win by making double-threat moves, each with two threats, as defined in Section 2. Recently, Lee (Lee, 2005), a Renju 3-dan player, presented some Josekis and Tsumegos for Connect6. Among them, he thought that W wins in the opening position in Figure 1 (below), and also showed many subsequent variations from this position. Three subsequent variations are shown in Figure 2, and winning sequences from these variations are respectively shown in Figure 3. These winning sequences indicate the importance of making single-threat moves in Connect6. For example, the moves at (14,15) and (18,19) in Figure 3 (a), (10,11) and (22,23) in Figure 3 (b), and (14,15) and (22, 23) in Figure 3(c).

This paper investigates new threat-based proof search techniques for Connect6 programs. These search techniques include dependency-based search and threat proof search. Based on these techniques, this paper solves many positions including those in Figure 2. In addition, the program with these new techniques is apparently stronger than our old program, which uses only double-threat moves in threat-based search and can already beat 70% players in the above game site.

This paper is organized as follows. Section 2 reviews the game Connect6 and some related definitions and properties. Section 3 describes threat-based proof search for Connect6 programs. Section 4 reports the experiments on solving positions. Section 5 concludes our work.

2. DEFINITIONS AND PROPERTIES

Definition 1. For Connect6, assume that one player, say W, cannot connect six. B is said to have *t* threats, if and only if W needs to place *t* stones to prevent B from winning in B's next move. ▌

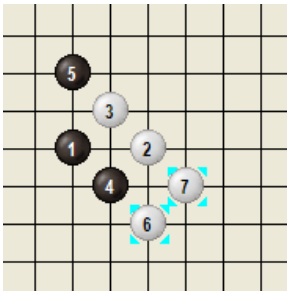
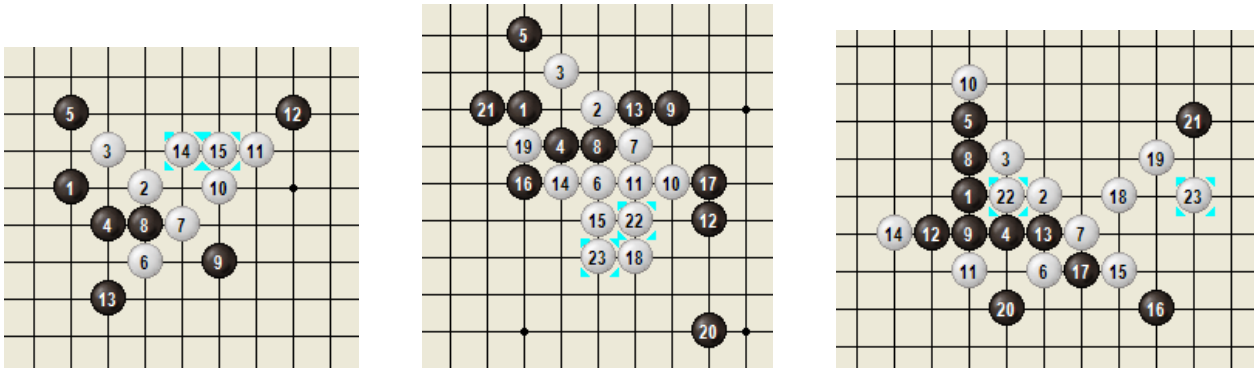
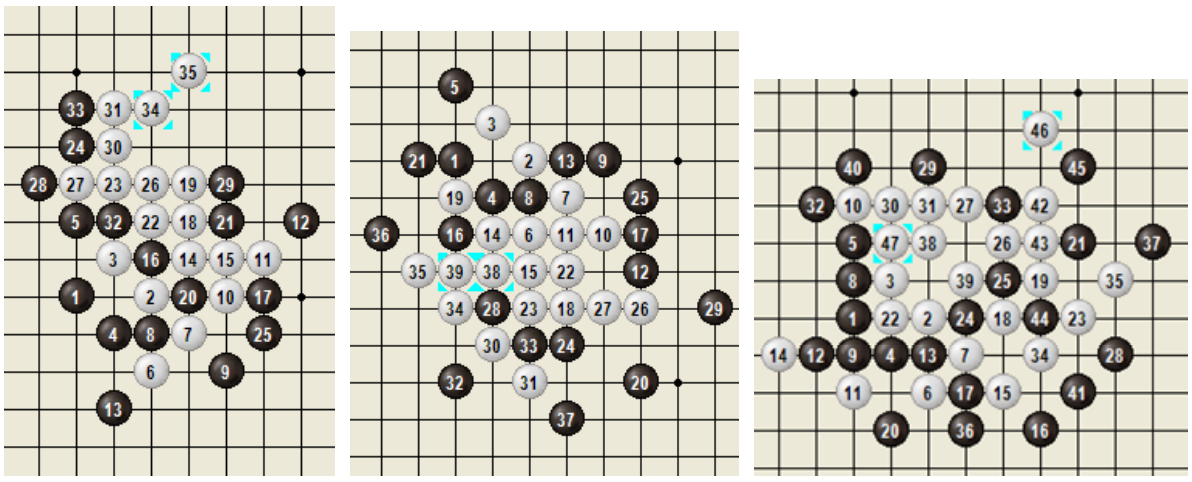


Figure 1: An opening position favoring W given by Lee (Lee, 2005).



(a) (b) (c)

Figure 2: Three subsequent variations from the position in Figure 1.



(a) (b) (c)

Figure 3: Winning sequences from the positions in Figure 2.

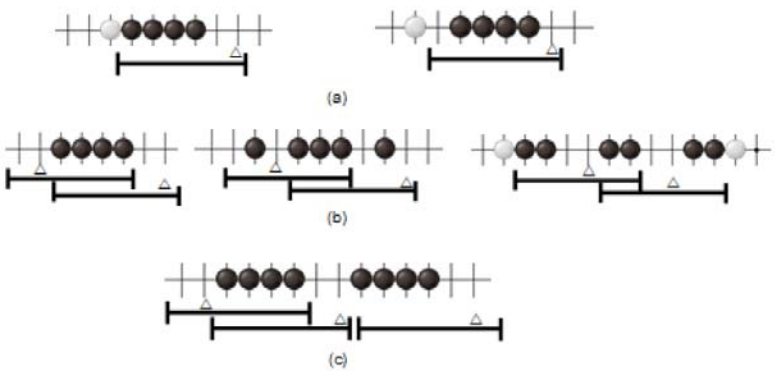


Figure 4: Threat patterns for Connect6. (a) One threat, (b) two threats and (c) three threats.

Threats are defined in Definition 1 as above. Figure 4 shows line patterns with one, two and three threats. A move is called a *single-threat move*, if there is one and only one threat after the move, and a *double-threat move*, if two. Based on the above definitions, we can derive the following properties for Connect6 (Wu and Huang, 2005).

1. One player can win the game if the player has three threats.
2. Consider some line on the board only. Placing one stone on this line increases threats by at most two.

Based on the second property, *live* and *dead* threats are defined in Definition 2.

Definition 2. In Connect6, a line includes a dead-*l* threat for one player, say B, if B only needs to add extra 4−*l* stones to create one threat. Similarly, a line includes a live-*l* threat for

B, if B only needs to add extra 4−*l* stones to create two threats. ▌

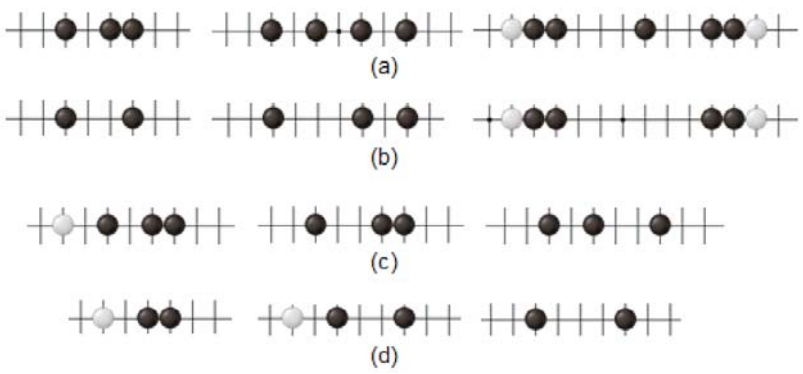


Figure 5: (a) Live-3, (b) live-2, (c) dead-3 and (d) dead-2 threats for Connect6.

Figure 5 illustrates live-3, live-2, dead-3 and dead-2 threats. The four threats are also called *potential threats* or *P-threats*, since one more move can create one or more threats. Among potential threats, live-3, live-2, dead-3 threats are also called *highly potential threats* or *HP-threats*, since one more move can make two or more moves. Among HP-threats, live-3 and dead-3 threats are also called *HP3-Threats*. For a HP3-threat, placing on one square of it can make one or more threats.

3. THREAT-BASED PROOF SEARCH FOR CONNECT6

Many proof search techniques, such as dependency-based search (Allis, 1994; Allis, Herik, and Huntjens, 1995), proof-number search (Allis, 1994; Allis, Meulen, and Herik, 1994), lambda-like search (Thomsen, 2000; Cazenave, 2001; Cazenave, 2003), were proposed to reduce the search tree size and prove whether a game position is a win for one player.

For simplicity of discussion, this player who is to be proved to win is called *Attacker* and the other *Defender* in the rest of this paper. For a game position *P*, a sequence of subsequent moves is called a *move sequence from P*. A move sequence from *P* is called a double-threat sequence from *P*, if the subsequent moves made by Attacker are all double-threat moves. A move sequence from *P* is called a *mixed-threat sequence* from *P*, if all the subsequent moves made by Attacker are either single-threat or double-threat moves.

Subsection 0 presents a threat space search framework for Connect6. Subsection 0 presents some schemes to detect whether positions are in a quiescent state. Subsection 0 presents dependency-based search for Connect6, while Subsection 0 presents the technique of threat proof search based on null move heuristics to find all the implicit threats.

Threat Space Search Framework

For Connect6, similar to other Connect games, such as Go-Moku, it is important to prove by a search tree whether Attacker wins in a game position. Such a search tree can be considered as an AND-OR search tree (Allis, 1994), instead. For a position, the corresponding search node returns the value **`true`**, if the position is a win for Attacker; and it returns the value **`false`**, if the position is a draw, a win for Defender or a quiescent. A position is a draw when all stones are filled up in the board without winners. If an infinite board is used for Connect6, there is no draw game. So, for simplicity, this paper ignores draw games, and considers quiescent positions, instead. A position is *quiescent* when it is *hard* for Attacker to win from the position. The routine **`IsQuiescent`**, described in more detail in Subsection 0, is designed to judge whether the current position is quiescent.

The positions to be moved by Attacker are represented as OR nodes, while the positions to be moved by Defender are as AND nodes. The operation steps of these search nodes are as follows.

1. If Attacker wins for this position immediately, return **`true`**.
2. If Defender wins for this position immediately, return **`false`**.

3. Call the routine **IsQuiescent**. If the routine returns **false**, return **false**.
4. In OR nodes, call the routine **EvaluateBestMoves**, which generates a list of nodes for several *best* moves by Attacker, and evaluates each node in the list. If some node returns **true** (win for Attacker), return **true**; otherwise, return **false**.
5. In AND nodes, call the routine **EvaluateAllMoves**, which generates a list of nodes for all the moves by Defender, and evaluates each node in the list. Note that if some node returns **false**, return **false**; otherwise, return **true**.

From the above operations, we can easily obtain Corollary 1 (below) that a position is a win for Attacker if the corresponding search node returns **true**. However, this does not imply that the search node must return **true** if Attacker wins in the position.

Corollary 1. For an AND-OR search tree as described above, if the returning value the tree is **true**, then Attacker wins. ▌

For the above operations, one problem is that the routine **EvaluateAllMoves** needs to generate all the moves for Defender. The routine is time-consuming for 19x19 boards and even impossible for infinite boards. Therefore, we need to change it to **EvaluateAllDefensiveMoves**, which only generates a set of defensive moves. If it can be proved that Attacker wins for all the other moves not in the set, then Corollary 2 can be obtained. Subsection 0 describes the technique of threat proof search based on null move heuristics to find all the defensive moves. This technique can be used in the routine **EvaluateAllDefensiveMoves**.

Corollary 2. For an AND-OR search tree using the routine **EvaluateAllDefensiveMoves** as described above, assume that Attacker wins for all the moves not in the set of defensive moves, generated by the routine. If the returning value of the tree is **true**, then Attacker wins. ▌

For the above operations, another problem is how many and which best moves need to be generated for Attacker in the routine **EvaluateBestMoves**. Normally, the number of *best* moves is usually limited to a threshold, say 10. For Connect6, since one move includes two squares, there are many combinations even for double-threat moves. So, the threshold is usually high, like 30, for finding solutions. However, for such a high threshold, the program may generate many quiescent moves, which usually result in a huge tree size. Therefore, it is important to filter moves. In this program framework, the routine **EvaluateBestMoves** simply uses the routine **IsQuiescent** to determine whether to continue the search.

Quiescence Detection Heuristics

As described in Subsection 0, our Connect6 program uses the routine **IsQuiescent** to determine whether positions are quiescent, and stops searching in quiescent positions in order not to traverse a huge search tree. For finding a winning double-threat sequence, **IsQuiescent** returns **true**, unless Attacker makes a move with two threats. Such an **IsQuiescent** is fast, but too conservative to prove some winning positions as those in Figure 2. The most aggressive way to design **IsQuiescent** is to return **false** in all cases, that is, try to find all the possible results. However, this is not realistic since it expands a tremendously large tree. This subsection presents an in-between design for **IsQuiescent**, based on our empirical experiences. In order not to expand a huge search tree, this routine is still somewhat in a conservative manner, but can be more aggressive by adjusting some parameters. This routine is described as follows.

As described in Subsection 0, this routine is especially important to limit the number of Attacker's moves generated in the routine **EvaluateBestMoves**. This paper only discusses the routine **IsQuiescent** called in the AND node to be moved by Defender. In AND nodes, Defender must have no threats. Otherwise, the operation step 2 of the AND node returns **false**. Let us discuss the heuristics in the following three cases, C1, C2 and C3, respectively.

Case C1: Attacker has two threats.

This routine **IsQuiescent** simply returns **false** in this case.

Case C2: Attacker has one threat only.

This routine **`IsQuiescent`** returns **`false`**, if one of the two non-quiescent conditions C21 and C22 holds, and **`true`**, otherwise.

- Non-quiescent condition C21: Defender has dead-3 threats but no live-3 threats, while Attacker has one HP3-Threat plus at least two additional HP-threats (without counting the single threat).

  For example, the move at (22, 23) by W (Attacker) in Figure 2 (b). Since Defender has only dead-3 threats, Defender places one stone to defend Attacker's single threat, while placing the other either to make a single-threat move or to defend one HP-threat. For the former, Attacker can place one stone to reply Defender's single threat, while placing the other to make threats in the next move. For the latter, since Defender has no threats, Attacker can use two HP-threats to make a double-threat move.

- Non-quiescent condition C22: Defender has no HP3-Threats, while Attacker has at least two additional HP-threats.

  For example, the moves at (14,15) and (22,23) in Figure 2 (c). Since Defender has no HP3-Threats, Defender places one stone to defend Attacker's single threat, while placing the other to defend one HP-threat. Thus, Attacker can still make threats in the next move.

Case C3: Attacker has no threat.

This routine **`IsQuiescent`** returns **`false`**, if one of the three non-quiescent conditions C31, C32 and C33 holds, and **`true`**, otherwise.

- Non-quiescent condition C31: Defender has only one dead-3 or live-2 threat and no more HP-threats, and Attacker has one HP3-Threat plus at least two HP-threats.

  It is possible for Defender to make a threat without subsequent threats. For example, the move at (6,7) by W (Attacker) in Figure 1. As long as Attacker maintains three HP-threats including one HP3-threat, it is still likely that Attacker gets some chances to fight back. On the other hand, if Defender can increase the number of HP-threats, then Defender takes the initiative and the position becomes quiescent.

- Non-quiescent condition C32: Defender has dead-2 threats and no HP-threats, and Attacker has one HP3-Threat plus at least two HP-threats.

  For example, the move at (10, 11) in Figure 2 (c). Since Defender has no HP3-Threats, Defender places two stones either to form a threat or to defend Attacker's HP-threats. For the former, while placing one stone to defend the threat, Attacker can still place another make a new threat. For the latter, since Defender can block two HP-threats, Attacker can make threats from the remaining HP-threat.

- Non-quiescent condition C33: Defender has no P-threats.

  Since Defender cannot make threats in the next move, there is no need to worry about Defender's counter threats. This happens initially. For example, the move at (2, 3) in Figure 1.

Dependency-based Search

Dependency-based search, also called threat space search, (Allis, 1994; Allis, Herik, and Huntjens, 1995) is an important search technique for Go-Moku to reduce the tree size. This search technique can also be applied to Connect6 with some modifications. One issue for dependency-based search is to let Defender place stones on all the defensive squares at a time. For a double-threat move, all the defensive moves together covers a set of defensive squares, called the *reply set*, which has at most four defensive squares. If Defender places all stones in the reply set and Attacker still wins, then this implies that Attacker still wins for all the defensive moves. For example, in Figure 7 (below), W places four stones for each

double-threat move, such as squares 6, 7, 8 and 9 for the double-threat move at (4,5). Since all the three kinds of defensive moves at (6,8), (6,9) and (7,8) covers the four squares and such a defense, four at a time, will greatly reduce the search tree

However, on the other hand, placing more than two defensive stones for a double threat may make it harder to find winning threat sequences in some positions, where some winning threat sequence actually exists. For example, in Figure 2 (b), if let Defender (B) place on the reply set for W's double-threat move at (28, 29), then W cannot win due to 34 occupied by B. This problem can be solved by detecting whether the moves with more than two defensive stones refutes potential winning sequences, and then to search these defensive moves individually again.

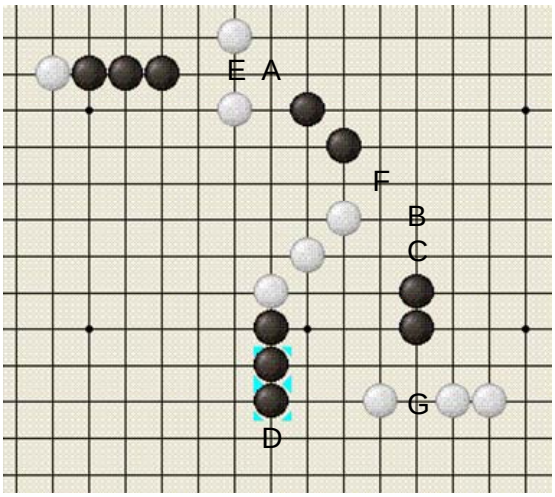


Figure 6: A position for dependency-based search.

Another important issue for dependency-based search is to reduce the tree size by traversing threat-space search trees, like Go-Moku (Allis, 1994; Allis, Herik, and Huntjens, 1995). For Connect6, this paper focuses on the effects of single threats and double threats. In this subsection, dependency-based search for Connect6 is illustrated by the position in Figure 6, where a correct winning sequence for B is to place stones on both A and D and then on both B and C.

In threat-space search trees, paths from the roots to leaves are called *threat dependency sequences*. In a threat dependency sequence, each element (S, n, γ) denotes three attributes, respectively indicating the squares to play (at most two squares), the number of threats to be created, and whether a defensive move refutes the sequence. The element for placing one stone on a dead-3 threat includes one square and one created threat. The element for placing two stones on a live-2 threat has two squares and two created threats. The element for placing one stone on a live-3 threat has one square and two created threats. Note that there are no elements for placing two stones on a dead-2 threat.

For the threat dependency tree starting from A, a threat dependency sequence SQ_A is {(A, 1, ⊥), (B, 1, γ), (C, 2, γ)}. The rules of generating sequences are as follows. In the sequence, the first element with ⊥ indicates that all the defensive moves such as E do not refute the sequence, while the second element with γ indicates that some defensive move, say at F, refutes the sequence. For an element with one square and with γ, the sequence can still take one extra element with one square only (but no more), e.g., (C, 2, γ), since it is possible that B makes a move on both squares at a time. For the tree starting from B or C, a threat dependency sequence SQ_{BC} is {(B&C, 2, γ)}. The sequence contains no more elements since the element with γ has already two squares and the move at G refutes the sequence. For the tree starting from D, a threat dependency sequence SQ_D is {(D, 1, ⊥)}.

A threat dependency sequence for Connect6 is called a *winning threat sequence* or a WT_0 *sequence*, if the sequence can be partitioned into consecutive subsequences with the following conditions.

- In the last subsequence *SQ*, the total threat number is more than two, and, in each of

other subsequences, the total threat number is two.
- In each subsequence except the last one, the last element is not marked with γ.
- In each subsequence, the total number of squares is at most two (normally two).

Each subsequence represents a move made by Attacker. Obviously, if a threat dependency sequence is a WT_0 sequence, Attacker wins as in Corollary 3. For example, both {(X&Y, 2, $\perp$), (Z, 1, γ), (W, 2, γ)} and {(X, 1, $\perp$), (Y, 1, $\perp$), (Z, 3, $\perp$)} are WT_0 sequences, but the sequence SQ_A is not.

Corollary 3. If a threat dependency sequence is a WT_0 sequence, Attacker wins. ▌

Furthermore, a threat dependency sequence is called a WT_i *sequence*, if the sequence can be partitioned into consecutive subsequences with the above three properties except for that the first property is modified as follows.

- In the last subsequence *SQ*, the total number of threat numbers is more than two, and in each subsequence in front of *SQ*, the total number of threat numbers is either one or two. The number of subsequences whose total threat numbers are one is *i*.

Each subsequence with one threat in total represents a single-threat *sub-move* made by Attacker. A sub-move is to place on one square only. Thus, for a WT_i sequence, if the board has other *i* independent single-threat sub-moves, Attacker wins as in Corollary 4. For the above example, the threat dependency sequence SQ_A is a WT_1 sequence, and Attacker (B) wins since SQ_D has an independent single-threat sub-move.

Corollary 4. If a threat dependency sequence *SQ* is a WT_i sequence and there exists at least *i* single-threat sub-moves independent of one another and SQ, Attacker wins. ▌

Threat Proof Search

Threat proof search includes lambda search (Thomsen, 2000), null move heuristics (Allis, 1994; Wu and Huang, 2005), and abstract proof search (Cazenave, 2001; Cazenave, 2003) are usually used to determine implicit threats, such as Go-Moku and Go, by preventing from searching all the moves in a large board. These search techniques share the same threat strategy as follows. First, Defender makes a null move that places neither stones on the board, and then apply dependency-based search to finding winning double-threat sequences for Attacker. If some winning double-threat sequence is found, we need to determine the relevance zone (Thomsen, 2000) for Defender to defend. If the relevance zone is much smaller than the board size, a great amount of time can be saved. For Connect6, this is especially important since an infinite board is considered. For Connect6, we define relevance zones in Definitions 3 and 4, and then can easily obtain Corollary 5 (below).

Definition 3. For a given position *P* to be moved by Defender, a set of squares *S* is called a $R^{(2)}(P)$-Zone, if the following condition holds: If Defender places neither stones in *S*, he must lose. ▌

Definition 4. For a given position *P* to be moved by Defender, a set of squares *S* is called a $R^{(1)}(P,s)$-Zone, if the following condition holds: If Defender places one stone at *s* and the other not in the set *S*, he must lose. ▌

Corollary 5. For a given position *P*, let *S2* be a $R^{(2)}(P)$-Zone, and $S1(s)$ be a $R^{(1)}(P,s)$-Zone, where $s \in S2$. Defender loses in *P*, if Defender loses for all the moves at (s,s'), where $s \in S2$ and $s' \in S1(s)$. ▌

From above, for a position *P*, the routine **`EvaluateAllDefensiveMoves`** can be designed for Defender by considering the following three cases. The first case is that Attacker has already two threats in *P*. In this case, Defender simply tries all the defending moves for the two threats.

The second case is that Attacker has one threat in *P*. In this case, Defender first places the first stone, say at *s*, to refute the threat. Then, find a $R^{(1)}(P,s)$-Zone, denoted by $S1(s)$. Thus, for all s' in $S1(s)$, (s,s') are defensive moves.

The third case is that Attacker has no threat. In this case, first find a $R^{(2)}(P)$-Zone, denoted by *S2*, and then finds $R^{(1)}(P,s)$-Zones, *S1*(*s*), for each $s \in S2$. Thus, for all $s \in S2$ and $s' \in S1(s)$, (s, s'), are defensive moves.

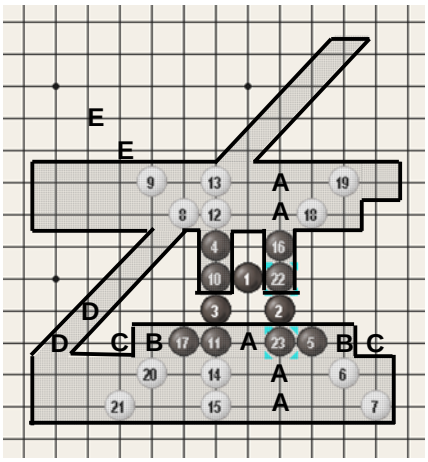


Figure 7: A winning double-threat sequence after a null move by *W*.

Now, we want to investigate the algorithms $A^{(2)}$ and $A^{(1)}$ of finding $R^{(2)}(P)$-Zone and $R^{(1)}(P,s)$-Zone, respectively, for a given position *P* and square *s*. We also illustrate the algorithms of building these zones by using an initial position, denoted by P_{623}, only including the three squares, 1, 2 and 3 in Figure 7. The position can be viewed as an initial move of Connect(6,2,3).

The algorithm $A^{(2)}$ for building $R^{(2)}(P)$-Zone first finds a winning double-threat sequence after Defender makes a null move in *P*. For example, Figure 7 shows a winning double-threat sequence from P_{623} after a null-move by W. Then, the algorithm uses the routine **`BuildZone(2)`** to build the shadowed zone *S2* in Figure 7. The routine can be generalized as **`BuildZone(i)`** by using the following three rules.

1. All black and white stones are included in the zone.
2. All defensive squares for the final threats are included in the zone.
3. For *i* empty squares, if Defender places stones on all of them and can form a threat, these squares are included in the zone. For example, when *i*=2, *D*s in Figure 7 are in the zone. Placing two stones on both *D*s builds a counter threat, also called an *inversion* (Thomsen, 2000). However, those *E*s are not in the zone, because for squares 8 and 9, we actually place one stone only and thus two extra white stones at *E*s cannot form a threat.

Note that in Figure 7 W places on four squares 6, 7, 8 and 9 for the double-threat move at (4,5) to reduce the tree size as mentioned Subsection 0. However, in case that the algorithm still needs to separate the three defensive moves at (6,8), (6,9) and (7,8), the algorithm still uses the above **`BuildZone(2)`** to build a zone for each defensive move. Then, the zone *S2* is the union of these zones.

Built from the above routine, *S2* for P_{623} satisfies a $R^{(2)}(G_{623})$-Zone for the following reasons. From above, placing two stones outside *S2* cannot block the winning double-threat sequence from Rules 1 and 2 and cannot form a counter threat from Rule 3. Therefore, Attacker still can win the game by playing the same sequence. Thus, we can obtain the following corollary.

Corollary 6. For a given position *P*, the zone *S2* built by the above algorithm is a $R^{(2)}(P)$-Zone. ▌

The algorithm $A^{(1)}$ for building $R^{(1)}(P,s)$-Zone, similar to $A^{(2)}$, first lets Defender make a *semi-null move*, which means to place on the square *s* only, and then find a winning double-threat sequence for Attacker as above. Then, the algorithm simply calls the routine **`BuildZone(1)`** to build a zone *S1*(*s*).

For example, for position P_{623} and square 4 in Figure 7, Figure 8 (below) shows a winning

double-threat sequence after the semi-null move by W; and the shadowed zone, denoted by $S1(4)$, is a $R^{(1)}(G_{623},4)$-Zone. From above, placing one stone at 4 and the other outside $S1(4)$ cannot block the winning threat sequence from Rules 1 and 2 and cannot form a counter threat from Rule 3. Therefore, Attacker still wins the game by playing the same sequence.

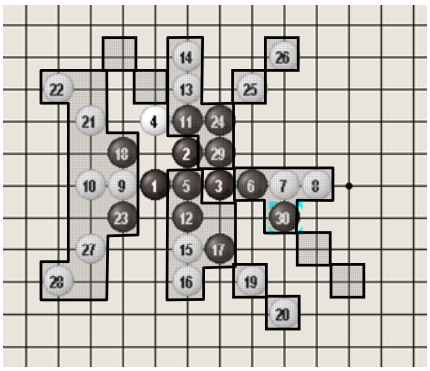


Figure 8: A winning double-threat sequence of *B*'s after a semi-null-move.

Similarly, suppose that the algorithm finds winning double threat sequences for Attacker by separating all the defensive moves. The zone $S1(s)$ is the union of these zones, built for individual defensive moves respectively. Thus, we can obtain the following corollary.

Corollary 7. For a given position P and square s, the zone $S1(s)$ built by the above algorithm is a $R^{(1)}(P,s)$-Zone. ▌

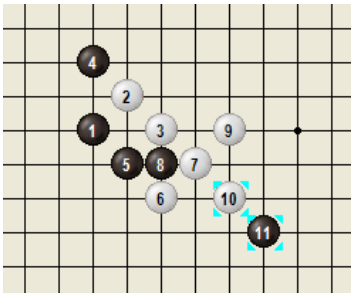


Figure 9: Semi-null moves at 8 and 11.

Now, an interesting question is what if the algorithms $A^{(2)}$ and $A^{(1)}$ cannot find winning double-threat sequences. For example, if we want to prove for the opening position, denoted by OP_1, in Figure 1, we first apply the algorithm $A^{(2)}$ to find a $R^{(2)}(OP_1)$-Zone, which surely includes square 8 in Figure 9. Then, we apply the algorithm $A^{(1)}$ to find a $R^{(1)}(OP_1,8)$-Zone. However, W can win by some winning mixed-threat sequence, but unfortunately cannot for any double-threat sequence.

The above example illustrates the need to make a null or semi-null move inside another null or semi-null move, like λ^2-tree search (Thomsen, 2000). However, λ^2-tree-like search does not always work for Connect6. For example, if we want to solve Connect(6,2,2), B needs to make null moves twice. Since W can actually place four stones on any squares to form a counter threat to refute the winning sequence of B. Thus, all squares are in the relevance zone. The goal of building a smaller relevance zone fails.

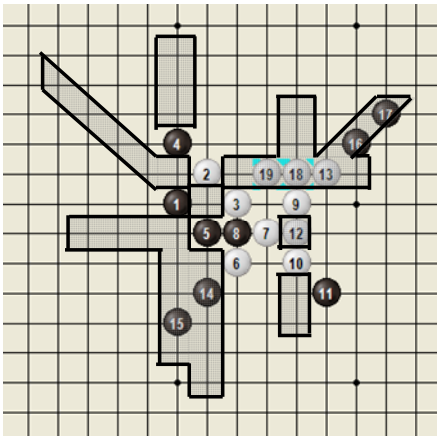


Figure 10: A winning double-threat sequence after two semi-null moves at 8 and 11.

Now, investigate how to build a $R^{(1)}(OP_1,8)$-Zone. For the semi-null move at 8, let W make a single-threat move at (9,10) as shown in Figure 9, and then let B make another semi-null move, say on 11. Then, after finding a winning double-threat sequence as shown in Figure 10, we use the routine **`BuildZone(2)`** to build the relevance zone *S*, as the shadowed zone in Figure 10. An intuition about the reason for using **`BuildZone(2)`** instead of **`BuildZone(1)`** is that the two stones skipped by B can be placed on any two squares to defend W's attack, like a null move in $A^{(2)}$.

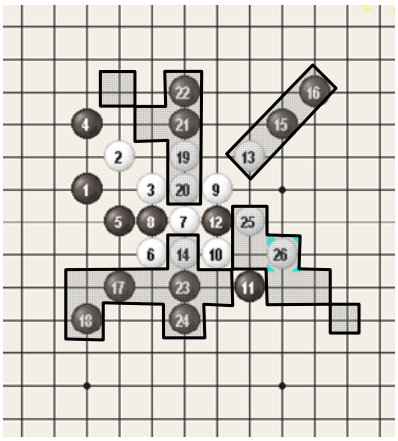


Figure 11: A winning double-threat sequence after the semi-null-move at 8 and the move at (11, 12).

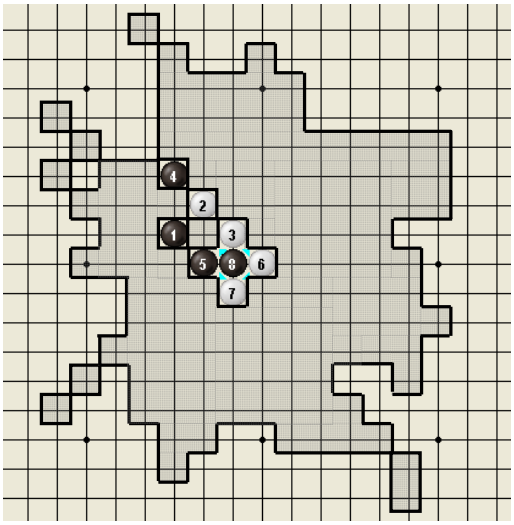


Figure 12: The relevance zone for the semi-null move at 8.

After obtaining the zone S, B makes moves (11, s) for all s in S. For example, in Figure 11, for the move at (11, 12), we obtain a winning double-threat sequence and the relevance zone, shadowed in Figure 11. Then, for all s in S, we use the same method to obtain relevance zones $S(s)$. After all $S(s)$ are obtained, the union of these S(s) is the relevance zone for the semi-null move at 8, as shown in Figure 12. The theory behind the above method is discussed in the rest of this section.

Definition 5. For a given position P, if Defender has skipped placing i stones, the position P is called an i-level position and is denoted by $P^{(i)}$. These skipped stones are called *null stones*. ▌

From Definition 5, a game position P without skipping Defender's stones is also $P^{(0)}$, a 0-level position. Once Defender makes a null move in a position $P^{(i)}$, the new position is $P^{(i+2)}$; and once Defender makes a semi-null move in a position $P^{(i)}$, the new position is $P'^{(i+1)}$.

Consider a sequence of moves $\{M_{00}, M_{01}, \ldots, M_{0,k0}, M_{10}, M_{11}, \ldots, M_{1,k1}, M_{30}, M_{31}, \ldots, M_{3,k3}\}$, where M_{10} is a semi-null move by Defender, M_{30} is a null move by Defender, and others are non-null moves. Let $P_{ij}^{(i)}$ denote the position after making the move M_{jj}. Then, the corresponding position sequence is $\{P_{00}^{(0)}, P_{01}^{(0)}, \ldots, P_{0,k0}^{(0)}, P_{10}^{(1)}, P_{11}^{(1)}, \ldots, P_{1,k1}^{(1)}, P_{30}^{(3)}, P_{31}^{(3)}, \ldots, P_{3,k3}^{(3)}\}$.

Definition 6. For a given position $P^{(i)}$, a set of squares S is called a relevance zone of $P^{(i)}$ or a R($P^{(i)}$)-Zone, if the following condition holds: If Defender places i stones back and none of the i stones are in S, Defender must lose. ▌

For each position $P^{(i)}$ in a position sequence, derive a R($P^{(i)}$)-Zone in a reversed direction as follows.

- Assume that $P^{(i)}$ is the last position in a position sequence. If Attacker wins, the zone can be built from the routine **`BuildZone(i)`**. If Defender wins, the zone includes all squares on the board.
- Assume that $P^{(i)}$ is not the last position in a position sequence for the rest of items. If Attacker is to move in $P^{(i)}$ and wins in the successor of $P^{(i)}$, the zone is the same as that of the successor.
- If Defender is to move in $P^{(i)}$, the zone is the union of R($PS_j^{(i)}$)-Zone, for all $PS_j^{(i)}$, where $PS_j^{(i)}$ are successors of $P^{(i)}$ and i-level positions.

Corollary 8. For a given position $P^{(i)}$, the zone derived by the above method is a R($P^{(i)}$)-Zone.

From above, Corollary 8 is obtained. The proof is omitted in this version. Besides, in the case that $i \geq 4$, the routine **`BuildZone(i)`** creates the whole board as the zone due to Rule 3. This explains why it is hard to build the relevance zone when solving Connect(6,2,2).

Now, we want to investigate the operation in the position $P^{(i)}$ again where Defender is to make a null or semi-null move. For simplicity of discussion, we only discuss the case that Defender makes a semi-null move in this position. In this case, the successive position becomes $P^{(i+1)}$. The zone for $P^{(i+1)}$ derived from above is a R($P^{(i+1)}$)-Zone, denoted by R^{i+1}. Similar to $A^{(1)}$, the algorithm lets Defender make moves by placing the null stone on all square in R^{i+1}. Thus, the successive position is back as an i-level position. And, the zone for $P^{(i)}$ is the union of the relevance zones of these successors.

Since R^{i+1} is a R($P^{(i+1)}$)-Zone, Attacker wins in $P^{(i+1)}$ if none of i+1 null stones are placed back in the zone. So, one of the i+1 null stones should be placed back in R^{i+1} to possibly prevent Attacker from winning. Consider the following two cases: (1) the null stone for the semi-null move in $P^{(i+1)}$ is placed inside R^{i+1}, and (2) one of the i null stones in position $P^{(i)}$ is placed inside R^{i+1}. In the first case, for each square in R^{i+1}, Defender already tries in the algorithm. In the second case, it is implied that the zone for $P^{(i)}$ need to include R^{i+1}. Interestingly, for the above algorithm, since all squares in R^{i+1} are tried by Defender and the relevance zone of each position must include the placed squares according to Rule 1, the zone for $P^{(i)}$ derived from the above algorithm must include R^{i+1}. Thus, we conclude that all defensive moves are tried.

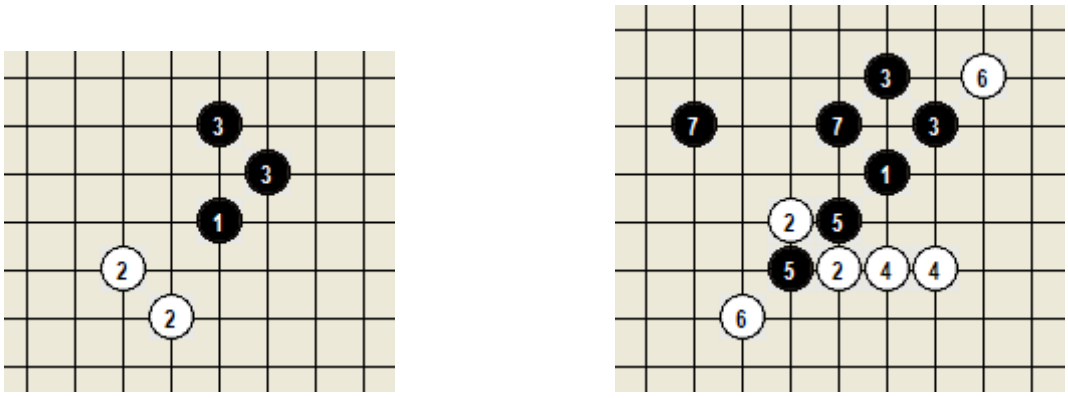
4. EXPERIMENTS

Based on the search techniques described in Section 3, we developed a new Connect6 program. The new program is apparently stronger than our old program, which uses only double-threat moves in threat-based search and can already beat 70% players in the game site (ThinkNewIdea Inc., 2005). Our Connect6 program is also modified to solve many positions. Let P1, P2 and P3 respectively denote the positions after the moves at (8, 9) in Figure 2 (a), (12,13) in Figure 2 (b), and (16, 17) in Figure 2 (c). Our program solves all of them with W winning. The statistical data are listed in Table 1. The CPU of the machine we used for the statistical data is AMD Athlon 64 1.8G Hz.

	P1	P2	P3
The total CPU time (in seconds)	288.15	481.46	584.82
The total number of visited nodes	38,179	45,267	47,479
The longest winning sequence	13	13	15

Table 1: Statistics for solving the games in Figure 2.

Besides, we also use the threat proof search technique in Subsection 0 to find all the defensive moves by B in the position in Figure 1. The total number of the defensive moves for this position is 2341. So far, we have solved 939 defensive moves of them with W winning. Among these solved defensive moves, 425 defensive moves are solved with mixed-threat sequences (which also include single-threat moves in the winning sequence).



(a) (b)

Figure 13. (a) The TX-H9 Joseki. (b) The key position based on our techinque.

Most importantly, we use this technique to solve a popular joseki, TX-H9 Joseki, with B winning as shown in Figure 13 (a). Among all the variations (Connect6 Organziation, 2007), the position as shown in Figure 13 (b) is proved based on the threat proof search technique.

5. CONCLUSION

This paper investigates new threat-based proof search techniques for Connect6 programs. The contribution of this paper is summarized as follows.

- Propose a quiescent detection heuristics for solving Connect6 position (in Subsection 0).
- Propose a new dependency-based technique for Connect6 (in Subsection 0).
- Propose a new threat proof search technique (in Subsection 0), like λ^i-tree search, to generate all the defensive moves.
- Use the above techniques to solve many Connect6 positions, some with mixed-threat winning sequences (in Section 4).

Based on the work in this paper, we expect to solve the opening in Figure 1 soon, if W does win in this position. Since the game Connect6 is quite new, the professional players are not still far less than those of Go, Renju, Chess, etc. It is very important to develop some search techniques to help us understand more about this game.

6. REFERENCES

Allis, L. V. (1994). Searching for solutions in games and artificial intelligence, Ph.D. Thesis, University of Limburg, Maastricht.

Allis, L. V., Herik, H. J. van den, and Huntjens, M. P. H. (1995). Go-Moku Solved by New Search Techniques. Computational Intelligence: An International Journal, 12 (1), pp. 7–23.

Allis, L.V., Meulen, M. van der, Herik, H.J. van den(1994). Proof-number search, Artificial Intelligence 66 (1) 91–124

Cazenave, T. (2001). Abstract Proof Search. Computers and Games (eds. T. A. Marsland and I. Frank), Vol. 2063 of Lecture Notes in Computer Science, pp. 39–54, Springer. ISBN 3–540–43080–6.

Cazenave, T. (2003). A Generalized Threats Search Algorithm. Computers and Games, Vol. 2883 of Lecture Notes in Computer Science, pp. 75–87.

Herik, H. J. van den, Uiterwijk, J.W.H.M., Rijswijck, J.V. (2002). Games solved: Now and in the future. Artificial Intelligence, Vol. 134, pp. 277-311.

Lee, T.W. (2005) Joseki and Tsumegos for Connect6 (in Chinese). http://groups.msn.com/connect6/.

ThinkNewIdea Inc. (2005) CYC game (in Chinese). http://cycgame.com.

Thomsen, T. (2000). Lambda-search in game trees - with application to Go. ICGA Journal, Vol. 23(4), pp. 203–217.

Uiterwijk, J.W.H.M., Herik, H.J. van den. (2000). The advantage of the initiative, Information Sciences 122 (1)
43–58.

Wu, I-C. (2005) Homepages of Connect6. http://connect6.csie.nctu.edu.tw.

Wu, I-C., Huang, D.-Y. (2005) A New Family of k-in-a-row Games. The 11th Advances in Computer Games Conference (ACG'11), Taipei, Taiwan.

計畫成果自評部份

本計畫第一個最重要的成果是：我們發展了 dependency-based search, null-move heuristics, threat proof search 等搜尋技術，並用**此技術證明了一個不少高手下過的定石（參見 Figure 13(a)），為黑必勝**。這對六子棋界棋士而言，是一個震撼的結果。

本計畫第二個最重要的成果是（主要在本計畫第二年度）：我們專注於利用這些技術改良我們的「交大六號」六子棋程式，**使之獲得第十三屆國際奧林匹亞賽局競賽冠軍，為我國獲得唯一的一面金牌及獎牌。**

從這兩個成果，我們很確信地自評：此計畫的執行成果相當優異。