Dependency-Based Search for Connect6

I-Chen Wu(✉), Hao-Hua Kang, Hung-Hsuan Lin, Ping-Hung Lin, Ting-Han Wei, and Chieh-Min Chang

Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
{icwu, kangbb, stanleylin, bhlin, tinghan, aup}@java.csie.nctu.edu.tw

Abstract. Allis proposed dependency-based search (DBS) to solve Go-Moku, a kind of five-in-a-row game. DBS is critical for threat space search (TSS) when there are many independent or nearly independent TSS areas. Similarly, DBS is also important for the game Connect6, a kind of six-in-a-row game with two pieces per move. Unfortunately, the rule that two pieces are played per move in Connect6 makes DBS extremely difficult to apply to Connect6 programs. This paper is the first attempt to apply DBS to Connect6 programs. The targeted program is NCTU6, which won Connect6 tournaments in the Computer Olympiad twice and defeated many professional players in Man-Machine Connect6 championships. The experimental results show that DBS yields a speedup factor of 4.12 on average, and up to 50 for some hard positions.

Keywords: Connect6 · NCTU6 · Dependency-based search · Threat-space search

1 Introduction

Dependency-based search (*DB Search*), proposed by Victor Allis *et al.* [1, 2], is a search method which explores search under dependency constraints. It was successfully used to solve games such as Double-Letter Puzzle, Connect-four and Go-Moku [3]. Go-Moku is a kind of five-in-a-row game without prohibited moves. A generalized family of *k*-in-a-row games [9, 10] were introduced by Wu *et al.*, and Connect6 is an interesting one in the family with the following rules [14]. Two players, named *Black* and *White*, alternately place two pieces on empty *squares*[1] of a 19 × 19 Go board in each turn, except that Black, who plays first, places one piece initially. The player who gets *k* consecutive pieces of his own first wins. The game is a tie when the board is filled up without either player winning.

Threats are the key to winning these types of games. For example, in Go-Moku, a four (a threat) forces the opponent to defend, or the opponent loses. *Threat space search* (*TSS*) is an important winning strategy for these games, where a winning path is found through threats. For example, in the Go-Moku community [2], *victory-by-continuous-fours* (*VCF*) is a well-known strategy, whereby a winning sequence is

[1] Practically, stones are placed on empty intersections of Renju or Go boards. In this paper, when we say squares, we mean intersections.

achieved through continuously playing fours (threats). Since the opponent's choice of possible replies is limited, the search space based on threats is greatly reduced. Thus, TSS can search much deeper than regular search methods.

While TSS shows its promise in finding the winning sequence, further improvements are possible when there are independent or nearly independent TSS areas. An example was given in [2] (cf. Diagram 3a of [2]). Hence, Allis *et al.* [11, 12] proposed DBS to help minimize the state space of TSS. The method removes unnecessary *independent threats* and focuses on *dependency relations*. However, some threats may be independent at a given point in time but mutually dependent later on. Allis *et al.* also proposed a method called "combination" to help solve this problem.

The above issue is also critical to Connect6 as well as other *k*-in-a-row games. For Connect6, similarly, there may be independent or nearly independent TSS areas, especially during the end game phase where there are usually many independent TSS areas. Unfortunately, independent TSS areas are more likely to become mutually dependent for Connect6 due to the property that two pieces are played for every move.

This paper is the first attempt to apply DBS to Connect6 programs. We give our definitions and notation for DBS in Connect6 in Sect. 2, propose to construct a dependency-based hypergraph to identify dependencies in Sect. 3, and propose some methods for DBS in Sect. 4. Our experiments were run based on our Connect6 program, named NCTU6, which won Connect6 tournaments [5, 10, 13, 15–17] several times from 2006 to 2011, and defeated many top-level human Connect6 players [6] in man-machine Connect6 championships from 2008 to 2011. Our experimental results in Sect. 5 show that NCTU6 with these methods was speeded up about 4.12 times in average when compared to the original NCTU6. Also, for some hard positions, the speedup was up to 50 times. Section 6 makes concluding remarks. Due to paper size limitation, the background for Connect6 is omitted. We directly follow the terminologies in [9, 10], such as threats, single-threat, double-threat, victory by continuous double-threat-or-more moves (VCDT), and victory by continuous single-threat-or-more moves (VCST).

2 Definitions and Notation

This section gives definitions and notations related to TSS. For simplicity of discussion, let the attacker indicate the player who we want to prove to win, and the defender the other. Let a move $m = (s, s')$ be defined as a move from a position s to another position s'. The set of pieces of move m is denoted by $P(m)$, and the set of threats generated by move m is denoted by $T(m)$. A sequence of moves ψ is

$$\psi = \langle m_1, m_2, \ldots, m_{2t} \rangle, \text{ where } t \in \mathbb{N}.$$

In ψ, all m_{2i+1} moves are attacker moves, while all m_{2i} moves are defender moves. If all attacker moves are double-threat-or-more, the sequence is called a *continuous double-threat-or-more sequence* or a *CDT*. If the last attacker move m_{2t-1} is three-threat-or-more, the attacker wins in this CDT sequence, which is called a *lead-to-win CDT*. If all attacker moves are one-threat-or-more in a sequence, it is a *CST*. In this paper, only CDT is discussed for simplicity.

For example, a sequence $\langle m_a, m_b, m_c, m_d, m_e, m_f, m_g, m_h\rangle$ is illustrated in Fig. 1(a) below. $P(ma)$ contains the two black pieces marked with a. Additionally, $T(ma)$ contains a double-threat, four black pieces highlighted with a blue rectangle in Fig. 1(a). The move m_c contains two single threats (STs) to form a double-threat (DT), so $T(m_c)$ contains eight black pieces. The sequence is a CDT since all Black moves are double-threats, and is a lead-to-win CDT since the last Black move m_g has three threats.

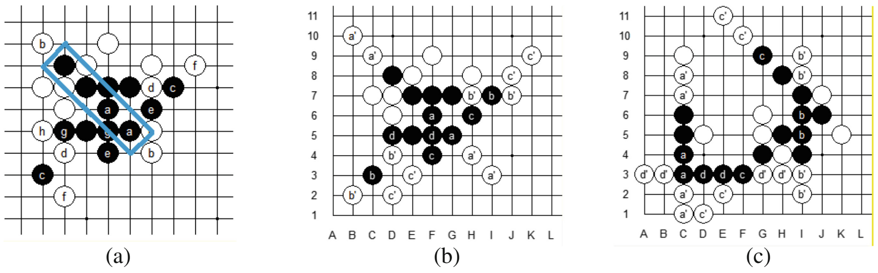

Fig. 1. (a) A lead-to-win CDT, (b) a CCDT with the same attacker moves as (a), and (c) another CCDT.

A lead-to-win CDT ψ does not imply its initial position, denoted by $r(\psi)$, is winning, since a variety of defensive moves are possible for the defender. In order to prove that a given initial position r is winning, we need to prove the following: $\exists m_1, \forall m_2, \exists m_3 \ldots, \forall m_{2t}$ such that a sequence $\langle m_1, m_2, m_3, \ldots, m_{2t}\rangle$ starting from r are CDTs. In the paper [17, 18], it is clear to see that the complexity becomes dramatically smaller if we conceptually allow the defender to play all defensive positions at once, which we refer to as a *conservative defense*. For example, all the White moves in Fig. 1(b) are conservative defenses. Thus, we define the sequence of moves with conservative defenses in Definition 1.

Definition 1: A CDT $\psi = \langle m_1, m_2, \ldots, m_{2t}\rangle$ is called a *conservative CDT* or *CCDT*, if all defender moves m_{2i} are conservative defenses. Since conservative defenses are fixed and unique, we can combine both moves $\langle m_{2i-1}, m_{2i}\rangle$ into a macro-move M_i. Let M_i^A denote attacker move $m_{2i-1,}$ M_i^D defender move m_{2i}. Then, a CCDT can be represented as a sequence of macro moves Ψ as follows.

$$\Psi = \langle M_1, M_2, \ldots, M_t\rangle \text{ where } t \in \mathbb{N}. \qquad \square$$

Since conservative defenses contain all possible defender moves, one important property is: if a CCDT sequence Ψ is lead-to-win, the initial position $r(\Psi)$ is a win for the attacker, and Ψ is also a *VCDT*. As illustrated in Fig. 1(b), $\Psi = \langle M_a, M_b, M_c, M_d\rangle$ is a VCDT, where $M_x = \langle m_x, m_{x'}\rangle$ for all $x = a, b, c, d$.

Definition 2: In a CCDT Ψ defined as above, a macro-move M_j is said to *depend on* M_i, denoted by $M_i \prec M_j$, if $P(M_i^A) \cap T\left(M_j^A\right) \neq \emptyset$, where $i<j$. $\square$

Let us illustrate a dependency relation using the CCDT in Fig. 1(b). We have $M_a \prec M_b$ due to the square at F6. Similarly, we have $M_b \prec M_c, M_a \prec M_d$ and $M_c \prec M_d$, but not $M_b \prec M_d$, denoted by $M_b \nprec M_d$. Another illustration is in Fig. 1(c), where $M_a \prec M_d$ and $M_c \prec M_d$, but $M_a \nprec M_b$ and $M_a \nprec M_c$, implying M_b and M_c are independent of M_a. Thus, M_d is a *combination move* from M_a and M_c.

3 Dependency-Based Hypergraph

For Go-Moku or Renju [8], the dependencies of moves can be represented by a directed graph straightforwardly. However, for Connect6 (where each move includes two pieces), if we directly draw the dependencies between macro-moves, the dependency graph becomes very sophisticated. In order to simplify the dependency graph, we split those double-threat moves with two single-threats into *half-moves*. Now, we classify all double-threat moves into three categories as follows.

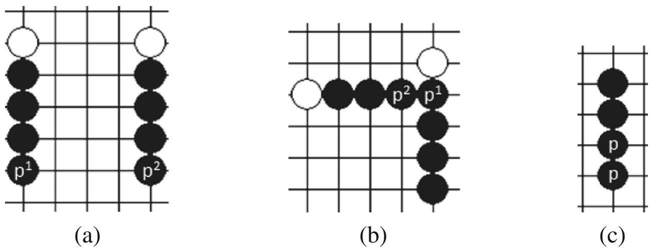

Fig. 2. Three categories of double-threat moves. (a) $\mathcal{T}^{I}_{ST}$ (b) $\mathcal{T}^{D}_{ST}$ (c) $\mathcal{T}_{DT}$.

- $\mathcal{T}^{I}_{ST}$: the set of double-threat moves M with two independent single-threats. Each of the two attacker pieces, p^1 and p^2 in $P(M^A)$, generates one single-threat independently. More specifically, let the macro-move be divided into two *half-moves*, $M^1 = M^{1A}, M^{1D}$ and $M^2 = \langle M^{2A}, M^{2D} \rangle$, where $M^{1A} = \{p^1\}$, $M^{2A} = \{p^2\}$, and M^{1D} and M^{2D} are the sets of conservative defenses to M^{1A} and M^{2A} respectively. An example is shown in Fig. 2(a). Obviously, there is no dependency between M^1 and M^2. Both M^1 and M^2 are *primitive macro-moves* or *primitive moves*, defined as the moves with the least number of pieces which can generate threats.
- $\mathcal{T}^{D}_{ST}$: the set of double-threat moves M with two dependent single-threats. Let p^1 and p^2 denote the two attacker pieces in $P(M^A)$. Without loss of generality, let M^1 be a half-move containing p^1 and generating one single-threat, and let M^2 be another containing p^2 and generating another single-threat depending on M^1. An example is shown in Fig. 2(b). According to the definition of dependency, we have $M^1 \prec M^2$. Both M^1 and M^2 are also primitive moves.
- $\mathcal{T}_{DT}$: the set of other double-threat moves, which are usually generated from live twos. An example is shown in Fig. 2(c). In this case, the macro-move cannot be divided into two half-moves since each piece alone would not form a threat. The move M itself is a primitive move.

For a VCDT sequence $\Psi = \langle M_1, M_2, ..., M_t \rangle$, let all macro-moves be translated into primitive moves as above. Then, the dependencies of these primitive moves can be drawn into a directed acyclic hypergraph, called a *dependency-based hypergraph* (*DBH*) in this paper. Formally, a DBH $\mathbb{G}$ is defined to be $(\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ is the set of vertices and $\mathbb{E}$ is the set of hyperedges. Each vertex V in $\mathbb{V}$ represents a position. The root, V_r, represents the initial position and has an indegree of 0. All other vertices have an indegree of 1; that is, for every V other than V_r, there is a single corresponding incoming hyperedge E.

Each hyperedge $E = (\{V_1, V_2, ..., V_k\}, V)$ in $\mathbb{E}$ represents a primitive move, where $\{V_1, V_2, ..., V_k\}$ are the sources and V is the destination. Let E_i be the corresponding hyperedge of V_i. Then, the condition $E_i \prec E$ must hold. On the other hand, for every primitive move E' where $E' \prec E$, one of the hyperedges $E_1, E_2, ..., E_k$ must represent E'. However, if E does not depend on any moves, then $E = (\{V_r\}, V)$. For consistency, let E^A denote the attacker move of E and E^D the defender move.

```
Procedure BuildPsi($\Psi$)
 1:  for every macro-move $M \in \Psi$ do
 2:      AddMove($M$)
 3:  end for
end
```

```
Procedure AddMove($M$)
 1:      if $M \in \mathcal{T}_{DT}$ then
 2:          create hyperedge $E$ for the DT;
 3:          AddE($E$);
 4:      else
 5:          create hyperedges $E^1$ and $E^2$ for each ST;
 6:          // without loss of generality, let $E^1 \prec E^2$ if $M \in \mathcal{T}_{ST}^D$.
 7:          AddE($E^1$);
 8:          AddE($E^2$);
 9:      end if
end
```

```
Procedure AddE(E)
 1:  Let $\mathbb{V}_{source} = \emptyset$ be the set of source vertices of $E$;
 2:  for every active hyperedge $E' \notin \mathbb{E}$ do
 3:      if $P(E'^A) \cap T(E^A) \neq \emptyset$ then
 4:          add the destination vertex of $E'$ to $\mathbb{V}_{source}$;
 5:      end if
 6:  end for
 7:  if $\mathbb{V}_{source} = \emptyset$ then set $\mathbb{V}_{source} = \{V_r\}$;
 8:  if $E \notin \mathbb{E}$ then add $E$ into $\mathbb{E}$;
 9:  mark $E$ as active;
end
```

The DBH $\mathbb{G}$ is constructed from macro moves one by one. For simplicity, we first consider building $\mathbb{G}$ from moves in a given CCDT $\Psi = \langle M_1, M_2, ..., M_t \rangle$ by the procedure BuildPsi shown as above. In practice, during the search, moves are added to $\mathbb{G}$ as they are traversed, so BuildPsi is not actually used. Each move M is checked in AddMove and the corresponding hyperedge and vertex are generated. In the case that $M \in \mathcal{T}_{DT}, M$ is transferred into one hyperedge E and inserted into $\mathbb{G}$ via AddE. In the case that $M \in \mathcal{T}_{ST}^I \, U \, \mathcal{T}_{ST}^D$, M is split into two half-moves and then respectively translated into two hyperedges E^1 and E^2, which are then inserted into $\mathbb{G}$ (let $E^1 \prec E^2$ if $M \in \mathcal{T}^D_{ST}$) using AddE. In the rest of this paper, $\mathbb{E}(M)$ denotes the set of hyperedges corresponding to M, namely $\{E^1, E^2\}$ for both $\mathcal{T}^I_{ST}$ and $\mathcal{T}^D_{ST}$, and $\{E\}$ for $\mathcal{T}_{DT}$.

In the procedure AddE for adding a hyperedge E, find all the hyperedges E' in $\mathbb{G}$ that satisfy $E' \prec E$, then let the destination of E' be one of the sources of E. If no dependent hyperedges are found, put V_r into the set of sources of E.

Furthermore, DBH can be constructed incrementally during threat-space search. Assume we have searched up to M_t in a CCDT $\Psi = \langle M_1, M_2, \ldots, M_t \rangle$. The hyperedges (in $\mathbb{E}(M_t)$) added via AddE are marked as active in line 9 of AddE. When the search on M_t is finished and the search recurses to M_{t-1}, we simply mark as inactive the hyperedges that were marked as active for M_t. The active hyperedges form a subgraph of $\mathbb{G}$, denoted by $\mathbb{G}(\Psi)$, which consists of all the hyperedges corresponding to primitive moves of $M_1, M_2, \ldots, M_t$. Note that the inactive hyperedges are still in $\mathbb{G}$. For illustration, the DBH that is constructed for the CCDT in Fig. 1(b) is shown in Fig. 3(a), and another DBH resulting from Fig. 1(c) is in Fig. 3(b).

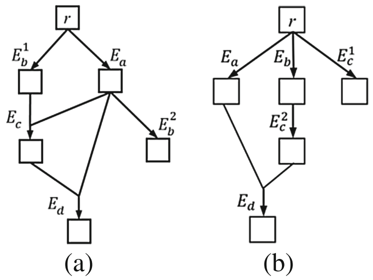

Fig. 3. (a) A DBH constructed for the CCDT in Fig. 1(b), (b) another for the CCDT in Fig. 1(c).

Definition 3: E' is said to be *reachable* from E if there exists some path from E to E', denoted as $E \ll E'$. Similarly, V' is said to be *reachable* from V if there exists some path from V to V', denoted as $V \ll V'$. □

All vertices (hyperedges) reachable from a node V (hyperedge E) indicate the threat space induced from V (E). For example, E_d shown in Fig. 3(b) is reachable from E_a, E_b and E_c^2. The above described DBH has an important feature in that the number of vertices in a DBH is in general much smaller than the number of the corresponding TSS search nodes.

4 Methods of DB Search

In NCTU6, a threat-space search routine, named IsVCDT, is used to check whether the current position is a win for the attacker by VCDT within a limited depth. The routine is described as follows.

```
Procedure IsVCDT($\Psi$, depth)
  1:  If the current position is winning return winning;
  2:  If depth=0 return not winning;
  3:  for each macro-move $M$ do
  4:      if WillSearch ($\Psi$, $M$) then
  5:          let $\Psi'$ be $\Psi$ and append $M$ into $\Psi'$
  6:          AddMove($M$);
  7:          isWin = IsVCDT($\Psi'$, depth-1);
  8:          mark $\mathbb{E}(M)$ as inactive;
  9:          if isWin return winning;
 10:      end if
 11:  end for
 12:  return not winning;
end
```

In lines 3 to 8 of IsVCDT, all candidate macro-moves are investigated in WillSearch (Ψ, M) to decide whether the macro-moves M are to be searched. Previously, in NCTU6, the routine WillSearch (Ψ, M) is in general true, except for some clearly bad moves or up to a limited number of moves. In this paper, the routine is modified to filter more macro-moves using dependency-based search techniques. Four methods for filtering are proposed in the following four subsections respectively.

4.1 Method F1

Method F1 only allows dependent moves to be searched. Given a CCDT $\Psi = \langle M_1, M_2, \ldots, M_{i-1} \rangle$ and the next candidate M_i, WillSearch (Ψ, M_i) returns true (i.e., M_i is to be searched) if and only if the following condition holds. For all M_j, $j \leq i$, there exists some hyperedge $E_j \in \mathbb{E}(M_j)$such that $E_j \ll E_i$, for every $E_i \in \mathbb{E}(M_i)$.

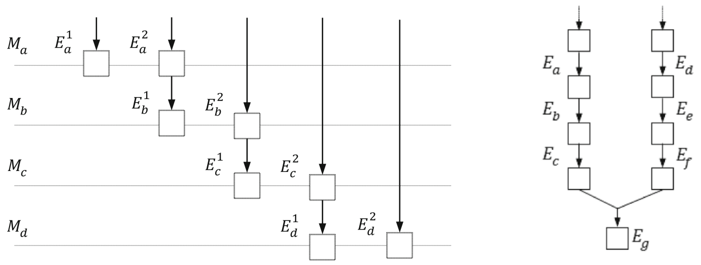

Fig. 4. (a) A DBH with 4 moves M_a, M_b, M_c, M_d, and (b) a DBH with a combination macro-move E_g.

In F1, we only traverse the moves which depend on their precedent moves and are reachable from all ancestors in order to reduce the number of nodes to search. Consider the DBH in Fig. 4(a), where a variety of sequences can be searched from the same eight primitive moves $E_a^1, E_a^2, E_b^1, E_b^2, E_c^1, E_c^2, E_d^1, E_d^2$ Examples include $\langle M_{aa}^{12}, M_{bb}^{12}, M_{cc}^{12}, M_{dd}^{12} \rangle$, $\langle M_{ab}^{12}, M_{cc}^{12}, M_{da}^{12}, M_{bd}^{12} \rangle$, $\langle M_{ac}^{12}, M_{da}^{12}, M_{bb}^{12}, M_{cd}^{12} \rangle$, etc., where M_{xy}^{mn} indicates a macro-move from E_x^m and E_y^n and M_x is M_{xx}^{12}. Using F1, the search will traverse from M_a to M_b, but since E_c^1 is not reachable from E_a^1, the search cannot continue past M_b.

Also, this method cannot be applied to cases where a combination move is required, such as the example in Fig. 4(b). Another example can be observed in Fig. 3(b), where WillSearch($\langle M_b \rangle$, M_c) returns true, but WillSearch($\langle M_b, M_c \rangle$, M_a) returns false, as does WillSearch($\langle M_a \rangle$, M_b), since $M_a$and M_b are mutually independent. Thus, the winning sequences $\langle M_a, M_b, M_c, M_d \rangle$ and $\langle M_b, M_c, M_a, M_d \rangle$ cannot be found. Intuitively, F1 focuses purely on reducing the search size, but the reduction comes at the cost of being unable to find sequences requiring combinations.

4.2 Method F2

Method F2 allows, in addition to those in F1, macro-moves that can form a combination move later. Before discussing F2, we define two types of zones below.

Definition 4: Each hyperedge E is associated with an *attacking zone*, denoted by $Z^A(E)$, which is a union set of locations of attacker pieces $P(E^{'A})$ for all hyperedges $E^{'}$ reachable from E (i.e., $E \ll E^{'}$). For each macro-move M, $Z*(M)$ denote the *to-be-combined zone*, which is a set of locations of the board which are on the lines containing one of the pieces in $P(M^A)$. □

For example, in Fig. 5(a), the marked squares form the attacking zone $Z^A(E_b)$. Note that the locations of single-threats like G9, F10 and E11 are not in the zone since the threat does not depend on E_b. In Fig. 5(b), the to-be-combined zone $Z^*(M_a)$ is the collection of squares that are covered by red lines[2]. The intersection of both $Z^*(M_a) \cap Z^A(E_b)$ contains the locations E2 and F3. This implies that some of subsequent macro-moves from M_b may combine with M_a at E2 or F3.

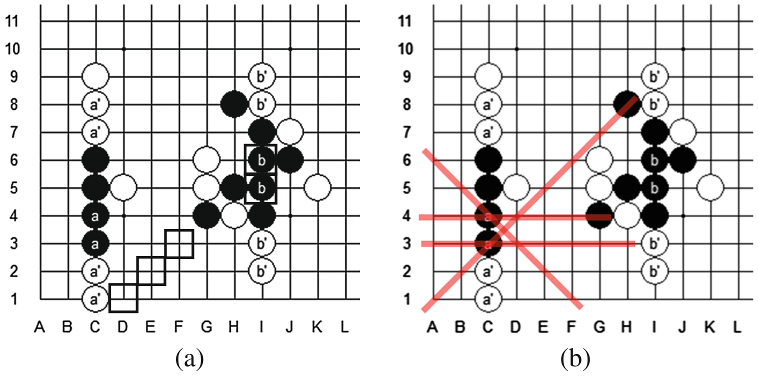

Fig. 5. An example of how combination detection is implemented. (a) $Z^A(E_b)$ (b) $Z^*(M_a)$ (Color figure online).

[2] In actual implementation for the zone $Z^*(M_a)$, all lines centered on $P(M_a^A)$ are not longer than 6 in all directions. Moreover, the lines are shortened when encountering opponent pieces. The zone is denoted by Z^*, since these lines form a star-like shape from the attacking pieces.

In Method F2, WillSearch(Ψ, M_i) returns true if Method F1 returns true. Additionally, WillSearch(Ψ, M_i) returns true if the following condition holds: there exists some $E_i \in \mathbb{E}(M_i)$ such that $Z^*(M_{i-1}) \cap Z^A(E_i) \neq \emptyset$. Thus, Method F2 allows TSS to switch to another move M_i whose descendants may be combined with the current move M_{i-1}. For example, in Fig. 5(a), since the intersection $Z^*(M_a) \cap Z^A(E_b)$ is not empty, the program is allowed to play M_b.

One issue is the sequence in which moves are played. Consider the example illustrated in Fig. 4(b). For the move E_g, which is a combination of E_c and E_f, Method F2 plays E_d after E_a, E_b, E_c, but does not play E_d immediately after E_a, E_b. This ensures that the playing sequence is narrowed-down to a unique sequence, eliminating the various different sequences that can also lead to E_g.

4.3 Method F3

Method F3 allows, in addition to the previous methods, moves that can block inversions. Inversions are defensive moves that result in the defender generating threat(s). To avoid inversions, the attacker needs to play moves that block defender threats while still generating a double-threat move of its own. Threat-blocking can be classified into post-blocking and pre-blocking. For post-blocking, the attacker blocks defender threats in the next immediate move, while for pre-blocking the attacker blocks the threats in advance.

In Method F3, WillSearch(Ψ, M_i) returns true if F2 (including F1) returns true. Additionally, it returns true in the following ways. For post-blocking, i.e., M_{i-1} is an inversion, it is straightforward for WillSearch(Ψ, M_i) to return true if M_i can block opponent threats while generating a double-threat for the attacker.

Before examining pre-blocking, we define the *defending-inversion zone*, denoted by $Z^I(E)$. Each hyperedge E is associated with a set of locations $Z^I(E)$ that potentially blocks defender threats $T(E^{'D})$ for all hyperedges $E^{'}$ reachable from E (i.e., $E \ll E^{'}$).

Assume that $P(M_{i-1}) \cap Z^I(E_i) \neq \emptyset$, where $E_i \in \mathbb{E}(M_i)$. Then, M_{i-1} may pre-block E_i's descendants that end up becoming defender threats. In Method F3, WillSearch(Ψ, M_i) also returns true in this case for potential pre-blocking.

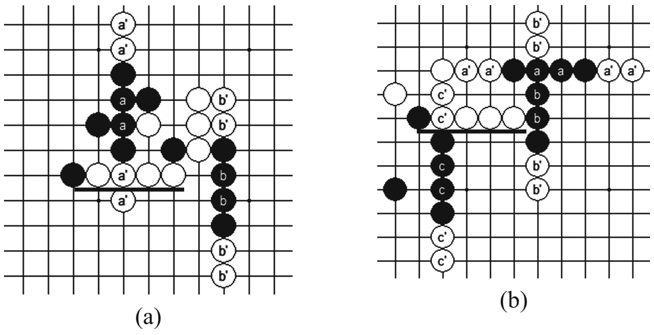

Fig. 6. (a) An example of post-block. (b) An example of pre-block.

The method is illustrated by two examples in Fig. 6, where the inversions are underlined. In Fig. 6(a), M_b will be searched since it blocks the inversion generated by M_a^D. In Fig. 6(b), M_b blocks the inversion generated by M_c^D in advance and we can then search M_c after M_b. If we searching M_c first, post-blocking is not possible since the attacker has to play two moves, M_a^A and M_b^A, to block the inversion. Therefore, the order in which moves are searched is critical for pre-blocking.

In actual implementation, we do not support a complete set for $Z^I(E)$. The larger $Z^I(E)$ is, the higher the chance to switch search order and the larger the branching factor. In our implementation, we only consider continuous inversions for $Z^I(E)$.

4.4 Method F4

Using the above three methods, we still encounter a problem: the number of moves derived from the three methods (even for Method F1) is still large in the case of macro-moves in $\mathcal{T}_{ST}^I$. Method F4 is proposed to filter more redundant moves in $\mathcal{T}_{ST}^I$.

Consider an example where many independent half-moves (single-threats), say $E_1, E_2, \ldots, E_n$, are available but only E_1 depends on the last move M_{i-1}. Based on F1, all macro-moves including E_1 and any other E_j are all legal candidate moves. In the case that all the other E_j are single-threats without any further development, the search still runs once for each of these E_j. However, it is clear for human players to search E_1 with some E_i only once.

In order to solve this problem, Method F4, basically following F3, is proposed to reduce the search space. Assume that F3 returns true and the candidate move M_i is in $\mathcal{T}_{ST}^I$. For simplicity, let $\mathbb{E}(M_i) = \{E_1, E_2\}$, and E_1 depend on the last move M_{i-1} without loss of generality. Method F4 checks the following cases. First, in the case that the half-move for E_2 is actually not created in the DBH yet, WillSearch(Ψ, M_i) returns true. Second, in the case that E_2 is in the DBH already, investigate the intersection of $Z^A(E_1)$ and $Z^A(E_2)$. If the intersection is not empty, then WillSearch(Ψ, M_i) returns true, since it is likely to combine both half-moves later. WillSearch(Ψ, M_i) returns false otherwise.

5 Experiments

For our experimental analysis, we collected 72 testing positions from Taiwan Connect6 Association [9] and Littlegolem [7]. The 72 positions all have VCDTs. Our experiments were done on a computer equipped with CPU Xeon E31225 3.1 GHz, 4 GB DDR3-1333 memory, and Windows 7 x64 version.

The four methods, F1, F2, F3 and F4 described in Sect. 4 are incorporated into the original NCTU6 for testing, denoted by PF1, PF2, PF3 and PF4 respectively. In this subsection, we want to compare the performances of these four programs with the original NCTU6. All five programs were set to a search depth of 10 macro-moves with the branching factor set to 50 without time limits.

From Sect. 4, clearly, NCTU6 is required to search more moves than the four programs PF1 to PF4, PF3 is more than PF2, and PF2 is more than PF1. Hence, the

Table 1. The comparison among the five programs.

Programs	Move count	Time (S)	Solve positions
NCTU6	12,694,339	1,050	72
PF1	55,691	11	55
PF2	5,648,130	571	70
PF3	7,273,329	799	72
PF4	2,852,988	255	72

following are expected: NCTU6 is slower than PF1 to PF4, but more accurate; PF3 is slower than PF2, but more accurate; and PF2 is slower than PF1, but more accurate.

Table 1 shows the experimental results fit the above expectation. Program PF4 solved all 72 positions while performing better than NCTU6, PF2 and PF3. The average speedup of PF4 over the original NCTU6 was 4.12, and the speedup for some hard positions reached up to 50.3. Figure 7 (below) shows the computation times of all the 72 positions for NCTU6 and PF4. The position IDs are sorted in the order of the computation times for NCTU6.

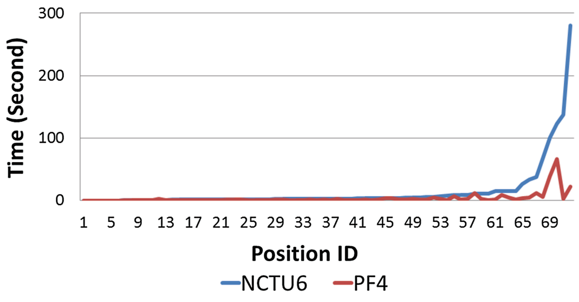

Fig. 7. The computation time on each position by original NCTU6 and NCTU6 with DBS.

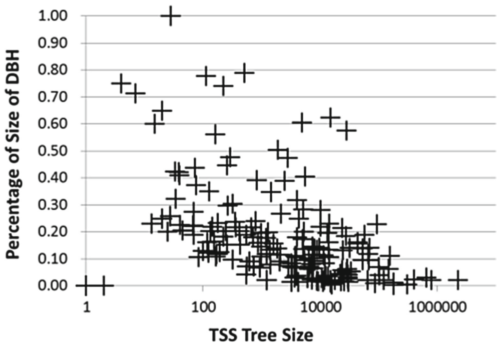

Fig. 8. The percentage of DBH over TSS tree size for all benchmark.

Table 1 also shows that the average computation time for each searched move is 82.7 ms in NCTU6, while it is 89.4 ms for PF4. This implies that the incurred overhead is about 8.1 % of the computation time.

Now, we want to investigate the memory size of DBH with respect to the TSS tree size. We use the number of hyperedges to indicate the memory size for DBH, and the number of searched moves to indicate the TSS tree size. Figure 8 shows the percentage of DBH size over TSS tree size for all testing positions. We can see that the ones with large TSS tree sizes usually have lower percentage of memory requirement for DBH.

6 Conclusion

For Connect6 and other *k*-in-a-row games, DBS is critical for TSS when there are many independent or nearly independent TSS areas, a situation that is especially common during end game. Unfortunately, the rule that two pieces are played per move in Connect6 makes DBS extremely difficult to apply to Connect6 programs.

This paper is the first attempt to apply DBS to Connect6 programs. We propose four DBS methods, F1 to F4, to reduce the search space of TSS in NCTU6 while still successfully finding solutions. The experimental results show that method F4 yields a speedup factor of 4.12 on average, and up to 50 for certain hard positions.

There are still open issues that need to be addressed. Consider an example where there are currently at least two independent TSS areas. Within each area, single threats may exist deep within the search tree. One major issue is that it is difficult to associate two of these independent single-threats as a double-threat move. Another issue is, as pointed out in Subsect. 4.4, it is hard to find more pre-blocking inversions without loss of performance. Also, a future direction for study is to apply DBS to other games such as Go to solve life-and-death problems.

Acknowledgements. The authors would like to thank the National Science Council of the Republic of China (Taiwan) for financial support of this research under contract numbers NSC 97-2221-E-009-126-MY3, NSC 99-2221-E-009-102-MY3 and NSC 99-2221-E-009-104-MY3.

References

1. Allis, L.V.: Searching for solutions in games and artificial intelligence. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands (1994)
2. Allis, L.V., van den Herik, H.J., Huntjens, M.P.H.: Go-Moku solved by new search techniques. Comput. Intell. **12**, 7–23 (1996)
3. van den Herik, H.J., Uiterwijk, J.W.H.M., Rijswijck, J.V.: Games solved: now and in the future. Artif. Intell. **134**, 277–311 (2002)
4. ICGA (International Computer Games Association). http://ticc.uvt.nl/icga/
5. Lin, H.-H., Sun, D.-J., Wu, I.-C., Yen, S.-J.: The 2010 TAAI computer-game tournaments. ICGA J. **34**(1), 51–55 (2011)

6. Lin, P.-H., Wu, I.-C.: NCTU6 wins in the Man-Machine Connect6 championship 2009. ICGA J. (SCI) **32**(4), 230–233 (2009)
7. Little Golem website. http://www.littlegolem.net/
8. Renju International Federation: The International Rules of Renju. http://www.renju.net/study/rifrules.php
9. Taiwan Connect6 Association: Connect6 Homepage. http://www.connect6.org/
10. TCGA Association: TCGA Computer Game Tournaments. http://tcga.ndhu.edu.tw/TCGA2011/
11. Allis, Private Communication (2012)
12. Thomsen, T.: Lambda-search in game trees - with application to Go. ICGA J. **23**(4), 203–217 (2000)
13. Wu, I.-C., Huang, D.-Y., Chang, H.-C.: Connect6. ICGA J. **28**(4), 234–242 (2006)
14. Wu, I.-C., Huang, D.-Y.: A new family of k-in-a-row games. In: The 11th Advances in Computer Games Conference (ACG'11), pp. 180–194, Taipei, Taiwan, (2005)
15. Wu, I.-C., Lin, P.-H.: NCTU6-Lite wins Connect6 tournament. ICGA J. **31**(4), 240–243 (2008)
16. Wu, I.-C., Lin, P.-H.: Relevance-Zone-Oriented proof search for Connect6. IEEE Trans. Comput. Intell. AI Games (SCI) **2**(3), pp. 191–207 (2010)
17. Wu, I.-C., Yen, S.-J.: NCTU6 wins Connect6 tournament. ICGA J. **29**(3), 157–158 (2006)
18. Yen, S.-J., Yang, J.-K.: 2-Stage Monte Carlo tree search for Connect6. IEEE Trans. Comput. Intell. AI Games (SCI) **3**(2), pp. 100–118, ISSN: 1943–068X, (2011). doi:10.1109/TCIAIG.2011.2134097