Chapter 5

Go-Moku

In this chapter we discuss the application of pn-search and db-search to go-moku. In the previous chapter we stated two goals for chapters 4 and 5, which we repeat here. The first goal is to explain in detail how pn-search and db-search have been applied to two combinatorially complex problems. The second goal is to show that qubic and go-moku can be solved, thereby positively answering our first research question (cf. section 1.4) for two specific games.

In several ways, qubic and go-moku are related games, with go-moku being the more complex one. The relationship between qubic and go-moku is expressed in the organization of this chapter: almost every section has a corresponding section in chapter 4. We mention this relationship for readers who are particularly interested in the application of db-search or pn-search. Comparing corresponding sections on qubic and go-moku may provide additional insight in these algorithms.

The chapter is organized as follows. In section 5.1 we provide a background to investigations in go-moku. The rules of go-moku and common strategies are presented in section 5.2. The application of db-search to go-moku is described in section 5.3. The role of pn-search in the solution of go-moku is explained in section 5.4. The results of our investigations are presented in section 5.5.

5.1 Background

Among the games of the Olympic List, go-moku has the simplest rules: two players (black and white) alternate placing stones on a 15×15 square lattice with the goal of obtaining a line of exactly five consecutive stones of the player's color. While its roots lie in China and Japan, it is also popular in several countries of Europe and the former Soviet-Union. Part of go-moku's popularity must be ascribed to the fact that it can be played with pencil and paper, allowing it to be played virtually everywhere (including classrooms) by virtually everyone (including bored students).

In Japan professional renju players (renju being a complicated variant of go-moku) have studied go-moku in detail and have stated that the player to move first (black) has an assured win (Sakata and Ikawa, 1981). These statements are sometimes accompanied by a list of main variations, such as the 32-page analysis in Sakata and Ikawa (1981). Close examination of these analyses reveals that in each position only a small number of white moves are analyzed. For example, after black's first move at the center of a 15×15 board, white has 35 distinct moves, of which 2 are adjacent to black's first move, ignoring symmetrically equivalent moves. In Sakata and Ikawa (1981) only the variations after the 2 moves adjacent to black's first move are discussed. As far as we know, prior to this work no complete proof of black's win in go-moku has been published.

Until this study, all go-moku programs have been defeated at least once or been in a lost position when playing black. As an example of the latter we mention the game between the go-moku 1991 world-champion program *Vertex* (black) and the program *Polygon* (white). *Vertex* maneuvered itself into a position provably lost for black (Uiterwijk, 1992a). As an aside we note that *Polygon* played its first move non-adjacent to the first black stone, indicating that finding a win in such a variation may not be entirely obvious.

Summarizing, go-moku is assumed to be a first-player win but, as far as we know, no complete proof has been published nor has any go-moku program ever been shown to be unbeatable when playing black.

At this point we reiterate our remark of section 4.1. In earlier publications we have used the term *threat-space search* for the application of db-search to qubic and go-moku. In this text we only use the term db-search.

5.2 Rules and strategies

Go-moku is a two-player game, related to the well-known trivial game of tic-tac-toe. While in tic-tac-toe players must create a line of three consecutive markers of their color on a restricted 3×3 board, in go-moku players must create a line of five on a practically unrestricted lattice. Through the years, several variants of go-moku have been developed, which are described in detail in section 5.2.1. Next, threats and threat trees are discussed in section 5.2.2. Finally, in section 5.2.3 some insight is given into the way human go-moku experts think.

5.2.1 Rules

In go-moku, simple rules lead to a highly complex game, played on the 225 intersections of 15 horizontal and 15 vertical lines. Going from left to right the vertical lines are lettered from a to o; going from the bottom to the top the horizontal lines are numbered from 1 to 15. Two players, black and white, move in turn by placing a stone of their own color on an empty intersection, henceforth called a *square*. Black starts the game. The player who first makes a line of five consecutive stones of her color (horizontally, vertically or diagonally) wins the game. The stones once placed on the board during the game never move again nor can they be captured. If the board is completely filled, and no one has five-in-a-row, the game is drawn.

Go-moku variants

Many variants of go-moku exist; they all restrict the players in some sense, mainly reducing the advantage of black's first move. We mention four variants.

Non-standard boards In the early days the game was played on a 19×19 board, since go boards have that size. Some people prefer to think of go-moku as being played on an infinite board. However, a larger board increases black's advantage (Sakata and Ikawa, 1981), which resulted in the standard board size of 15×15.

Free-style go-moku An overline is a line of six or more consecutive stones of the same color. In this variant, an overline is regarded as a win.

Standard go-moku In the variant of go-moku played most often today, an overline does not win (this restriction applies to both players). Only a line of exactly five stones is considered as a winning pattern.

Renju A professional variant of go-moku is renju. White is not restricted in any way, e.g., an overline wins the game for white. However, black is not allowed to make an overline, nor a so-called *double three* or *double four* (cf. Sakata and Ikawa (1981)). If black makes any of these patterns, she is declared to be the loser. Renju is not a symmetric game: to play it well requires different strategies for black and for white. Even though black's advantage is severely reduced, she still seems to have the upper hand.

We have investigated both free-style go-moku and standard go-moku. We remark that in this chapter we discuss free-style go-moku unless it is explicitly stated otherwise.

Opening restrictions

In an attempt to make the game less unbalanced, opening restrictions have been imposed on black. We mention two such restrictions.

Professional go-moku In professional go-moku, black is forced to make her first move in the center of the board. White must play her first move at one of the eight squares adjacent to black's first move. Black's second move must be outside the set of 5×5 squares centered by black's first stone.

Professional renju In professional renju, the game starts with two players which are named *temporary black* and *temporary white*. Temporary black plays her first move at the center of the board, while temporary white plays her first move adjacent to the black stone on the board. Due to symmetry, there are only two distinct first moves for temporary white. For each of these two, there are 12 selected squares where temporary black is allowed to play her second move. Thus, there are 24 possible 3-ply sequences in this variant. Next, temporary white may choose between playing black or white for the remainder of the game. Temporary black automatically plays the other color. Then, white plays her second move. Finally, black selects two squares for her third move and gives white the choice between these two. From there, the game continues according to the rules of standard renju.

In our research we have investigated variants of go-moku without any opening restrictions.

5.2.2 Threats and threat trees

We describe the four types of threats in go-moku, followed by a discussion of threat trees and winning threat trees.

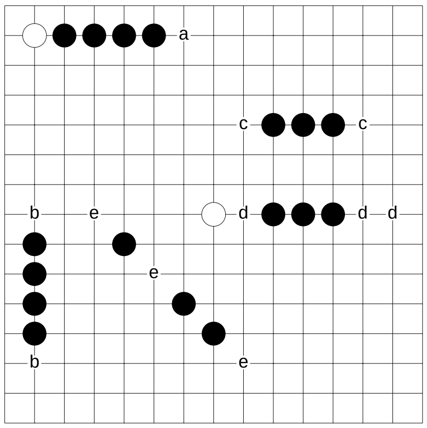


Figure 5.1: Threats in go-moku.

Threats

In go-moku a threat is an important notion; the main types have descriptive names: the *four* (figure 5.1a) is defined as a line of five squares, of which the attacker has occupied any four, with the fifth square empty; the *straight four* (figure 5.1b) is a line of six squares, of which the attacker has occupied the four center squares, while the two outer squares are empty; the *three* (figure 5.1c and 5.1d) is either a line of seven squares of which the three center squares are occupied by the attacker and the remaining four squares are empty, or a line of six squares with three consecutive squares of the four center squares occupied by the attacker and the remaining three squares empty; the *broken three* (figure 5.1e) is a line of six squares of which the attacker has occupied three non-consecutive squares of the four center squares, while the other three squares are empty. A winning pattern, i.e., a line of five squares, all occupied by one player, is named a *five*.

If a player constructs a four, she threatens to win on the next move. Therefore, the threat must be countered immediately at the empty square of the four. If a straight four is constructed, the defender is too late, since there are two squares where the attacker can create a five at her next move (unless, of course, the defender has the opportunity to win at her next move). With a three, the attacker threatens to create a straight four at her next move. Thus, even though the threat has a depth of two moves, it must be countered immediately. If an extension at both sides is possible (figure 5.1c), then there

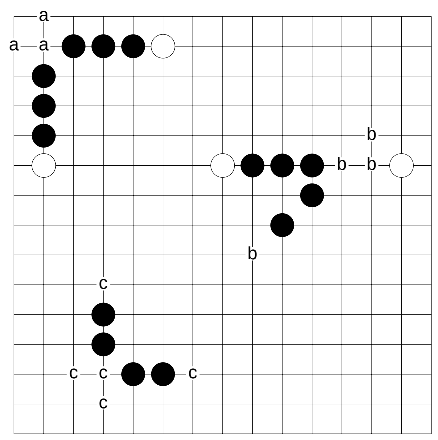
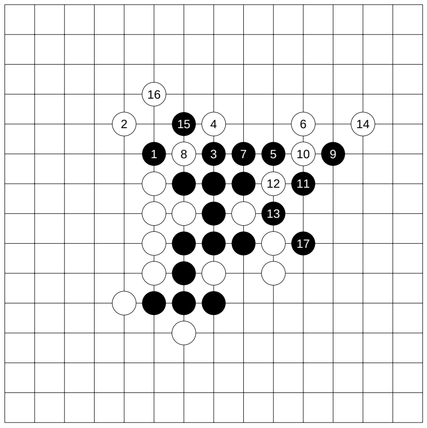


Figure 5.2: Complicated threats.

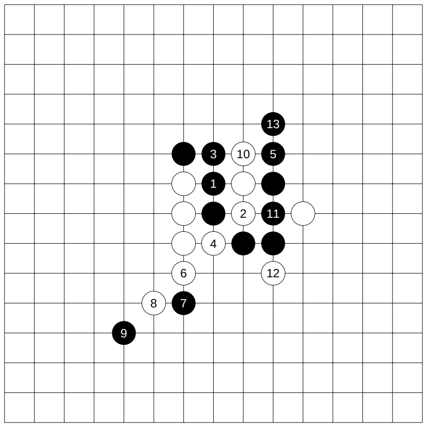
are two defensive moves: both directly adjacent to the attacking stones. If only one extension is possible then three defensive moves are available (figure 5.1d). Moreover, against a broken three, three defensive moves exist (figure 5.1e).

We remark that more complicated threats exist, which threaten to win in two or more moves. Three examples are shown in figure 5.2, in each of which black threatens to play at the intersection of the two lines of black stones. In figure 5.2a, black threatens to create a double four, in figure 5.2b, black threatens to create a four-three, and in figure 5.2c, black threatens to create a double three. Each of these is a winning pattern. White can counter the threats of figure 5.2 in 3, 4 and 5 possible ways, respectively.

In our research we have not included the patterns of figure 5.2 as threats for three reasons. First, the large number of defensive moves per threat does not combine well with our transformation of the winning threat-tree search to a single-agent search, as described in section 5.3.1. Second, recognizing threats which consist of a single line on the board can be performed more efficiently than recognizing threats which consist of combinations of lines. Third, the threats shown in figure 5.2 are only a small sample of the complete set of more complicated threat patterns, making inclusion of all possible threats of go-moku a complex task. In Uiterwijk (1992b) a program based on a large set of threat patterns is described.



(a) Fours only.



(b) Threes only.

Figure 5.3: Winning threat variations

Threat trees

To win the game against any opposition a player needs to create a *double threat* (either two threes, two fours, or a three and a four). In most cases, several threats are executed before a double threat occurs. A tree in which each attacker move is a threat is called a *threat tree*. A threat tree leading to a (winning) double threat in each variation is called a *winning threat tree*. A variation in a winning threat tree is called a *winning threat variation*. Each threat in the tree forces the defender to play a move countering the threat. Hence, the defender's possibilities are limited.

In figure 5.3a a position is shown in which black can win through a winning threat variation consisting of fours only. Since a four must be countered immediately, the whole sequence of moves is forced for white.

In figure 5.3b a position is shown in which black wins through a winning threat variation consisting of threes, twice interrupted by a white four. As mentioned earlier, white has at each turn a limited choice. During the play, she can create fours as is shown in figure 5.3b. Still, her loss is inevitable.

5.2.3 Human strategies

During the second and third Computer Olympiad (Levy and Beal, 1991; Van den Herik and Allis, 1992), we observed two human expert go-moku players (A. Nosovsky, 5th dan and N. Alexandrov, 5th dan). These Russian players are involved in two of the world's strongest go-moku playing programs (*Vertex* and *Stone System*). While observing the experts, it became clear that they are able to find quickly sections on the board where a winning threat tree can be created, regardless of the number of threes which are part of the winning threat tree. The depth of these winning threat trees are typically in the range of 5 to 20 ply.

The way a human expert finds winning threat trees so quickly can be broken down into the following four steps.

1. A section of the board is chosen where the configuration of stones seems favorable for the attacking player. It is then decided whether enough attacking stones can collaborate making it useful to search for a winning threat tree. This decision is based on a "feeling", which comes from a long experience in judging patterns of stones (cf. De Groot (1965)).

2. Threats are considered, and in particular the threats related to other attacking stones already on the board. Defensive moves by the opponent are mostly disregarded.

3. As soon as a variation is found in which the attacker can combine her stones to form a double threat, it is investigated how the defender can refute the potential winning threat variation. Whenever the opponent has more than one defensive move, an examination is made to see whether the same threats work in all variations of the threat tree. Moreover, it is investigated whether the opponent can insert one or more fours, effectively neutralizing the attack.

4. If only a few variations of the tree do not lead to a win via the same threat variation, an examination is made to see whether the remaining positions can be won via other winning threat trees.

In practical play, a winning threat tree often consists of a single set of attacking moves applicable to each variation of the tree, independent of the defensive moves.

We remark that the size of the state space is considerably reduced by first searching for one side (the attacker). Only if a potential winning threat tree is found is the impact of defensive moves investigated. This approach is supported by the analyses given in (Sakata and Ikawa, 1981). When presenting a winning threat tree, they only provide the moves for the attacker, thus indicating that the set of attacking moves works irrespective of the defensive moves. Possible fours which the defender can create without refuting the threat tree can be neglected altogether

In positions without winning threat trees, the moves to be played preferably increase the potential for creating threats or, whenever defensive moves are called for, the moves chosen will reduce the opponents potential for creating threats. The human evaluation of the potential of a configuration is based on two aspects: (1) direct calculations of the possibilities, (e.g., if the opponent does not answer in that section of the board) and (2) a so-called good shape (i.e., configurations of which it is known that stones collaborate well).

In section 5.3 we model the above thinking process in our application of db-search to go-moku.

5.3 Applying db-search

As mentioned before, threat trees play a dominant role in go-moku. To play go-moku well, it would be advantageous to have a module which determines whether a winning threat tree exists. Our application of db-search to go-moku is restricted to searching for winning threat trees.

This section consists of four parts. First, in section 5.3.1 we describe how the adversary-agent state space, if restricted to a subset of all possible threat trees, can be transformed into a single-agent state space. Second, in section 5.3.2 we illustrate how the single-agent state space thus created for go-moku fits in the framework for db-search as presented in chapter 3. Third, in section 5.3.3 we discuss properties of the single-agent state space for go-moku which have not been included in the framework of section 5.3.2. For each of these properties it is explained how our implementation of db-search handles them. Fourth, in section 5.3.4 heuristics are described which lead to a significantly improved efficiency, at the cost of a slightly reduced efficacy. Fourth, in section 5.3.5 we describe the additional requirements necessary to apply db-search to standard go-moku instead of free-style go-moku.

5.3.1 A single-agent search in go-moku

Our description of the single-agent state space in go-moku consists of a set of definitions, an interpretation of the definitions, and the transformation of the adversary-agent state space to a single-agent state space.

Definitions

In the previous sections we have introduced the concept of threats, threat trees and winning threat trees. For our application of db-search to go-moku, we formally define the notions *threat* (definition 5.1), *reply* (definition 5.2), *threat sequence* (definition 5.3), *potential winning threat sequence* (definition 5.4) and *winning threat sequence* (definition 5.5).

Definition 5.1 *A* threat *in* go-moku *is a move by the attacker creating a five, a straight four, a four, a three or a broken three. A five and a straight four are called* double threats*, while a four, three and broken three are called* single threats*. The* squares related to a threat *are the 5 (five and four), 6 (straight four, three, broken three) or 7 (three) squares in the line of the threat (cf. section 5.2.2).*

Definition 5.2 *A* reply *to a threat T in* go-moku *is the set of defender moves R, such that each element of R counters T. Against a five and a straight four, R is empty, against a four, R consists of one move, against a three R consists of two or three moves, and against a broken three, R consists of three moves.*

Definition 5.3 *A* threat sequence *in* go-moku *is any sequence of moves $(a_1, d_1, a_2, d_2, \ldots, a_n, d_n)$, with $n \geq 1$, such that each a_i, $1 \leq i \leq n$ is a single threat, and each d_i is the reply to a_i.*

Definition 5.4 *A* potential winning threat sequence *in* go-moku *is any sequence $(a_1, d_1, \ldots, a_n, d_n, a_{n+1}, d_{n+1})$, such that $(a_1, d_1, \ldots, a_n, d_n)$ is a threat sequence, a_{n+1} is a double threat and d_{n+1} is the reply to a_{n+1}.*

Definition 5.5 *A* winning threat sequence *in* go-moku *is a potential winning threat sequence $(a_1, d_1, \ldots, a_n, d_n, a_{n+1}, d_{n+1})$, for which it has been checked that the defender cannot counter the threat sequence by:*

1. *interjecting a sequence of threats the attacker must respond to, leading to a win for the defender*

2. *interjecting a sequence of threats the attacker must respond to, leading to occupation of a square related to a threat a_i, before the defender has played the reply to d_i.*

Interpretation

Here we elaborate on the definitions presented above. Definition 5.1 defines threats in accordance with the definitions of section 5.2.2. The only difference is our inclusion of the five as a threat, and naming the straight four and the five double threats. The reason for doing so is explained below.

When a double three is created, it is assumed that the defender counters one of them, allowing the attacker to convert the remaining three into a straight four at the next move. When a double four is created, it is assumed that the defender counters one of them, allowing the attacker to convert the remaining four into a five at the next move. When a four-three is created, depending on the threat countered by the defender, the attacker can create either a five or a straight four. Thus, we may recognize double threats one move after they appear in the form of straight fours or fives.

The definition of a reply forms a crucial step in our conversion of the adversary-agent state space of go-moku into a single-agent state space. Human strategies imply that often threat trees are found such that in each variation the same attacking moves are played. In other words, the choice between defensive moves in such threat trees is irrelevant. We convert these threat trees to threat sequences by allowing the defender to play *all* defensive moves as a single reply. In figure 5.4, we have depicted such a winning threat sequence, consisting of four threats. After black 1, white has the three-move reply 2. After black 3, white has the two-move reply 4. After black 5, white has the three-move reply 6. Black 7 creates a straight four, to which the reply set is empty.

Clearly, in free-style go-moku, having extra stones on the board is never a disadvantage. Thus, if a variation wins for the attacker when the defender is allowed to play replies consisting of multiple stones, then the variation wins also if the defender is forced to select one stone from each multiple-stone reply.

Positions exist in which the multiple-stone reply leads to counter play for the defender, while the attacker would win in all variations through the same attacking moves if the defender were restricted to playing one stone per reply, but these are rare.

A potential winning threat sequence as defined in definition 5.4 has investigated only *local* defensive moves, i.e., after each threat, it is assumed that the defender must immediately counter the threat. A winning threat sequence has also been checked for *global* defensive moves, i.e, that the squares not related to the threat sequence have been investigated for their influence

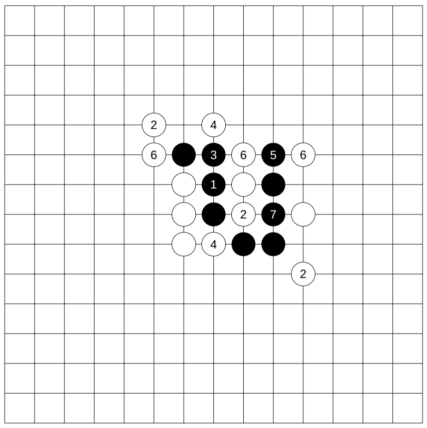


Figure 5.4: White defending with multiple-stone replies

on the success of the threat sequence.

Adversary-agent vs. single-agent

As we have seen, in (winning) threat sequences, each reply by the defender is implied by the previous attacker move. Therefore, we may conceptually merge these two moves into a single meta-move.

The state space created by these meta-moves is no longer an adversary-agent state space, but instead a single-agent state space. In the remainder of this section, when discussing meta-moves, we assume that the attacker move and defender move in a meta-move are made simultaneously.

5.3.2 A db-search framework for go-moku

In this section we define a db-search framework for the single-agent state space of go-moku, defined in the previous section. We mention that the framework only involves *local* defensive moves, while ignoring *global* defensive moves. Global defensive moves of a position will be discussed in section 5.3.3. The terminology introduced in chapter 3 is used throughout this chapter.

Notation

Lines of five, six and seven squares play an important role in go-moku. For notational purposes, we define the following sets.

$$\begin{aligned}
G_5 &= \{\{s_1, s_2, \ldots, s_5\} \mid s_1, \ldots, s_5 \text{ form a line of five squares}\} \\
G_6 &= \{\{s_1, s_2, \ldots, s_6\} \mid s_1, \ldots, s_6 \text{ form a line of six squares}\} \\
G_7 &= \{\{s_1, s_2, \ldots, s_7\} \mid s_1, \ldots, s_7 \text{ form a line of seven squares}\}
\end{aligned}$$

We mention that on a 15×15 board, G_5 has 572 elements, G_6 has 500 elements and G_7 has 432 elements.

Furthermore, we define a linear order on the squares of the go-moku board, such that $a1 < a2 < \ldots < a15 < b1 < \ldots < o15$. Clearly, the outer squares of a line are always minimal and maximal within the line, with respect to this ordering.

Attributes

The set U of all attributes is defined as follows. $U = \{S(i, x) \mid a1 \leq i \leq o15 \wedge x \in \{\circ, \bullet, \cdot\}\}$. Attribute $S(i, x)$ represents the fact that square i is occupied by the attacker ($\circ$), occupied by the defender ($\bullet$), or empty ($\cdot$). It can easily be checked that U has 675 elements.

Operators

The operator f_{FI,g_5} (five), for $g_5 = \{s_1, \ldots, s_5\}$ and $g_5 \in G_5$, is defined as follows.

$$\begin{aligned}
f^{pre}_{FI,g_5} &= \{S(s_1, \circ), S(s_2, \circ), S(s_3, \circ), S(s_4, \circ), S(s_5, \cdot)\} \\
f^{del}_{FI,g_5} &= \{S(s_5, \cdot)\} \\
f^{add}_{FI,g_5} &= \{S(s_5, \circ)\}
\end{aligned}$$

The operator f_{SF,g_6} (straight four), for $g_6 = \{s_1, \ldots, s_6\}$ and $g_6 \in G_6$, and $s_1 < s_2, s_3, s_4, s_5 < s_6$, is defined as follows.

$$\begin{aligned}
f^{pre}_{SF,g_6} &= \{S(s_1, \cdot), S(s_2, \circ), S(s_3, \circ), S(s_4, \circ), S(s_5, \cdot), S(s_6, \cdot)\} \\
f^{del}_{SF,g_6} &= \{S(s_5, \cdot)\} \\
f^{add}_{SF,g_6} &= \{S(s_5, \circ)\}
\end{aligned}$$

The operator f_{FO,g_5} (four), for $g_5 = \{s_1, \ldots, s_5\}$ and $g_5 \in G_5$, is defined as follows.

$$\begin{aligned} f^{pre}_{FO,g_5} &= \{S(s_1,\circ), S(s_2,\circ), S(s_3,\circ), S(s_4,\cdot), S(s_5,\cdot)\} \\ f^{del}_{FO,g_5} &= \{S(s_4,\cdot), S(s_5,\cdot)\} \\ f^{add}_{FO,g_5} &= \{S(s_4,\circ), S(s_5,\bullet)\} \end{aligned}$$

The operator f_{BT,g_6} (broken three), for $g_6 = \{s_1, \ldots, s_6\}$ and $g_6 \in G_6$, and $s_1 < s_2, s_3, s_4, s_5 < s_6$ and s_4 neither minimum nor maximum in $\{s_2, s_3, s_4, s_5\}$, is defined as follows.

$$\begin{aligned} f^{pre}_{BT,g_6} &= \{S(s_1,\cdot), S(s_2,\circ), S(s_3,\circ), S(s_4,\cdot), S(s_5,\cdot), S(s_6,\cdot)\} \\ f^{del}_{BT,g_6} &= \{S(s_1,\cdot), S(s_4,\cdot), S(s_5,\cdot), S(s_6,\cdot)\} \\ f^{add}_{BT,g_6} &= \{S(s_1,\bullet), S(s_4,\bullet), S(s_5,\circ), S(s_6,\bullet)\} \end{aligned}$$

The operator f_{T2,g_7} (three with 2 reply moves), for $g_7 = \{s_1, \ldots, s_7\}$ and $g_7 \in G_7$, and $s_1 < s_2 < s_3, s_4, s_5 < s_6 < s_7$, is defined as follows.

$$\begin{aligned} f^{pre}_{T2,g_7} &= \{S(s_1,\cdot), S(s_2,\cdot), S(s_3,\circ), S(s_4,\circ), S(s_5,\cdot), S(s_6,\cdot), S(s_7,\cdot)\} \\ f^{del}_{T2,g_7} &= \{S(s_2,\cdot), S(s_5,\cdot), S(s_6,\cdot)\} \\ f^{add}_{T2,g_7} &= \{S(s_2,\bullet), S(s_5,\circ), S(s_6,\bullet)\} \end{aligned}$$

The operator f_{T3,g_6} (three with 3 reply moves), for $g_6 = \{s_1, \ldots, s_6\}$ and $g_6 \in G_6$, and $s_1 < s_2, s_3, s_4, s_5 < s_6$ and s_2 either minimum or maximum in $\{s_2, s_3, s_4, s_5\}$, is defined as follows.

$$\begin{aligned} f^{pre}_{T3,g_6} &= \{S(s_1,\cdot), S(s_2,\cdot), S(s_3,\circ), S(s_4,\circ), S(s_5,\cdot), S(s_6,\cdot)\} \\ f^{del}_{T3,g_6} &= \{S(s_1,\cdot), S(s_2,\cdot), S(s_5,\cdot), S(s_6,\cdot)\} \\ f^{add}_{T3,g_6} &= \{S(s_1,\bullet), S(s_2,\bullet), S(s_5,\circ), S(s_6,\bullet)\} \end{aligned}$$

The set of all operators U_f is defined as follows.

$$\begin{aligned} U_f = \; & \{f_{FI,g_5} \mid g_5 \in G_5\} \cup \{f_{SF,g_6} \mid g_6 \in G_6\} \cup \{f_{FO,g_5} \mid g_5 \in G_5\} \cup \\ & \{f_{BT,g_6} \mid g_6 \in G_6\} \cup \{f_{T2,g_7} \mid g_7 \in G_7\} \cup \{f_{T3,g_6} \mid g_6 \in G_6\} \end{aligned}$$

We mention that on a 15×15 board, U_f contains 3076 operators, of which each can be applied in more than one way, resulting in a total number of 23596 possible applications of operators.

Initial state and goal states

The initial state consists of exactly 225 attributes, one per square indicating the contents of the square. Each possible configuration of black, white and empty squares in which neither player has occupied a line of five can serve as initial state. The set U_g of goal states is independent of the initial state, and is defined as follows.

$$\begin{aligned} U_g = \; & \{ \; \{S(s_1,\circ), S(s_2,\circ), S(s_3,\circ), S(s_4,\circ), S(s_5,\circ)\} \mid \\ & \quad \{s_1,s_2,s_3,s_4,s_5\} \in G_5\} \cup \\ & \{ \; \{S(s_1,\cdot), S(s_2,\circ), S(s_3,\circ), S(s_4,\circ), S(s_5,\circ), S(s_6,\cdot)\} \mid \\ & \quad \{s_1,s_2,s_3,s_4,s_5,s_6\} \in G_6 \wedge s_1 < s_2,s_3,s_4,s_5 < s_6\} \end{aligned}$$

In other words, each state containing a five or straight four is a goal state. U_g is not singular.

Properties of the go-moku framework

The framework we have described above is monotonous. Furthermore, we can easily restrict ourselves to non-redundant paths. If U_g were singular, our U_k would be complete.

We can create a singular $U_{g'}$, by defining a special goal attribute G and operators which transform any element of U_g into G, which would result in a complete U_k. A discussion of the completeness of U_k would be premature, however, since so far we have ignored global defensive moves.

5.3.3 Go-moku specific enhancements to db-search

The db-search framework for go-moku presented in the previous section focuses only on the local defensive moves. For those moves we defined replies such that each defender move was forced, allowing us to transform the search into a single-agent search.

A search for global defensive strategies is only necessary to investigate whether a potential winning threat sequence is correct. Thus, given such a threat sequence, it should be investigated whether the defender has alternatives to the local reply to refute the threat sequence. To investigate the global defensive strategies, we perform single-agent searches, this time fixing the *attacker* choices. After each attacker move specified in the threat sequence, the resultant position is investigated for a global defensive strategy by the defender. We describe the investigations in four steps.

First, we define the *threat categories*, which play an important role in determining for each position the types of global defensive moves available. Second, we describe two ways in which global defensive moves may successfully counter a potential threat sequence. Third, we describe a set of parameters for db-search. Fourth, we describe how the module searching for winning threat sequences is composed of a series of db-searches.

Threat categories

The operators defined in section 5.3.2 can be divided in three categories. Category 0 consists of the five, category 1 of the straight four and four, and category 2 consists of the three and the broken three. Using these categories we can state exactly what kind of global defensive moves may be interjected by the defender while countering a threat sequence. Against a threat from category i, only threats from categories j can be used as global defensive moves, with $j < i$. Thus, against a five no global defensive moves exist, against a (straight) four only a five can serve as global defensive move, while against a three or broken three, both fives, straight fours and fours may serve as global defensive moves.

The above relation between global defensive moves and threat categories can easily be verified by noting that each threat in category i threatens to win in exactly i moves.

Global defensive strategies

In section 5.3.1 we have listed two ways in which the defender may successfully counter a threat by interjecting global defensive moves. First, she may create a sequence of threats leading to a win. Second, she may create a sequence of threats leading to the occupation of a square in the threat sequence.

Here we describe how db-search can be used to determine whether such a global defensive strategy exists. Our application of db-search for this purpose is such that we may erroneously decide that a defensive strategy exists, thus rejecting a winning threat sequence for the attacker, but that we will never overlook the existence of a defensive strategy.

To prevent confusion arising from the terms attacker and defender in this context, we assume here that player A has found a potential winning threat sequence, and we investigate whether player B has a global defensive strategy after move a_i by A. Three remarks concerning the application of db-search to search for global defensive strategies for player B are in order.

1. The goal set U_g for player B should be extended with singleton goals for occupying any square in threat a_j or reply d_j, with $j \geq i$.

2. If B finds a potential winning threat sequence (i.e., a global defensive strategy against the potential winning threat sequence of A), this threat sequence is not investigated for counter play of player A. Instead, in such a case we always assume that A's potential winning threat sequence has been refuted.

3. In the application of db-search for player B, only threats of categories less than the category of the threat played by A may be applied. Thus, in a db-search for player B, only threats having replies consisting of a single move are applied.

If we examine the description of db-search for B, we may find that the search is monotonous and contains no redundant paths. As argued before, U_g can be easily transformed into a singular $U_{g'}$, without a conceptual difference in the resulting U_k. Since any sequence found for player B is accepted as refutation of the potential winning threat sequence of A, we claim that if application of db-search does not find a global defensive strategy, such a strategy does not exist for player B.

We stress this point as it is a vital element in the process of solving go-moku: we must ensure that in no position we accept a threat sequence as winning, if the threat sequence could be refuted.

Parameters to db-search

Above, we have seen that db-search is used to find potential winning threat sequences as well as to investigate whether the defender has a global defensive strategy refuting a potential winning threat sequence. These searches are all performed by the same module, whose parameters are listed below.

1. The *position* to which db-search is to be applied.

2. The *attacker*, i.e., the player for whom a db-search is applied.

3. The *goal squares*, i.e., the set of squares, which, if one is occupied by the attacker, terminates the search.

4. The *defensive check option*. This is a Boolean value indicating whether a potential winning threat sequence should be investigated for counter play.

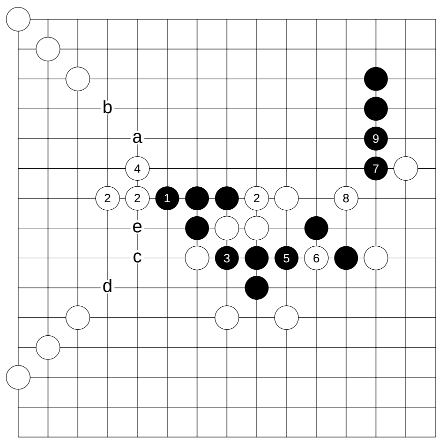


Figure 5.5: White refutes a potential winning threat sequence.

5. The *maximum category*, i.e., only threats of this category and lower categories may be applied.

The winning threat sequence module

Here we present a step by step description of the winning threat sequence module, with the aid of the position in figure 5.5.

To find the winning threat sequence for black in the position before black 1 of figure 5.5, db-search may be called with (1) that position; (2) attacker black; (3) the empty set of goal squares; (4) the defensive check option at value true; (5) maximum category 2. If the potential winning threat sequence shown in figure 5.5 is found, db-search will be called five more times, after black 1, black 3, black 5, black 7 and black 9. The parameters to db-search after, for instance, black 1 are: (1) the position after black 1; (2) attacker white; (3) the set consisting of the 28 squares related to the threats black 1 (7 squares), black 3 (5 squares), black 5 (5 squares), black 7 (5 squares) and black 9 (6 squares); (4) the defensive check option at value false; (5) maximum category 1.

After black 5, which is of category 1, black can only use a defensive strategy involving threats of category 0, i.e., fives only. However, to create a five after black 5, white should have created several fours after black 1 (of category 2), followed by the local defensive reply white 2. Therefore, we need to try threats of category 0 after black 5, for all positions which could arise after sequences of fours by white, in earlier global defensive strategy searches.

Indeed, if white, instead of playing 2 immediately after 1, interjects move *a* (followed by black's forced reply *b*) and move *c* (followed by black's forced *d*), then after white 2, black 3, white 4 and black 5, white can create a five at *e*.

Summarizing, to find the global defensive strategies, after each attacker move of category 2, a search for category 1 for the defender should be performed, while after each attacker move of category 1, a search for category 0 for the defender should be performed, from every position which could be reached by interjecting defender fours after previous threats of category 2 by the attacker.

5.3.4 Heuristically improving the efficiency of db-search

As we have argued before, the module which searches for winning threat sequences will only return a winning threat sequence if the winning threat sequence is guaranteed to lead to a win for the attacker. The opposite is not true: not all winning threat sequences will be found. This is caused by our acceptance of a global defensive strategy, without investigating whether the defensive strategy itself can be countered.

In the context of winning threat *trees* our search is far from complete, as we only find winning threat *sequences*, i.e., threat trees in which each variation leads to a win through the same attacking moves, in the same order.

In this section we present three heuristics which significantly increase the efficiency of our winning threat sequence module, at the cost of another (small) reduction in efficacy. Each of the heuristics, if at all applicable, is *not* applied during searches for global defensive strategies, in order to ensure that all existing refutations of potential winning threat sequences are found.

Global refutation

Our first heuristic for increasing the efficiency of db-search is based on the existence of *global refutations* in some positions. A global refutation is a configuration on the board which refutes all winning threat sequences of the attacker. An artificial example is depicted in figure 5.6.

Black to move has a large number of distinct potential winning lines at her disposal, each starting with a three. For instance, black 1 creates a double three immediately. White 2, however, creates a double four, thus successfully countering the three created by black 1. Alternative lines for black, such as

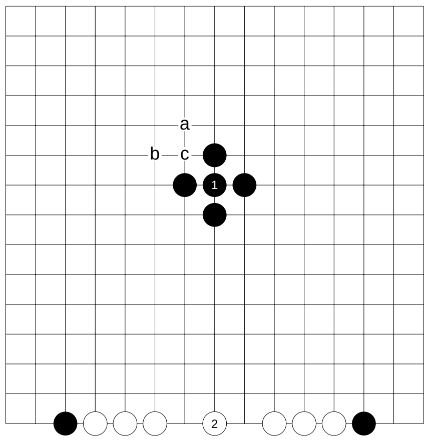


Figure 5.6: Global refutation of all potential winning lines.

black a, black b and black c, again creating a double three, are all also refuted by white 2.

Thus, while db-search, focusing on local defenses, finds many potential winning threat sequences, each of these is refuted by the search for global defensive strategies. Finding *all* several hundreds or thousands of potential winning threat sequences in such a position is clearly a waste of time.

As heuristic to recognize those positions, we check at the end of each db-search level the number of potential winning threat sequences investigated so far. If this number exceeds a preset threshold T, the search is terminated. Experiments showed that $T = 10$ leads to a largely increased efficiency, at a small cost in efficacy.

We remark that while searching for global defender strategies, the first potential winning threat sequence found is accepted as refutation. The search is therefore not influenced by this heuristic.

Category reduction

The category reduction heuristic is designed for a special type of global refutations. Let us suppose that the defender has a threat T_{c_1} of category c_1. If the attacker creates a threat T_{c_2} of category c_2, then either (1) $c_2 < c_1$, or (2) T_{c_2} should counter T_{c_1}, or (3) T_{c_1} is a refutation of T_{c_2}. As the search for potential winning lines does only consider local replies, countering T_{c_1} by T_{c_2} will only occur by accident.

Ignoring the option that this may happen, we obtain the category reduction heuristic: if in a node N of the db-search DAG, the defender has a threat of category c_1, for each descendent of N the attacker is restricted to threats of categories less than c_1.

We remark that this heuristic is switched off while searching for global defender strategies.

Restricted threes

The definitions of operator f_{T3,g_6} (three with 3 reply moves) and operator f_{T2,g_7} (three with 2 reply moves) imply that if the latter is applicable, the former is too. While in most positions where both are applicable they are interchangeable, operator f_{T2,g_7} is superior in that its reply consists only of 2 moves, thus diminishing the chances for counterplay. Only in rare occasions are both applicable, while only f_{T3,g_6} leads to a winning threat sequence.

To prevent the creation of threat sequences with as only difference the occurrence of f_{T3,g_6} instead of f_{T2,g_7}, we restrict application of f_{T3,g_6} to lines where f_{T2,g_7} is not applicable.

We remark that while searching for global defender strategies, only threats of categories 0 and 1 are applicable. The search is therefore not influenced by this heuristic.

5.3.5 Additional requirements for standard go-moku

Standard go-moku differs from free-style go-moku in the value of overlines: an overline is a win in free-style go-moku, while it is not in standard go-moku.

To apply our winning threat sequences module, as described in the previous sections, to standard go-moku, a few additional requirements are necessary. We discuss these requirements briefly.

First, we introduce the concept of a line extension. Second, we describe how a line extension influences a db-search for potential winning threat sequences. Third, we describe the influence of line extensions to the search for global defensive strategies.

Extensions

For each line $g \in G_5$, a square c is an *extension* of g, if $g \cup \{c\} \in G_6$. Similarly, for each line $g \in G_6$, a square c is an extension of g, if $g \cup \{c\} \in G_7$. We mention that the extension of a line $g \in G_7$ is defined analogously, after the set G_8 has been defined. The *extension set* of a line g, i.e., the set of all extensions of g consists of 0, 1 or 2 elements, depending on the position of g on the board, with respect to the board edge.

Line extensions and winning threat sequences

A winning threat sequence in standard go-moku must meet all the requirements for a winning threat sequence in free-style go-moku. An added requirement is that at the moment of execution of threat a_i, the squares in the extension set of a_i must not be occupied by an attacker stone.

An attacker stone may be placed at the extension of a threat in three distinct ways.

1. The stone was present in the initial position.
2. The stone is played while executing an earlier threat in the threat sequence.
3. The stone is played as forced response to a defender threat.

The first and second way of placing an attacker stone at a threat extension is checked during the db-search for potential winning threat sequences: an operator can only be applied if the extension squares are empty or occupied by the defender. During the combination stage of db-search, we ignore the occupation of extensions. Instead, after a potential winning threat sequence has been found, the extensions of all threats in the threat sequence are examined.

Line extensions and global defensive strategies

The third way of placing an attacker stone in a threat extension provides the defender with an extra global defensive strategy. This strategy fits as follows within the parameters provided to db-search. In addition to the set of goal squares provided for free-style go-moku, the set of extensions to the threats which have not yet been executed by the attacker is passed to db-search. A refutation of the potential winning threat sequence has been found, if one of the extensions has been occupied by the attacker (i.e., the player whose potential winning threat sequence is being examined).

Special attention must be paid to the multiple-stone replies. While having extra stones on the board does not harm a player in free-style go-moku, it may harm a player in standard go-moku. To ensure that each global defensive strategy is found, we perform the db-search for global defensive strategies as a free-style go-moku search. Thus, a potential winning threat sequence in standard go-moku may be refuted through a sequence of defender threats containing overlines.

5.4 Applying pn-search

To apply pn-search to go-moku, we need to convert the go-moku game tree into an AND/OR tree. This is described in section 5.4.1. Furthermore, we describe the enhancements to basic pn-search adopted for our go-moku implementation in section 5.4.2.

5.4.1 Go-moku as an AND/OR tree

Pn-search (as described in chapter 2) is an AND/OR-tree algorithm. To apply it to go-moku, we represent positions where black is to move as OR nodes, and positions where white is to move as AND nodes. A win for black is represented by the value true, while a draw and a win for white are represented by the value false. Thus, proving the pn-search tree means that black can win in the root positions, while disproving the pn-search tree means that white can achieve at least a draw.

In each OR node, black is to move. As evaluation function at such a node, we apply db-search with black as attacker. If db-search finds a winning threat sequence, the node evaluates to true, otherwise to the value unknown. In each AND node, white is to move. The same procedure as in OR nodes is applied, this time with white as attacker. If a winning threat sequence is found, the node evaluates to false, otherwise to the value unknown. A node representing a position with all 225 squares occupied and neither player having a winning configuration, is a draw, and therefore obtains value false, without applying db-search.

5.4.2 Enhancements

The above description explains how standard pn-search is applied to go-moku. However, five enhancements have been added to speed up the search. The enhancements are discussed in this section.

Transpositions

A DAG is created instead of a tree, using the algorithm described in section 2.3.3. This ensures that if a position has already occurred in the DAG, or if a position is equivalent through automorphisms to another position in the DAG, the position is not investigated again. We test for the 8 standard automorphisms of a square board.

Restricting black's moves

In go-moku, the average branching factor is more than 200. Most of these moves are unrelated to the battle at the center of the board and should be ignored. However, since we want to *prove* the value of the root position, we cannot simply ignore moves using heuristic selection functions.

A large reduction of the branching factor at the OR nodes can be made, however. Since we want to prove a win for black in the root position, it is sufficient to prove for each internal OR node that (at least) *one* child leads to a win for black. For each internal AND node *all* children must be proved.

Using these properties, we may at each OR node restrict black to, say, the N most-promising children, using a heuristic ordering function. If in the restricted game tree a proof of black's win is found, the same proof is valid in the full game tree. In our investigations presented in section 5.5 we have restricted black in each OR node to the 10 most-promising children. Before the ordering function is applied, we first restrict the set of all legal moves to the set of moves which counter the threats of the opponent, as described in the next section.

The heuristic ordering function used is rather simple: each square is assigned 4 points for each three with a two-stone reply, 3 points for each three with a three-stone reply, 2 points for each broken three, 2 points for each *open two*, which is defined as two black stones in the center of an otherwise empty line of 6 and 1 point for each broken two, which is defined as two black stones with a one-square gap in the center three squares of an otherwise empty line of 7. Among all children, the 10 children with the highest score are selected.

No points are given for the creation of a four. Creating a four is often only a strong move if it stops a threat of the opponent, or if it creates a winning threat sequence. Since a node is only expanded if no winning threat sequence exists and it is ensured that we select the 10 best moves among the moves which counter the existing threats of the opponent, there was no need to assign any points for creating fours.

Clearly, a thorough analysis of the strategic knowledge of experts would have led to a more refined move-ordering function. As we show in section 5.5, the function described here was sufficient for our purposes.

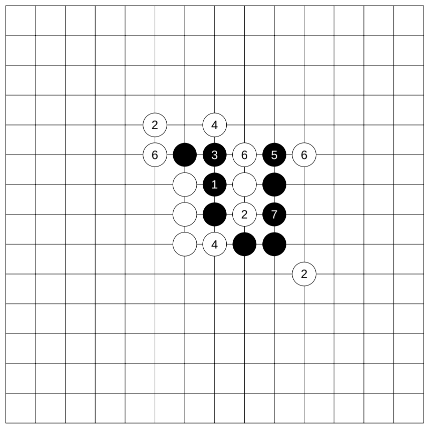


Figure 5.7: Black threatens to win by moves 1 through 7.

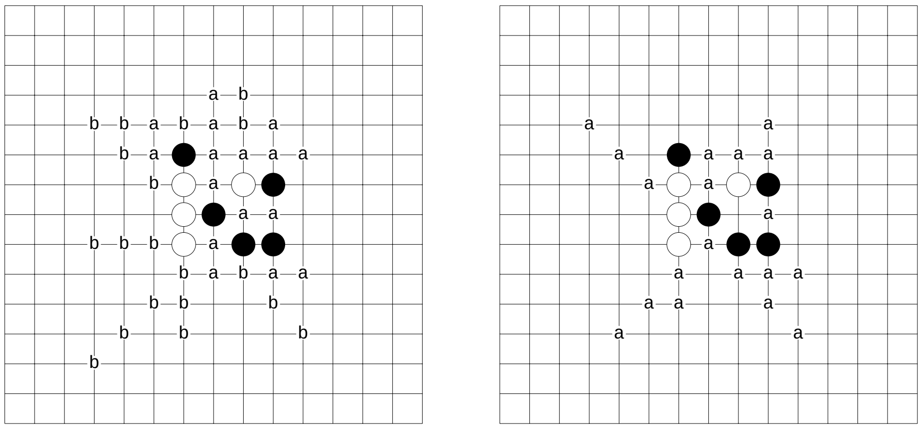
Related squares

As stated before, most of the approximately 200 legal moves per position are unrelated to the battle at the center of the board and should be ignored. Although we cannot ignore moves by white using heuristic selection functions, we may try to apply a winning threat sequence found as reply to one move to a large number of other moves. In this section we describe how this is done in a reliable way.

For each winning threat sequence of the attacker, we define the set of related squares as follows. An empty square c is *related* to a winning threat sequence in a given board position, if the threat sequence no longer wins, if c would have been occupied by the defender.

Before we use the notion of related squares, we introduce the term *implicit threat*, for any position where a player threatens to win through a winning threat sequence. In figure 5.7 black threatens to win through the threat sequence consisting of black 1 through 7. Therefore, the position is an implicit threat for black.

Now let us suppose that we have algorithms to determine whether a position is an implicit threat, and that we can determine for each winning threat sequence the set of related squares. Given a position with white to move, which is an implicit threat for black, we determine the set of squares related to the winning threat sequence. Then, it follows directly from the definition of related squares that we may restrict white to these related



(a) Superset of related squares (b) Related squares

Figure 5.8: Replies to the threat sequence of figure 5.7

squares.

Clearly, by determining implicit threats and sets of related squares in an efficient way, we could speed up our search. To determine an implicit threat, it suffices to make a null-move for the opponent (white in figure 5.7) and to apply db-search to find a winning threat sequence for black. Determining the exact set R of squares related to a winning threat sequence is computationally expensive. Instead, we determine a superset S of the set of related squares. The set consists of all squares meeting one of the following two conditions.

1. The square is in one of the lines of the threats in the winning threat sequence.

2. The square may be used in any counter threat by the opponent, in any of the global defensive strategy searches performed to investigate the winning threat sequence.

Using db-search S can be determined efficiently. Without proof we state that $R \subseteq S$. For empirical evidence of this claim we refer to section 5.5.4.

In figure 5.8a, we have shown the set S for the threat sequence of figure 5.7. The squares labeled a are part of the lines of the threats. The squares labeled b may, together with white stones on the board or the defensive moves in the threat sequence, form new defensive threats for white.

Iterated related squares

The related-squares concept can be used to even further reduce the set of white moves to be examined. After having determined the superset S of the set of related squares, an element s of S is selected. A white stone is placed at s, and the position is investigated with db-search. If no winning threat sequence is found, a child is added to the tree for s. Otherwise, the superset S_1 of squares related to the newly found winning threat sequence is determined. Only squares in $S \cap S_1$ need further be investigated, since all moves at other squares lead to a win through one of the two winning threat sequences found so far. This procedure is repeated until all moves have been examined.

In figure 5.8b we have marked the set of squares for which child nodes are grown. Of the 35 related squares of figure 5.8a (set S), only 19 squares (set R) remain in 5.8b.

The null-move heuristic and the related-squares heuristic are applied for both players in the pn-search DAG. For the attacker in the search (the player for which we select only 10 moves per node) we first determine the set of counter moves using the heuristics of this section, and then order the moves according to the move-ordering function of the previous section. Of course, if less than 10 counter moves exist, these are all selected.

The implicit-threat heuristic

The branching factor of go-moku is such that the search tree may become quickly intractable. To force black to select moves where white has a restricted number of moves, we evaluate a position which is not an implicit threat for black to false. Only early in the game tree (i.e., when there are less than 9 stones on the board), if no black move leads to an implicit threat, is the above restriction lifted.

We have found that no later than move 11 in the game, black can ensure that each move is an implicit threat. By enforcing this restriction, the size of the search tree is significantly reduced.

Heuristic (dis)proof number initialization

During our initial experiments, we have used the standard proof and disproof numbers initialization of 1 each. While studying the trees grown, it became apparent that pn-search tended to pursue some deep lines longer than desirable. This is mainly caused by continuously executing threats, without creating a potential for a winning threat sequence.

In qubic, as described in chapter 4, we chose to remove all threatening moves for the attacker from the search tree. We could safely do so, since our db-search implementation for qubic searched the full space of threatening sequences. The incompleteness of db-search in go-moku with respect to the space of all threat trees blocks a similar approach in go-moku. Instead, we have opted to attach a small penalty to all deep lines. At each frontier node the proof and disproof numbers are initialized to the number of full moves made from the root position. Thus, at depth d, the proof and disproof numbers are initialized to $1 + \lfloor d/2 \rfloor$.

This heuristic initialization ensures that forcing lines are not searched too deeply (before sufficient alternatives have been tried), without interfering with the essence of pn-search.

5.5 Solving go-moku

The program *Victoria* consists of the pn-search algorithm described in the previous section, using db-search as evaluation function. In this section we describe how *Victoria* solved both free-style go-moku and standard go-moku. First, we describe the I/O of *Victoria*. Second, it is explained how the game tree was split in several hundreds of subtrees. Third, we present statistics regarding the search process. Finally, we discuss the reliability of our results.

5.5.1 Victoria's I/O

The input to *Victoria* consists of (1) A go-moku position; (2) The game variant (free-style go-moku or standard go-moku); (3) The player to move; and (4) The maximum tree size for pn-search.

The output of *Victoria* consists of (1) the value upon termination of pn-search (true, false, unknown) (2) a database containing a record for each position in the solution tree. The database returned by *Victoria* is empty unless the value true was returned. For each record in the database representing a position with black to move, at least one child position will also be represented in the database. For each record in the database representing a position with white to move, only child positions are represented in the database in which black does not have a winning threat sequence.

The database created by *Victoria* served two purposes. First, the merged database of all subtrees investigated should provide us with a solution tree for the full go-moku game tree. Second, the databases created by solved subtrees were used as transposition table for pn-searches. We have seen several occasions where a search of several hundreds of thousands of nodes without transposition tables was reduced to a mere few thousands nodes, by hitting the database early during the search.

5.5.2 Subdividing the game tree

We have divided the go-moku game tree into several hundreds of smaller problems. The main reason for doing this is that the size of the go-moku game tree is such that we could not solve it through a single pn-search, due to the limits imposed on pn-search by the size of our computer's working storage.

We remark that by splitting the game tree into subtrees, part of the solution process has been performed by hand. Most of these moves have been made with the aid of Sakata and Ikawa (1981), while others where suggested by the proof and disproof numbers of failed pn-searches. The number of black moves selected by hand (several hundreds) is less than one percent of the total number of black moves in the solution tree (many tens of thousands).

5.5.3 Statistics

In this section we present the statistics of running pn-search on go-moku. As mentioned before, we have subdivided the problem in several hundreds of subtrees, each of which was individually solved. Since each completed search extended the database of solved positions, the number of positions searched partly depend on the order in which the subproblems were solved.

Execution time

Our calculations were performed in parallel on 11 SUN SPARCstations of the Vrije Universiteit in Amsterdam. Each machine was equipped with 64 or 128 megabytes internal memory, ensuring that pn-search trees of up to 1 million nodes would fit in internal memory, without slowing down the search by swapping to disk. The processor speed of the machines ranged from 16 to 28 MIPS. Our processes could only run outside office hours. As a result, sometimes processes which had not finished at 8am were killed, and had to be restarted at 6pm. Still, over 150 CPU hours per day were available for solving go-moku. In the figures below, we have not included CPU time spend on processes which were killed in the morning and restarted in the evening, nor have we included the CPU time spent on test runs during which we discovered bugs in our software (see also section 5.5.4). Thus, the time mentioned indicates the amount of time necessary to solve go-moku without interruptions, using the final version of *Victoria*.

Free-style go-moku was solved using 11.9 days of CPU time, while standard go-moku (thus banning wins through overlines) was solved with 15.1 days of CPU time.

Pn-search tree size

The summed size of all pn-search trees built during the calculations (again excluding terminated processes and runs of initial versions of the program) for free-style go-moku is 5.3 million. For standard go-moku, 6.3 million nodes were grown.

Comparing these figures with the execution time necessary for the solutions, we see that both variations ran at the speed of approximately 5 nodes per second. The rejection of potential winning lines involving overlines, resulted in the creation of a 20% larger search tree.

Db-search evaluations

For each internal node of the pn-search tree, 10-20 independent db-searches (excluding global defensive strategy searches) were performed on the average, resulting in, between 50 and 100 db-searches per CPU second. Multiplied by the total calculation time, the number of independent db-searches executed to solve go-moku lies between 50 million and 130 million.

Solution size

The solution tree found by *Victoria* for free-style go-moku is slightly smaller than the solution tree found for standard go-moku: 138,790 versus 153,284 database records. Comparing these numbers with the total size of the pn-search trees, we find that 1 out of every 40 nodes created participates in the solution. The deepest variation in both solution trees is 35 ply.

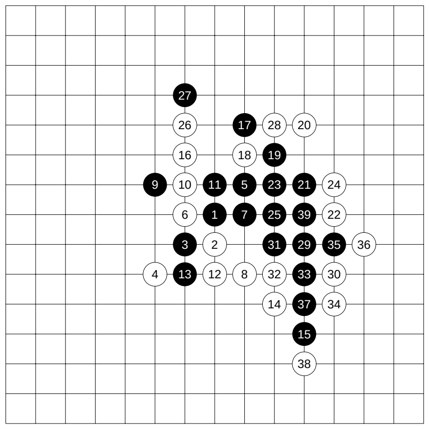
depth	free-style	standard	depth	free-style	standard
0	1	1	18	1351	1885
1	1	1	19	1094	1590
2	35	35	20	710	1125
3	35	35	21	594	954
4	7227	7242	22	408	641
5	6824	7251	23	327	506
6	20859	22749	24	193	296
7	20239	21078	25	154	241
8	20686	22056	26	85	159
9	20550	21898	27	74	128
10	8959	10015	28	40	67
11	8637	9570	29	35	54
12	5246	6015	30	7	19
13	4778	5492	31	7	18
14	2999	3663	32	1	8
15	2647	3282	33	1	6
16	2173	2810	34	1	1
17	1811	2392	35	1	1

Table 5.1: Nodes per tree depth in go-moku solutions.

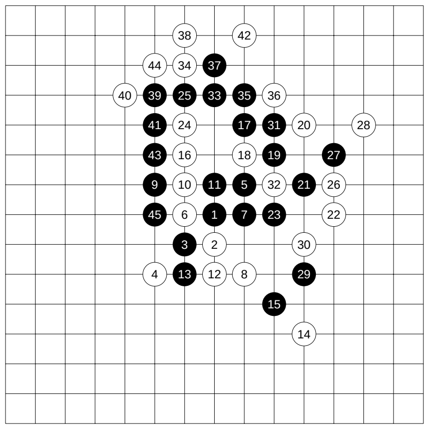
In table 5.1 we have listed the number of nodes per depth for both solution trees. We remark that for each position with black to move, only one child position needs to be included. Due to transpositions, the number of nodes at each odd ply should therefore be less or equal to the number of nodes at the preceding even ply. The only exception in the table, ply 5 for standard go-moku, is caused by the fact that we have included several options for black for some opening positions in our set of positions to be checked by pn-search.

Deep winning lines

The combination of db-search and pn-search makes it difficult to determine the *maximin* of go-moku (i.e., the length of the game after optimal play of both players). Both db-search and pn-search do not aim at finding the shortest winning paths, while the longest path found by the combination of the algorithms may well be different from the game leading to the single



(a) Free-style go-moku



(b) Standard go-moku

Figure 5.9: Deep variations

position at level 35 of the solution tree. Even though the games leading to positions at level 35 of the solution trees do not necessarily show optimal play of either side, we have depicted two of these games in figure 5.9.

5.5.4 Reliability

In section 4.5.4, we have explained the hazards of solving games through large computer programs. The same hazards mentioned there exist in go-moku in even greater form.

Our go-moku implementation consists of almost 20,000 lines of C-code. Approximately half is dedicated to the X-interface created by Loek Schoenmaker, while the other half consists of db-search, pn-search, database look-up and database creation, automorphism management, etc. Errors in programs this size are virtually unavoidable. Many errors have been created and corrected during implementation and testing of the program, but there is no guarantee that all bugs have been found.

A further source of error is the complexity of the calculation process. We used 11 SPARCstations in parallel to solve each of the several hundreds of subtrees. These 11 SPARCstations created their own databases when solving a position, while they all used one large database as transposition table. After solving a position, the transposition table should be extended with the newly created small databases. A locking mechanism was created to ensure that no databases would be corrupted. Still, computers going down at critical moments introduced the possibility that data would get lost. This, in fact, has happened during our calculations.

To ensure the completeness of the solution found, we have created a module which examines the final database created. For each position with black to move a successor position must be present in the database. For each position with white to move, for each legal move either a winning threat sequence must exist, or the successor position must be present in the database. The only common element with the solving process is db-search. Thus, an error in db-search may go unnoticed, while all other parts, including pn-search and the related-squares generator, are eliminated from the checking process. Using the database checking module, we have both located missing database parts, due to computers failing at critical moments and human error, and have found an error in our related-squares generator. The final investigations, however, for both the free-style go-moku and standard go-moku variants were successful.

The correctness of our db-search implementation is based on meticulously testing all possible types of counterplay, including intricate ways in which the opponent forces the attacker, after a sequence of fours, to occupy an extension square of a threat in the threat sequence. After the final database creation, which was checked and accepted by the database-checking module, no errors have been found in this part of the program. Therefore, go-moku should be considered a solved game.