Marco Kunze (makunze@cs.tu-berlin.de) **2004/12/29**
Sebastian Nowozin (nowozin@cs.tu-berlin.de)

An AI for Gomoku/Wuziqi
$\alpha - \beta$ and more...

How to think 20 moves ahead.

1 Introduction

This document details our solution to the second of two projects of the "Artificial Intelligence" course in fall 2004 at the SJTU.

Like the first project, the problem given to the students is easy to describe, but difficult to solve. The students are asked to write a program playing the game of Wuziqi, which is also known by Gobang, free-stlye Gomoku or "Five in a row".

1.1 The game

The standard playing rules are the following: On a board of 15x15 rows and column two players, black and white, in turns place one stone of their color on a free field until either the field is full or one player has won. A player has won if he has five or more stones of his color in a straight row, be it horizontal, vertical or diagonal.[1]

The game is interesting for two reasons. The first is that the rules are very easy to understand and everybody can play this game to some degree. The second reason - and the challenging part of the excercise - is that the game can develop a deep strategy, with moves having to be planned ahead 10 or more plys ahead to win successfully or even to just defend successfully against a strong opponent. World class Gomoku player plan ahead more than 25 plys.

Gomoku has a seemingly large search space due to many possible moves on a 15x15 board. Additionally, it has been shown that generalized Gomoku is PSPACE-complete, which means that Gomoku played on an board with n fields, the complexity increases without bounds for increasing n. The net result is, that without any clever tricks, Gomoku even on a 15x15 board is computationally very difficult to solve.

Albeit this difficulties Gomoku has been solved by L. Victor Allis in 1992 [1] by using a new method he coined "dependency-based search", which we will describe in detail below.

We apply $\alpha - \beta$ search with some modifications and the dependency-based search method to create a strong AI player, with no limit of planning ahead for some situations of the game.

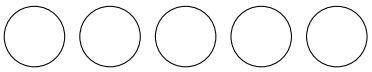
2 Basic definitions

We use a number of definitions in the following sections, which are explained here.

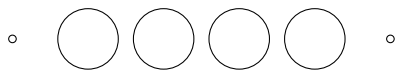
- *Threat.* A threat is a move to which you have to react. The most simple example is a *four*, which is a series of four same-coloured stones, with one bordering field being free. If the the opponent does not react or does not win himself in the next move, the player owning the threat has won. We use a limited set of possible threats, which we will discuss in detail now. We assume the attacker always has the white color. All cases are shown with one example, but there are more cases than the one shown, because of symmetry.

[1] There are more advanced rules in variations of this game, for example in "standard Gomoku" a player is only granted a win if he has *exactly* five stones in a row. Also, professional "Renju" is a more complex variant restricting movements even further.

– *Five.* A five is a simple row of five stones and the player has won.

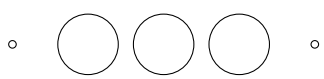


– *Straight four.* A straight four is a row of four stones with both ends free. If the other player has not won in his next move, this situation is a sure win.

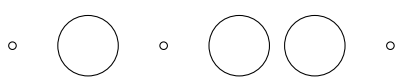


– *Four.* If the other player cannot win in his next move, he has to defend against the four, which is a row of four stones, but with one end closed by the opponent and the other end being free.

– *Three.* The three is a more long term threat to the opponent and although he has to react to it, he has more choice in the possible defending moves to make. If undefended, the three can be extended to a straight four, which is a sure win.



– *Broken three.* Using the broken three, the attacker can exert great control about the opponents stones. It has to be defended against, just as with the three.



- *Threat category.* To each type of threat we associate a threat category (or class), which equals the number of undefended moves necessary to win. The five has a class of 0, the straight four and the four have class 1 and the three and broken three have class 2.
- *Threat sequence.* A threat sequence is a series of single attacker moves and one or more defending moves. The last attacking move reaches a goal state and has no defending moves. A goal state is either a five or the undefendable straight four.

3 Alpha-Beta search

We use a standard four level (two pair move) alpha-beta search stage as detailed in [2]. The state space searched is the board itself and the operations are single moves.

3.1 The static evaluation function

The basic static evaluation function basically uses three measurements to rate a move:

1. The possibilities the move gives us to build rows of five stones in the future.

   The space around the newly set stone is the lowest measurement we apply for every move. That means, if no other criteria work, we will set the stone that we, we can build most possible combinations of rows of five stones over it.

2. Building up long lines (or combinations of long lines) to work towards threats and to prefer special kinds of threats.

   This is done pretty easily: The row of own stones built up by the new stone is counted and assigned by a multiplier. This for example has the advantage, that a "straight four" will be preferred to a "four" with a hole.

3. The identification of a winning situation.

   This is obvious.

Besides the pure measurements and assigned points, the static evaluation function has to be fast.

In order to need not to evaluate the whole board in every move, we just consider the space around a newly set stone for the rating. This means, we look at the 4 lines of 9 pieces which lead through the new stone, building up a star. In order to avoid the calculation useless every time we want to rate a move, we generate the values before the game starts into a table with something around 260000 entries, which is much less and faster than doing this during the execution of the alpha-beta algorithm.

This fast way of rating single nodes allows us not just to call the static evaluation function in all the leaf nodes, but in every single node. During the alpha-beta-search, the nodes are sorted (higher successors for MAX-nodes and lower successors first for MIN-nodes) allowing a much faster and higher cut-off rate during the alpha-beta-procedure [2].

3.2 Supporting db-search: statically evaluating threats

The alpha-beta method tries to support the db-search. This is done by favoring the creation of new high category threats. Our assumption is that other AI players will be unable to track a large number of high category threats more successful than we are able to do using db-search. Additionally the large number of threats on the board will create many possible threat sequences of which we hope there will be a winning one.

During the static evaluation function, threats therefore are higher rated than anything else (except win of course), and in threats of the same category, longer lines of own stones are preferred.

Since the threat detection for the static evaluation function has to be much faster, but less generalized than during the db-search, it has it's own methods for this purpose, allowing a fast pattern matching on the same local lines as for the normal static evaluation function, also using a cached look-up table.

3.3 Keeping the branching factor small: finding interesting fields

For the speed of the alpha-beta-procedure, it is crucial to keep the branching factor small. Since the branching factor is decided by the new number of interesting moves (which means the number of moves we should consider to take in the next step) we have to find ways to exclude as many fields as possible.

This is done in three steps:

1. Using "binary dilation" on the board.

   Binary dilation, originally coming from the field of digital image processing, allows us to select just the fields around already set stones, using a fixed pattern. This way we just consider the 16 fields (starformed) around already set stones as interesting.

2. Looking for threats.

   If we are already forced to block threats, we don't need to consider other fields than the blocking ones, except...

3. Looking for own possible threats.

   If we are able to build up threats of lower categories than the opponent threats, we can force him before he turns his threat into a win, so these moves can be considerer.

This method allows us crucial branching limitations in threat situation (which we are looking for) and in end-game situations. The threats and interesting moves ar also locally evaluated, never have to been recalculated for the whole board, to let us be fast.

4 Dependency-based search

Dependency-based search is a single-agent search algorithm to explore a space state under certain constraints. It was invented by Victor Allis specifically to solve thinking games such as Connect-four and Gomoku.

Now, following [1] we will discuss two points that are crucial for the understanding of how we use db-search in Gomoku. The first is how the adversary-agent problem of Gomoku is converted to a single-agent search space. The second is how db-search works on this single-agent search space.

4.1 Converting Gomoku to a single-agent search space

The idea of converting the adversary search space to a single agent one, is a simple and clever one: we allow the defender to make all possible defending moves at once. Then, every threatening move is combined with all the possible moves to defend against this threat and the whole pair-move is combined as one operation in the search space.

Of course, allowing the opponent to make more than one move at a time will give him a huge advantage. But the advantage to safely plan ahead a large number of moves far outweights this disadvantage. That is, we are denied the ability to recognize a large number of cases where we could have won, but without this simplification we would not have the computational ability anyway. What remains is a small portion of the possible cases from which we can try to find a win.

4.2 Applying db-search

The theoretic framework for db-search provided in [1] is quite large and generic. We implemented this framework completely and besides solving the Gomoku with it, we solved a simple problem known as "Double Letter Puzzle" for testing purposes. Below we will shortly describe the operation of db-search, although a larger discussion would take to much time.

4.2.1 The basic idea

The basic idea of db-search is to use operators to transfer between states in the state space. The already discovered states are kept in a directed acyclic graph (DAG). The constraints to apply an operator on a state already discovered are the following:

- The operator is valid.

  The operators preconditions are valid on the given state and the operator can be applied.

- The application of the operator *depends* on the previous one.

  This condition is always assumed true if the state is the root state, as there was no previous operator.

  Otherwise, if there was a previous operator leading to this state, the application of the new operator examined has to depend on the result of the previous operator. That is, without the previous operator applied before, the operator would not work. This is one important part where useless explorations of the space state are limited.

- The application of the operator has to *create new dependent operators*.

  The exploration of the state space is greatly limited by this condition. Only operators are applied, if they depend on the previous operator *and* if their application creates at least one operator that is dependent of the result of the operator considered.

4.3 Implementation

The db-search description is quite simple and intuitive, but an efficient implementation of this semantics is another matter. One pseudo-code implementation is given in the original paper by Allis, which we used as basic design guideline for our from-scratch implementation. The really difficult part in implementing db-search were a number of predicates (such as `NotInConflict` given in the original paper) and meta-operators (such as `Combine`). The implementation we use works in two stages, a *dependency following* and a *state combination* stage. An in-depth discussion would take too much time, but suffice to say, the three conditions listed above - also known as "meta operator" used in the theoretical db-search framework - are ensured by successively executing this stages in pairs, called levels.

For Gomoku, we often have seen one single application of the dependency following stage following more than 10 moves in a row. The largest sequences we have seen being checked extended more than 25 moves. In no case observed by us the search has required more than four levels.

On an algorithmic level our implementation is very space and time efficient, often exploring less than a thousand states and taking less than a few seconds to explore all possible threat trees. However, we did not profile and manually optimize the code so far, which could still result in improvements of an order of magnitude.

Note that the winning threat sequence search is not complete. This means it will not find all winning threat sequences or miss them while searching. However, if it finds one, it guarantees it is not refutable. As such, it is used only in conjunction with the $\alpha - \beta$ search, not as replacement.

One caveat: we did not implement complete three- and four-state combinations in the db-search combination stage, which can lead to problems, see below. This is because we ran out of time to devise an efficient implementation and our implementation already took considerable time to complete.

4.4 Winning threat sequence search

The winning threat sequence search is called with a board position. The goal condition for the search to succeed is a *five* or *straight four* pattern on the board. As long as this pattern is not found and the search space is not exhausted, we continue to search.

Also, as an extra condition to ensure timely execution, we added a timeout mechanism. If the timeout given in milliseconds is exceeded, the search is cancelled and no winning threat sequence is returned.

The basic operation of the search is the following:

1. Search for potential winning threat sequences

The db-search is started with the board as parameter and proceeds expanding the search tree using the given constraints. For each new state reached it is checked if it is a goal state. If it is no goal state, the search proceeds. If it is a goal state, a potential winning threat sequence is extracted by converting all the operators on the path from the root state to the goal state into moves. Then

2. For each potential winning threat sequence found, we check if it can be defended against by the opponent. Defending means, that by making a good choice on his defending moves, he can build threat sequences himself. His threat sequences are a danger to us for two reasons:

   - They could lead to a win for the opponent.
   - They could lead to occupation of squares we would need ourselves later in our threat sequence or occupy defending squares ahead of time.

   To check for this, we start a second db-search, this time trying to defend against our own attacks. Note, while the attack sequence search we did for ourselves is not complete, this defending search has to be complete to be sure our attack is working every time. See the next subsection for an explanation why this is so.

   The defending db-search works like this:

   (a) Make an attacker move

   (b) If the attacker has won with this move, we mark the sequence as unrefutable.

   (c) Try to find a potential winning threat sequence, but limit the applicable threats to a class one lower than the attacker threat.

   It would make no sense to search for threats with a class as high as or higher than the attacker has challenged us with, as a challenge with a class as high as his would mean he is always a class lower than we are in the next move, leading to a win for the attacker. On the other hand, for example he challenges us with a class 2 threat, we can successfully defend ourselves with both a class 1 (if he does not react, we win in the next move) and class 0 (we won) threat.

   (d) If the threat sequences are potential winning sequences or if they lead to an occupation of any attacker or defender square later in the attackers threat sequence, we return and mark the threat sequence as refuted.

   (e) If there are still moves left in the attackers threat sequence, advance to the next move and start at (a).

   If a winning threat sequence has been found, we abort the search and return early with the sequence.

4.4.1 Completeness of the defending db-search

The defending threat sequence search has to be complete in that it never misses any defending sequence the opponent can make. This requires completeness in the db-search algorithm. However, as we have seen that the attacker winning threat sequence search is not complete, how can we make the defending one complete?

This is achieved by restricting the number of applicable operators drastically: only operators with a category below the attackers one are allowed. As the highest category is 2, for threats that will win in two moves if not defended against, the maximum possible threat category the defender can apply is 1. However, to all threats of class 1, there is always a reply using only one stone, which does not require our "multiple-defender-stone" assumption, which makes the attackers search incomplete. Hence the defending search is complete.

However, in our implementation a small part of the db-search algorithm is missing. This leads to some rare cases not being explored in the search tree although they should be. We have seen no such incident in a real game, but there are artificial situations in which this happens. As this

is a vulnerability in our AI player, we will not describe it in too much detail here which situations could trigger this, but we think they are unlikely to occur in a real game our player plays.

4.5 In game operation

The winning threat sequence search is called whenever an opponent move has been made. Then, after the alpha-beta and interesting moves module is ran, we use the remaining time to run a time-constrained db-search, giving the current board position.

If the winning threat sequence finds no winning threat sequence, the alpha-beta result is used. If it does find a sequence, this guarantees us a win:

- We have won in no more than n moves, where n is the depth given in the threat sequence plus the number of opponent threats on the board.
- We have a move sequence which the opponent cannot refute at the last move.
- We know all possible defender moves in advance that can delay our win. If the defender does not make any of this moves, we fall back to $\alpha - \beta$ search and finding a new winning threat sequence. In almost all cases the $\alpha - \beta$ search will find a guaranteed win already, as the latest threat was not defended against. In rare cases it might not find a good move and a new much shorter threat sequence is returned by the winning threat sequence search.

The threat sequence is stored and followed as long as the defender makes one of the known defending moves. We expect almost all opponent AIs to blindly following the threat sequences expected defending moves until at least near the end.

References

[1] L. Victor Allis, "Searching for Solutions in Games and Artificial Intelligence", Ph.D. thesis available online at `http://www.cs.vu.nl/~victor/thesis.html` and available as book (ISBN 90-9007488-0)

[2] Nils J. Nilsson, "Artificial Intelligence - A New Synthesis", book (ISBN 7-111-07438-6)