Five-In-Row with Local Evaluation and Beam Search

Jiun-Hung Chen and Adrienne X. Wang
jhchen@cs
axwang@cs

Abstract

This report provides a brief overview of the game of five-in-row, also known as Go-Moku, the basic structure of our program, and the heuristics we develop to improve the efficiency of searching. The basic search algorithm is min-max search with alpha-beta pruning. With a sorted successor list at each level, we are able to explore beam search which bounds the number of searching branches and analyze the improvement.

Introduction

The game of five-in-row, also known as Go-Moku, is a two-player game. Its rules are simple, but they lead to a highly complex game. On a board of n horizonal lines and n vertical lines, two players, Black and White, take turns to mark their own color on one of the empty squares. Black starts the game. Once a marker is placed on the board, it can not be moved to another square later. A marker can not be taken over by the other player, either. The player who creates a line of five consecutive markers of his/her color wins. If no one creates five-in-rwo before the board is completely filled, the game is drawn.

Five-In-Row variants

Five-in-row is a game with long history. Some professional players claim that this game is a won game for the player who moves first. Many variants of five-in-row exist. They all try to restrict Black's move to reduce the advantage of moving first. However, none players nor existing five-in-row programs can prove that this claim is true. Here we briefly introduce some of the variants to this game.

1. **Non-standard boards** People in early days play on a board bigger than the size of 15×15. However, it's believed that larger board would increase the first player's advantage.
2. **Standard Five-In-Row** This is the most popular five-in-row these days. If a player creates a line of six or more consecutive markers, this line does not win. A line of five consecutive markers is the only winning pattern.
3. **Renju** Renju is the professional five-in-row. It restricts Black from winning with a line of six or more consecutive markers or double threats, while does not place any restriction on White.

We design our program to play the non-restricted five-in-row due to its simple rules.

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Previous Work

1. **Heuristic Search** Some previous implementation of five-in-row employed heuristic searches. The decision is based on maximizing the value of evaluation function.

   $Eval(i,j) = AttackValue(i,j) * (AttackFactor + 16)/16 + DefenseValue(i,j) + randomfunction$

   The $AtttackValue$ is the number of lines affected by the move, and the $DefenseValue$ is the number of opponent's lines affected if the move is taken by the opponent. The $AttackFactor$ is to emphasize the move that would lead to winning for sure. $randomfunction$ adds randomness to the program's behavior.
2. **Threat-Space Search** The five-in-row program $Victoria$ uses Threat-space search (Allis, van den Herik, & Huntjens 1993). It is modelled to formalize the human search strategy. It searches for the moves that can lead to a winning threat sequence.
   We use the idea of threats in our program to evaluate the utility of the game after one action is taken.

Basic Structure

We implement this game using min-max search algorithm with alpha-beta pruning. Besides alpha-beta pruning, several heuristics are implemented to limit the search space.
The successor list is limited to the actions which are useful to the player. We sort the successor list based on the evaluation scores. The most promising action is searched first. We then use local beam search to reduce the branch factor furthermore. At the same time, to save the time of computation of evaluation function, we evaluate each action when generating the successors instead of evaluating the state globally at each cut-off point.

Experiments

Since we implement several techniques to speed up min-max search, we measure the improvement at the end. The measurement is based on the branch factor, which is the indicator of the search space complexity. The experiments include comparison of different search depths, and improvement of beam search.

Heuristics

We design the evaluation function to be an estimate of the expected utility of the game based on the local situation after a mark is placed on the board. This is simpler than the global evaluation on complexity of computation, but still maintains high accuracy level. To make the estimate as accurate as possible, we introduce the idea of threats.

Threats

In five-in-row, each winning sequence consists of one or more threats. Threats are early indications of potential winning. Some threats are not refutable. Having those types of threats is equivalent as having a five-in-row. The other threats can be refuted. The other player has to take a defensive move or create a more powerful threat in order not to lose. Our evaluation on the entire state relies heavily on the type of threats and number of threats. Being able to identify them helps us to predict the result.
We define 5 types of threats in our evaluation function.

Types of Threats

Winning Threat The threat of **five-in-row** 1 is the winning threat. The player who creates this threat wins.

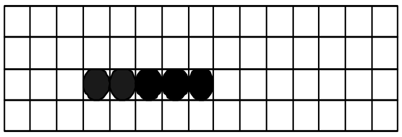

Figure 1: five-in-row threat

Non-Refutable Threat The threat of **straight-four** 2 is a line of six squares, of which one player has occupied the four center squares, while the two outer squares are empty. This threat is not refutable. The other player may mark one of the empty squares in his next move, but marking the other one would create a five-in-a-row. In our program, whenever such a threat appears, we claim that the player will win this game.

Refutable Threats We call them refutable threats because whoever creates them threatens to win, but it's not an assured win. Refutable threats can be saved if the opponent takes action immediately. Assume Black creates a refutable threat. This threat forces White to either take defensive moves or create more powerful threats in order not to lose. White

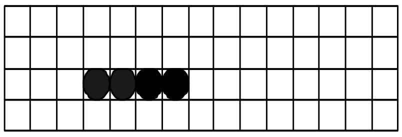

Figure 2: straight-four threat

can force Black to defend by creating a threat/threats which can lead to winning after less number of steps. We will talk about these ”more powerful threats” in the section of Category Reduction. If White doesn't have more powerful threats, and chooses to defend, his/her successive actions are limited to the set of defensive moves.

1. **four-in-row** The threat of **four-in-row** 3 is a line of five squares, of which one player has occupied the four consecutive squares, while the fifth square is occupied by the other player. Unlike the above two types of threats, this is a refutable threat. There is one possible defensive move to the other player. He/She may mark the square on the other side of the four consecutive squares in his/her next move so that the four existing marks won't grow into five-in-row.

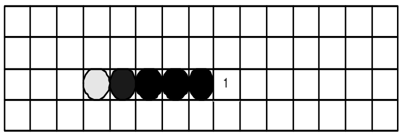

Figure 3: four-in-row threat. 1 is the defensive move

2. **three-in-row** The threat of **three-in-row** 4 is a line of five squares, of which one player has occupied the three center squares, while the two outer squares are occupied by the other player. This is also a refutable threat. There are two possible defensive moves. The other player may mark one of the two empty outer squares.

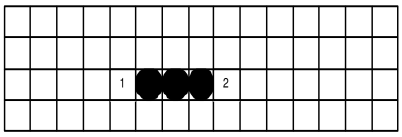

Figure 4: three-in-row threat. 1 and 2 are the defensive moves

3. **broken-three** The threat of **broken-three** 5 is a line of six squares, of which one player has occupied three non-consecutive squares of the four center ones, while the other three squares are empty. Three possible defensive moves can be taken by the other player. He/She may mark one of the two outer squares or mark the middle square.

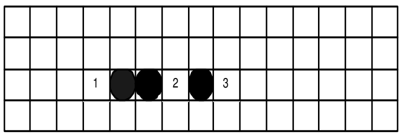

Figure 5: broken-three threat. 1, 2 and 3 are the defensive moves

Category Reduction

We mentioned earlier that when the opponent has one or more threats, the player may choose to defend or create more powerful threats. We use category to represent the power of each type of threats. Intuitively, if a threat requires less steps to win, it is considered to be more powerful.

Category 0 five-in-row
This threat requires 0 step to win.

Category 1 straight-four and four-in-row 6
These two threats require 1 more step to win.

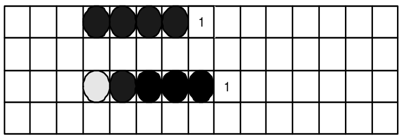

Figure 6: Threats in Category 1. Both need to mark square 1.

Category 2 three-in-row and broken-three 7
These two threats require 2 more steps to win.

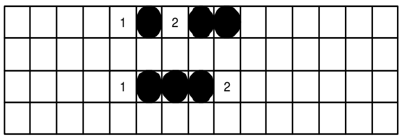

Figure 7: Threats in Category 2. Both need to mark square 1 and square 2.

Clearly, each threat of category n can be refuted by a threat of category $n-1$ for $n = 1, 2$. A player can choose to attack instead of defend if he/she can create such a threat. In Figure 8, White creates a three-in-row, which is in Category 2. Black can extend his/her existing three marks into four-in-row, which is in Category 1, and force White into defense because there is one more step for Black to win, while White needs two more steps. This is also called global refutation.

Double Threats

If one player creates more than one threats, and the sets of defensive moves available to the other player have no intersection, we call these threats double threats. For example, if Black creates a three-in-row threat and a broken-three threat,

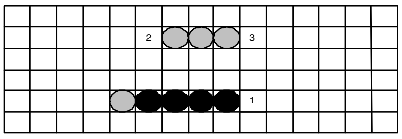

Figure 8: Black's Category1 threat can refute White's Category2 threat

there are two possible outcomes. If they share the same defensive move, this combination of threats is refutable, and therefore is as powerful as a single threat (see Figure 9). If their defensive moves don't have intersection, it becomes a non-refutable double threat (see Figure 10).

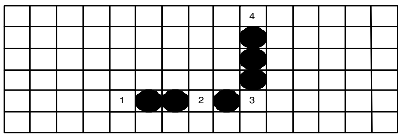

Figure 9: two threats that can be saved by one action

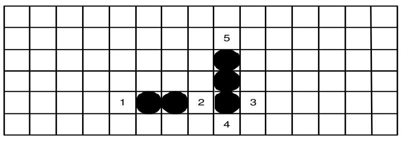

Figure 10: double threat

Evaluation function

Since a winning sequence must consist of some type of threats, we design our evaluation function base on the number and types of threats. This evaluation is an estimate of the local state. When generating successive actions of a state, we evaluate each action mainly on the new threats that would be created if the player would take that move. We assign different weights to each type of threats.
In addition, when there is no threat in the current state, number of two-in-rows and single marks which could potentially be extended into threats in successive states become important. A line of three consecutive marks with two different marks on both ends will not help us win the game because it can't grow in either direction. This line is rather dead. We don't assign any weight on these lines.

$Eval = w_1 \times \#$ five-in-row $+w_2 \times \#$ straight-four $+w_3 \times \#$ four-in-row $+w_4 \times \#$ three-in-row $+w_5 \times \#$ broken-three $+w_6 \times \#$ two-in-row $+w_7 \times \#$ single marks

The weights on non-refutable threats are much heavier than the weights on refutable ones, whose weights are substantially heavier than the non-threat lines.
This evaluation function reflects the new threats created after taking a possible action. It is relatively simple since

it's a local evaluation. This evaluation correctly predicts how likely one action may lead to winning.

Reducing Searching Space

The efficiency of min-max search depends on the branching factor b. It performs a complete depth-first search of the game tree. Let d be the maximum depth of the tree, the time complexity of the min-max search is $O(b^d)$. The complexity of seaching increases exponentially with the number of branches at each level. We propose several ideas to cut the branching factor b.

Eliminate actions of no real value

We can greatly improve the efficiency by reducing the number of successors. The key to reduce search space is to eliminate moves that are easily determined to be of no real value. A move is not useful if the player marks far away from any of his/her own marks and any of its opponent's marks.

$successors = \{(x_1, y_1) | \exists x, y$ where (x,y) is an existing mark s.t. $|x-2| \leq x_1 \leq |x+2|$ and $|y-2| \leq y_1 \leq |y+2|\}$

All valuable successors are within two steps from any existing mark on the board. We only need to evaluate a very small set of actions on the board during the initial stage of the game. Instead of considering nearly 225 possible actions, we reduce the number of branches to less than 30. This is very important because it helps most if number of branches is small at a higher level in the hierarchy.

Defensive moves vs. Global refutation

A substantial reduction comes from the determination of the defensive moves connectd to a threat. However, instead of choosing from these defensive moves, the player may globally refute the opponent's threats by creating a threat with reduced category. When the opponent has some types of threats, the player's actions are limited to either the set of defensive moves or the set of global refutations. The successor list will contain only these moves since the other moves will for sure lead to loss of the game. The branch factor b under such a circumstance is normally less than 3.

Sorted Successor List

The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. If the most promising action is examined first, we would be able to prune a lot of branches. Theoretically, if we assume that the best successor is always the first in the list, it turns out that alpha-beta needs to examine only $O(b^{d/2})$ nodes to pick the best move, instead of $O(b^d)$ for minimax. This is equivalent to reducing the branching factor to $\sqrt{b}$ instead of b. (Russell & Norvig 2003)
We sort the successors based on the evaluation of the local utility if this action would be taken.

Local Beam Search

Because we sort the successor list on evaluation scores, the best moves are likely to be searched first. We could bound the branching factor b with a limit k. The search complexity is further reduced to $O(k^d)$ (Russell & Norvig 2003).

Experimental results and Discussions

We implement our program in C++. The execution is real-time if search depth is no more than five. However, when we search more than five levels in the game trees, computation becomes very intensive and memory load is very high. Hence, we limit search to be at most five levels in following experiments.

We create an experiment to demonstrate how sorted successor list can improve the efficiency of alpha-beta pruning. Each test tasks consists of 100 games and we take the average. The opponent is a player who picks an action randomly from the successor list. This random player gets to pick defensive moves and global refutation moves when the other player creates threats. Figure 11 shows the average branching factor b with different search depth d. It turns out that sorting does help to reduce the search space while it still maintains similar performance (see Figures 12 and 13). However, when we generate the successor list, we limit the list to defensive moves and global refutation moves when the other player creates threats. It substantially cuts the size of the list in both experiments (sorted and unsorted list). Because threats appear often in a game, it becomes a big distortion in the measurement. We are able to show that sorting helps to improve the efficiency to some extent. Figure 12 reveals the fact that the deeper we search in the game tree, the more likely we win the game. Our evaluation function tries to maximize the winning possibility for each player, so attacking moves always get higher score. This best explains the experimental result in Figure 13. If the search depth is odd, the leaf node is an action for MAX. The player is more likely to pick an attacking move, and therefore finishes the game sooner.

We create another experiment to measure the improvement of beam search. In the first test case, we search the game tree with depth 3 using min-max search with alpha-beta pruning. In the second test case, we put a bound on the number of successors. The average branching factor b of min-max search with alpha-beta pruning is 13.6, while the the average branching factor b of beam search varies from 4.81 to 6.28. Beam search helps to cut the branching factor b. However, it does not lower the performance, which proves that our evaluation function accurately estimates the state utility. In Figure 14, we demonstrate that beam search could substantially improve the efficiency while still get high percentage of winning.

Conclusions and Future Work

We notice that local beam search substantially improves the efficiency of searching when the successor list is sorted based on an accurate evaluation function. However, it is not easy to design such a good heuristic. In this project, we design the heuristic using the idea of threats, which models the strategy human players use. We believe that the idea of threats may be used on some other games with similar nature, like connect-four and tic-tac-toe. Although this idea is proved to work really well in the game of five-in-row, it turns out to be very complicated in computation. Applying some of the AI techniques, like reinforcement learning and pattern database, on heuristic design may be more efficient and promising.

Reinforcement learning has been applied in game search design and obtained very successful results. For example, Tesauro proposes arguably the best player in the world for TD-Gammon (Tesauro 1992). Many other successful applications can be found in (Sutton & Barto 1998).

We plan to apply Q-learning algorithm (Russell & Norvig 2003) for five-in-row. We only evaluate a local state, which at most is a 9×9 binary board. Because the number of possible states is extremely large, we dynamically keep the states with highest frequency in our table. For each state, there are two possible outcomes. One is to occupy the center position and the other is not to occupy the center position. For each action, we give some reward or punishment. An action which can create more threats or refute the opponent's threats in the next state is greatly rewarded. On the other hand, an action which allows the opponent to have a chance to create threats is heavily punished. The degree of reward or punishment depends on the category described in **Category Reduction**.

After learning, we find the action a^* for state s^* in the successor list.

$$(s^*, a^*) = \arg \max_{s \in successors, a \in A} Q^*((s, a) \qquad (1)$$

where A is the set of possible actions and Q^* is Q-value learned from Q-learning.

Acknowledgments

We greatly thank Miao Xu for his valuable help in many directions.

References

Allis, L.; van den Herik, H.; and Huntjens, M. 1993. Go-moku and threat-space search. In *Report CS 93-02, Department of Computer Science, Faculty of General Sciences, University of Limburg.*

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: a modern approach*. Prentice-Hall.

Sutton, R. S., and Barto, A. 1998. *Reinforcement Learning An Introduction*. MIT Press/Bradford Books.

Tesauro, G. 1992. Practical issues in temporal difference learning. In *Machine Learning*, 257–277.

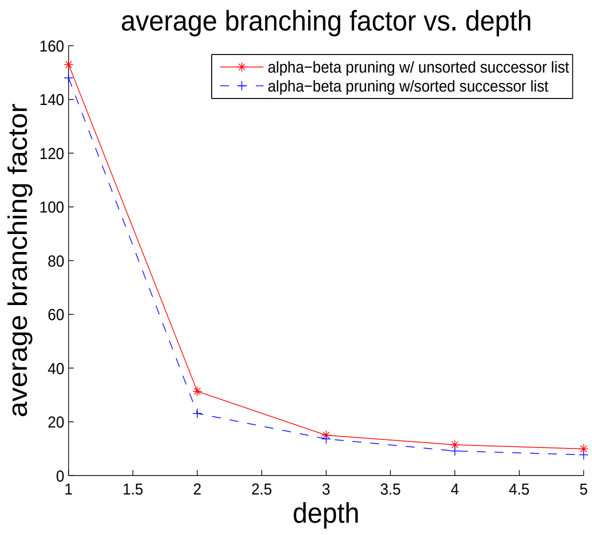

Figure 11: Branching factor vs. different search depth

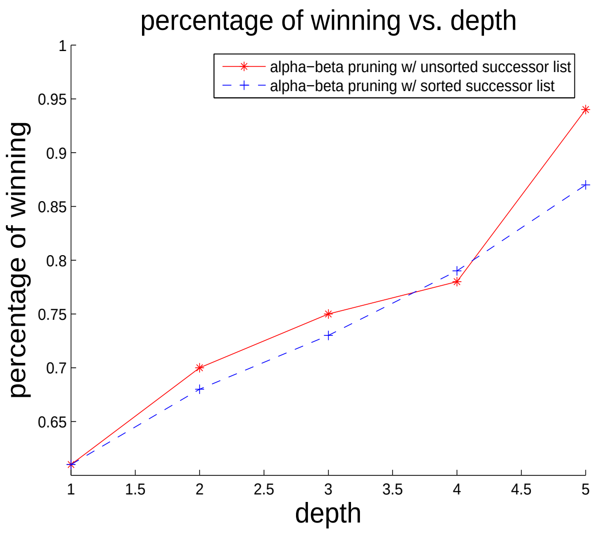

Figure 12: Percentage of winning vs. different search depth

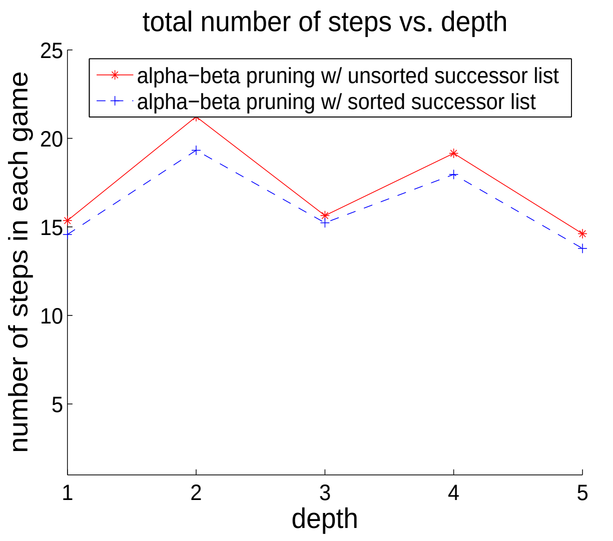

Figure 13: Steps of each game vs. different search depth

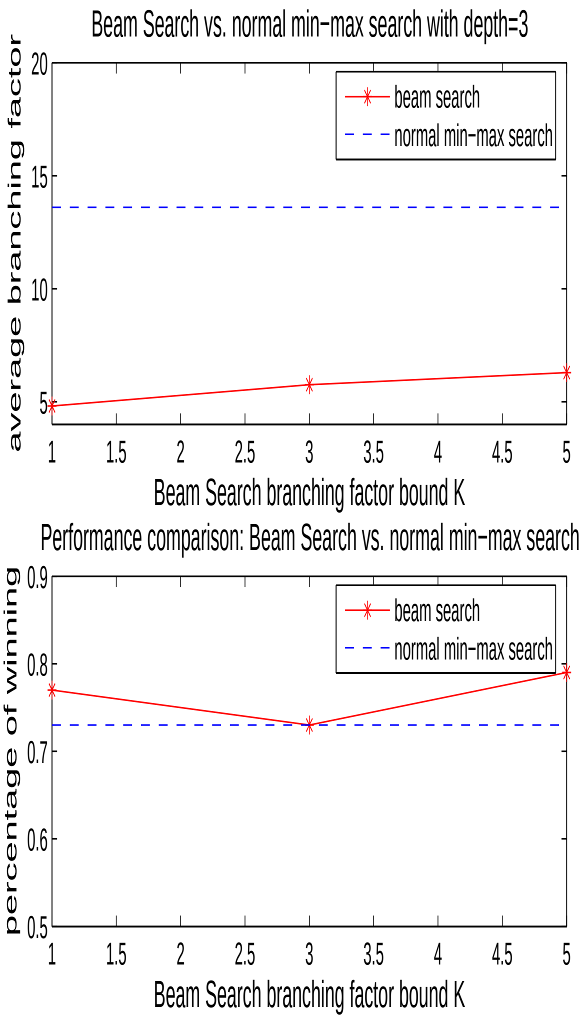

Figure 14: Branching factor with beam search