CS221 Project Final Report
Gomoku Game Agent

Qiao Tan
qtan@stanford.edu

Xiaoti Hu
xiaotihu@stanford.edu

1 Introduction

Gomoku, also know as five-in-a-row, is a strategy board game which is traditionally played with Go pieces on a go board with 15×15 intersections. The rules for Gomoku are similar to Go, where two players place stones alternatively on the intersections of the board. The winner is the first player to get 5 of their stones in a row, either vertically, horizontally or diagonally. Human players strategies usually involve blocking opponents lines, keeping his line unblocked, and perform multiple attacks at the same time.

It has been well known that standard Go-Moku favors the player to move first.[1] Thoeritically, L.V. Allis[2] has been proved that the first player wins without restrictions in standard board. And it was argued that a smaller board size decreases the advantage of first player.

Threat space search, proposed by L.V. Allis[2] is a common strategy used in Go-Moku, which finds deep winning lines of play based on threats. However, in some cases where a winning line exists, threat space search is not guaranteed to find it. To address this problem, proof-number search[3] is proposed. It is proved that pn-search was able to find a forced win for the player to move first.

The scope of this project is develop an AI agent for Gomoku using tree search based approach with alpha beta pruning implemented. We will focus on developing good heuristic functions which are mainly inspired by human players strategy as evaluation function to speed up the computation.

2 Methods

The Gomoku can be modeled as follows. We can explore different policies under this framework and evaluate how efficient and effective each policy performs.

Players	black (player 1), white (player 2)
State	position of all pieces and whose turn it is
Actions	all positions that the next player make place stone on
IsEnd(s)	wheter s has a five-in-row or there is no legal actions
Utility(s)	$+\infty$ if agent wins, $-\infty$ otherwise

2.1 Random Policy

A random policy is a very naive and simple strategy that the agent just randomly places its piece to a position on the board if the position has not been occupied any pieces.

2.2 Baseline Policy

The baseline policy is defined based on several naive strategy used by human beginners. For each round, the agent have three options:

- Randomly picks an intersection if it's first round and it's the `black` player
- Make an effective block if it's a force move. Force move is the scenario where we find the opponent has formed an open row of three, or a row of four with at most one blocked end. If multiple forcing sequences are found, we prioritize them with the length of sequence so that it's guaranteed we first block sequences with number of four. For sequence with same length, baseline agent randomly picks one effective block to eliminate the threat.
- If none of the criteria is met above, the agent just randomly picks an intersection that isn't occupied

2.3 Minimax Policy

2.3.1 Algorithm

$$V_{max,min}(s) = \begin{cases} Utility(s), & \text{isEnd(s)} \\ max_{a \in Action(s)} V_{max,min}(Succ(s,a)), & \text{Player(s) = agent} \\ min_{a \in Action(s)} V_{max,min}(Succ(s,a)), & \text{Player(s) = opp} \end{cases}$$

we may also use minimax policy to tackle this problem. Notice here that the standard gomoku game has a branching factor up to 225 (15×15), which leads to an enormous game tree. It is critical that we adopt techniques such as depth-limited search and alpha-beta prunning with a reasonable evaluation function.

2.3.2 Evaluation Function

To speed up the search, we would like to define an evaluation function based on the idea of **counting winning windows**. A winning window is defined as a window containing consecutive five positions, either horizontally, vertically or diagonally, which has no opponent pieces at all but contains non-zero number of agent pieces. The intuition is that winning windows reflect opportunity positions where placing agent pieces could possibly lead to the victory. A state with a large number of winning windows should be preferable as such state reflects a large possibility of winning.

A snapshot of a Gomoku board (partial view) is shown in Figure 1. The window enclosed by the blue line is a valid winning window because it contains no opponent pieces but four agent pieces. On the contrast, window enclosed by the red line is not considered to be a valid winning window as it has one opponent piece inside. Notice that in the latter case, there is no such a way that the agent can win by placing one piece inside the red window.

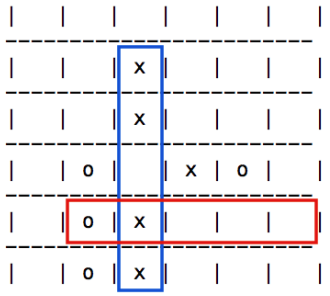


Figure 1: Example of winning windows

One caveat is that each winning window cannot be treated equally. We differentiate winning windows by number of agent pieces inside the window. Intuitively, the agent should be more likely to win if the board contains a large number of winning windows, most of which contains number of agent pieces approaching to five.

To compute all possible winning windows inside a board, we slide a mask of size five in all horizontal, vertical and diagonal direction and advance the mask to the next position by incrementing the start position of the mask

by one. For each mask, we check whether the window under the mask is a valid winning window and classify each window by number of agent's pieces it contains.

Based on the intuition above, we are able to define an evaluation function for each state. For each state s, consider a feature vector $\phi(s)$:

$$\phi(s) = [\phi_1(s), \phi_2(s), \phi_3(s), \phi_4(s)]$$

where $\phi_i(x)$ = number of winning windows containing i agent's pieces.

For example, in Figure 1, number of winning windows that contain one agent's piece is six, contain two agent's pieces is two, contain three agent's pieces is one and contain four agent's pieces is one. So the feature vector $\phi(s)$ in Figure 1 will be $[6, 2, 1, 1]$.

The evaluation function $Eval(s)$ therefore can be defined as:

$$Eval(s) = \begin{cases} \infty, & \text{isEnd(s) and agent wins} \\ -\infty, & \text{isEnd(s) and opponent wins} \\ \sum_{i=1}^{4} i\phi_i(s), & \text{otherwise} \end{cases}$$

Notice that in the evaluation function above, we hard coded the weight of the feature $\phi_i(s)$ to be i, where is a reasonable number to capture winning possibility of different winning windows but may not be the optimal one. We may try to learn the optional weight in later stages as we proceed on.

2.3.3 Optimizations

The evaluation function defined in Section 2.3.2 computes the count of winning windows based on the knowledge of a full chess board. We have found that a naive implementation of evaluation function would result in enormous increase in computational time and became the bottleneck of the computation efficiency when the agent tried to determine the next optimal action to be taken at a given state.

In a naive way of evaluating the value for an arbitrary state, the whole chess board needs to be traversed, resulting in a computational time in $O(N^2)$ where N is the board size. To search for the next optimal action at a given state, we need to go through such naive computation at each leaves at the search tree which have depth being the max search depth. Given the branching factor of the Gomogu board being N^2, searching for the next optimal action given a state would require $O(N^4)$ time even when the search depth is only limited to 1.

To optimize the computation of the evaluation function, we consider to cache the value of winning windows count and incrementally update the cache value whenever a new action is made either by the agent or the opponent.

- Initialization: At the start of the game, initializing the winning window count to the one corresponding to an empty Gomoku board.
- Update given an action: Whenever an action changes the board state (no matter it is from the agent or the opponent), recalculate the value changes on winning windows which enclosing the positions which have a distance with the position in action no greater than four in horizontal, vertical and diagonal directions. Add the value changes to the cached winning window count.

Since the update step only costs $O(1)$ time, now state evaluation is equivalent to return the cached value of the winning window count, requiring $O(1)$ time as well. By keeping a global copy of the winning window count and only adding incrementally modifications on the global copy, we can successfully reduce the search time for the next optimal action to $O(N^2)$ and eliminate a lot redundant computation.

2.4 TD Learning

In fact, the performance of our agent is highly dependent on the weight vector for evaluation function. We proposed to use temporal difference(TD)-Learning to find an optimal weight that gives us highest winning rate.

In particular, we can approximate current utility (sum of future rewards) using two methods, evaluation function described 2.3.2, and Monte Carlo simulation. The former is based on our view of the game, whereas the latter is one a statistically value based on experiments (exploration). An optimal weight vector, thus evaluation function, is expected to be as close to Monte Carlo results as possible. And the objective function is

$$(V(s;w) - \text{Utility}(s))^2$$

To obtain the optimal weights that minimize the objective function, we compute the stochastic gradient descent of the objective function given a sequence generated by the Monte Carlo simulation. Thus, On each (state, action, consequent state), we have,

$$V(s;w) = w \cdot \phi(s)$$
$$w \to w - \eta(V(s;w) - (r + \gamma V(s';w)))\nabla_w V(s;w)$$

Below are the three points we found critical for TD learning process:

- The policy you training/playing against. We choose the baseline policy for opponent, based on two reasons. First, the opponent policy must operates as a min-node based on the assumption of minimax policy. Second, learning from Monte Carlo sequences gives us optimal policy against the simulated world, i.e. our agent is optimal against baseline policy.
- Actions that trigger an update. One challenge of the agent problem is that the future reward of current state, is also affected by opponents action. Thus we update the weight on actions both from agent and opponent.
- Regularization of weights is a must. During the experiments, TD learning may produce an arbitrarily large weight vector depending on the initial weight. We adopt gradient decaying to avoid possible overfitting.

3 Experiments

A custom made Gomoku game engine is implemented in Python. We also implemented a baseline policy, a random policy and a minimax policy with alpha beta pruning and heuristic function. We have also invited some intermediate level human player to play with our agent and treated them as the oracle of our model.

We are interested in measure how well our agent plays against different policies. For each policy implemented, we allow our agent to play against the opponent using the random, baseline or human policy. Each policy will be evaluated based on:

- Winning rate
- Average number of steps to win
- Average computational time for deciding the next step to be taken, denoted by $t_{\pi(s)}$ in the subsequent sections.

We have conduct 1000 rounds of games for each pair of policies on a 2.4 GHz Intel Core i5 Machine.

4 Results

The performance of the baseline policy is shown in Table 1 where the baseline agent has played with an agent using random policy for 1000 rounds of games.

The experiment results of a minimax agent playing against the baseline agent are summarized in Table 2. DLS is the abbreviation for Depth Limited Search and the maximum search depth is denoted by d_{max}. Learnt weights represent the optimal weights learnt by TD Learning.

$t_{\pi(s)}$ (in seconds)	Winning rate	Average number of steps to win
0.00064	52%	20.15

Table 1: Performance of Baseline Policy (against Random Policy)

Notice that due to extremely long computational time for naive depth limited search (without any pruning and optimizations), it is not feasible for depth limited search to even complete a game in a reasonable amount of time, thus the measures for the naive depth limited search in terms of winning rate and number of steps to win are missing from the results shown.

Approach	d_{max}	$t_{\pi(s)}$ (in seconds)	Winning rate	Number of steps to win
DLS	1	180.91	-	-
DLS + Pruning	1	0.737	97.8%	10.89
DLS + Pruning + heuristic optimization	1	0.139	99.2%	10.97
DLS + Pruning + heuristic optimization	2	12.61	100%	9.49
DLS + Pruning + heuristic optimization + Leant weights	1	0.079	98%	11.10

Table 2: Performance of Minimax Policy (against Baseline Policy)

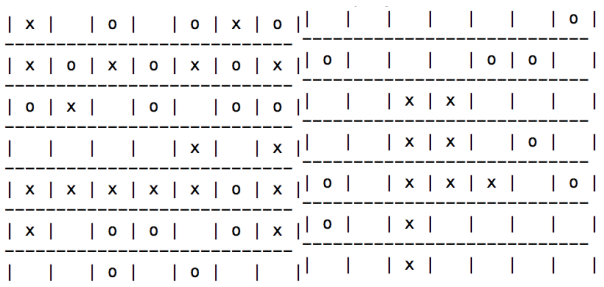
We also tried to measure the winning rate of the best policy we have so far in a game against a human player. The agent using DLS + Pruning + heuristic optimization approach can always win when playing against a intermediate level of human player.

5 Analysis

As shown in Table 2, pure depth limited search without pruning is extremely computational costly in practice even with the maximam search depth being 1. With $d_{max} = 1$, given a state in the search tree, a total number of N^2 of actions needs to be considered in the worst case, where N denotes the width of the game board. For each action considered, we also need to compute the heuristic value associated with the state lead by the action taken, which requires $O(N^4)$ time without the optimization discussed in Section 2.3.3. As a result, given a current state, we spend $O(N^6)$ time choosing the next action which maximizing the value for the agent. This number will grow exponentially with an increase in d_{max}.

By using the alpha-beta pruning strategy, we are able to save a substantial amount of computational time while achieving very good performance in terms of winning rate compared with the baseline policy. This shows the power of branch elimination brought by alpha-beta pruning in practical search implementations.

The high winning rate that the minimax agent achieves can be contributed to the fact that the counting winning window strategy used in approximating the value of a state is very effective. Two snapshots of the partial game boards at the end state are shown in Figure 2. The minimax agent tends to group its pieces compactly in order to achieve a higher heuristic value, which indeed can increase the winning rate of a player. On the contrast, the pieces of a baseline agent are more scattered around which can be one important contributors of the low winning rate.



(a) Baseline Policy (b) Minimax Policy

Figure 2: Snapshots of the (partial) game board in the end state with agent using different policies

Assume the agent always holds black pieces (x). Also assume that the opponent holds white pieces (o) and uses the random policy.

We have also tried to optimize our program further in terms of computational complexity and found that evaluation of heuristic function has becomes a bottleneck of our system. As discussed in Section 2.3.3, a possible optimization on heuristic evaluation can potentially reduce the complexity from $O(N^2)$ to $O(1)$. Thus, by implementing such optimizations, we are able to reduce the computational time per action by 81.2%.

To further increase the winning rate of our agent, we have also tried to increase the maximum search depth d_{max} which should intuitively give a more precise approximation of the value associated with the current state. The results in Table 2 show that increasing the search depth indeed improves the winning rate, which results in a minimax agent that can beat the baseline agent all the time, *i.e.* winning rate equals 100%.

However, there is a trade-off between search depth and computational time per action. Despite of the nearly perfect winning rate achieved by the larger search depth, the computational time per action increases exponentially as we increases the search depth.

Finally, we have tried to obtain the optimal weights in the heuristic function approximation through TD Learning. When trained againt different opponent policies, the weights diverge in different directions due to randomness in learning and limited training time. For minimax as opponent, the training gives slight learning signal since we restrict the board size and the game ends in tie easily. As shown in Table 2, we get reasonable results using baseline opponent, which achieves 98% for winning rate. Another experience we gained is that regularization plays a critical role in learning weights.

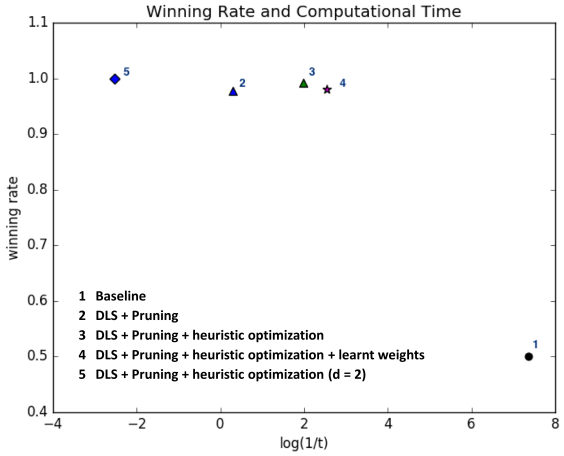


Figure 3: Performance and Complexity Tradeoff

To illustrate the trade-off between the winning rate and computational complexity of all approaches, we have

summarized the corresponding numbers in Figure 3. Approaches with high winning rate (large value in y-axis) and low computational complexity (large value in x-axis) are preferred. It shows that DLS + Pruning + Heuristic Optimization with hard-coded wights and with learnt weights are the two best approaches when considering the performance and complexity trade-offs.

6 Conclusion and Future Work

In this project, we have explored different approaches to develop an AI agent for Gomoku game based on depth limited search. We have emphasized on developing good evaluation functions to estimate the value of a state and coming up with efficient implementations that have low computational omplexity. It turns out that when competing with the baseline agent, we have achieved a high winning rate (up to 99.2%) using a minimax agent that follows depth limited search approach with alpha-pruning and heuristic optimizations implemented.

Despite the good performance of the minimax agent against the baseline, we haven't thoroughly experimented our agent with experienced human players. To obtain an optimal agent, we consider the following directions:

- Extract more features by expert knowledge: Our current feature space only contain five features, each corresponding to the count of winning window that contains a specific number of pieces. We might consider extend our feature space by learning from some dataset that contains previous games engaging human expert players. For example, we may consider adding features that reflect possible winning patterns that involve multiple directions.
- Leverage advanced regularization skills for TD Learning: It is observed that TD learning can over-fit to the data. In this project, we use gradient decaying to avoid overfitting and arbitrarily large weights. In future work we might also consider other specialized regularization methods for TD Learning, such as l1 regularized off-policy convergent TD-learning method(RO-TD), which has shown promising experimental result with little computation cost.

References

[1] Wu, I-Chen, and Dei-Yen Huang. "A New Family of k-in-a-row Games." Advances in Computer Games. Springer Berlin Heidelberg, 2005.

[2] Alus, L. V., and M. P. H. Huntjens. "GOMOKU SOLVED BY NEW SEARCH TECHNIQUES." Computational Intelligence 12.1 (1996): 7-23.

[3] Allis, L. Victor, Maarten van der Meulen, and H. Jaap Van Den Herik. "Proof-number search." Artificial Intelligence 66.1 (1994): 91-124.