An Intelligent Agent for Connect-6

Sagar Vare, Sherrie Wang, Andrea Zanette
{svare, sherwang, zanette}@stanford.edu
Institute for Computational and Mathematical Engineering
Huang Building 475 Via Ortega
Stanford, California 94305

December 17, 2016

Abstract

Connect-6 has been gaining popularity since its introduction to the Computer Olympiad. It is a highly complex search problem, with a branching factor of $O(65,000)$. Here we design an AI to play Connect-6. To tackle the complexity of the game, we present a beam search algorithm with a simple yet powerful evaluation function. The agent beats commercially available AI agents for Connect-6.

Introduction

Connect-6 is a two-player strategy game similar to Gomoku. The game is played on a 19×19 lattice board (like Go), and the objective is to connect six stones in a line. Since its 2003 introduction in Taiwan [1], Connect-6 has become increasingly popular, especially in the Middle East. In 2006, it was added as a tournament to the Computer Olympiad.

In this project, we design an intelligent agent to play Connect-6. The primary challenge is the game tree's enormous branching factor, which due to the game rules is $O\left(\binom{19^2}{2}\right) \approx 65,000$. We will explore methods such as greedy search, beam search, crafting a good evaluation function, and optimizing feature weights stochastically. Our code is at: `https://github.com/andreazanette/cs221-project`

Game Rules

Connect-6 consists of two players, Black and White, alternately placing stones on empty intersections of a Go-like 19×19 board.

Game play is as follows.

1. In the first move, Black places 1 stone on the board.
2. Subsequently, White and Black take turns placing 2 stones.
3. A player wins if he is the first to place 6 or more connected stones in a horizontal, vertical, or diagonal line on the board.

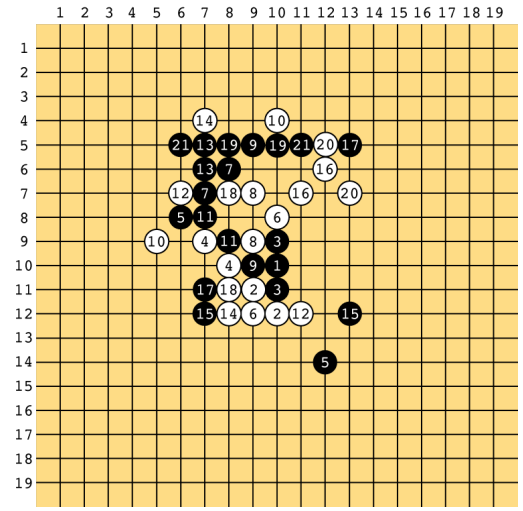

Figure 1: Connect-6 is played on a 19×19 board akin to Go. Here our baseline algorithm (Black) defeats an Android app's beginner AI in 21 moves.

The first player places only one stone in the first move for fairness: at any given move, a player always finishes with one more stone on the board than his opponent.

Complexity

Since Connect-6 is played on a 19×19 board, its state space complexity is on the same order as that of Go at $O(10^{172})$, and much larger than that of Chess (Table 1).

Since its creation, the *branching factor* of Connect-6 has been identified as a major difficulty for a computer agent playing the game. There are (at most) $19^2 = 361$ positions available for each stone. Placing two stones per move results in a branching factor of $O\left(\binom{361}{2}\right) \approx 65,000$ at each ply.

Game	State space	Branching factor
Chess	$O(10^{18})$	$O(140)$, average 35
Go	$O(10^{172})$	$O(361)$, average 250
Connect-6	$O(10^{172})$	$O(64\ 980)$

Table 1: State space and game tree complexities for Chess, Go, and Connect-6. Our challenge is to reduce the branching factor of Connect-6 to make search tractable.

The set of potential moves is thus much higher than for a game of alternating single moves like Go, making it intractable to perform any exhaustive search even up to a very small depth. Thus, to the best of our knowledge most computer programs for Connect-6 use a combinations of heuristics with handcrafted features ("threat detection"), Monte Carlo search, alpha beta pruning, and others.

Previous Work

Interest in Connect-6 started around 2002, when it was settled that Gomoku (Connect-5) is not a fair game [8]. By following a particular strategy, one can always win in Gomoku; by contrast, it is believed that Connect-6 is a fair game [1].

Around 2006, Connect-6 was introduced to the Computer Olympiad [6] which led to a spree of AI agents that played the game efficiently. Unfortunately implementations from the Computer Olympiad are not public, but its introduction to the competition inspired further research. Zhang *et al.* [5] explored the application of chess strategies to Connect-6 and developed the MTD and "deeper-always" algorithms. Most of these algorithms are based on tree search with a minimax agent [7]. Likewise our best agent employs a search-based algorithm, though in the scope of this project we have not played against these other cited search agents.

In addition to search, another strategy for playing Connect-6 is the threat-based strategy [1]. Rather than consider the consequences of each move at a given board, it explicitly defines what types of moves lead to a winning state, and aims to produce or prevent such moves. This basic strategy is the crux of our baseline algorithm, and we will describe it further in the Methods section.

Model

Objective

We aim to design an AI agent that plays Connect-6 at the level of amateur human players. Along the way, we will compare our agent against publicly available commercial AI agents. We will also analyze our performance by testing our agent's runtime at different search depths and branching factors.

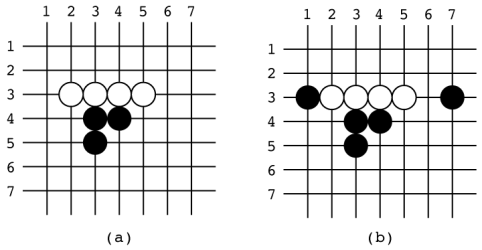

Figure 2: An example of an (a) input board and (b) output stone placement decided by a game-play strategy.

Game Class

We formalize the game using a state-based model, where a state is the board configuration (location and color of all stones), an action is a sequence of two stones (one move), and the successor of a state-action pair is the new board configuration. The initial state is the empty board, and a state is an end state if a player has won the game.

Our Connect-6 class houses the game parameters, the board, a list of all moves made, and the functions relevant to gameplay. These include placing stones, removing stones, switching players, testing if the game has ended, obtaining winning moves, obtaining win-blocking moves, and more.

Specifically, we store the board as a dictionary, where the key is the board location and the value is the player $\{+1, -1\}$ who has placed a stone at the location.

Input and Output Behavior

The input to each of our algorithms is the state of the board, and the output is the location of stones to be placed by the current player.

For example, suppose it is black's turn in the game in Figure 2, and our algorithm can block opponent winning moves. The input and output are:

- Input:
  {`(3,2):-1, (3,3):-1, (3,4):-1, (3,5):-1, (4,3):+1, (4,4):+1, (5,3):+1`}
- Output: `[(3,1), (3,7)]` (or `[(3,6)]`, or `[(3,1), (3,6)]`, to block white)

Methods

In this section we describe the main ingredients adopted to create our agent for Connect-6. The majority are ways to reduce the branching factor of the game tree to make search tractable.

Threat Detection

The first and most basic strategy we employ is the threat-based strategy.

The strategy checks every window of 6 spaces on the board; if the window has ≥ 4 opponent stones, no stones that belong to the current player, and no marked stones, it marks the rightmost (or leftmost) empty space in the window. This space is called a "threat", because failure to block the space can result in the opponent winning in his next move. If the number of threats is > 2, the player can lose the game next turn (and will if the opponent plays optimally).

In Figure 3 we provide examples of White having a single threat in pattern A, a double threat in pattern B, and a triple threat in pattern C. That is, White needs at least one stone to stop the threat imposed in pattern A, and two stones to defend the threat in pattern B. But in pattern C, White needs at least three stones to block Black; if Black plays optimally, White has lost the game.

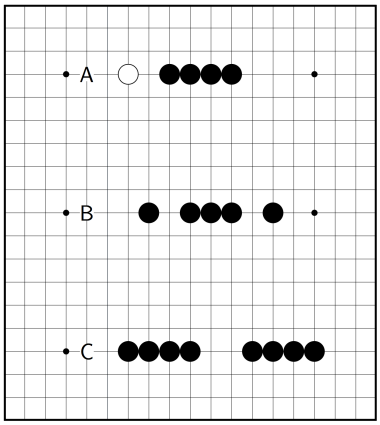

Figure 3: Example of one (A), two (B) and three (C) threats, respectively.

Active Area Identification

As discussed, if we naively assume an agent may play anywhere on the 19×19 board, the high branching factor of $\approx 65,000$ precludes the possibility of examining the game tree to any meaningful depth to find the best move.

Thankfully, not all regions of the board are equally good options for a player's next move. Unlike Go, Connect-6 possesses *high localization* — it is better to place stones near existing stones on the board than far away.

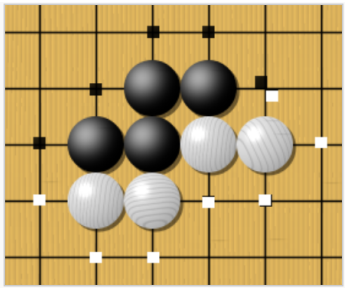

Figure 4: Active area for the given board with $b = 1$.

Thus we believe it is natural to restrict moves in the game tree to an "active area", which we define as spaces within b horizontal, vertical, or diagonal steps from existing stones. Put differently, location (x, y) is a potential move if and only if there is already a stone within b lattice spaces from (x, y). This belief is supported from our own experience playing against commercial AIs; we never experienced any state in which it is advantageous for a player to place isolated stones, nor did we witness any AI playing such a move. Figure 4 highlights the active area for $b = 1$.

For our beam search agent, we search within an active area of $b = 2$. The choice of 2 units is not arbitrary; it is the same as the number of stones a player places per move, and therefore the number of spaces away from existing stones that we must consider to find all possible winning moves.

The resulting active area identification algorithm is below.

Algorithm Active Area Identification

Require: board state
1: **for all** empty locations (x_i, y_i) on the board **do**
2: **for all** locations (x_j, y_j) within distance 2 from (x_i, y_i) **do**
3: **if** location (x_j, y_j) is a stone **then**
4: add (x_i, y_i) to active area
5: **end if**
6: **end for**
7: **end for**

This active area identification is one of the key ingredients of our search agent, as it reduces the branching factor from $O\left(\binom{361}{2}\right)$ to under $\binom{150}{2} \approx 11,000$ for moves in typical games, allowing our agents to make moves in reasonable periods of time.

Beam Search

Even with a branching factor of $\approx 11,000$, backtracking search becomes impossible as games re-

quire multiple moves to arrive at an end state. To make search possible, alpha-beta pruning is often used in the context of adversarial games. Unfortunately for us, alpha-beta pruning requires a search of ply depth at least 3, which is also prohibitively deep for even the lower branching factor.

We therefore turn to beam search, a version of breadth first search that keeps the B most promising candidates at a given search depth. Greedy search is a special case of beam search, with $B = 1$.

At a given board state, we generate all m possible actions and evaluate their successor boards using a function discussed shortly. We then keep the B best successors, and search deeper down these branches by generating the opponent's possible moves. For each of the *player's* B branches, we keep the opponent's single best move, because we assume he is a minimizing agent. In the next ply, we are back to evaluating the player's moves, and keep again the B best branches at a ply depth of 3, and so on.

Algorithm Minimax Beam Search

Require: board state s, ply depth p, beam size B

```
player ← current player
bestMoves ← []
while p > 0 do
    bestChildren ← []
    for m in bestMoves do
        for all moves a in active area do
            evaluate successor board succ(s, a)
        end for
        sort successors
        if current player is player then
            add all move sequences to bestChildren
        else
            add 1 move sequence resulting in lowest
            successor evaluation score
        end if
    end for
    bestMoves ← best B move sequences in
    bestChildren
    switch player
    p = p − 1
end while
```

Evaluation Function

The only definite measure of a board state in Connect-6 is whether a player has won or lost, corresponding to evaluations of $+\infty$ and $-\infty$, respectively.

Using only winning and losing as evaluation means that we must perform search until games end. Although beam search scales linearly with ply depth searched, in actual game play searching to game completion can be prohibitively slow.

To overcome this, we implement an evaluation function, which tells us how good a board is based on features other than win and loss.

Like the threat-based strategy, the evaluation function needs to take into account the key role that "threats" have in Connect-6. We define a *forced move* as a move that the opponent must play as a result of an existing threat. One threat forces the opponent to defend himself with one stone and can rapidly escalate to a two-threats condition. Two threats deserves special attention since it forces the opponent to use both his stones to defend, depriving him of the opportunity to attack or to carry out strategies of his own. If the player can force the game repeatedly and arrive at a state where three threats are present concurrently, then the player has won the game.

After careful observation of how Connect-6 is played with commercial AIs we realized that there are two key ingredients for a successful evaluation function.

- An evaluation function must correctly identify the number of threats posed by each player's stones. Not all 4- and 5-length sequences pose the same number of threats.
- It is good to create a *positional advantage* by means of an underlying connectivity of stones, for example multiple 3-in-a-row sequences, for later use.

While the former point can be addressed with a careful implementation of the evaluation function, the latter is far more challenging. It means that an AI needs to have some *long-term strategy*. However, this is in contradiction to the *short-term tactics* of the threat-based search.

If the AI focuses exclusively on its own long-term planning, it might inadvertently let the opponent create a sequence of threats that have to be stopped with moves useless for its original plan. Therefore a successful AI will build a good network of stones *while playing forcing moves* to disrupt the opponent's strategy.

To this aim we rely on the concept of s-index, detailed below.

s-index

The s-index is a feature of a board state based upon how well-connected it is for a given player. We take all windows of size six *where there are no stones of the opponent*, and count the number stones of the current player (the player's stones need not be contiguous). If there are s stones of the player, then we call it one count of a s-index (Figure 5).

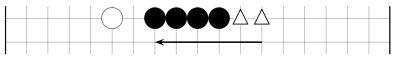

Figure 5: An example of one 4-index in the window defined by the arrow. The number of triangles marks the 4-index count for this sequence.

Notice that the 4-in-a-row sequence in figure 5 yields a 4-index of 2. One is identified by the arrow, the other is obtained by moving the window one unit to the left (it still contains 4 stones of the player and none of the opponent). If the window is moved two units to the left then it would contain no s-index, for any s because there is an opponent stone in the window.

By contrast, an empty space in place of the white stone would result in a 4-index of 3. Thus, this procedure effectively encodes the existence of an opponent stone on one side, which is a less favorable condition for Black.

In summary, the number of s-index gives an idea of the connectivity of one's stones. We both want larger s to be weighted more in the evaluation function computation as they are closer to being 6-in-a-row, and a higher count of each s-index. This evaluation function theoretically balances strategy and tactics if the weights are set appropriately.

Weight Selection

After playing many games, we understood that each longer sequence is exponentially more important. To ensure that our AI agent understands early in its search that leaving an opponent's 4-index sequence of the game is equivalent to losing, we assign a large negative weight to an opponent's 4-index (Table 2).

We also capture other observations gleaned through our own gameplay:

- A 4-index is as good as a 5-index, because they both create the same number of threats.
- It is often more valuable to destroy an opponent's s-index than create our own s-index.

In our first search agent, these insights have been hand-crafted into the weights. We improve upon this with automatic weight selection in the weight optimization section to follow.

Fast Evaluation

The evaluation function just described re-examines the board anew each time it is called, which makes it too slow ($\approx 0.2s$) to use even in a search with depth 1. However, notice that when a stone is placed in the board, only the areas around that location are affected. In other words, we can update the evaluation function by recomputing the s-index in 4 directions, each of window size 12.

Player Weights		Opponent Weights	
2-index	$1e2$	2-index	$-1e2$
3-index	$1e3$	3-index	$-1e4$
4-index	$1e6$	4-index	$-1e10$
5-index	$1e6$	5-index	$-1e10$
6-index	$1e30$	6-index	$-1e29$

Table 2: Player and opponent weights chosen manually for our first search agent.

Specifically we perform the following update at each move:

- If previously there was a s-index in this location of the player, then we change the dictionary holding the counts, by increasing the count of $s+1$-indices by one, and reducing the count of s-indices by one.
- If previously there was an s-index of the opponent, then by placing this stone we have destroyed it, so we reduce the count of the opponent's s-index by one.

Code Parallelization

To achieve further speedup, we parallelized the computation performed at the leaves of the search tree. In other words, we compute the evaluation function for different final successors in parallel. Since this affects only part of the code we achieved a sub-linear reduction in the run time.

Algorithm Computing features

```
Require: Active area of the board
for every locations X in the active area
  for ever direction D
    for every six length window in direction D containing X
      if window contains stones from both players
        pass
      else if window contains stones from one player
        s = sum of stones in window
        Count(s-index)++
      else window contains no stones
        pass
return Counts
```

Algorithm Fast Updates

Require: Computed Features and player's move
for every stone X in the player's move
  for ever direction D
    for every six length window in direction D containing X
      if window contains stones from both players
        pass
      else if window contains stones from player's
        s = sum of stones in window
        Count(s-index)--
        Count($s+1$-index)++
      else if window contains stones from opponent's
        s = sum of stones in window
        Count(s-index)--
      else window contains no stones
        pass
  place stone X on the board
remove all the placed stones
return Counts

Results

Baseline Agent

Our most basic agent decides upon a move by following steps in sequence:

Algorithm Baseline Agent

Require: threat detection
if possible to win in this move
  play the winning moves
if number of threats ≥ 0
  for every location in possible threats
    play a stone at location
if any moves left
  play to increase the largest sequence

Performance

Our baseline agent defeats Level 1 of the Google Play commercial AI "Connect6". The agent plays surprisingly well by blocking wins almost optimally, and ensuring that the opponent cannot win easily. The game between the agent and Level 1 commercial agent is shown in 2, and analyzing the game we see that the simple idea of extending the largest sequence produces a decent player.

The baseline agent was, however, defeated by the commercial AI on Level 2.

Beam Search Agent

Performance

The beam search agent with an s-index-based evaluation function performs remarkably well, beating the commercial AI at its hardest level (Level 4), and also beating humans who are amateur Go players.

The game between the commercial AI and our agent is shown in 6. Analyzing the game we see that our agent plays a good game by disrupting the underlying structure of his opponent, always forcing opponent moves and blocking optimally whenever possible.

We play against our agent ourselves and find that it chooses some really clever moves. After observing many games, we see that the agent plays aggressively by forcing the moves of the opponent at nearly every move and then drives towards creating three threats.

Runtime

We compared the beam search agent's run time over different ply depths and beam sizes. The results of our comparison are shown in Figure 8. These results were obtained by playing our beam search agents with given beam size and ply depth against our baseline agent, and then averaging over multiple games. We see that the run time increases as the game progresses, because the active area grows as more stones are placed. We also note that the run time scales linearly with the beam size and the ply depth, as expected in beam search.

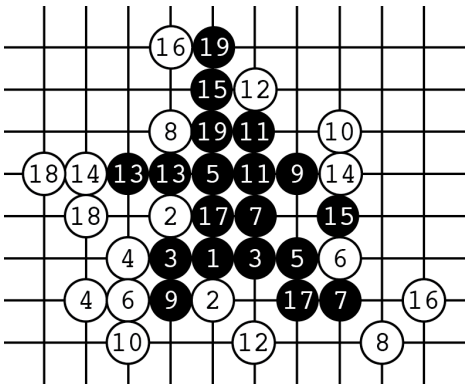

Figure 6: Beam search agent defeating the commercial AI at its hardest level.

Weight Optimization

We use ideas similar to particle filtering by creating multiple agents with randomly selected weight values and making them play against the agent — call him Agent 0 — with handcrafted weights.

Next, we select the agents that defeated Agent 0 and create more agents similar to but slightly different from these agents. This results in obtaining weights that are better than those we handcrafted.

One challenge that we faced during optimization is how to update the baseline agent that all particles play against in a computationally efficient manner. We believe this to be an interesting extension that we would want to pursue in the future.

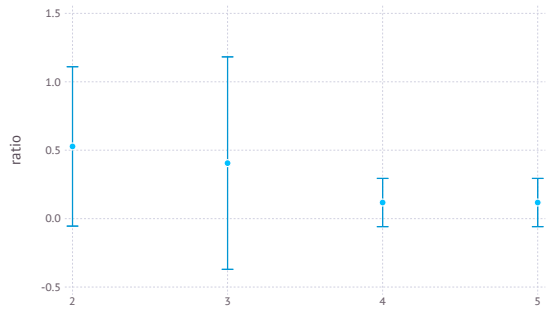

Figure 7: Ratios of player to opponent weights for each s-index that result in agents that defeat Agent 0.

Figure 7 shows the distribution of weights of the agents that beat Agent 0. We see there can be quite a spread of weights, but can glean some basic insights:

- On average we should value destroying our opponent's 2-index twice as much as creating our own.
- We should value destroying our opponent's 3-index sequence twice as much as creating our own three length sequence.
- We should value destroying our opponent's four length sequence 4 times as much as creating our own (similarly for the 5-index sequences). This ratio means we will always destroy opponent's 4- and 5-indices.

These insights are in line with our intuition of the game.

Conclusion

The aim of this project is to create an AI to play Connect-6 using techniques learned in CS221. We successfully achieved this goal and created a surprisingly strong AI which is able to beat the most difficult commercial AI we could find, as well as some humans. In particular, it beat everybody during the poster presentation for CS221 including (amateur) Go players and people who have played Connect-6 before.

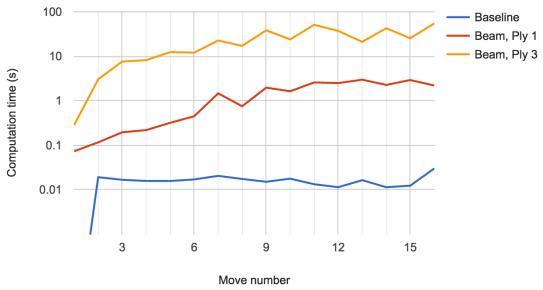

Figure 8: Run time of beam search agents with different ply depths

The high branching factor of Connect-6 precludes searching in the game tree to any meaningful depth. We address this problem in three ways by (1) searching in the "active area" of the game, (2) relying on a strong evaluation function to balance strategy and tactics, and (3) using beam search instead of searching the entire action space.

We further implemented ways to improve the speed of the game, through the fast update method and parallelizing the evaluation of final successors. This speed-up makes the agent able to play against other agents in real time.

Next, we experimented with the beam sizes and the depth of search to see if we can intelligently design the depth to get a trade-off between computational time and the strength of agent play. So far, we find that a ply depth of 1 yields the best trade-off. Deeper search agents, while they outperforming ply depth 1,, are prohibitively slow.

Lastly, we improved the evaluation function by automatically discovering new weights through particle filtering.

Future Works

- The exceptional results that we obtained with a strong evaluation function and parallelized search were still constrained by computational time, making deeper searches impossible. One possible solution is to implement our search in a lower level language such as C++.
- We would also like to experiment more with methods such as cross entropy and reinforcement learning to learn better weights for our evaluation function.
- We can also explore efficient search ideas such as Monte Carlo Tree Search.

References

[1] Wu, I. C. & Huang, D. Y. (2005). A new family of k-in-a-row games. *Proceedings of the 11th international conference on Advances in Computer Games.* 180-194.

[2] Hsieh, M. Y. & Tsai, S. C. (2007). On the fairness and complexity of generalized k-in-a-row games. *Theoretical Computer Science. 385*(1-3), 88-100.

[3] Sano, Kentaro, and Yoshiaki Kono. FPGA-based Connect6 solver with hardware-accelerated move refinement. *ACM SIGARCH Computer Architecture News. 40*(5), 4-9.

[4] Pluhár, András. "The accelerated k-in-a-row game." Theoretical Computer Science 270.1 (2002): 865-875.

[5] Zhang, Ruimei, Changcheng Liu, and Chuandui Wang. "Research on connect 6 programming based on mtd (f) and deeper-always transposition table" 2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems. Vol. 1. IEEE, 2012.

[6] Wikipedia contributors. "Computer Olympiad." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 22 Jul. 2016. Web. 22 Jul. 2016.

[7] Müller, Martin. "Global and local game tree search." Information Sciences 135.3 (2001): 187-206.

[8] Alus, L. V., and Huntjens. "Go Moku solved by new search techniques." Computational Intelligence 12.1 (1996): 7-23.