Analysis of Computational Agents for Connect-k Games.

Michael Levin, Jeff Deitch, Gabe Emerson, and Erik Shimshock.

Department of Computer Science and Engineering
University of Minnesota, Minneapolis.
emers089@umn.edu, levi0190@umn.edu,
deitc004@umn.edu, shims004@umn.edu

Abstract. We develop and evaluate several approaches to game-playing agents for the Connect-k family of games. Connect-k is considered a "fair" variant of Go-Moku or Pente, and as such is more suited to game-theory investigations than its solved (provably biased) relatives. We compare a number of agents programmed to play connect-k,, based on several different strategies and computational methods. We identify an error in the threat algorithm used by Connect-k's designers, and demonstrate an improved algorithm to solve this problem.

1. Introduction

In this paper, we examine the efficiency of several playing agents designed for the game Connect-6 (explained in section 1.1). We compare our agents to each other and to the program designed by I-Chen Wu and Dei-Yen Huang, who first published the specifications for Connect-k games.

Connect-6 is more interesting than related, but "solved" games such as Pente, and Connect 4. For each of these games, one player maintains an advantage by always having at least as many pieces on the board as their opponent. This imbalance, and it's solution, is discussed in more detail in the next section. By eliminating the unfair condition, Connect-6 requires intelligent playing behavior rather than forced advantages in order to win, and thus provides a higher degree of challenge for playing agents.

This paper is organized as follows. Section 1.1 defines the game and provides terminology for discussing game state. Section 1.2 discusses generalized strategies for winning Connect-6. Section 2 covers methods for counting threats on game boards. Section 3 lays out the different agents we developed, while section 4 discusses search methods. Section 5 demonstrates the results of competitive trials between our own agents and the agent program "NCTU6" developed by Wu and Huang [10].

1.1. Game Description.

Connect-k, or more properly, Connect(k,p,q) is a variant of Pente (Go-Moku), and Connect-four. Previous work regarding Connect-k can be found in [10], [11], and [8]. Connect-k consists of two players, White and Black, who alternate in placing stones on a board. The first player to sequentially align a given number of stones in any row, column, or diagonal is the winner of the game.

The following is a list of definitions regarding Connect-k:

k - The number of pieces a player needs to place sequentially to win.
p - The number of pieces placed by each player during all but the first turn of the game.
q - The number of pieces placed by the first player in the first turn of the game.

Threat - An arrangement of stones that will lead to a victory for a player in their next turn if not blocked.
Threat count - The number of stones that a player must place in order to neutralize all the threats that the opposite player has created.
Sequence – Any 2 or more adjacent stones of the same color in a line.

Connect-6 is played with k = 6, p = 2, and q = 1. Playing in this manner attempts to ensure fairness. If Connect-6 were to be played in a way where one piece was played on each turn (p = 1) the first player would have an advantage. This is because the first player would always have one more, or as many, pieces on the board as the second player. With p = 1 and q = 1, each player will always have one more piece on the board after finishing his or her turn. Without additional rules and restrictions, experienced players or properly designed algorithmic solutions can always win when playing first [8]. In general, for Connect-k Wu and Huang give an argument to show that a value of p = 2q ensures fairness [11].

1.2. Strategies.

Defensive strategies mainly focus on preventing an opponent from creating k-length sequences. This can be done by directly blocking individual rows, columns, and diagonals where multiple stones exist, or more proactively by looking ahead and blocking locations which could become part of multiple sequences in a single move. A purely defensive strategy does not try to win, it only tries to prevent its opponent from winning. Offensive strategies include building up one's own sequences and "breakout" in which a player makes moves away from the main grouping of pieces. Taking the initiative is an important goal, as it can force the opponent to defend and give the player more freedom in piece placement [9]. Creating multiple threats is also important, as a threat count greater than p cannot be countered by the opponent's next turn, leading to a win [5]. A purely offensive strategy does not try to prevent its opponent from winning, it merely focuses on trying to create a win for itself. Mixed strategies use a sensible combination of offense and defense by attempting to locate and block the opponent's sequences and threats while building sequences and threats of its own.

2. Identifying and Counting Threats.

2.1. Wu and Huang's strategy

The developers of Connect-k, I-Chen Wu and Dei-Yen Huang, propose a threat-counting strategy in their whitepaper [11]. This strategy is based on the idea that "Threats" are sequences within one move of being length-k. Player W (white stones) is considered to have t threats if player B (black) must place t stones in order to prevent a win on W's next move.

Wu and Huang's strategy is to create at least p+1 threats so that the opponent is unable to block all in their move, forcing a win. They use a sliding window on individual lines (consisting of columns, rows, and diagonals) to check the existence of stones on a given line. For the defensive player, the object is then to identify and block potential threats before they can grow in number, as well as blocking immediate potential wins by the opponent.

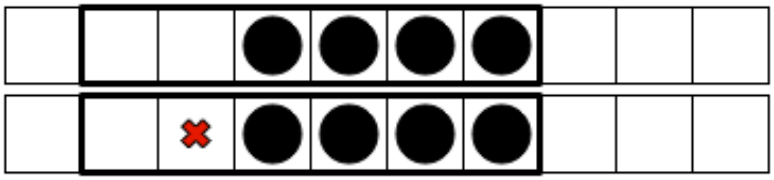
The threat counting algorithm described by Wu is as follows: (from page 7 of "A

New Family of *k*-in-a-row Games") [10].

1. For a line, slide a window of size six from the left to right.
2. Repeat the following step for each sliding window.
3. If the sliding window contains neither white stones nor marked squares and at least four black stones, add one more threat and mark all the empty squares in the window. Note that in fact we only need to mark the rightmost empty square. The window satisfying the condition is called a threat window.

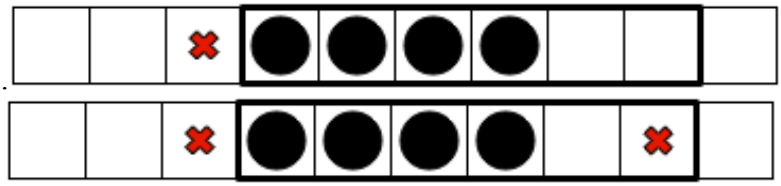
The following series of figures illustrate the operation of Wu's sliding window. We start by sliding the window of size six from left to right.

In the following window, all of the conditions of step 3 are met. (At least four black stones, no white stones, no marked tiles.) The algorithm then increments the threat counter (currently one) and marks the rightmost available tile.



As the scan continues, the next window sees four black stones and no white stones. Since there is a marked tile the conditions of step 3 do not hold, and therefore there is no new threat here.

Proceeding one window to the right finds that all three conditions are again true. The algorithm marks the rightmost empty tile and updates the counter to two.

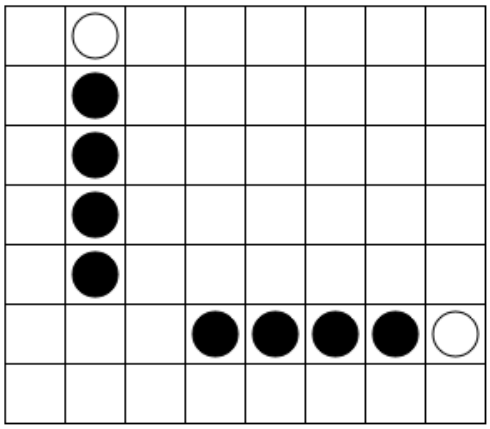


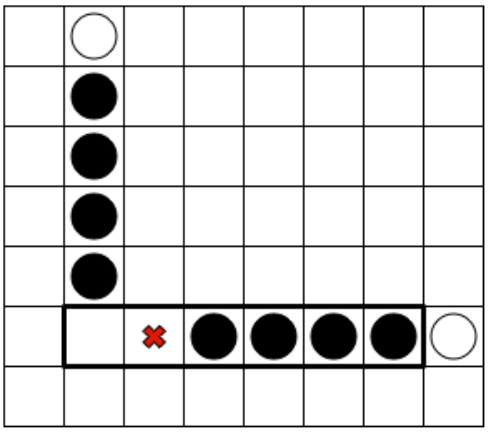
There are now no further threats in this game board space.

2.2 Error in Wu and Huang's Algorithm.

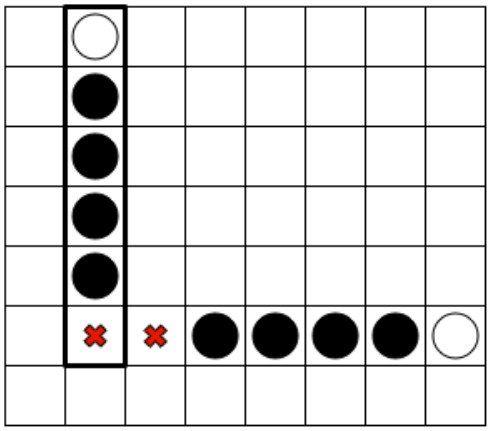
The method shown above works correctly along a single line of tiles, but fails to properly identify all threats in some cases when used in two dimensions. Real-life and simulated games often have intersecting threat areas on multiple lines in different dimensions.

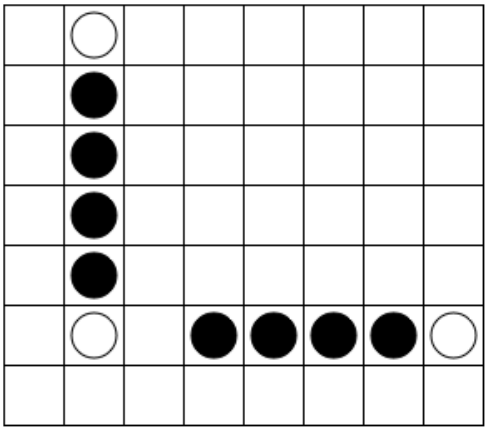
As an example of this error, consider the following situation with nearby stone sequences on multiple dimensions. We assume the agent using the algorithm is playing as White. In this case, Wu and Huang's algorithm, scanning horizontally, will identify, increment, and mark the threat shown below.





When then scanning vertically, the algorithm finds a second threat and marks it as shown at left below. The algorithm considers this case to be two threats, but in reality there is only a single threat which White can neutralize by placing a stone as shown below at right.

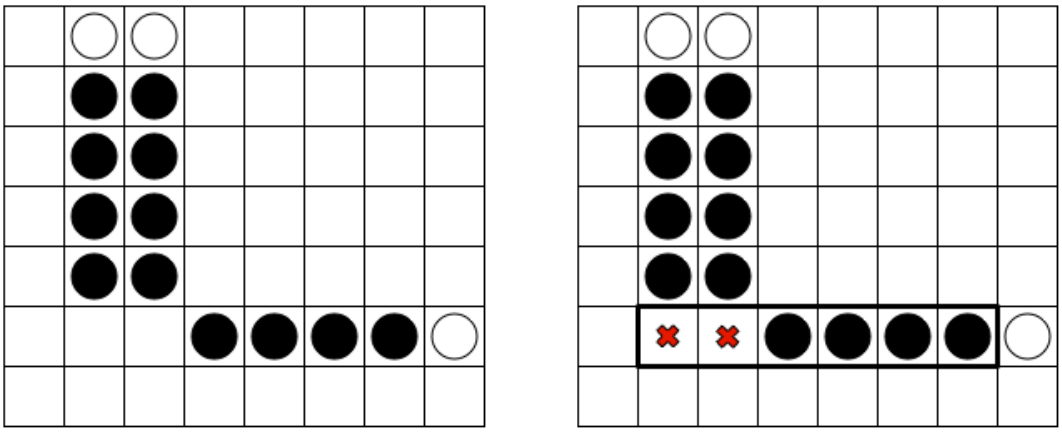




It is easy to see that if the horizontal window identifying the first threat had marked both tiles instead of simply the rightmost one, the vertical window would not have counted the threat a second time. This would have resulted in the correct calculation of one threat. It is worth noting that allowing one window to mark multiple tiles does not mean that one window can see multiple threats. One window can only see at most one threat.

2.3 Additional problems

The improved method for marking multiple tiles is not perfect. The board arrangement shown at left below demonstrates a situation where this method fails. In this case, given we start with a horizontal scan, the first threat will be counted and two marks will be made as shown at right.



As a result of having marked both tiles at once, neither of the two vertical threats will be counted. The threat counter will report one threat, but it will in fact take two stones to neutralize all the threats on this board.

The underlying problem comes from the fact that when we are marking a tile, we are essentially saying, "I see a threat that I can neutralize by playing *here*," and then marking that spot on the board. However, in some situations there are multiple locations where placing a stone could neutralize the threat. The agent will not know which location makes the most sense to mark without doing a deeper analysis, so it simply plays on both. In other words, the threat counting algorithm assumes that we can play two stones at once to neutralize a threat, however obviously this is not the case.

At present, our Sequences AI (described in section 4) uses this slightly improved threat counting algorithm. It is assumed to be better than the original since there are few situations where it will fail, but a clever opponent could use the problem discussed here to exploit the Sequences agent. A better approach is described in section 3.2

3. Playing Agents for Connect-k

3.1. Framework

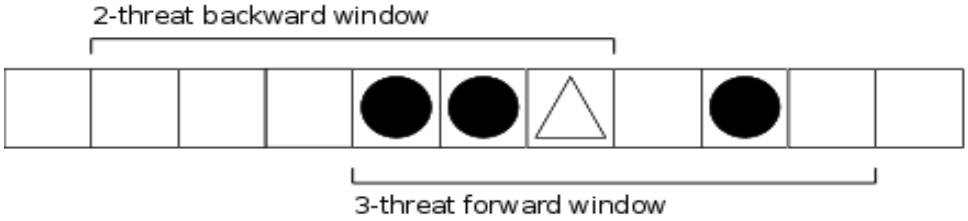
A Connect-k program has been written to serve as a framework for studying this problem. The program is a fully-featured Connect-k game, playable by two humans, by a human and a user selectable agent, or by two agents. In addition to offering a game-playing platform, an API has been created that guides the development of new AI agents, making it relatively easy to add new agents to the framework.

Our code is theoretically capable of handling any board size and any values of k, p, and q, although memory and processing limitations will force some realistic upper limit. At present we are limiting our investigations to standard Go game boards of size 19x19 with k = 6, p = 2, and q = 1.

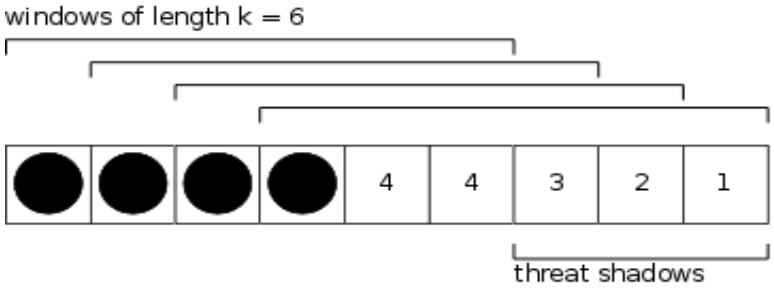
3.2 An algorithm to find threats correctly

We have developed an algorithm to find generalized threats. A generalized threat includes not only immediate threats that require one move to win, but also sequences which require multiple turns to win. This algorithm will accurately show when a move will build multiple sequences as well:

- For every possible horizontal, vertical, and SE and SW diagonal of length at least k:
  - Keep a line cache the length of the line. The line stores two threat values for every tile.
  - For every tile within the line:
    - Check for threats in this fashion in the forward and backward direction:



      - Try to find the largest window on the line that includes the current tile as the starting tile and is of length at most k. This window must contain either empty pieces or pieces of one of the two players.
      - Now similarly find the largest window on the line that ends with the current tile. This time require that this window contain only pieces of one player (if the previous window contains a player's pieces, use that player).
      - Return the largest number of pieces one player has within a window of k inside of the area we scanned. This is the threat value.
    - Set the cache line threat value for this tile to be the largest threat value of the two threat checks.
    - To prevent "threat shadows", that is the reuse of a larger sequence in counting an adjacent smaller sequence (for instance marking a 3-threat by counting all but one piece from a 4-threat window), we have larger threats "dominate" smaller threats. If there is a larger threat value within a k distance of the current tile, we do not set smaller threats on the line.



    - If both threat window checks' forward and backward windows' lengths sum to at least k, set the second threat value in the cache line to the smaller threat value. This means that if we play at this tile, we will create (or block) two valid sequences rather than one.
    - For both threat values, if they are non-zero, convert them to integers via this formula:

      $n_t = 2^{c(t-1)}$

      Add the resulting value for both threats t to the current (initially zero) board values. The constant c must be large enough to prevent overflow (we use 6) of lower tier threats into higher tiers during the addition step and small enough such that the integer produced for the largest threat fits

within the integer size limits of a modern computer. Do this for every threat in the cache line.

Optionally after all the threat values are counted on the board, we may re-prioritize the threats. For instance, building sequences greater than k - p without enough moves left to win is not advantageous. In some cases it is more advantageous to build multiple smaller sequences than simply extend the largest sequence.

3.3. Agents

The crux of this work has been the development of intelligent agents capable of playing Connect-k. Agent programs have been designed to interface easily with the game program, and can be set to compete against each other or against human players in any generalized k, p, q domain. The set of agents programmed to date include the following;

Adjacent

Adjacent places a piece randomly in a tile that is adjacent to an existing piece. This is a proof-of-concept agent and was not intended to be a competitive program. It is useful for testing search techniques, especially for small board sizes where searching is effective enough to find a win quickly.

Windows

Windows uses a sliding-window scheme for identifying sequences (and threats), based on algorithms described in [2] and [11]. This AI tries to block multiple windows that give the opponent chances to make sequences.

Threats

Threats uses the threat detection algorithm directly as a utility function. It attempts to offensively create sequences of pieces that can lead to the creation of multiple threats. If the agent can create a threat count greater than p, it will force a win.

Monte Carlo

Monte Carlo picks the next move based on which one wins the most random games in a simulated setting. The reasoning behind this approach is that stronger moves will win more random games. The motivation for this approach comes from the successful use of this same approach in Go [3].

Sequences

Sequences determines the utility of each possible move by calculating a board utility that would result if the move were played, and then chooses the move with the highest utility. This agent uses the windowing method to traverse rows, columns, and diagonals on the board for each player (color) Each window seen by the agent is checked and given a score as follows (assuming it is White's turn):

- Windows containing any black stones are given a score of zero.
- A window containing any white stones r is given a score, equal to the number of stones squared. By squaring, we assign higher scores to lines or groups of stones than to scattered singletons.

- If a window contains four or more white stones and no marked tiles, we note that White has a threat in the window and mark all empty tiles. Marks for each player are recorded separately.
- The process is then repeated for Black, keeping threat windows for black and white separate.

The data collected by this process is then used to compute the board utility. This is done as follows, assuming it is White's turn:

1. If White has an immediate win available (sufficient moves left in the turn to make a k-length sequence), the current board is given the maximum utility.
2. If any threats exist from the opponent, the board is given a low utility. The more threats, the lower the utility.
3. If white has P+1 threats, it is possible for white to win on the next turn. This board is given a very high utility, but less than the utility of an immediate win.
4. If none of these conditions are met, the next move is chosen based on the following utility calculation: The agent takes the sum of White's window scores, minus the sum of Black's scores times a defensive constant..

Thus, the total utility for the board is equal to:

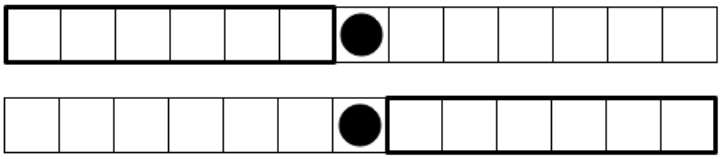
$$U = \sum_{windows} CP^2 - c \times \sum_{windows} OP^2$$

Where CP is the current player and OP is the opponent player.

The defensive constant 'c' allows different weight to be applied to the opponent's pieces, in order to make the agent more offensive or more defensive. By changing c, we cause the current player to value its opponent's scores higher, so that it will prefer to block the opponent's sequences before creating its own.

Scanning the entire board is computationally expensive, especially when playing in domains higher than the standard 19x19 board. We note, as does Muller in [7], that placing a stone on the board only affects a small subset of windows, so it is necessary only to re-scan those windows which have changed after every turn. Data regarding scores, marks, and threats from the unchanged board areas can be cached and reused.

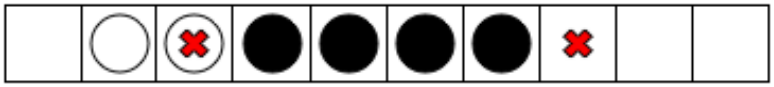
The local updates require us to scan all windows containing new pieces, as well as one window on each side of each new piece. The starting and ending windows for each new piece in a single direction are shown below.



This means that in the horizontal, vertical, and both diagonal dimensions, we need to scan a total of 32 windows (8 for each direction). Simply scanning windows containing new pieces would correctly count sequence lengths, but would possibly not correctly count threats. As an example of this, consider the following simplified board configuration.

As the sliding window proceeds from left to right, the agent playing as White will see and mark the threat as follows:

The agent then places a new stone on the left side of the threat (on a marked tile):



When using the incremental method, agents remember which marks were created by the current window, so when rescanning an area, the current window checks if it has already placed marks there. If the agent finds any marks left by the current window, they are removed to reevaluate the threat conditions (So if a window sees, marks, and counters a threat, it will then un-mark other spaces it associated with the threat and re-check for other threats). Marks left by different windows are not removed, but treated normally.

If the agent then re-scans only windows which contain the new stone, we encounter the following situation:

The agent has removed its previous marks, so when checking the last window containing the new stone, it sees the sequence of four black stones as having been blocked, and will not correctly identify the threat on the right. If we want the agent to identify this threat, we must slide the window one step beyond the new stone:

The problem with checking only those windows which contain new stones stems from the multiple ways in which a single threat can be marked. It is possible that the window which first sees a threat will encounter a situation where it no longer considers it a threat, but a different window will.

By itself, the utility function would sometimes value multiple k-1 length sequences more than single k-2 sequences, which is not always optimal. However, the algorithm works well when told how to handle specific cases of immediate-win and imminent threats.

Mesh

The Mesh agent attempts to create large clumps of stones. The motivation for this strategy comes from the observation that creating clumps of stones allows freedom to create multiple threats. The more adjacent stones belonging to one player, the greater chance that player has of creating p+1 threats. A contrived example of this would be a 3x3 square of stones. In such a situation a player would have 8 locations in which to create threats, after only playing 9 stones.

To create clumps of stones, the agent treats the board as an elastic surface in which pieces are affected by gravity. Placing a piece (stone) will cause a depression in the board. The more stones in a region of the board, the "deeper" the depression in the surface. The depth of each tile is calculated by adding the weight of the stone at that tile (if any) to the average weight of the four nearest stones in each of the eight radials surrounding the tile. These sets of equations are solved iteratively and the lowest tile that has not yet been played in is chosen.

Opening Book

Opening Book is not a full agent, in the sense that it only gives move recommendations up to a fixed number of turns into a game. More specifically, it can make move recommendations for common early game situations. These recommendations are based on analysis of a large number of previous played games whose moves and results have been recorded in an external file, as in pattern-based learning [6]. To make its decision about which *k*th turn move to recommend, it starts by finding all observed games which had the same game state after (*k*-1) turns, I.E. games whose board looked the same after (*k*-1) turns (Note that in this section, a move refers to placing p pieces on the board). It then looks at the *k*th turn moves in the observed games as well as the games' final outcomes, and chooses the best observed move to recommend (see later in this section for a discussion of the meaning of "best" in this situation).

The Opening Book agent learns by building what we call a statistics tree from the observed games. A node at depth *k* in the tree contains the following information:

- an encoding of the board state (after *k* turns),
- statistics on the eventual outcomes of observed games which had this board state, recorded as win, tie, and loss counts for player 1,
- links to children nodes (at depth *k*+1) with each link labeled by the observed move which causes the board state change.

The root of the tree is the node representing an empty board, and since every game starts with an empty board, the outcome statistics will be the win/tie/loss counts over all observed games. A specific observed game is thus represented as a walk down the tree from the root, following the links corresponding to the moves that occurred in that game. It is important to point out that this statistics tree is not exactly a tree in the traditional sense because there may be multiple paths from the root to a specific node. This is because the order in which moves occur doesn't affect the board state--that is if player 2 makes move A, player 1 responds with move B, and then player 2 makes move C, the board state is the same as if the moves occurred as C, B, A.

The statistics tree is built from a set of observed games by enumerating over each game. A game is added to the tree by walking down the tree from the root, adding links for each observed move. These links either connect to existing nodes if the resulting board state is already in the tree, or newly created nodes if the resulting state isn't in the tree (the search for existing states only occurs at a specific depth in the tree since it takes an exact number of turns to get to a given board state). As the walk down the tree occurs, the relevant win/tie/loss count for each node is incremented depending on the outcome of the observed game. Once a statistics tree has been built for a specific set of observed games, the tree structure can be saved to a file so that the tree can be loaded later without having to enumerate over all the games again.

The state of a board can be encoded in a few different ways. One example is encoding a board as a large positive integer. Consider a board with *m* rows and *n* columns. We can index the board tiles so that the tile at row *r* and column *c* has index:

$$index(r,c) = (r-1)n + (c-1)$$

For example the tile at row 1 column 1 (top left) has index 0, the tile at row 2 column 1 has index *n*, and the tile at row *m* column *n* (bottom right) has index (*mn*-1). We can then encode the state of a board as:

$$state = \sum_{i=0}^{mn-1} v_i 3^i$$

where v_i is 0,1, or 2 depending on whether the ith tile is empty, has a black piece, or has a white piece respectively. With a reasonable sized board these state encodings exceed the capacity of most programming language's fixed-size integers, so arbitrary precision integer libraries must be used (e.g. Gnu MP for C or C++ or BigInteger for Java).

Another way to represent a board state is to encode it using a pair of sorted lists. One list holds the sorted tile indices of the black pieces and the other list holds the sorted tile indices of the white pieces. These tile indices can be defined as previously, but unlike the integer encoding method it isn't necessary for the indices to be non-negative. In fact it is possible to use this fact to take advantage of some of the symmetry of the game. When the game parameter q=1 (I.E. player 1 plays one piece on the first turn) then it doesn't really matter where player 1 plays their first piece (for this reason most players just play their first piece right in the middle). Letting r_0 and c_0 be the row and column of player 1's first piece, we can define our indices relative to this position using the definition:

$$index(r,c) = (r - r_0)n + (c - c_0)$$

Since this is a useful symmetry to take advantage of and it is an easy modification to implement (it is merely subtracting a constant from the previous definition) we chose this method to encode our board state.

Once the statistics tree is created (or loaded) it can be used to generate move recommendations. The recommendation for a given board situation is determined by first searching the correct depth of the tree for the node with the corresponding board state encoding (the depth is determined by the turn, which is determined by the number of pieces on the board). If no node has the same board state, no recommendation can be given. Once the corresponding node is found, the win/tie/loss statistics for the children of that node are used to determine the recommended move. A desirable move is one that has a high win to loss ratio, but also has a high enough number of games which contained the move so that the ratio is a meaningful measure (e.g. a move which was observed to cause 1 win and 0 ties or losses has a win to loss ratio of 1.0, but may have just been a fluke). With the need to balance both of these characteristics (good ratio and enough observed games), we were not able to come up with a completely satisfactory ordering relation on the moves. As an example, it is hard to tell if a move with a w/l ratio of 70% in 10 observed games or one with a w/l ratio of 100% with only 2 observed games is the 'better' choice.

Unable to come up with a 'best' ordering, we settled on the following method. We define s to be the threshold of significance for number of games observed (we use s=4). Consider all moves which have been observed at least s times, and pick the move which has the highest win to loss ratio among these. If the ratio is better than 50%, recommend this move. Otherwise, look at the moves which have been observed less than s times, and recommend the move with the most number of wins, with ties going to the move observed the least number of times (e.g. 2 wins out of 2 games is better that 2 wins out of 3 games, but 2 wins out of 3 games is better than 1 win out of 1 game).

The Opening Book agent is limited in several ways. It is restricted to giving recommendations for only a fixed number of turns. The first reason for this is that after enough turns, the size of the statistics tree becomes prohibitively large. The second reason is that as the number of turns increase, the number of observed games matching the situation is likely to decrease, resulting in less sound recommendations. This second reason also limits the Opening Book agent in the sense that if none of the observed games

match the current situation, it cannot make recommendations. For these reasons, the Opening Book agent is more useful as an early-game aid to a human player, or when combined with another agent that can analyze later game moves.

4. Search Methods

In addition to the agents listed above, a Minimax search with alpha-beta pruning has been implemented, and can be used by any agent. To take advantage of the search interface, the agent must provide a list of moves with a utility assigned to each move. To reduce the branching factor of the search, only the moves with the highest utility are explored further at each level of the search. The number of top moves considered is a variable that can be set at run time. A Minimax search allows each agent to be able to look ahead in the game and choose the best moved based on where it approximates the game will be in the future [5].

Allowing multiple moves per turn presents a unique problem to the search algorithm. For simplicity and generality across different values of p, a utility function will process one board at a time and return a list of moves. However, it does not return *sets* of moves. The search algorithm, despite searching one level at a time, must keep track of what move combinations it has already searched. We can solve this problem by marking a tile on the board (on the search stack) as searched, and avoiding tiles that have been marked this way in succeeding function calls. Note that the searched markers must be cleared when a player's turn ends.

5. Agent Performance

By running a game with two agent players, the relative performance ability of each agent can be compared. We have compared each of our agents to our other agents, to our own (inexperienced human) play, and to the Connect-6 program “NCTU6” developed by Wu and Chang [10]. NCTU6 was the champion program of the 2006 ICGA Connect-6 tournament and is considered to be the premier Connect-6 program, it provides an excellent baseline for the efficiency and performance of our advanced agents.

NCTU6 does however have a few noticeable limitations. It is deterministic, meaning that once an exploitable game is found, it can always be reproduced. There appears to be a bug in the program related to incorrect use of multiple threats. We suspect the program also suffers from the error in the sliding window algorithm described in section 3.2.

At present, the publicly available version of NCTU6 is limited to depth-3 search, which means that a competitive agent would need to perform better at the same search depth. However, we believe NCTU6 also uses a pattern-matching database, so we feel that one of our agents using only search but with a deeper depth should still be considered a fair opponent to the NCTU6 program.

Without search, both Threats and Sequences are formidable opponents to inexperienced human players. Humans have trouble keeping track of the many sequences within a complex board, giving computer opponents an edge in longer games.

Using our program's tournament mode, we tested our most advanced AIs against each other. The Sequences utility function won 77% of its games against Threats. Monte Carlo performed a bit better against Sequences, which won 70% of the time. When Threats is enabled with search to depth 4, it will beat Sequences 52% of the time. Because sequences is a slower utility function than Threats, it was prohibitive to thoroughly test it using a deep search, however, limited testing did show improvement.

We could not perform automated tests with NCTU6, but we were to play the program against ours by manually placing pieces. Threats was unable to defeat NCTU6, however Sequences was able to win about 10% of the time.

6. Future Work

There are a number of additional approaches which could be applied towards the Connect-k domain. Genetic search algorithms show a great deal of potential for developing efficient strategies via evolution. A simple way to apply genetic learning would be for more accurate results in the Monte Carlo agent, but this would involve an even heavier processing burden. Monte Carlo could also be improved by caching the results of random games to improve per-move computation requirements.

The mesh algorithm shows potential, but is very easy to beat in its present incarnation. Adding basic defensive and offensive behavior could improve the agent's play. Strategic positioning in addition to clumping would allow a wider range of responses to board conditions and should make the agent "smarter" about creating multiple threats.

The Threats utility function has potential to be improved by prioritizing moves. A clever playing agent could assign different weights to threat values based on the current game scenario. For example, building sequences longer than k-p without enough moves left to win is not always preferred, as such a sequence will likely be blocked. It may be better to create multiple small sequences than to extend the longest sequence.

We intend to release our code as an open-source project, including the Connect-k game framework and collected agent programs. Eventually we hope to produce an agent which is advanced enough to be competitive in the international gaming community. Other types of algorithms which could be applied to this area include reinforcement learning and statistical learning, as well as neural network agents. It remains to be seen which branch of artificial intelligence is most suited to this specific domain.

7. Conclusions

We found that our "Threats" and "Sequences" agents were the two strongest approaches. Relative performance of these two agents was based on the search depth available to the Threats algorithm. Monte Carlo is also a relatively strong approach, although slow in performance. The Sequences algorithm is our best candidate for a competitive playing agent, as it is occasionally able to beat the current world champion program. By utilizing our corrected threat detection technique, we have developed a more accurate and extensible framework for connect-k agents than previously existed.

References:

1. Allis, L. V., and Huntjens, M.P.H. (1993) Go-Moku and Threat-Space Search *Report CS 93-02*, Department of Computer Science, Faculty of General Sciences, University of Limburg, Maastricht, The Netherlands.

2. Batalov, D.V, and Oommen, B. J. (2001) On Playing games without knowing the rules *IFSA World Congress and 20th NAFIPS International Conference, 2001. Joint 9th* Vol 4, pp 1862.

3. Bouzy, B., and Helmstetter, B. (2003) Developments of Monte Carlo Go. *Advances in Computer Games* Vol 10.

4. Bryant, B. and Miikkulainen, R. (2006) Evolving Stochastic Controller Networks for Intelligent Game Agents *2006 IEEE Congress on Evolutionary Computation* Vancouver, BC, Canada.

5. Cazenave, T. A (2003) Generalized Threats Search Algorithm *Lecture Notes in Computer Science* Vol 2883 pp. 75-87.

6. Epstein, S. L., Gelfand, J. and Lesniak, J. (1996). "Pattern-Based Learning and Spatially-Oriented Concept Formation with a Multi-Agent, Decision-Making Expert." *Computational Intelligence*, 12 (1): 199-221.

7. Muller, M. (1995) Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory". Dissertation, Swiss Federal Institute of Technology, Zurich.

8. Pluhar, A. (2002) The accelerated k-in-a-row game *Theoretical Computer Science* Vol 270, No 1, pp. 865-875.

9. Uiterwijk, J.W.H.M, and Van den Herik, H.J. (2000) The advantage of the initiative *Information Sciences* pp. 43-58 v122.

10. Wu, I-Chen and Huang, Dei-Yen. (2005) A New Family of k-in-a-row Games *The 11th Advances in Computer Games Conference (ACG'11)*, Taipei, Taiwan.

11. Wu, I-Chen, Huang, Dei-Yen, and Chang, Hsiu-Chen. (2005) Connect6 *ICGA Journal* Vol. 28, No 4, pp. 234-241.