Heuristics and Threat-Space-Search in Connect 5

Veselin Kulev and David Wu

December 11, 2009

Abstract

Our goal was to write a strong agent for Connect 5 using a combination of search, heuristics, and threat-based win/loss detection, and to examine the effects of these methods on the performance of the agent. Although Connect 5 has already been solved (on a 15x15 board), it remains an interesting domain for testing out the effects of heuristic pruning and dependency-based search in highly tactical games. We produced a reasonably strong agent for Connect 5 and discovered interesting directions for further investigation in pruning and tactical search.

1 Introduction

Our goal for the CS182 final project was to write a strong agent for playing Connect 5. We believed that Connect 5, being both easy to implement and yet nontrivial and tactically complex to play well, would be a good testing ground for experimenting with various methods for game-tree search, pruning, and heuristic evaluation.

There is surprisingly little published material on Connect 5, especially considering the simplicity and well-known nature of the game. This is perhaps due to the fact that it was solved for the 15x15 board a decade and a half ago. As shown by L.V. Allis through a combination of proof-number search and deep threat-search, Connect 5 is a win for the first player [1]. Nonetheless, even on modern hardware, we have found that playing Connect 5 well is highly nontrivial, and remains a very interesting testbed for tactical and heuristic search.

We will first explain the rules for Connect 5, and then present the first major algorithm that we used to approach the problem, Alpha-beta search, and discuss the results. We will present a number of heuristics that we tried, and discuss how these heuristics affected the performance of our agent. We will then present our second major algorithm, threat space search, and discuss results for this algorithm, and conclude with an overall evaluation of our agent.

2 The Game

2.1 Connect 5

Connect 5 is a 2-player adversarial game that takes place on an initally empty NxN grid. Players take turns placing a single piece of their color on any empty location on the grid. The game ends when a player forms a row, column, or diagonal of at least 5 consecutive pieces of his or her color, in which case that player wins, or when the entire grid is filled with neither player having done this, in which case the game is a draw. An example game is shown in Figure 1.

In practical play, Connect 5 strategy revolves heavily around making "threats", lines of 3 or 4 pieces that if not blocked, will allow the opponent to win in one or two moves. Figure 2 displays some common types of threats.

Typically, a player will win by finding a forced sequence of moves that generates two threats at once, which guarantees a win, since the opponent will be unable to defend against both in one move. For instance, Figure 3 depicts a simple forced winning sequence. Absent a such a winning sequence, players will typically try to place pieces in "favorable" configurations that have the potential to generate numerous threats in the future, while blocking their opponent from doing the same.

It is worth noting that the first player has a large advantage in Connect 5. Indeed, as mentioned in the introduction, Connect 5 has been proven to be a first-player win on a 15x15 board, and in empirical play, the first player advantage is large. This has led to the introduction of many variants of the game with additional rules to handicap the first player. However, we have found that among time-limited and imperfect players, though the first-player advantage is very noticeable, there is nonetheless plenty of variation in results. Therefore, we have chosen for simplicity to consider only the version of Connect 5 just described, without any additional restrictions. During test matches between our agents, we merely alternate who plays first.

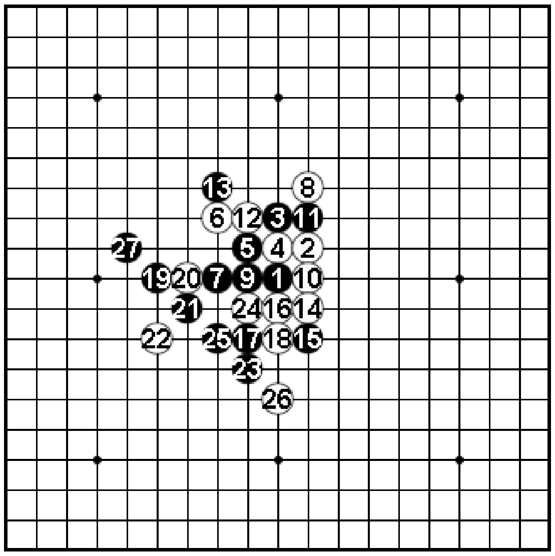


Figure 1: Example game - Black wins on move 27.

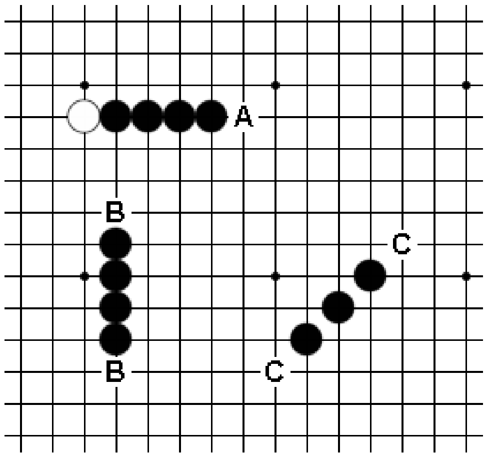


Figure 2: Example threats. Top left - black has four in a row and threatens to make five. White is forced to block at a or else lose. Bottom left - black has a *live-four* with threats at both b's. White cannot block both and will lose on the next turn. Right - black has a *live-three* and can form a double-four by playing at either c. White must block at one of the c's or else lose.

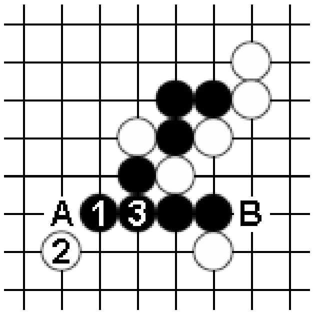


Figure 3: Black makes four-in-a-row with 1, forcing White to block at 2. Then, Black makes two simultaneous threats at 3, and will win next turn.

3 Algorithm 1 - Alpha Beta Search and Heuristics

3.1 The Basic Algorithm

The first major algorithm that we implemented was alpha-beta search, which we used throughout the project as our base algorithm. We chose alpha-beta search because it was a logical and common algorithm for two player abstract strategy games like Connect 5. Moreover, it provided a flexible base upon which we could plug in and try a variety of heuristics, including move ordering, pruning, and evaluation heuristics.

For generating a move in a given position, we also implemented iterative deepening. When generating a move, our agent simply performs an alpha-beta search to depth 1, then depth 2, then depth 3, and so on, up to the maximum specified depth or until it is interrupted by hitting a maximum allowed time for the search. The agent then uses the results of the deepest uninterrupted search.

This allows us to perform equal-time comparisons between different configurations of heuristics and parameters, and allows us to specify time-based restrictions on a search as opposed to merely specifying a search-depth. Since the time needed for each successive depth of search increases exponentially, this adds very little overhead to the agent, since the repeated work is minimal.

3.1.1 Results

	Depth 3	Depth 4	Depth 5
Avg Nodes	50873	1346021	?
Avg Time	0.054	1.527	?

Figure 4: Average performance on 100 test positions on a 13x13 board. Depth 5 could not finish.

As seen in Figure 4, our vanilla alpha-beta searcher could not get very far on its own. The branching factor for Connect 5 is very large, since any empty point on the board is a legal move. Our algorithm was not capable of searching even the first few test positions we had to a depth of 5. Moreover, even depth 5 would be barely enough to detect even simple forced wins that would be immediately obvious to a human, such as that in Figure 3 earlier, which is already a depth 5 win. This is far from being a strong player.

We attempted to use a variety of heuristics and additional methods to improve the performance of our base searcher.

3.2 Evaluation

Since our alpha-beta search was depth-limited, we needed a heuristic for evaluating the board at the once the search reached the maximum depth without reaching in a winning, losing, or drawn position.

The heuristics we tried included:

- Random - return a random value. This was included as a baseline.

- Simple (Line based) - for each unblocked line of n consecutive pieces of the current player, add 4^n points, and subtract 4^n points for each such line of the opponent. Encourages creating and blocking of threats.
- SimpleDef (Asymmetric line based) - same as the above line-based heuristic, but subtracts double the points for the opponent's threats. Introduced when we noticed SimpleEval often ignoring opponent's threats to extend its own, causing the opponent to win, due to horizon effects of low search depth.
- EmptyNeighborsDown (Neighbor based) - for each pair of adjacent player pieces, add 1 point, for each pair of adjacent opponent pieces, subtract one point. The idea is to value clusters of pieces more to generate more threats.
- EmptyNeighborsUp (Neighbor based) - the reverse of the previous function. The idea is to spread pieces out more.
- Neighbors (Neighbor based) - for each empty position adjacent to a player piece add 1 point, and subtract 1 point for each adjacent to a opponent. Encourages a game style that spreads pieces around the board. This might help generate more possible combinations of threats.

3.2.1 Results

	Rand	Simple	SimpleDef	EmpNeiD	EmpNeiU	Nei
Random	-	11-84-5	6-86-8	46-52-2	42-57-1	71-31-8
Simple	84-11-5	-	37-36-17	99-0-1	83-17-0	99-1-0
SimpleDef	86-6-8	36-37-17	-	95-1-4	85-15-0	96-0-4
EmpNeiD	52-46-2	0-99-1	1-95-4	-	35-65-0	38-61-1
EmpNeiU	57-42-1	17-83-0	15-85-0	65-35-0	-	75-25-0
Nei	31-71-8	1-99-0	0-96-4	51-38-1	25-75-0	-

Figure 5: Tournament between agents with different evaluation heuristics, results reported as Wins-Losses-Ties from the perspective of the agent listed on the left.

We ran a tournament involving between different versions of our alpha-beta searcher using each of these different heuristics.

As shown in Figure 5, of the six heuristics we tried, the line-based heuristics, Simple and SimpleDef, performed the best. This makes sense, as these are perhaps the most "natural" heuristics for the game, counting points for building lines of pieces, and encouraging the agent to block lines of its opponent's, since these actions most directly contribute to the goal of the game - to build a line of 5 pieces.

It was also interesting to note the poor performance of the neighbor based heuristics. Initially, we had some intuition that clustering of pieces should have at least some correlation with favorable and unfavorable positions in the game, since clusters of pieces tend to generate more coordinating threat possibilities, while it was also suggested that perhaps more scattered pieces would be better at defense and also making more distant threat possibilities. Although we never expected any of these heuristics to perform particularly well, it surprising to see that two of them were statistically no better than random when matched against it! Moreover, the third heuristic, Neighbors, in fact performed worse than Random. The reason becomes clear in watching games using this heuristic, while okay at defending, an agent using Neighbors is reluctant to place its pieces adjacent to one another, preventing it from winning.

It is also interesting to note that the two line-based agents performed equally well, while also having the highest number of draws of any match in the tournament. It is unclear whether this is due to the defensive nature of the SimpleDef heuristic against an opponent that will not simply lose most of the time, or whether it is due to the fact that as play in Connect 5 improves, draws becomes slightly more common. We did not have time to run tests to investigate further on this point, however.

Overall, we found that the simple and natural heuristics for the game gave the best results.

3.3 Move Ordering Heuristics

We next examined a number of ordering heuristics, to try to increase the effectiveness of alpha-beta pruning by searching better moves first. By searching these moves first, we can obtain more optimal running evaluations in our search, resuling in tighter alpha-beta windows, enabling us to perform more cutoffs in the tree.

The heuristics we tried included:

- SimpleOrder (line-based) - add 2^n points for playing in a line with n pieces in a row of either player. This encourages blocking and forming threats.
- NeighborOrder (neighbor-based) - add a point for each adjacent opponent piece and subtract a point for each adjacent player piece. Encourages positions with high Neighbors evaluation heuristic.
- HistoryOrder (dynamic) - Whenever a search for a given node finishes, mark the best move in a large table that is shared across branches of the search. Order moves by how frequently they were the best moves in other branches of the search. Taken from [4].

3.3.1 Results

	Avg Nodes	Avg Time
No Ordering (Simple eval)	5508303	6.702
SimpleOrder (Simple eval)	228992	0.382
NeighborOrder (Simple eval)	1262594	1.631
HistoryOrder (Simple eval)	245332	0.256
SimpleOrder (Neighbors eval)	1386467	4.326
NeighborOrder (Neighbors eval)	675191	2.263
HistoryOrder (Neighbors eval)	636534	1.551

Figure 6: Average peformance on 100 test positions on a 13x13 board at a search depth of 5.

The results for the various move ordering heuristics were quite interesting. Whereas the line-based heuristic and the neighbor-based heuristic performed very well when used together with their corresponding evaluation functions, they performed extremely poorly when paired with the other evaluation function. This makes intuitive sense, as each ordering heuristic to favors moves that are more likely to lead to positions that are highly-evaluated by their corresponding evaluation heuristics. This might have been expected, but it is still interesting to see the sharp contrast exhibited here in practice.

Surprisingly, the history heuristic outperformed them both, despite being a general heuristic that does not even use any domain knowledge about Connect 5! This suggests that the ability for a heuristic to take into account on-line information as it is discovered by the search is very powerful. The history heuristic alone was able to dynamically adjust it's valuation of the moves according to the actual data discovered by the search, and the correlation between good and bad moves across different branches of the search tree in Connect 5 is apparently strong enough to make this a very potent ordering heuristic, allowing the depth 5 searches to run more than *twenty times faster* on average! Moreover, because of its generality, it can be used regardless of which evaluation heuristic we choose to use.

It is also interesting to note that the SimpleOrder heuristic outperforms HistoryOrder in terms of the number of nodes generated, yet still loses. This is mostly likely because the history heuristic is far cheaper to compute, requiring only a table lookup, while SimpleOrder, in order to score points for pieces in lines, needs to iterate over the lines of the board to determine the counts of pieces in a row. This close balance is not an artifact of low-depth searches. As seen in Figure 7, the same pattern holds at depth 7, where SimpleOrder searches fewer nodes but takes longer than HistoryOrder.

	Avg Nodes	Avg Time
SimpleOrder	6771025	12.498
HistoryOrder	8304484	9.103

Figure 7: Average peformance on 100 test positions on a 13x13 board at a search depth of 7.

The overall difference between using an ordering heuristic and not using one is quite large, worth almost two additional steps of search, which is a massive improvement considering the exponential branching factor. This demonstrates that alpha-beta search depends very heavily on a good ordering to make full use of beta cutoffs.

Nonetheless, we found that ordering alone was not enough. Even with the history heuristic, the agent was unable to search too much deeper, due to the astronomical branching factor (169 for 13x13, 361 for 19x19). While we expected that the threat searcher that we would be implementing might compensate for the deficiencies in the particular lines that lead to immediate wins or losses, it would not help as much in more general tactics that didn't immediately force a win or loss, and we felt that it was possible to do better in general. Therefore, we turned to a set of inadmissible heuristics aimed at generating fewer moves, to prune the search space.

3.4 Move Generation Heuristics

We observed that essentially always, our agent would play only moves next to existing moves. In practice, we never observed an agent making a move further than radius 2, which makes sense, considering that distant moves would be unhelpful for extending existing threats or blocking those of one's opponent. We therefore considered the following pruning methods:

- Radius2 - Generate only moves within radius 2 of existing moves (that is, a maximum of 2 spaces away in horizontal and/or vertical distance). In practice, we *never* observed any agent making a move further than radius 2 from existing moves, so we feld this would be a good and relatvely safe pruning heuristic.
- Line2- Generate only moves within radius 2 and in line with existing moves. This is the same as Radius2, but does not include knight's move relationships.

3.4.1 Results

	Avg Nodes	Avg Time
AllLegal	8304484	9.103
Radius2	3691037	6.799
Line2	2737824	6.069

Figure 8: Average peformance on 100 test positions on a 13x13 board at a search depth of 7.

	All	Radius2	Line2
All	-	44-44-12	45-44-11
Radius2	44-44-12	-	42-46-12
Line2	44-45-11	46-42-12	-

Figure 9: Tournament between agents with different move generation methods, results reported as Wins-Losses-Ties from the perspective of the agent listed on the left.

We were surprised to discover that such pruning had very little effect on the performance of the agent. One would think that such a pruning would allow a drastic reduction in the search space, even on a board as small as 13x13, especially since the reduction would occur at *every level* of the search tree. Yet, it only gave a speedup of 1.5 times at depth 7, and in a tournament between the three agents, there was almost no statistical difference between using radius-based pruning or simply searching all moves.

We suspect that this lack of improvement is *not* because good moves are being lost by this type of pruning - indeed, this pruning almost never changed the actual move that an agent chose to make. Tentatively, we suspect that there are several effects that are combining here to produce the unimpressive results. Firstly, we note that often, many possible moves that can be made in a position are at a radius of 2 from existing pieces, and if even one such move is made at any point in a branch of the search, all moves that are radius 2 from *that* move will now suddenly be considered. This allows the search to very quickly include distant areas of the board, despite the pruning. Secondly, we suspect that the board is relatively too small to make radius pruning completely effective perhaps a board larger than 13x13 will show a sharper effect. These would be interesting directions for further investigation.

4 Algorithm 2 - Threat-Space Search

The second major algorithm that we implemented was that a directed threat searcher.

In practice, nearly all the wins and losses in games between experienced Connect 5 players occur when one player finds a sequence of threats, forcing the opponent's move at each step, ending in the formation of two simultaneous threats, which assures a win, since the opponent can only prevent one of them at a time. These winning sequences of threats can be 10 or 20 moves, or even longer, and finding and preventing them is a key component of high-level play. Yet, Due to the exponential branching factor of the game, it is impossible to detect these sequences with a normal, full-width alpha-beta search, since we cannot search deep enough. As we've seen, the search quickly becomes impractical beyond depth 7.

However, it is quite possible that a more directed search could detect such sequences, since within any threat sequence, the branching factor is very low, since only a few moves will typically generate threats and the opponent's moves are forced at each step. This motivated us to examine a special, threat-based search algorithm that considers only threats and responses to threats.

4.1 Definitions

Our algorithm attempts to prove a win for one player at a time. That player is the *attacker*, and that player's opponent is called the the *defender*. A *threat* for a player is any square on the board where that player could win immediately by playing there, by creating five in a row. A *forcing move* is any move of the attacker that creates a open line of four pieces and thereby creates a threat. This forces the defender to respond by playing on the square of the threat or else immediately lose. We note that if any forcing move generates two simultaneous threats, then the attacker wins, since the defender cannot prevent both. A *forcing sequence* is any sequence of one or more forcing moves, along with the defender's forced responses.

4.2 Our Threat-Search Algorithm

At each step, the algorithm maintans and adds to a list of forcing sequences. The list is initially empty.

The algorithm proceeds as follows:

If the attacker has an existing threat already on the board, then we halt and report a win for the attacker, since the attacker can simply play there to win. If the defender has an existing theat on the board, then we halt and conclude nothing.

Othrewise, we first find all forcing moves for the attacker, adding them to the list of forcing sequences.

Then, then we "chain" from all forcing sequences found so far. That is, for each forcing sequence

F found so far, we play all moves of F and all forced responses by the defender. If these responses do not cause the defender to have a winning line of five pieces or to have a threat, then we find all forcing moves m that depend on the last move of F, and for each such move, we add the new forcing sequence $F + m$ to the list.

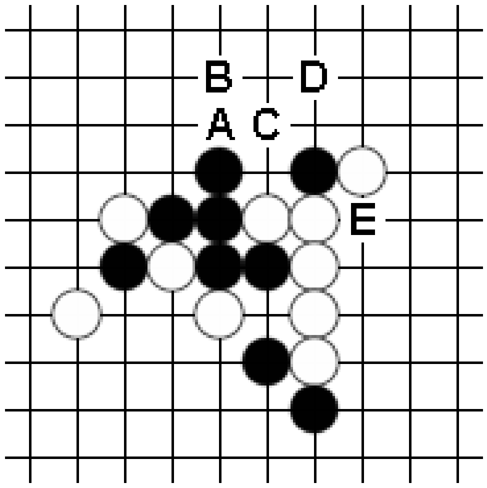
Additionally, we attempt to "merge" forcing sequences found so far. That is, for each forcing sequence F and G, if the last move of F and the last move of G line on the same row, column, or diagonal and are close enough together to potentially enable new forcing moves, and if the squares involved in the moves of each sequence are disjoint, and if combining the defender's responses would not cause the defender to have a threat or a winning line of five pieces, then we add the forcing sequence $F + G$ to the list.

We iteratively merge and chain from forcing sequences in the list to produce more sequences. If at any point, we find a sequence that generates two simultaneous threats, then we halt and report a win for the attacker.

Otherwise, we halt when no new forcing sequences can be found, or when the total number of forcing sequences exceeds some constant limit.

4.3 Example

We show a specific case of how a threat search would proceed with the following example:



In this position, since neither player has any existing threats, our algorithm would first generate all possible forcing moves for Black, paired along with the forced responses by White. There are four of them, $\{(A, B), (B, A), (C, D), (D, C)\}$.

Then, all possible forcing moves that could be chained from each of these are considered. However, no such moves are possible. Combinations such as that of black A, followed by white B, followed by black C, are not considered to be chains, since black's forcing move of C does not depend on A.

However, such combinations are explored during merging. During merging, independent forcing sequences that result in black gaining pieces in line with one another are considered. For instance, the forcing moves at A and C both share the same row, so they are combined. Three combinations are added, AC, BC, and BD, giving a list:

$\{(A, B), (B, A), (C, D), (D, C), (AC, BD), (BC, AD), (BD, AC)\}$, where (AC, BD) indicates that black plays at A and then C, with white responding at B and then D.

Then, during the next chaining step, a new forcing move is possible chaining from (BC, AD). Black can play at E to create four in a row. Since this move generates two simultaneous threats, Black wins.

Separating the chaining and merging steps serves to limit the combinatorial possibilities by only considering combinations of independent threats that could possibly produce other threats, rather than considering all possible combinations.

4.4 Integrating the Threat Searcher

While the threat searcher is highly effective in detecting certain forced wins, it cannot take the place of a main alpha-beta search for choosing moves in general positions where there may be no immediate winning forcing sequences. Therefore, we use it as a subroutine in the alpha-beta search. At each node in the search, we apply the threat searcher, and if the threat-searcher proves a win, then we can terminate that branch of the alpha-beta search immediately, and otherwise, we proceed to expand the tree according to alpha-beta as usual.

4.5 Results

	NoTS	TS10	TS50	TS100	TS150
NoTS	-	36-64-0	26-72-2	25-70-5	25-72-3
TS10	64-36-0	-	38-54-9	28-60-12	20-68-12
TS50	72-26-2	54-38-9	-	46-45-9	41-50-9
TS100	70-25-5	60-28-12	45-46-9	-	39-50-11
TS150	72-25-3	68-20-12	50-41-9	50-39-11	-

Figure 10: Tournament between agents with different threat-search settings, results reported as Wins-Losses-Ties from the perspective of the agent listed on the left. NoTS - no threat search. TS(X) - threat search with a maximum of X forcing sequences before quitting

To test the effectiveness of our threat searcher in combination with alpha-beta search, we ran a tournament between versions of our agent with different caps on the number of forcing sequences allowed for the threat searcher and also without the threat searcher, with the results in Figure 10.

The threat-searcher proved to be quite effective. In a direct match against the version with no threat-searcher, the versions of the agent using the threat-searcher won far more games than they lost, approaching a 3 to 1 winning ratio with a cap of 150 forcing sequences.

Moreover, increasing the cap on the number of forcing sequences allowed seems to be almost uniformly beneficial - searchers with a higher cap almost always tend to beat those limited to searching fewer forcing sequences, despite taking somewhat longer to run per node in the alpha-beta search. It would be interesting to see how far this trend continues. We initially set the cap due to problems in some positions where a cluster of related forcing moves would generate combinatorial explosions of possible foricng sequences, causing very long runtimes. Yet, it seems that at least up until a limit of 150, searching more forcing sequences if they are possible in a position, at the expense of the alpha-beta search, is generally better. This makes sense, as the threat-search is far more directed in exploring the areas of the search tree that are critical to the result of the game, since it directly deals with narrow lines leading to forced wins and losses.

Our threat-searcher is actually somewhat more limited than the one described by L.V. Allis in [2], due to the fact that it does not generally search “threes”. As shown earlier in Figure 2, a line of 3 pieces open on both ends will very often be forcing, because it can be converted to a double threat in one more move. However, handling threes is quite a bit more challenging, since the opponent has several possible responses, and because it is a delayed threat - the opponent can also interject his own forcing moves by creating lines of 4 before responding. It would be interesting to compare our results with that of implementing a more complete multi-layered dependency search capable of handling delayed threats.

Nonetheless, our threat searcher in its current form has already proven to be effective and a useful component for our agent. In informal testing and play, with the threat searcher enabled, our agent is capable of detecting wins and losses well beyond its alpha-beta search depth, even often finding wins that are five or even seven moves beyond the maximum depth reached by the alpha-beta search, and is significantly better at handling the tactics that alpha-beta alone could not detect.

5 Notes on Testing

In testing our algorithms and heuristics in the above sections, we ran two types of tests.

In the first type, we ran various agents with different heuristics on a suite of 100 test positions to compare the running speed of each version. These tests were shownas simple comparisons between time and number of nodes, and were used when the heuristics being tested only affected the speed of the agent, rather than the moves that it produced.

In the second type of test, when the properties being tested did affect the moves produced, we played various agents in tournaments against one another, beginning from a set of 50 different starting board positions. Initially, we tried playing agents from the starting, empty position, but we found that despite some amount of randomization, our agents were generally too deterministic, and would play the same game over and over with only minor variations. This made it hard to tell whether a change was beneficial, or whether it merely caused the agent just by chance to play a different deterministic variation that caused it to win. As a result, we varied the initial position to obtain better statistics. To avoid any first or second player biases, we played two games from each position, one with each agent playing first, for a total of 100 games in each matchup. Games were played at a time control of 0.1 seconds per move.

All test positions were generated by playing a depth 1 searcher against itself with heavy randomization hundreds of times, and then choosing random positions from these games, discarding any positions that had immediate, short forced wins. For the tournaments, positions were chosen that were within the first 20 moves of the random games played.

6 External Testing

We were eager to test our AI against others online, only to find that, surprisingly, there are relatively few freely available programs online that play Connect 5 well. The best player that we managed to find online is "Stahlfaust - a Gomoku AI", by Marco Kunze. Stahlfaust takes a similar approach as our agent, using a combination of alpha-beta search and threat-based search.

Using the strongest settings for our agent, we manually ran a total of eight games between our agent and Kunze's on 15 by 15 boards. The first four games were played allowing each program a maximum of two seconds of computation time, alternating the first player. All four games were won by our agent quickly and decisively, in less than 25 moves, with a clear tactical advantage within the first 10 to 15 moves. During the second four runs, both agents were allowed 8 seconds of computational time. Of those four runs, our agent won three and lost one of the games in which it played second.

Although there are undoubtedly much stronger agents available (especially, since Connect 5 has been solved on 15x15!), this nonetheless suggests that our agent is at least competitive with others. Moreover, playing against it personally, we find that our agent is very difficult to beat. Even going first, we can only win occasionally, and only when our agent is set to play very quickly and we play very slowly and carefully.

Our final searcher currently uses a threat-space search with a sequence limit of 150, both of the simple line-based evaluation and ordering heuristics that we analyzed earlier, and despite the relatively neutral results, the "Line2" move generation heuristic.

7 Conclusion

We believe we have succeded in our goal of writing a strong connect 5 player. By employing simple heuristics and threat-space search, we have produced an agent tremendously better than the one we started, to the point where it usually catches *us* in tactical mistakes, as opposed to the other way around. While we are by no means experts in the game, our agent has come a long way in a short time, and along the way, we have uncovered a number of promising results for further investigation. In the process, we have learned a great deal about tactical and game-tree search.

As a minor goal, wherever possible, we designed our game representation and our agent to support more generalized connection games, and in particular, Connect 6. In Connect 6, at least six pieces in a row, column, or diagonal are required to win, and players play two moves per turn, except that the first player plays only one move on the first turn, to eliminate the first-player advantage. Connect 6 is an recent and interesting variant that, unlike Connect 5, appears to be well-balanced with no first-player advantage. And whereas Connect 5 has been solved, Connect 6 is likely far from a solution or even from expert computer play, due to a drastically higher branching factor. In this respect, it poses an interesting challenge for modern AI [5].

Unfortnately, our agent plays Connect 6 quite poorly, failing to see "obvious" threats only two or three turns deep, and it is clear that many of the heuristics and methods we have used for Connect 5 are insufficient alone to handle Connect 6. Extending some of the techniques we have explored here and finding new methods to handle this new game would be a particularly fascinating future direction of investigation.

References

[1] Allis, L. V. Van den Herik, H. J. Huntjens, M. P. H., *Go-Moku Solved by New Search Techniques.* COMPUTATIONAL INTELLIGENCE -OTTAWA-, 1996: VOL 12; NUMBER 1, pages 7-23.

[2] L.V. Allis, H.J. van den Herik, J. van den Herik M.P.H. Huntjens. *Go-Moku and Threat-Space Search.*

[3] Mller, Martin. *Global and local game tree search.* 2001.

[4] Schaeffer, Jonathan. *The History Heuristic and Alpha-Beta Search Enhancements in Practice.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1989, vol 11.

[5] Wu, I-Chen and Huang. *A New Family of k-in-a-row Games.* ICGA Journal (SCI), Vol. 28, No. 4, pp. 234-241, December 2005.