# Monte-Carlo Tree Search for Poly-Y

Lesley Wevers  Steven te Brinke

*University of Twente, The Netherlands*

**Abstract**

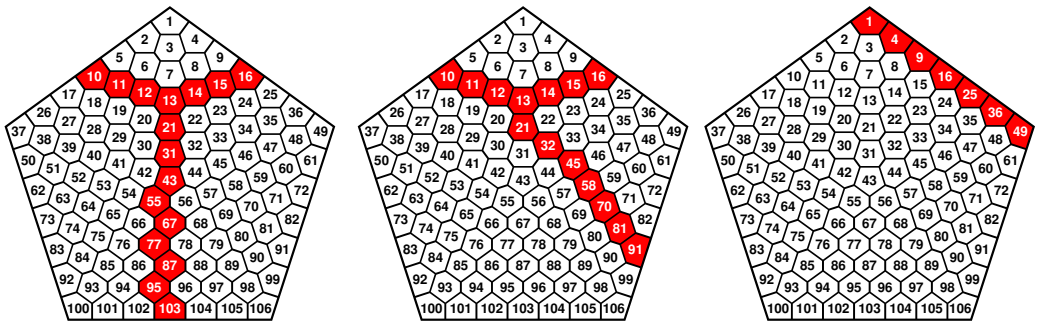
Monte-Carlo tree search (MCTS) is a heuristic search algorithm that has recently been very successful in the games of Go and Hex. In this paper, we describe an MCTS player for the game of Poly-Y, which is a connection game similar to Hex. Our player won the CodeCup 2014 AI programming competition. Our player uses MCTS with the all-moves-as-first heuristic, and detects basic heuristic patterns to defend virtual connections in Poly-Y. In the CodeCup, we can only use 30 s single-core computation time per game, whereas in Hex, 5 to 30 minutes of multi-core computation time is common. We improve the performance of our player in the early game with an opening book computed through self play. To assess the performance of our heuristics, we have performed a number of experiments.

## 1 Introduction

We created a program that plays Poly-Y, which is a generalization of the game Y, which in turn is a generalization of Hex. Our program won the CodeCup[1] 2014, which is an annual international AI board game programming competition organized by the Dutch Olympiad in Informatics. In contrast to other AI programming competitions, the CodeCup has major restrictions in computing resources: Programs only get 30 seconds of total playing time, 64 megabytes of memory, and cannot use multi-threading. This requires programs to use lightweight techniques for gameplay. The source code of our program (Lynx) is freely available[2], and it is possible to play against our program online[3].

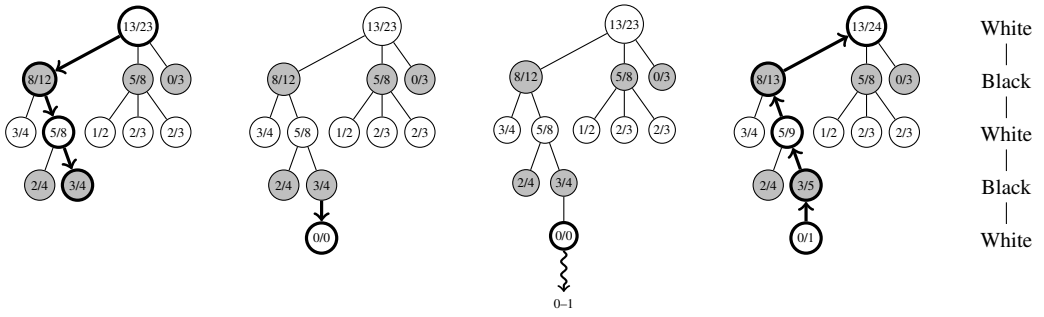
**Poly-Y** In contrast to Y, which is played on a three-sided board, a Poly-Y board can have any odd number of sides greater or equal to three. An $(n, r)$-board consists of an $n$-sided center cell, with $r$ rings of cells around it. For the CodeCup, we played on a $(5, 7)$-board, which has 106 cells (see Figure 1).

[1] CodeCup website: `http://www.codecup.nl/`
[2] Lynx source code: `https://github.com/lwevers/lynx/`
[3] Online playable version of our player: `https://maksverver.github.io/lynx/` (thanks to Maks Verver)



**(a)** Captures one corner  **(b)** Captures two corners  **(c)** Captures two corners

**Figure 1:** Y-structures that capture (a) the top corner or (b), (c) both top and right corners.




**(a)** Selection **(b)** Expansion **(c)** Playout **(d)** Backpropagation Player

**Figure 2:** Monte-Carlo Tree Search. Nodes are labelled with the number of wins/playouts for White.

Poly-Y is played between two players, which we call *White* and *Black*. Players alternatingly place stones in empty cells, starting with White. The goal of the game is to capture the majority of the corners. A corner is captured by creating a Y-structure (examples are shown in Figure 1), which is a connected group of stones that (1) connects two adjacent edges, and (2) connects to any other edge on the board. The second rule ensures that once a corner has been captured by some player, it remains captured by that player. Since a full board has all corners captured and the board has an odd number of corners, no game can end in a draw.

As is the case in Hex, in practice, White has a strong first-player advantage. To mitigate this, there is a *swap rule* stating that Black may choose to swap sides as its first move. That is, the first move of White becomes the first move of Black. The *swap rule* results in more balanced games in practice: White will neither open with a very strong move, since then Black will swap, nor a very weak move, which allows Black to win easily.

Our player is based on the Monte-Carlo tree search (MCTS) algorithm [8], which has already been very successful in Hex and Go. We leverage work on MCTS Hex players to construct an MCTS player for Poly-Y, for which we describe the details in section 2. MCTS uses random playouts in self play to evaluate the strength of game states. Guiding these random playouts toward more realistic games can improve the quality of an MCTS player. In section 3, we discuss how we do random playouts, and we discuss a number of heuristics for Poly-Y to guide these playouts. In Hex it has been found that MCTS is weak in the very early stages of the game. To improve the performance of our player at these early stages, we have constructed an opening book, as we discuss in section 4. Finally, in section 5 we show the results of experiments performed on our Poly-Y player to show the effectiveness of our techniques.

# 2 Leveraging MCTS in Hex to Poly-Y

Others have already explored MCTS and strategies for Hex [2]. We leverage these strategies and adapt them to a Poly-Y player. Compared to the Hex players, which usually have 5 to 30 minutes of thinking time per game, our player only has 30 seconds of computation time. With this in mind, we have chosen to only use lightweight techniques.

**Pure Monte-Carlo Search** Traditional minimax search does not lead to strong play in the game of Hex, because of the high branching factor, and the difficulty of evaluating the strength of game states. Monte-Carlo search is an alternative method for evaluating the strength of game states. The idea of pure Monte-Carlo search is to compare the relative strength of moves in self play by measuring the average outcome of a number of random playouts from the states corresponding to these moves.

**Monte-Carlo Tree Search** As the basis for our player, we use the Monte-Carlo tree search algorithm (MCTS) [8], which has already been very successful in the game of Hex. The idea of MCTS is to combine pure Monte-Carlo search with a game tree similar to minimax. Like in minimax, in MCTS a game tree is constructed where nodes correspond to game states, and edges correspond to moves. Each node stores the total number of playouts performed from this node, and the number of winning playouts.

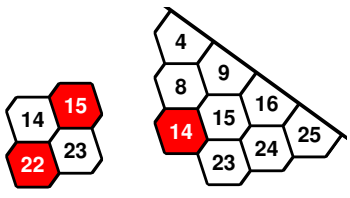
The MCTS algorithm works in four steps. The first step (Figure 2a) selects a promising node to be explored. Selection recursively selects the move with the highest *score*, which is computed based on statistics kept in the node. One challenge in the MCTS algorithm is to find a good scoring function that balances between exploration of moves that have few samples, and exploitation of the best moves found so far. This problem is an instance of the multi-armed bandit problem [3]. A common solution is to use the UCT (Upper Confidence Bound 1 applied to trees) algorithm [12] to score nodes. However, we use the all-moves-as-first heuristic to score nodes instead, as discussed in the next paragraph. After selecting a move to be explored for which no node exists, the tree is expanded (Figure 2b) by creating a leaf node, from which a Monte-Carlo playout (Figure 2c) is performed. To reduce the overhead of MCTS, our player performs 32 playouts when creating a new node. Finally, the result of the playout is propagated back up to the root (Figure 2d), while the statistics of all nodes on the way are updated.

**All Moves as First** If we consider a random playout, we alternatingly fill the board with random moves for both players. The order of these moves does not matter for the end results. The idea of the all-moves-as-first (AMAF) heuristic [5] is that any of these moves could have been the first move, and we can use this observation to gather information about many moves from a single playout. In effect, AMAF increases the number of playouts that we can perform by the number of empty cells on the board.

We now briefly review the AMAF heuristic. We say that a playout is made *from node* $n$ if the playout was made in $n$ or in any of its descendants. The *move that corresponds to node* $n$ is the move performed to reach $n$ from its parent. In the AMAF algorithm, each node $n$ in the tree maintains two separate (wins, samples)-pairs: a direct pair $(d_w, d_s)$ and an indirect pair $(i_w, i_s)$. The *direct* pair maintains the cumulative wins $d_w$ and samples $d_s$ of all playouts made from $n$. The *indirect* pair maintains the cumulative wins $i_w$ and samples $i_s$ of all playouts made from any *sibling* of $n$ where the move corresponding to $n$ was played in the playout by the player who's turn it is. In the MCTS selection phase, we compute the *direct win rate* of a node as $w_d = d_w/d_s$, and the *indirect win rate* as $w_i = i_w/i_s$. Note that the indirect win rate $w_i$ is based on playouts where the move was played in a different context, and is less reliable than the direct win rate $w_d$. We compute the score of a node using alpha-AMAF [10] as $w = \alpha \times w_d + (1 - \alpha) \times w_i$. For our player, we empirically found that $\alpha = 0.75$ produces the best results. Alternatively, the RAVE heuristic can be used, which increases $\alpha$ as more direct samples become available. However, we found that RAVE is not beneficial with the low number of playouts that we can produce, which corresponds to the findings of Helmbold and Parker-Wood [10].

One benefit of AMAF is that we can quickly explore moves. Like in MCTS Hex [2], we found no need for UCT exploration anymore, instead relying completely on AMAF for the exploration. Moreover, we found that AMAF quickly removes inferior moves.

**Heuristics** In Hex, patterns have been discovered that simplify the analysis of the game. One of these patterns is a bridge, as shown in Figure 3a. No matter what the opponent does, we can always create a connection between cells 15 and 22. If the opponent plays cell 14, we play cell 23 to create a connection, and vice versa. Cells 15 and 22 are said to be *virtually connected*. Another example of a virtual connection is shown in Figure 3b, where cell 14 is virtually connected to the edge of the board. These virtual connections also apply to Poly-Y.




**(a)** Bridge **(b)** Level-2 connection

**Figure 3:** Patterns that guarantee a connection (a) between stones and (b) to side.

In MCTS Hex players, virtual connections are used in two ways. First, virtual-connection search can be used to solve the game board [9]. However, this search is PSPACE-complete [11], and is therefore not feasible in limited time. H-search is a weaker form of virtual-connection search [1], and has been successfully applied in analyzing game positions for MCTS Hex [2]. However, H-search is still computationally expensive, which makes it infeasible for our player, because we have only 30 seconds per game. Second, virtual connections can be used in an even weaker form, by recognizing basic patterns on the board, and applying fixed responses. This technique is used in MCTS Hex to increase the quality of playouts by defending basic virtual connections [2]. We also use patterns to increase the quality of playouts in Poly-Y, as discussed in section 3.

In Hex, a board can be analyzed for inferior and dominating moves, which are provably irrelevant for the remainder of the game [2]. This technique can also be used for Poly-Y. For instance, playing

at the edge is generally weak, unless a (virtual) connection to the edge is made by this move, or when preventing the opponent from making a (virtual) connection. For our player, we found that the AMAF heuristic is already very effective at pruning weak moves, so we do not perform explicit inferior and dominating move analysis.

**Win Condition** For our win check, we use a flood-fill algorithm to find connected edges and captured corners. An alternative method is bitwise-parallel reduction (BPR). For the game of Y, BPR can speed up the win check by a factor of up to 15 compared to flood fill (Path) [4]. For Hex, this speedup is lower, because a Hex board has to be encoded into a larger Y board. Similar to Hex, we can encode a Poly-Y $(5, r)$-board into a Y board of height $3 \times r$. However, BPR only checks for a single Y-structure, therefore our encoding can only detect Y-structures's between two edges around a specific corner, so the reduction must be applied to five rotations of the board. Together with the encoding overhead, the expected performance of BPR is similar to flood fill. Thus, BPR is not helpful for Poly-Y and the simpler flood fill is the algorithm of choice.

# 3 Monte-Carlo Playouts for Poly-Y

The effectiveness of the Monte-Carlo method depends both on the quantity and the quality of playouts. Completing games in a purely random fashion does not correspond to the way players react in reality. For example, if a vital virtual connection is attacked, a rational player will defend this connection. This problem has already been observed in MCTS Hex [2], where heuristics are used to guide the Monte-Carlo search towards more realistic games. The idea of these heuristics is to maintain virtual connections. However, seemingly good heuristics do not necessarily improve gameplay, as shown in MCTS Hex players [2]. In this section, we describe the heuristics that we use for Poly-Y.

**Defending Bridges** As in MCTS Hex, we try to defend any bridge that is attacked by the opponent during a playout. In practice, a bridge may not be required for a win, and defending such a bridge wastes a move. We do not check for this in the playout, as this is too expensive. Instead, we rely on MCTS to avoid choosing such moves, since the tree search also considers moves that do not defend the bridge.

**Higher-level Virtual Connections** In MCTS Hex, patterns are also used for level-2 (Figure 3b) and level-3 virtual connections [2]. However, these patterns are quite large, and therefore slow to apply. Instead, we only check if the opponent tries to block us from reaching an edge, and avoid this by 'walking around the opponent' towards the edge. This pattern is computationally cheap to check, but is not exact in applying the level-2 and level-3 patterns. However, we found that for our player this simple pattern produces stronger play than more accurate level-2 and level-3 virtual connection patterns.

**Fillboard: Avoid Playing at the Edges** In purely random games, the order of moves does not matter. However, in the presence of heuristics, the order of moves affects the applicability of the heuristics. The fillboard heuristic is used in the MCTS Go program MoGo [6]. The idea of this heuristic is to distribute moves evenly over the board early in the game, resulting in a structure that can be exploited by other heuristics. We use a similar heuristic: not playing at the edges of the board early in the playout. The idea of this heuristic is that cells at the edges of the board are mostly filled in to defend virtual connections, and are usually bad moves if not part of a virtual connection. In our implementation, no randomly selected move will be at the edge of the board within the first 50 moves of the playout.

**Implementation** Performing more Monte-Carlo playouts leads to a stronger player, so our implementation should be as fast as possible. Therefore, we need to be able to apply the heuristics quickly. We only check for heuristic patterns around the last opponent move, and see if we need to respond to this move. To do this we compile our heuristics into a list of templates for each board position, which we can check in constant time each. We represent the board as two bit arrays, where each array corresponds to the stones of one of the players. A pattern consists of bitmasks that describe the state of the board, together with a move to play. Using bitwise operations, the bitmasks can be checked in constant time against the game state to see if the pattern matches. If multiple patterns match, we randomly pick one of the matching patterns. If no pattern matches, we pick a random move.

# 4 Opening Analysis

We found that the outcome of a game is usually determined in the first few moves of a game. For example, we have seen games where after 10 moves the game was already determined. However, Monte-Carlo tree search does not perform very well in the beginning of the game. To improve the performance of our player early in the game, we have constructed an opening book for a Poly-Y $(5, 7)$-board.

**Methodology** Computing a perfect opening book is computationally infeasible. Instead, we compute an opening book by evaluating the strength of game states after an opening sequence by measuring the win / loss ratio from this state in self play. Assuming that our opponents also use a variant of MCTS, we expect that the measured win rates are a good representative of our actual chance of winning. When we evaluate states that are deeper in the game tree, we also get more information about the strength of earlier moves. However, the number of opening sequences increases exponentially in the length of the opening sequence. Furthermore, it is computationally expensive to evaluate the strength of a game state. In our opening analysis, we use 5 seconds per player for the self plays, for a total of 10 seconds per game. We perform 256 self plays per state, which means that evaluating a single opening sequence takes 42.7 minutes of single-core computation time. Still, the variation in measurements is high, as the error decreases only by the square root in the number of self plays. Because the board is symmetrical, we only have to compute opening moves for one symmetry, so there are only 16 opening moves to consider. It takes 11.4 hours of single-core computation time to evaluate these 16 opening moves. If we want to look deeper, we have to consider all 105 responses the opponent can make. For this level-2 analysis, there are 1680 cases to consider, for a total of 49.8 days of single-core computation time. A level-3 analysis would require analyzing 174,720 states, for a total of 14.2 years of single core computation time. This shows that computing a deep opening book is infeasible. An optimization is to only investigate the most promising moves. For instance, we could stop analyzing a branch when our win rate in self play is higher than some threshold. Alternatively, a method such as UCT can be used to perform fewer playouts for game states that are less promising, and instead focus on the more promising game states [7]. An additional optimization is to prune states that cannot become better than the best state found so far with more samples, e.g., if it is outside the 95% error bounds of the best move. In our analysis we do not use these optimizations methodically, but we hand picked the best moves for further exploration.

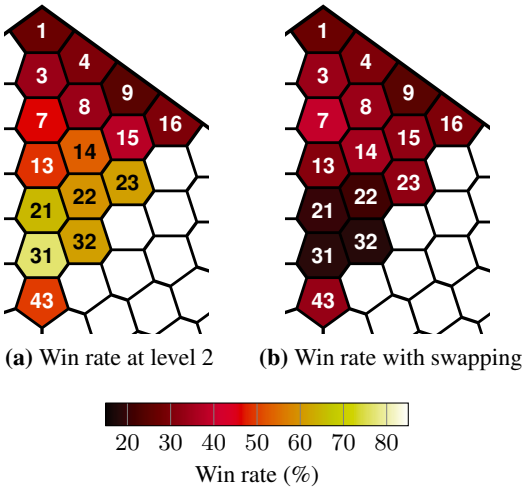
**Results** We have performed our analysis on a 64-core ($4\times16$-core 2.0 GHz) AMD machine by running 64 independent games in parallel, which allows us to perform the level-2 analysis in 18.7 hours. After performing the level-2 analysis, we decided on our opening move, and we decided for which opponent opening moves we would use the swap rule. For the moves where we use the swap rule, we computed the best level-3 responses for the top level-2 moves, which we can play if the opponent plays one of these top level-2 moves. For the moves where we do not swap, we did not perform any further computation. In case the opponent chooses to swap on our opening move, we can play the best level-2 move. Additionally, for the top level-3 moves we have also computed the best level-4 response, which we can use if we are Black. In total, we used two weeks of computation time to compute our opening book. The resulting opening book can be downloaded from our source repository[4]. Table 1 shows the win rate for White after analyzing the opening move (level 1), after analyzing the best response from Black (level 2), and after analyzing the best response for White (level 3). Figure 4a graphically depicts the win rates for White after the first move of Black, assuming Black does not swap. Figure 4b depicts the win rates for White either after the second move of Black when his first move was swap, or after the first move of Black otherwise.

The win rates we found varied wildly when analyzing the positions deeper. For instance, move 15 has a win rate of 49.6% at level 1, while at level 2 the win rate drops to 37.9%, and at level 3 it is back up to 67.6%. However, note that due to the way the opening book is constructed, these win rates do correspond to actual playing strength in self play against a version of our player without an opening book. To select a good opening move, we also have to consider the swap rule. Ideally, we want a move that leads to a 50% win rate at the deepest examined level. As we did not find such a move, we picked move 15 with a win rate hovering around 50%. Alternatively, moves 8 and 3 seem good options as well.

[4]Opening book: `https://github.com/lwevers/lynx/`

| Opening | Level 1 | Level 2 | Level 3 | Swap? |
|---|---|---|---|---|
| 9 | 40.8% | 24.6% | - | |
| 1 | 39.4% | 27.3% | - | |
| 4 | 38.8% | 30.0% | - | |
| 16 | 40.2% | 30.0% | - | |
| 8 | 45.4% | 33.6% | - | |
| 3 | 46.2% | 33.6% | - | |
| 7 | 59.4% | 46.5% | 60.5% | ✓ |
| 14 | 58.0% | 53.1% | 64.8% | ✓ |
| 43 | 66.5% | 49.6% | 66.8% | ✓ |
| 15 | 49.6% | 37.9% | 67.6% | ✓ |
| 23 | 63.7% | 61.3% | 67.6% | ✓ |
| 13 | 65.0% | 48.8% | 69.1% | ✓ |
| 22 | 64.9% | 59.7% | 78.5% | ✓ |
| 21 | 67.2% | 64.8% | 80.0% | ✓ |
| 31 | 79.4% | 76.6% | 81.3% | ✓ |
| 32 | 76.7% | 61.3% | 81.6% | ✓ |

**Table 1:** Win rate for all opening moves.




**(a)** Win rate at level 2 **(b)** Win rate with swapping

**Figure 4:** Win rate (a) without and (b) with swap rule.

We also use the results of the opening analysis to determine when to apply the swap rule. If the opponent opens with a move that has less than 50% win chance for us at the deepest level that we analyzed, we apply the swap rule.

# 5 Experimental Results

We have performed experiments with our player in self play, which we discuss in this section. In the experiments, our player alternatingly plays Black and White, except for the opening book experiment. Our player uses $\frac{\text{remaining time}}{12}$ time per move. As a baseline, our player uses 20 seconds of computation time for the whole game, which corresponds to the amount of time that our player used in the CodeCup competition. In the results, intervals show 95% confidence bounds, computed with the Wilson score interval with continuity correction [13].

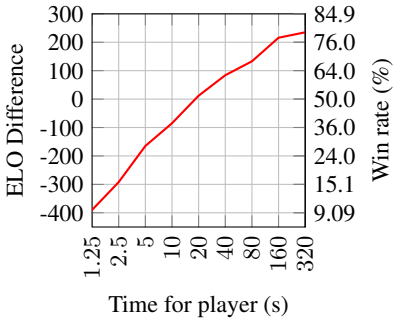
**Game Duration** We investigated the length of games from a sample of 256 games. The first games end by move 50, and by move 80 nearly all games have ended. Interestingly, games end earlier when our player plays without heuristics. When using heuristics our player gets to a winning state earlier, but the player does not try to end the game immediately because it considers all moves as winning.

**The Effect of Computation Time** Figure 5 shows the effect of computation time on the playing strength of our player. Playing strength is measured by the win rate against the baseline player. Additionally, the ELO gain or loss compared to the baseline is shown. We see that time has a large effect on playing strength. We can gain or lose this speed by the choice of algorithm, or the programming language implementation. We also see that spending more time leads to diminishing returns. This shows that it may be worthwhile to spend more time on heuristics and higher level analysis instead of playouts if more time is available.

**Comparing Heuristics** Figure 6 shows how various versions of our player compare to each other. Adding either AMAF or playout heuristics provides a significant improvement over a UCT version of our player. The AMAF version is better against the UCT version than the UCT with heuristics version. Interestingly, the UCT with heuristics version beats AMAF, showing that the win rates of different improvements cannot simply be multiplied to obtain the win rate when combining the improvements. This could be explained by AMAF effectively increasing the number of playouts, while heuristics lead a more targeted search in the tree. Combining both AMAF and playout heuristics leads to another large

| Time | Win rate | ELO |
|---|---|---|
| 1.25 s | 9.6 % ± 2.0% | -390 |
| 2.5 s | 15.7 % ± 2.4% | -292 |
| 5 s | 27.9 % ± 2.9% | -165 |
| 10 s | 38.0 % ± 3.1% | -85 |
| 20 s | 51.7 % ± 3.1% | 12 |
| 40 s | 61.9 % ± 3.1% | 84 |
| 80 s | 68.3 % ± 3.0% | 133 |
| 160 s | 77.6 % ± 2.7% | 216 |
| 320 s | 79.5 % ± 2.6% | 235 |

**Figure 5:** Win rate against 20 s (1024 playouts).



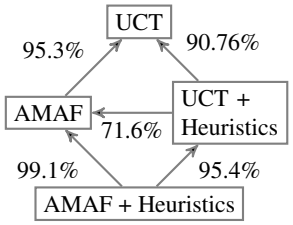




**Figure 6:** Win rates of various players (20 s, 10,000 playouts, ± 1% with 95% confidence).

| Time per player | **5 s** ($n$ = 20,000) | **20 s** ($n$ = 20,000) | **80 s** ($n$ = 10,000) |
|---|---|---|---|
| Bridges | 97.5% ± 0.23% | 98.8% ± 0.16% | 99.2% ± 0.21% |
| Higher-level | 67.3% ± 0.66% | 70.4% ± 0.64% | 72.4% ± 0.89% |
| Fillboard | 57.4% ± 0.69% | 58.7% ± 0.69% | 58.3% ± 0.97% |

**Table 2:** Incremental improvements of individual playout heuristics, with $n$ playouts.

improvement over both the AMAF and the UCT heuristics version. In our experiments, the UCT-only version was able to beat the AMAF + Heuristics version only five times out of 10,000 games.

Table 2 shows the incremental improvements for the individual playout heuristics compared to the *previous entry in the table*, where the bridges heuristic is compared to a baseline featuring AMAF without heuristics. That is, the higher-level version also contains the bridges heuristic, and the fillboard version also contains the higher-level and bridges heuristics.

We see that the bridges heuristic provides a dramatic increase in playing strength. The heuristic to move around the opponent (higher-level) also increases playing strength by a large amount, and the fillboard heuristic provides another small increase in playing strength. We also see that, as more time is used and more playouts are performed, the performance of the heuristics improve.

**Opening Book** Table 3 shows the effect of the opening book on playing strength when playing as White or Black. The no book version of our player does not use the opening book, but opens with move 15 and swaps according to Table 1. Without an opening book, we see that Black has a strong first-player advantage due to the swap rule. When we play using the opening book against a version that does not use the opening book, we see a large increase in playing strength as White. However, playing strength of Black does not increase significantly. We also see that with more computation time, White has less of an advantage from using the opening book. Finally, when both players use the opening book, White gains the advantage over Black.

# 6 Conclusions

Many techniques used in MCTS Hex are applicable to Poly-Y. However, our player is provided with only 30 seconds of time per game, which makes computationally expensive techniques such as virtual-connection search infeasible. Moreover, we did not implement techniques for pruning moves, such as dominating move analysis, since early experiments showed that the AMAF heuristic effectively covers

| Time per player | **5 s** | | **20 s** | | **80 s** | |
|---|---|---|---|---|---|---|
| Playing as | **White** | **Black** | **White** | **Black** | **White** | **Black** |
| No book vs. no book | 25.3% | 74.7% | 24.4% | 75.6% | 23.2% | 76.8% |
| Book vs. no book | 69.0% | 76.3% | 65.1% | 78.0% | 62.2% | 77.0% |
| Book vs. book | 56.9% | 43.1% | 57.4% | 42.6% | 54.5% | 45.5% |

**Table 3:** Effect of using the opening book (10,000 playouts, ± 1% with 95% confidence).

this. Instead, we rely on tree search together with patterns for protecting virtual connections in the random playouts. Additionally, we found that detailed level-2 and level-3 patterns in the random playouts, as used by some Hex players, are less effective than a simpler 'walking to the edge' strategy. Finally, to improve the performance of our player in the early game, we computed an opening book, which made our player significantly stronger against our player without an opening book.

**Future Work** Our player only uses three simple playout heuristics, but these improve playing performance by a lot. Future work may investigate whether other patterns improve playing performance. Additionally, given more computation time per move, our player may improve through virtual-connection search as used in MCTS Hex players. It would also be interesting to investigate very lightweight virtual-connection search techniques that can be used in very short games. Recently, Saffidine and Cazenave [14] show that for the game Y, product propagation is an alternative to MCTS in settings with tight time constraints, so it would be interesting to investigate how this performs for our Poly-Y player.

# References


[1] Vadim V. Anshelevich. A hierarchical approach to computer Hex. *Artificial Intelligence.*, 134(1-2):101–120, January 2002.

[2] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo tree search in Hex. *IEEE Trans. on Computational Intelligence and AI in Games*, 2(4):251–258, December 2010.

[3] Donald A. Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Springer, 1985.

[4] Cameron Browne and Stephen Tavener. Bitwise-parallel reduction for connection tests. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):112–119, June 2012.

[5] Bernd Brügmann. Monte Carlo Go. Technical report, Physics Department, Syracuse University, March 1993.

[6] Guillaume Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, and Olivier Teytaud. Adding expert knowledge and exploration in Monte-Carlo tree search. In *Advances in Computer Games*, volume 6048 of *LNCS*, pages 1–13. Springer, 2010.

[7] Guillaume Chaslot, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, Olivier Teytaud, and Mark Winands. Meta Monte-Carlo tree search for automatic opening book generation. *Proceedings of the IJCAI'09 Workshop on General Intelligence in Game Playing Agents*, pages 7–12, 2009.

[8] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *5th Int. Conference on Computers and Games*, volume 4630 of *LNCS*, pages 72–83. Springer, 2007.

[9] Ryan Hayward, Yngvi Björnsson, Michael Johanson, Morgan Kan, Nathan Po, and Jack van Rijswijck. Solving 7×7 Hex with domination, fill-in, and virtual connections. *Theoretical Computer Science*, 349(2):123–139, 2005.

[10] David P. Helmbold and Aleatha Parker-Wood. All-Moves-As-First Heuristics in Monte-Carlo Go. *Proc. of the 2009 International Conference on AI*, pages 605–610, 2009.

[11] Stefan Kiefer. Die Menge der virtuellen Verbindungen im Spiel Hex ist PSPACE-vollständig. Studienarbeit Nr. 1887, Universität Stuttgart, July 2003. In German.

[12] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Proc. 17th European Conference on Machine Learning*, ECML'06, pages 282–293. Springer, 2006.

[13] Robert G. Newcombe. Two-sided confidence intervals for the single proportion: comparison of seven methods. In *Statistics in Medicine*, volume 17, pages 857–872, 1998.

[14] Abdallah Saffidine and Tristan Cazenave. Developments on product propagation. In *8th International Conference on Computers and Games*, volume 8427 of *LNCS*, pages 100–110. Springer, 2014.