

# Monte Carlo Tree Search in Hex

Broderick Arneson, Ryan B. Hayward, Philip Henderson

**Abstract**—Hex, the classic board game invented by Piet Hein in 1942 and independently by John Nash in 1948, has been a domain of artificial intelligence research since Claude Shannon’s seminal work in the 1950s.

Until the Monte Carlo Go revolution a few years ago, the best computer Hex players used knowledge-intensive alpha-beta search. Since that time, strong Monte Carlo Hex players have appeared that are on par with the best alpha-beta Hex players. In this paper we describe MoHex, the Monte Carlo Tree Search Hex player that won gold in the 2009 Computer Olympiad.

Our main contributions to Monte Carlo Tree Search include using inferior cell analysis and connection strategy computation to prune the search tree. In particular, we run our random game simulations not on the actual game position, but on a reduced equivalent board.

## I. INTRODUCTION

Monte Carlo Tree Search (MCTS) is an exciting new development in the field of automated game players, attaining great success in a variety of games, most notably Go [10], [11], [13], [25].

Hex is the classic board game invented by Piet Hein in 1942 and independently by John Nash in 1948 [19], [31]. Hex has been a domain of artificial intelligence research since Claude Shannon’s seminal work in the 1950s [34].

Like Go, Hex has a large branching factor that limits automated players to shallow or selective search. Unlike Go programs, Hex programs have reasonably strong evaluation functions, and therefore straightforward alpha-beta techniques have been successful [3], [6], [28].

We began experimenting with the possibility of an effective automated Hex player based on the MCTS framework in early 2007. The resulting player, named MoHex, won silver and gold respectively in the 2008 and 2009 Hex Computer Olympiads [5], [6]. In the latter tournament MoHex was undefeated.

In this paper we describe the framework of MoHex, with particular emphasis on its algorithmic contributions which exploit mathematical properties of Hex:

- MoHex uses proven knowledge in the MCTS tree to prune inferior children, and
- MoHex computes Monte Carlo simulations from a reduced equivalent board, rather than the actual game board.

In §II we review the rules of Hex, and algorithms for finding inferior cells and connection strategies. In §III we discuss previous automated Hex players and review the success of alpha-beta based programs. In §IV we describe the basic framework of MoHex. In §V we describe our enhancements of MoHex, especially Hex-specific techniques

for pruning children in the MCTS tree, and for constructing reduced but equivalent boards which improve and accelerate MoHex’s game simulations. In §VI we analyze experimental data which measures the effectiveness of our techniques and the scaling of MoHex’s strength with greater computing power. In §VII we analyze MoHex’s performance against other automated Hex players. In §VIII we discuss the potential application of these techniques to related games, such as Y and Havannah. In §IX we review MoHex’s current limitations, and avenues for future research.

## II. RULES AND ALGORITHMS OF HEX

### A. Rules of Hex

Hex has simple rules: Black and White alternate turns, and on each turn a player places a single stone of their colour on any unoccupied cell. The winner is the player who forms a chain of their stones connecting their two opposing board sides. See Figure 1.

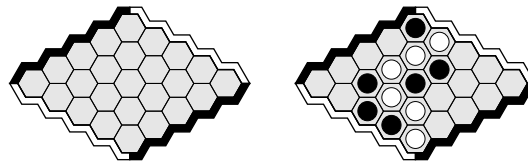


Fig. 1. An empty 5×5 Hex board and a game won by White.

It is never disadvantageous for players to have an extra stone of their colour on the board, and Hex can never end in a draw, so by Nash’s strategy-stealing argument Hex is a first-player win on all  $n \times n$  boards [31]. The first-player advantage is considerable and when unrestricted usually leads to an easy win, so Hex is often played with the swap rule: the first player makes Black’s first move, and the second player then chooses whether to play as Black or White. White makes the next move, and the players alternate turns thereafter. This variation is a second-player win, but in practice produces closer games than without the swap rule.

Like Go, Hex can be played on any  $n \times n$  board. The Computer Olympiad Hex tournaments use  $11 \times 11$  boards. Humans typically play on board sizes ranging from  $11 \times 11$  to  $19 \times 19$  [24], [27], with beginners often starting out on smaller boards. Automated Hex solvers have solved all opening moves on all board sizes up to and including  $8 \times 8$  [21].

### B. Inferior Cell Analysis

There are two main techniques in Hex inferior cell analysis: fillin and move pruning.

*Fillin* is the process of adding to a Hex position a set of stones that is guaranteed not to alter the position’s minimax

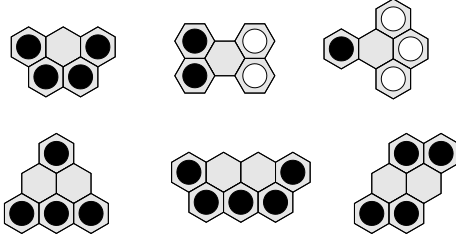


Fig. 2. Local fillin patterns. Top three patterns identify dead cells, and can be filled in with stones of either player. Bottom three patterns identify Black-captured regions, and can be filled in with Black stones.

value. There are two main categories of fillin: dead cells and captured regions. A *dead cell* is a cell that is provably useless for both players. A *captured region* is a set of cells on which one player has a second-player strategy that negates any benefit their opponent might gain from playing in the region. MoHex uses nineteen local patterns for identifying dead cells and captured regions; some are shown in Figure 2. MoHex also uses graph-theoretic algorithms and board decompositions to find larger classes of fillin configurations [21].

*Move pruning* is the process of omitting from consideration a legal move in the current game position. Combinatorial game theory allows reversible moves to be bypassed, and dominated moves to be pruned as long as some dominating move is considered [8]. This theory forms the basis for pruning various forms of Hex cell, including vulnerable, captured-reversible, fillin-dominated, and induced-path dominated cells [15], [16], [18], [22], [23]. As with fillin, MoHex uses (about 250) local patterns to find such cells. See Figure 3.

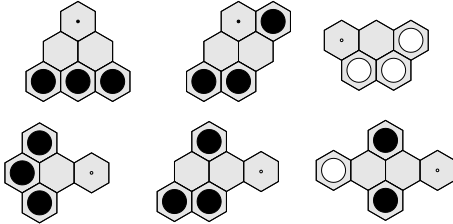


Fig. 3. Local inferior cell patterns. Empty cells can be pruned from consideration by Black based on their reversible/dominated relation to the corresponding dotted cells.

### C. Computing Connection Strategies

H-search is an algorithm that finds cell-to-cell connection strategies in a given Hex position [2], [4]. Starting with the base case of trivially connected adjacent cell pairs, H-search inductively builds larger connections by combining smaller ones in series and parallel until no further connection strategies are found. H-search is known to be incomplete; in particular, it misses some relatively simple connections with overlapping substrategies that humans can easily detect. Thus minor H-search extensions have been developed [20], [32].

A *virtual connection* (VC) is a second-player strategy for connecting two endpoints, and a *virtual semi connection* (SC) is a first-player strategy for connecting two endpoints. A *winning connection* is a strategy whose two endpoints are opposing board sides. The *carrier* of a connection strategy is the (minimal) set of empty cells required to carry out this strategy. See Figure 4.

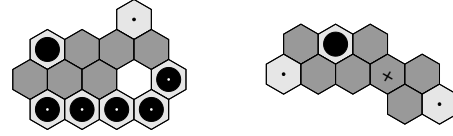


Fig. 4. A Black VC and a Black SC. Carriers are shaded, endpoints are dotted, and the first move of the SC strategy is  $\times$ .

If H-search finds a winning VC for either player, or a winning SC for the player to move, then this player can win by following the discovered connection strategy.

If H-search finds a winning SC for the player who just moved, then the player to move need only consider moves inside the carrier of this SC, since all other moves are provably losing; that is, all moves outside the winning SC's carrier leave the opponent a winning SC on their next turn. The *mustplay* for the player to move is the intersection of their opponent's winning SC carriers. By the same reasoning, all moves outside the mustplay are provably losing. See Figure 5.

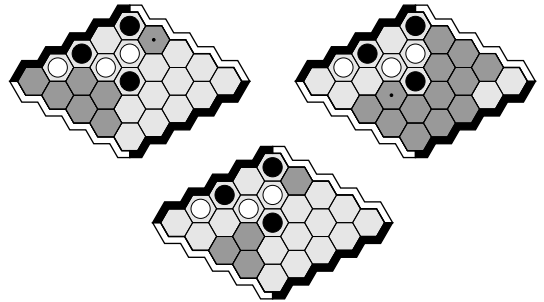


Fig. 5. Two White winning SC carriers and the corresponding mustplay for Black.

H-search prunes many losing moves and produces perfect endgame play, but is time-costly: efficient implementations can compute the connection strategies for about 25 positions per second on tournament-sized  $11 \times 11$  boards.

### III. ALPHA-BETA HEX PLAYERS

Together with E.F. Moore, Claude Shannon developed the first automated Hex player in the 1950s, an electronic circuit network which set positive charges for one player's stones, negative charges for the other player's stones, and then played at a certain saddle point. The computer played strong opening moves but sometimes erred in tactical situations [34]. Shannon also developed a computer to play Bird Cage, now known as Bridg-it, a game similar to Hex. This circuit

network set the resistance of one player's stones to zero, the resistance of the other player's stones to infinity, and then played at a cell with greatest voltage drop [12].

In 2000 Hexy won the first Computer Olympiad Hex competition [3]. Hexy's evaluation function uses an augmentation of Shannon's Bird Cage circuit in which extra wires are added which correspond to connections found by H-search [2]. Hexy uses this evaluation function in a straightforward alpha-beta search.

In 2003, 2004, and 2006 Six won the next three Computer Olympiad Hex competitions [17], [29], [35]. Six significantly refines the Hexy framework by improving H-search efficiency via heuristic limits, restricting the alpha-beta branching factor, tuning the evaluation function, and using a small amount of inferior cell analysis [28]. Six uses a truncated 2-ply alpha-beta search. Six is open source and has been played by many humans. It is generally considered to be a very strong, although not quite expert, player on board sizes up to  $11 \times 11$ , with near-perfect play on board sizes up to  $8 \times 8$  [27].

In 2008 Wolve won the fifth Computer Olympiad Hex competition, defeating the previous champion four games to zero [6]. Wolve was developed in conjunction with MoHex and shares the same codebase. Wolve improves on Six's design by modifying the evaluation function, improving H-search efficiency, using much more inferior cell analysis, using an opening book generated by self-play, and using a solver in parallel with the player. Like Six, the 2008 version uses a 2-ply search. The 2009 version of Wolve searches to 4-ply when time allows.

In summary, computer players using alpha-beta search have proven to be strong competitors against skilled human players, and until 2008 dominated computer Hex competitions.

#### IV. BASIC MOHEX FRAMEWORK

##### A. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a best-first search algorithm that is guided by the outcome of random game simulations. The algorithm is composed of three basic phases:

- 1) *tree traversal* from the root to some leaf node,
- 2) *random game simulation* from the leaf node's corresponding game position, and
- 3) *tree update* using information from the simulated game.

The basic algorithm is anytime, repeating these steps until no more time remains. After the tree traversal phase, the search tree is expanded by adding the children of the selected leaf node. When MCTS terminates, the child with the largest subtree (i.e. which produced the most simulations) is selected as the best move.

MoHex's MCTS is built on the codebase of Fuego, the Go program developed by Martin Müller et al. at the University of Alberta [10].

##### B. Tree Traversal and Update

MoHex uses the UCT framework combined with the all-moves-as-first (AMAF) heuristic to select the best child during tree traversal [14], [25]. Like Fuego, MoHex plays strongest when it uses an exploration constant of zero, effectively turning off UCT exploration and relying solely on the AMAF heuristic to find strong candidate moves.

The tree update phase updates the win/loss statistics for each move in the simulated game. Updates occur at each node along the path from the leaf to the root, thereby influencing leaf node selection in future tree traversals.

##### C. Random Game Simulation

Playing Hex until the board is completely filled results in the same win/loss outcome as stopping once one player has a winning chain, so the random game simulation need not check for game termination after each move. Hence Hex game simulations can be efficiently implemented: add all empty cells to an array, shuffle them randomly, and play the remaining moves in order. A consequence of this particular implementation is that each legal move's AMAF statistics are updated after each game simulation.

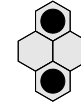


Fig. 6. Bridge rollout pattern: if White plays one empty cell, Black plays the other.

As with Go, it is beneficial to apply some knowledge during simulated games [9]. MoHex uses a single pattern during random game simulation: if a player probes any opponent bridge, then the opponent always replies so as to maintain the connection. See Figure 6. If multiple bridges are probed simultaneously, then one such bridge is randomly selected and then maintained.

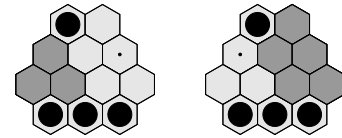


Fig. 7. 4-3-2 VC rollout pattern: if White plays a shaded cell, Black maintains the connection by playing the corresponding dotted cell.

Yopt, another MCTS Hex program, uses additional patterns based on another commonly occurring VC [33]. See Figure 7. However, in our tests MoHex showed no strength gain from such patterns.

#### V. MOHEX ENHANCEMENTS

##### A. Tree Knowledge

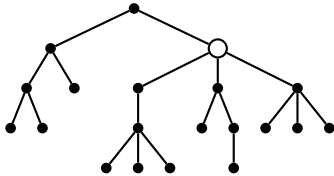
Like many other MCTS players, MoHex uses knowledge-intensive algorithms in important parts of the tree, as well as flags to indicate solved states [36].

Using a fixed “knowledge threshold”, if any node is visited often enough during tree traversal, then both inferior cell analysis and the H-search algorithm are run on that position. There are two possible outcomes: either fillin or H-search solves the position, or the position value is still unknown.

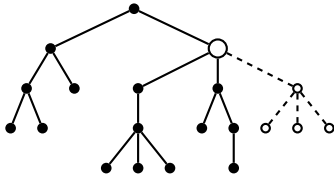
In the former case, all child subtrees are deleted, and the tree node is marked such that any tree traversal that encounters this node omits the random game simulation, and simply updates its ancestor nodes using the correct outcome.

In the latter case, subtrees corresponding to moves that can be eliminated via **inferior cell analysis** or mustplay results are pruned from the tree.

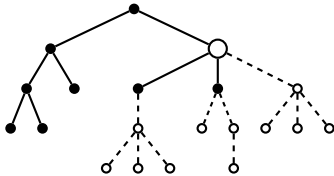
Furthermore, for every tree node that surpasses the knowledge threshold, its fillin computation is stored permanently and applied to every tree traversal. Since fillin is computed anew for each tree node, there can be disagreement between the fillin of a node and its child, so the descendant node’s fillin takes precedence, and any prior fillin knowledge is ignored.



1) Node reaches knowledge threshold; perform inferior cell analysis and H-search computations.



2) Prune children that are inferior or outside of mustplay.



3) Remove subtrees of all remaining children.

Fig. 8. Applying knowledge to the Monte-Carlo tree.

This fillin produces two benefits. Firstly, the random game simulations are shorter (since the number of empty cells has decreased), and thus allows more game simulations per second. Secondly, the accuracy of the game simulations should improve, since any resolved regions of the board are guaranteed to have the correct outcome.

Although each child node corresponding to a fillin move is deleted, a fillin move might still be available in some child’s subtree, possibly yielding an illegal game sequence in

which a fillin move is played. To avoid this, each unpruned child’s subtree is also deleted excepting their roots and all relevant statistics (e.g. UCT and AMAF data). Note that any subsequent tree expansions below the parent node will not conflict with the fillin. See Figure 8.

The knowledge threshold is typically small (e.g. the 2009 Olympiad version had threshold 50), so the size of any truncated subtree is small, and the subsequent loss of information is apparently more than compensated by the gain in performance.

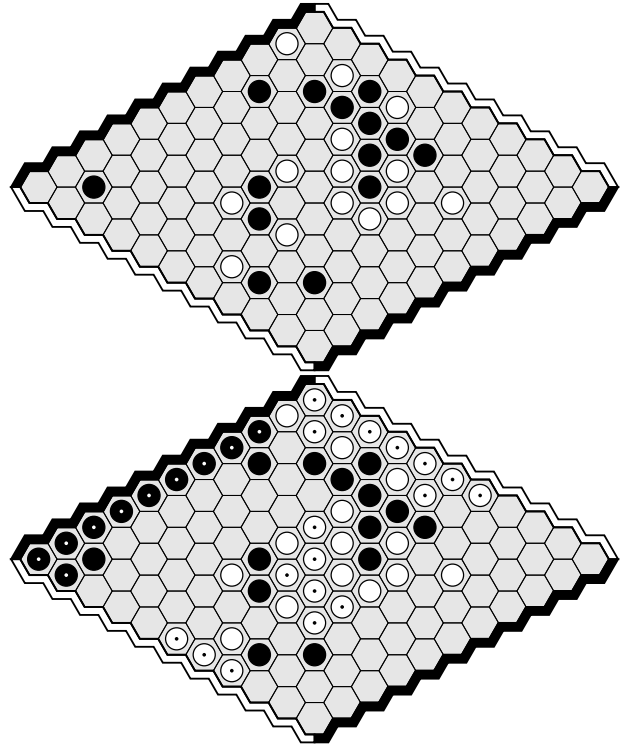


Fig. 9. A Hex position and its fillin reduced position.

For instance, consider the top Hex position in Figure 9. If MoHex evaluates this position for Black without knowledge, it consistently gets a score of 71-73% for 30s searches. By applying knowledge in the Monte Carlo tree, fillin produces the bottom Hex position in Figure 9, and the evaluation scores plummet to 1-8% for 30s searches. This behaviour also holds for other MCTS Hex players: this position was taken from an olympiad game between Yopt and Wolve, where Yopt’s evaluation score surpassed 90% in the endgame despite its losing position.

## B. Lock-Free Parallelization

MoHex uses the Fuego codebase, and so benefits from Fuego’s lock-free parallel MCTS [10]. MoHex’s knowledge computations are handled within this lock-free framework. Thus it is possible for different threads to perform duplicate knowledge computations concurrently, but this is extremely rare in practice.

### C. Parallel Solver

MoHex runs a Hex solver concurrently with the search. This produces perfect play whenever the game position can be solved quickly. This solver uses inferior cell analysis and H-search in the same way as MoHex: to identify state values, and prune inferior moves from consideration.

The solver is based on depth-first proof number (DFPN) search, and so also relies on these knowledge computations to restrict its branching factor and guide it to the strongest moves [1], [30]. It also uses a move ordering heuristic to temporarily prune the weaker moves of each node in the DFPN search tree, gradually revealing them as their sibling moves are proven to be losing. This technique reduces the combinatorial explosion of large branching factors, and also prevents the winning player from spending time exploring (heuristically) weaker moves.

Our solver can solve all  $8 \times 8$  openings in roughly 30 hours. It has also solved over half of the  $9 \times 9$  openings [7].

### D. Time Management

Over thousands of games between strong players (i.e. any of MoHex, Wolve, Six), the average  $11 \times 11$  game length is about 60 moves. Within a one-minute search, MoHex’s solver usually computes the game’s value by move 35, and almost always by move 45. This bounds the number of moves MoHex needs to generate in a game. Hex tournaments allow 30 minutes per player, so MoHex can easily allot one minute per move. In the 2009 Computer Olympiad, MoHex allotted 96 seconds per move.

## VI. EXPERIMENTAL RESULTS

Because of the swap rule, an automated Hex player needs to be able to respond competently to every opening move the opponent might select. Thus our testing iterates over all  $11 \times 11$  openings with the swap rule off, with each program playing once as Black and once as White for every opening. Thus one round consists of  $11 \times 11 \times 2 = 242$  games. In order to reduce the standard error to a small percentage, we typically run several rounds.

While this testing format is helpful in identifying weaknesses in our algorithm’s performance (e.g. openings where we perform poorly as both Black and White), it significantly dampens any strength gains obtained, as polarized openings (i.e. openings easily won by Black or White) are played twice, and essentially guarantee at least some wins for the weaker player. Thus the Elo gains reported in our experiments underrepresent the expected tournament play Elo improvement.

### A. Scaling

MCTS is a parallelizable anytime algorithm, so the scaling of its performance with respect to time and number of threads is important.

As with many other MCTS programs, MoHex’s strength scales logarithmically versus time, with each doubling of the game simulations producing roughly an additional 36 Elo of

strength: 8s/move MoHex defeats 1s/move MoHex 65.1% of the time. See Figure 10.

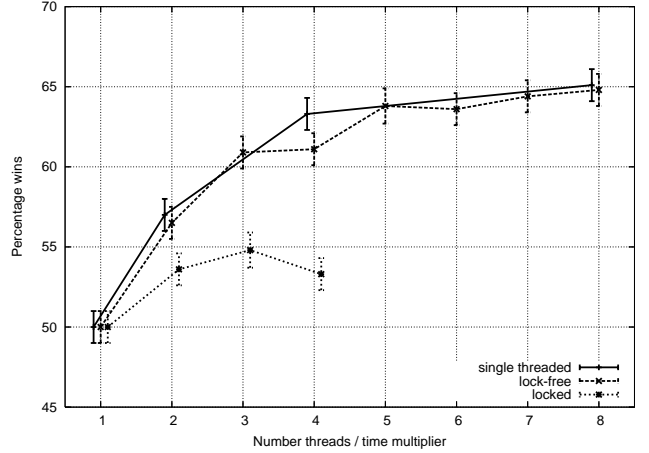


Fig. 10. Performance of locked, lock-free, and time-scaled single threaded MoHex against single threaded 1s/move MoHex.

As with Fuego, the lock-free version of multithreaded MoHex scales far better than the locked alternative. Indeed, the change here is even more dramatic than that in Go — scaling of the locked version is worse with two threads, and performance actually degrades with only four threads — presumably because the game simulations in Hex are so much faster than in Go, and the threads spend most of their time in the tree. See Figure 10.

### B. Heuristic Techniques

Both the bridge pattern and AMAF heuristic give major strength gains for MoHex. The bridge pattern produces a 105 Elo strength gain against a naive UCT implementation (with an optimized exploration constant of 0.7), and this improved version is surpassed by another 181 Elo by adding the AMAF heuristic (and turning off UCT exploration). Based on the scaling information above, this total strength gain is roughly equivalent to a 250-fold increase in computing time. See Figure 11.

Incrementally Added Feature	Win %	Elo gain
Bridge pattern	64.7% $\pm$ 1.4%	105
AMAF heuristic	73.9% $\pm$ 1.3%	181

Fig. 11. The bridge pattern and AMAF heuristic improve playing strength by 286 Elo.

We tested many inferior cell analysis patterns as game simulation patterns. Unfortunately, in all cases these patterns gave MoHex no strength gain. This provides evidence that provably correct information in game simulations can weaken MCTS players.

### C. Tree Knowledge

Adding tree knowledge to MoHex is roughly equivalent to doubling the number of game simulations. The optimal knowledge threshold seems to be 400 for the single threaded version, and to decrease proportionally with the number of

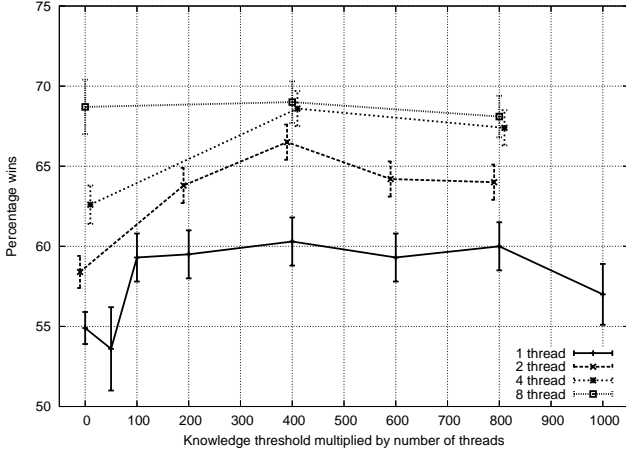


Fig. 12. Threaded 8s/move MoHex with knowledge against 2-ply Wolve. A knowledge threshold of zero means that no knowledge is computed.

threads. A very low knowledge threshold can worsen performance, due to the corresponding decrease in the number of rollouts. See Figure 12.

#### D. Opening Book

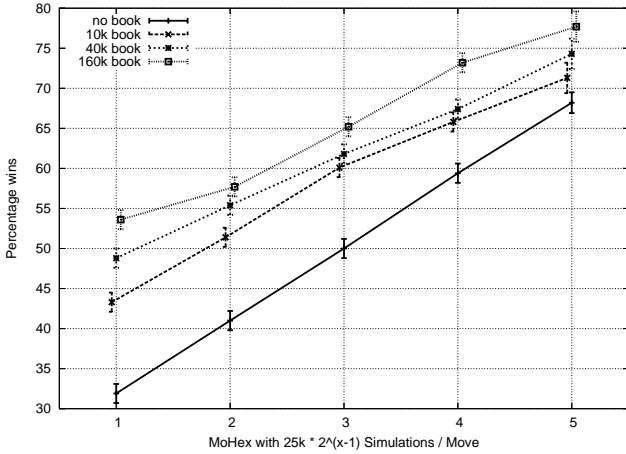


Fig. 13. MoHex with books of increasing size against 100k / move MoHex with no book.

MoHex’s opening play can be inconsistent, perhaps because there is so little existing structure to guide the random game simulations. We are investigating the construction of an opening book using Lincke’s method [26]. Our initial results are promising, with an opening book for  $9 \times 9$  Hex that was constructed in a day producing gains of 85 Elo, which is worth more than a doubling of simulations. As the book size increases, the strength gains grow logarithmically. See Figure 13.

### VII. TOURNAMENT PERFORMANCE

As mentioned earlier, MoHex won gold in the 2009 Computer Olympiad. Its opponents were Yopt, another MCTS-based player, and the previous alpha-beta based champions Wolve and Six. See §III. Although MoHex was undefeated,

it nevertheless had a few close games, three of which we briefly examine here. See Figure 14.

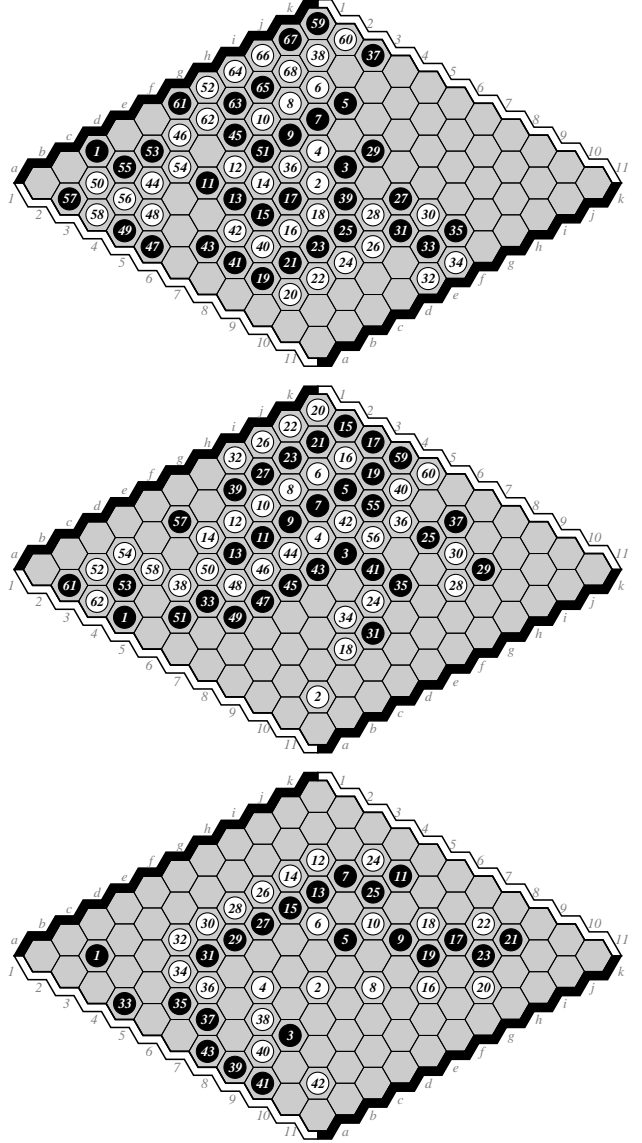


Fig. 14. MoHex 2009 Computer Olympiad games. From top to bottom, opponents are Wolve, Six, and Yopt. MoHex is White in all games.

Against Wolve, MoHex’s evaluation was close to 50% throughout the game, until its parallel solver indicated that the game was won. Wolve’s move 37 looks weak — k2 seems preferable to k3 — but Wolve had already lost the game at that point.

Against Six, the game’s progression is surprising, since MoHex’s opening play is weak and Six develops a strong wall of influence within 19 moves. Nevertheless, MoHex manages to play effectively, with an especially brilliant move 36. MoHex’s parallel solver generates moves 40 and beyond. Post-game analysis revealed the game was close: MoHex blundered on move 38 and Six blundered on move 39.

Against Yopt, the game is straightforward, with parallel

bridge structures and many probes of such connections. Post-game analysis indicates that Yopt played winning moves 27 and 29, but that move 31 is a blunder: d4 loses while c4 wins. MoHex maintains its winning position from this point on.

See the Olympiad report for further commentary [5].

#### A. Experimental Tournaments

Because Olympiad results are based on a very small sample size, we also ran our own tournaments between MoHex and the two previous champions, Six and Wolve. These tournaments omitted opening books and parallel solvers, so that the results measure only the relative strength of each player’s search engine. We did not run tests against Yopt since this program is not publically available. The results are summarized in Figure 15. Basically, MoHex dominates Six and is evenly matched with Wolve.

Opponent	MoHex Win %
Six	76.6 $\pm$ 3.6
Wolve	49.2 $\pm$ 3.2

Fig. 15. MoHex: performance against Six and Wolve.

### VIII. APPLICATION TO OTHER GAMES

The application of tree knowledge to MCTS, in particular the use of fillin, can be generalized to other classical board games. Most of Hex’s inferior cell analysis applies to Y (with only minor modifications relating to the fact that board sides are not owned by players in Y), and so a MCTS Y player could likewise use fillin and inferior cell knowledge in its Monte Carlo tree.

Fillin and inferior cell pruning can also be computed for Havannah, but the existence of rings as a winning condition (instead of just connecting paths, such as bridges and forks) makes this process far less straightforward. Our group has performed some initial research on this topic, but no implementation of these ideas has yet been performed. Due to the relative rarity of such patterns in Havannah, we expect this knowledge to yield only a minor gain, although it may be more useful in terms of improving a player’s parallel solver.

For games like Go where such theoretical pruning is extremely difficult and/or rare, perhaps heuristic fillin — such as preventing simulated games from playing in the territory of live groups and seki regions — could prove useful in improving accuracy and speeding up simulated games.

### IX. LIMITATIONS AND FUTURE WORK

Despite MoHex’s recent success, there are still many aspects that can be improved.

MoHex does not fully exploit the connection strategies it computes. Although H-search finds connection strategies, MoHex uses this information only for pruning and endgame play. In particular, simulated games do not use known connection properties. As expected, the near-random games

usually do not maintain important connections, especially when the carrier is large.

We have tested many algorithms that incorporate connection strategies into the simulations, including the following two: heuristically select connections to maintain during the simulation; at the root node, allow the choice of maintaining some subset of a given list of connection strategies.

Unfortunately, all of these techniques greatly worsened performance. The best variation to date involves using common responses in the Monte Carlo tree to guide the responses in the simulated game; investigations are still ongoing.

Given the strength of our parallel solver, stronger integration of the solver and MCTS could produce even greater benefits. Currently the solver indicates only whether the root position has been solved; the solver could instead inform MoHex of any tree nodes that it has solved. Furthermore, it might be desirable for the search tree to indicate which moves it currently prefers, thereby encouraging the solver to explore the most pertinent lines.

Other possibilities include using the AMAF heuristic to influence the random game simulations, ensuring that moves perceived to be strong are played early in the simulation. Also, heuristic initialization of moves in the Monte Carlo tree may ensure that strong moves are always explored or speedup the UCT convergence, although such methods have not yet been successful in Monte Carlo Hex.

### X. CONCLUSION

We have applied Monte Carlo Tree Search techniques to produce an automated Hex player that is on par with the best players produced via the classical alpha-beta approach. The potential for further improvement remains, especially in terms of better incorporating connection strategies in the tree and game simulations, improved knowledge computation in the tree (i.e. parallel solver updates), development of an opening book, and improving simulation quality.

#### Acknowledgements

We are indebted to Sylvain Gelly of the MoGo team and Martin Müller, Markus Enzenberger, David Silver of the Fuego team for many useful discussions regarding Monte Carlo Tree Search. We also thank Gábor Melis and Tristan Cazenave for useful discussions regarding their respective automated Hex players.

### REFERENCES

- [1] L. Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, Netherlands, 1994.
- [2] Vadim V. Anshelevich. The game of Hex: An automatic theorem proving approach to game programming. In *AAAI/IAAI*, pages 189–194, Menlo Park, 2000. AAAI Press / The MIT Press.
- [3] Vadim V. Anshelevich. Hexy wins Hex tournament. *ICGA Journal*, 23(3):181–184, 2000.
- [4] Vadim V. Anshelevich. A hierarchical approach to computer Hex. *Artificial Intelligence*, 134(1–2):101–120, 2002.
- [5] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. MoHex wins Hex tournament. *ICGA Journal*, 32(2):114–116, 2009.
- [6] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Wolve 2008 wins Hex tournament. *ICGA Journal*, 32(1):49–53, March 2009.

- [7] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Solving Hex: Beyond humans. Accepted to Computers and Games, 2010.
- [8] Elwyn Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*, volume 1–4. A.K. Peters, 2nd edition, 2000.
- [9] Rémi Coulom. Computing Elo ratings of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, December 2007.
- [10] M. Enzenberger and M. Müller. Fuego homepage, 2008. Date of publication: May 27, 2008. Date retrieved: March 14, 2010.
- [11] D. Fotland. The Many Faces of Go, version 12, 2009. Date retrieved: April 1, 2009.
- [12] Martin Gardner. *The 2nd Scientific American Book of Mathematical Puzzles and Diversions*, chapter 7, pages 78–88. Simon and Schuster, New York, 1961.
- [13] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go, 2006. Technical Report RR-6062.
- [14] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In Zoubin Ghahramani, editor, *ICML*, volume 227 of *ACM International Conference Proceeding Series*, pages 273–280. ACM, 2007.
- [15] Ryan Hayward, Yngvi Björnsson, Michael Johanson, Morgan Kan, Nathan Po, and Jack van Rijswijck. Solving  $7 \times 7$  Hex with domination, fill-in, and virtual connections. *Theoretical Computer Science*, 349(2):123–139, 2005.
- [16] Ryan B. Hayward. A note on domination in Hex. Technical report, University of Alberta, 2003.
- [17] Ryan B. Hayward. Six wins Hex tournament. *ICGA Journal*, 29(3):163–165, 2006.
- [18] Ryan B. Hayward and Jack van Rijswijck. Hex and combinatorics. *Discrete Mathematics*, 306(19–20):2515–2528, 2006.
- [19] Piet Hein. Vil de laere Polygon? *Politiken*, December 1942.
- [20] Philip Henderson, Broderick Arneson, and Ryan Hayward. Hex, braids, the crossing rule, and XH-search. In J. van den Herik and P. Spronck, editors, *ACG*, volume 6048 of *Lecture Notes in Computer Science*, pages 88–98. Springer, 2010.
- [21] Philip Henderson, Broderick Arneson, and Ryan B. Hayward. Solving  $8 \times 8$  Hex. In Craig Boutilier, editor, *IJCAI*, pages 505–510, 2009.
- [22] Philip Henderson and Ryan B. Hayward. Probing the 4-3-2 edge template in Hex. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2008.
- [23] Philip Henderson and Ryan B. Hayward. Captured-reversible moves and star decomposition domination in Hex. Submitted to *Integers*, 2010.
- [24] igGameCenter. [www.iggamecenter.com/](http://www.iggamecenter.com/), 2009.
- [25] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.
- [26] Thomas R. Lincke. Strategies for the automatic construction of opening books. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games*, volume 2063 of *Lecture Notes in Computer Science*, pages 74–86. Springer, 2000.
- [27] Little Golem. [www.littlegolem.net/jsp/](http://www.littlegolem.net/jsp/), 2009.
- [28] Gábor Melis. Six. [six.retes.hu/](http://six.retes.hu/), 2006.
- [29] Gábor Melis and Ryan Hayward. Six wins Hex tournament. *ICGA Journal*, 26(4):277–280, 2003.
- [30] A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Dept. of Information Science, University of Tokyo, Tokyo, Japan, 2002.
- [31] John Nash. Some games and machines for playing them. Technical Report D-1164, RAND, February 1952.
- [32] Rune Rasmussen and Frédéric Maire. An extension of the H-search algorithm for artificial Hex players. In Geoffrey I. Webb and Xinghuo Yu, editors, *Australian Conference on Artificial Intelligence*, volume 3339 of *Lecture Notes in Computer Science*, pages 646–657. Springer, 2004.
- [33] Abdallah Saffidine. Utilization d’UCT au Hex. Technical report, Ecole Normale Supérieure de Lyon, 2008.
- [34] Claude E. Shannon. Computers and automata. *Proceedings of the Institute of Radio Engineers*, 41:1234–1241, 1953.
- [35] Jan Willemson and Yngvi Björnsson. Six wins Hex tournament. *ICGA Journal*, 27(3):180, 2004.
- [36] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo tree search solver. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2008.