Monte-Carlo Hex

Tristan Cazenave[1] and Abdallah Saffidine[2]

[1] LAMSADE, Université Paris-Dauphine
Place Maréchal de Lattre de Tassigny, 75775 Paris Cedex 16, France
email: cazenave@lamsade.dauphine.fr
[2] Ecole normale supérieure de Lyon
46 allée d'Italie, 69364 Lyon cedex 07, France
email: abdallah.saffidine@ens-lyon.fr

Abstract. We present YOPT a program that plays Hex using Monte-Carlo tree search. We describe heuristics that improve simulations and tree search. We also address the combination of Monte-Carlo tree search with virtual connection search.

1 Introduction

Hex is a two-player board game that was invented by Piet Hein in 1942 and reinvented by Nobel-prize John Nash in 1948. Hex is the most famous connection game. Although one could definitely view it as a mathematical game, numerous theorems have indeed involved Hex, this game has succeeded on developing a strong playing community. The first book dedicated to Hex strategy was released a few years ago [2].

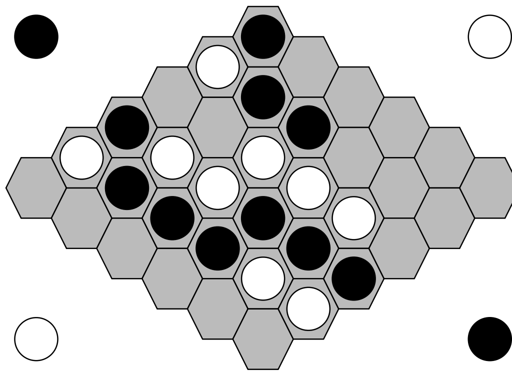

Fig. 1. Game of Hex won by Black

Rules are simple, the board is made of hexagons and is diamond-shaped. Opposing borders belong to the same player. Figure 1 gives a 6x6 board where the game is won by Black. Computer championships are held on 11x11 boards. Players take turns putting pieces of their color on the board. Once set, a piece is never moved nor removed. The

winner is the one player that manages to link his side with a continuous string of his pieces.

The simplicity of the rules helps proving some theoretical facts. Draws are impossible, that is every full board contains one and only one winning chain [20]. There exist a winning strategy for first player at the beginning of the game. The proof of this property is not constructive and uses John Nash strategy-stealing argument. One can understand how difficult it may be to construct an efficient Artificial Intelligence playing Hex by taking a look at the complexity of the associated decision problem: PSPACE-complete [12, 21].

Two pieces of advice are to be given to the beginner. On the one hand it is far quicker to cross the board with bridges, using bridges however provides the same resistance to being cut than the side to side travelling. On the second hand one's position value is that of its weakest link, because it is this weak link that the opponent may exploit to make his way. Finally it seems that a beginner may improve their playing level by trying to prevent their opponent to connect instead of trying to create their own connection.

1.1 Templates

As soon as a bridge is on the board, the groups it joins can be mentally linked. Remembering this shape makes it unnecessary to calculate that the groups are linked. The shape idea leads to the wider concept of template. Edge templates link groups to borders and inner templates link groups together. The Ziggurat is such an edge template 2.

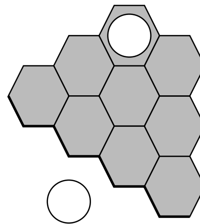

Fig. 2. The *Ziggurat* template links the White piece to the side.

The easiest way to show that a template truly links its groups is to make the connection threats explicit. To each of these threat is associated its carrier which is the set of locations whom owner may influence the outcome of the threat. Therefore to prevent the connection, the opponent needs to play one of the carrier locations. If there are many threats then the opponent has to play in the intersection of the carriers to prevent the template connection. If the intersection is the empty set, the template is proved valid. Or else it is still possible to try the few remaining moves, and study the case for each of them (see figure 5). A bridge connected to the side is a level two template since the distance to the side is two. The Ziggurat is a level three template that can be deduced from bridges based connection threats. Higher level templates also exist. Figure 3 gives a level four template. It can be deduced from bridges and Ziggurats (see figure 4).

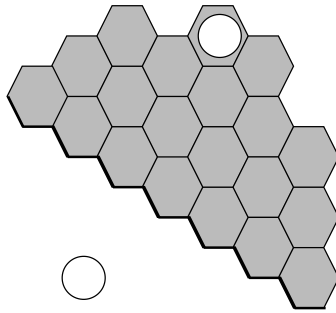

Fig. 3. The distance four template

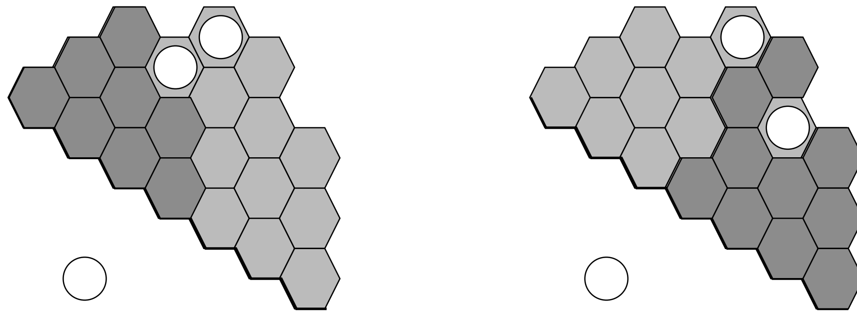

Fig. 4. Two White threats and their carriers.

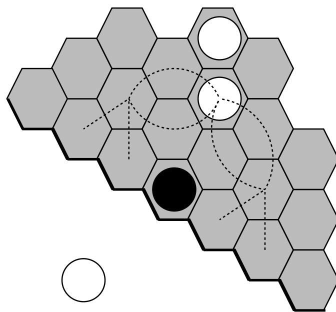

Fig. 5. Analysis of a Black response in the remaining cell.

1.2 Hex Programs

Surprisingly enough the evaluation function that lead to the best playing level up to now dates back to 1953 [1], it makes an analogy between the board and an electric circuit and then measure the electric resistance between opposite borders. Programs such as SIX, use a method of virtual connections in a way similar to that of V. Anshelevich.

When building our program YOPT, we chose to test a completely different method that uses Monte-Carlo simulations. Back in 2000 when we first tried Monte-Carlo methods with Hex, we did not use tree search and V. Anselevich virtual connections used to achieve better results. Recently however, the interest for Monte-Carlo methods for Hex raised with the success of methods developing trees along with Monte-Carlo simulations in the game of Go [10, 19, 13, 15]. Rémi Coulom, Lille 3 University, or Philip Henderson, Alberta University, among other researchers of this field have shown such an interest. As a consequence we developed a Monte-Carlo search program for Hex.

The following is organized as such: second section deals with Monte-Carlo tree search, third section presents its application to Hex, fourth section reveals experimental results

2 Monte-Carlo tree search

Recent improvements of Monte-Carlo methods applied to board games will be discussed in this section. We will particularly concentrate over UCT (Upper Confidence bounds for Trees) and RAVE (Rapid Action Value Estimation) algorithms.

2.1 Simulations

Numerous random simulations are required when applying Monte-Carlo methods to board games. A random simulation is random played game. Knowledge can be added to improve the reliability of the simulations. The random choices of move are biased towards good moves [10, 13, 4]. However, using knowledge during simulations does not always lead to an improvement of the playing strength. In our Go program, it happened indeed that while a knowledge-based pseudo-random player A usually played better than a random player B, once incorporated to the Monte-Carlo algorithm, the results were different. The Monte-Carlo player using simulations of player A was far worse a player than the Monte-Carlo player using player B. Using carefully chosen knowledge clearly improves the Monte-Carlo algorithm; some knowledge may look interesting at first sight, but happen to be counter productive.

In Hex, the knowledge used in random simulations is to always answer to moves that try to disconnect a bridge.

More elaborate knowledge such as inferior moves of the 4-3-2 edge template [17] could also be used to bias simulations.

We can detect the Ziggurat during simulations so as to always connect it. When the position of figure 6 occurs in a game, the simulation player will always win the game as Black if it uses the bridges and the Zigurrat.

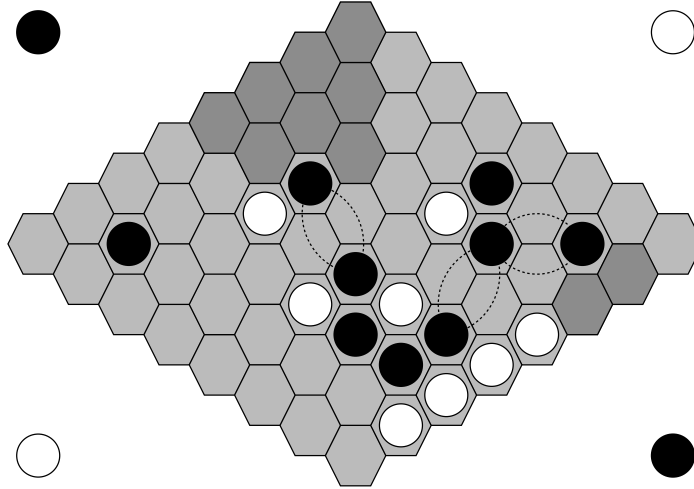

Fig. 6. A Hex game won by Black

2.2 Tree search

The UCT (Upper Confidence bounds for Trees) algorithm [19] consists in creating a tree whose root is the position to evaluate. Each branch is associated to a move and each node is associated to a position. For each node then number s of simulations that went through this node is registered altogether with the sum of the score of these simulations. Hence the average score μ of each node is available. It possible to associate a UCT value to each node using the formula:

$$\mu + C \times \sqrt{\frac{log(parent \to s)}{s}}$$

At the beginning of each random simulation, the algorithm UCT chooses to develop the moves that lead to the node which has the highest UCT value . The C constant allows to tune the exploration policy of the algorithm. The higher C is, the more the algorithm is likely to try moves with a relatively bad mean score.

In the RAVE (Rapid Action Value Estimation) algorithm [15], each node of the UCT tree is given a RAVE-value. The RAVE-value of a node n is the average score of the simulations in which the move associated to n has been played.

The heuristic that evaluates the mean of the simulations in which a move has been played whenever in the simulation, was already used in the first Monte-Carlo Go program GOBBLE [3]. It is usually named AMAF (All Moves As First). Sylvain Gelly improvement is to calculate AMAF at each node of the UCT tree and to combine efficiently UCT and AMAF values. Indeed when only a few simulations have been done for a node, the average score cannot estimate precisely the position's value, whereas the AMAF value for the associated move is calculated over a much higher number of simulations and therefore has a lower variance.

A bias is used to combine AMAF and UCT values. If rc is the number of games where the move was played, rw the number of games where it was played and the game was won ($\frac{rw}{rc}$ is then the AMAF value of the node), c the number of games of the node and m the mean value of this games, a coefficient $coef$ is calculated:

$$coef = 1.0 - \frac{rc}{rc+c+rc*c*bias}$$

The value of a move is then given by the formula:

$$val = m \times coef + (1.0 - coef) \times \frac{rw}{rc}$$

RAVE uses this formula to choose which move should be explored during the descent of the tree before each simulation.

Monte-Carlo tree search gave very good results in the game of Go. The current best Go programs such as MOGO [13, 15] and CRAZY STONE [10, 11] use Monte-Carlo tree search.

2.3 Combining Monte-Carlo Tree Search with a solver

At Hex it is important to detect virtual connections. We reimplemented Anshelevich algorithm [1]. The obvious way to combine with Monte-Carlo tree search is to call the virtual connection algorithm before the Monte-Carlo search in order to detect winning moves. A more elaborate combination is to use the mustplay regions [16] found by the solver in order to reduce the possible moves of the Monte-Carlo search. Analyzing virtual connections could be used at each node of the Monte-Carlo search. However, analyzing a position using virtual connections takes a non negligible time, so we only analyze nodes that have a sufficient number of simulations.

Combining exact values of solved position with Monte-Carlo tree search has been addressed by M. Winands et al. [22] and we reused their algorithm for the combination.

Combining the results of tactical search and Monte-Carlo search has already been successfully used for connections in the game of Go [5].

3 Experimental results

In order to test the different algorithms we have proposed we make them play against each other. Each match consists in 200 11x11 games, 100 with White and 100 with Black.

We first tested if giving the program more simulations improves it. The reference programs plays 16,000 simulations. Results are given in table 1. Clearly the level increases with the number of simulations.

The next experiment consists in finding the best constant for UCT. Using RAVE in Hex clearly improves on the UCT algorithm [8]. We see in table 2 that when RAVE is used the best UCT constant becomes 0 (the reference program in this experiment uses a 0.3 UCT constant).

In order to find if the use of templates was beneficial we tested the program with a fixed number of simulations and different templates use during the simulations:

Simulations	1000	2000	4000	8000	32000	64000
Percentage of wins	6%	11.5%	20%	33%	61%	68.5%

Table 1. Increasing the number of simulation improves the level.

Constant	0	0.1	0.2	0.4	0.5	0.6	0.7
Percentage of wins	61%	60%	55.5%	42%	41%	35.5%	32.5%

Table 2. Variation of the UCT constant.

- type1: completely random
- type2: only bridges
- type3: bridges and level 2 templates
- type4: bridges, level 2 templates and Ziggurats.

Results against type 3 are in table 3. However the time to match the Ziggurat makes the sequential algorithm slower and experiments with a fixed time per move resulted in a low winning percentage (32 %). Checking for Ziggurats during simulations could still be interesting using a parallel Monte-Carlo tree search algorithm [6, 14, 7, 9, 18] since parallel algorithms make simulations cost-less.

Templates	type1	type2	type4
Percentage of wins	22%	42%	71.5%

Table 3. Using templates during simulations

Table 4 makes the program with a RAVE bias of 0.001 play against programs with different biases. The best bias we found is 0.00025.

The next experiment consists in evaluating the combination with a solver. Both programs use bridges and level two templates during simulations. Virtual connection search is computed and used to prune nodes after 1,000 simulations. The program using virtual connections wins 69.5% of its games against the program not using virtual connections.

4 Conclusion

We have presented YOPT, a Monte-Carlo Hex playing program. Monte-Carlo tree search and the RAVE algorithm give good results at Hex. Moreover the combination of virtual connection search with Monte-Carlo search gives even better results.

In future work, we will work on a tighter integration of both paradigms as well as on the improvement of the simulations.

bias	0.0005	0.00025	0.000125
Percentage of wins	50.5%	59%	53.5%

Table 4. RAVE bias

5 Acknowledgement

We thank Philip Henderson, Rémi Coulom and Ryan Hayward for interesting discussion about Monte-Carlo Hex.

References

1. Vadim V. Anshelevich. A hierarchical approach to computer hex. *Artificial Intelligence*, 134(1-2):101–120, 2002.
2. Cameron Browne. *Hex Strategy. Making the Right Connections*. A K PETERS, 2000.
3. B. Bruegmann. Monte Carlo Go. Technical Report, http://citeseer.ist.psu.edu/637134.html, 1993.
4. T. Cazenave. Playing the right atari. *ICGA Journal*, 30(1):35–42, March 2007.
5. T. Cazenave and B. Helmstetter. Combining tactical search and Monte-Carlo in the game of go. In *CIG'05*, pages 171–175, Colchester, UK, 2005.
6. T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Computer Games Workshop 2007*, pages 93–101, Amsterdam, The Netherlands, June 2007.
7. T. Cazenave and N. Jouandeau. A parallel monte-carlo tree search algorithm. In *Computers and Games*, LNCS, pages 72–80, Beijing, China, 2008. Springer.
8. T. Cazenave and A. Saffidine. Utilisation de la recherche arborescente Monte-Carlo au Hex. *Revue d'Intelligence Artificielle*, 29(2), 2009.
9. Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2008.
10. R. Coulom. Efficient selectivity and back-up operators in monte-carlo tree search. In *Computers and Games 2006*, Volume 4630 of LNCS, pages 72–83, Torino, Italy, 2006. Springer.
11. R. Coulom. Computing elo ratings of move patterns in the game of go. *ICGA Journal*, 31(1), March 2008.
12. S. Even and R. E. Tarjan. A combinatorial problem which is complete in polynomial space. *Journal of the Association for Computing Machinery*, 23, October 1976.
13. S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in monte-carlo go. Technical Report 6062, INRIA, 2006.
14. Sylvain Gelly, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, and Yann Kalemkarian. The parallelization of monte-carlo planning. In *ICINCO*, 2008.
15. Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML*, pages 273–280, 2007.
16. R. Hayward. A puzzling hex primer. In *Games of No Chance 3*, volume 56 of *MSRI Publications*, pages 151–161, 2009.
17. P. Henderson and R. Hayward. Probing the 4-3-2 edge template in hex. In H.J. van den Herik et al., editor, *CG 2008*, volume 5131 of *LNCS*, pages 229–240, Beijing, China, 2008. Springer-Verlag.
18. H. Kato and I. Takeuchi. Parallel monte-carlo tree search with simulation servers. In *13th Game Programming Workshop (GPW-08)*, November 2008.

19. L. Kocsis and C. Szepesvàri. Bandit based monte-carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.
20. T. Maarup. Hex everything you always wanted to know about hex but were afraid to ask. Master's thesis, University of Southern Denmark, 2005.
21. Stefan Reisch. Hex ist pspace-vollständig. *Acta Inf.*, 15:167–191, 1981.
22. Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-carlo tree search solver. In *Computers and Games*, volume 5131 of *LNCS*, pages 25–36. Springer, 2008.