EfficientPlayouts for the *Havannah* Abstract Board Game

Sandro Könnecke and Johannes Waldmann

Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fb IMN, PF 301166, 04251 Leipzig, Germany

Abstract. We describe efficient algorithms and data structures for applying Monte Carlo game tree search to the abstract board game *Havannah.* This game, invented by Christian Freeling in 1979, combines the themes of connection and capture (surrounding territory), thus it is somewhere between Hex and Go. We discuss

- how to find the winner of a playout efficiently,
- how to make a reasonable set of playouts.

To compute the winner, we use a Disjoint Set implementation for on-line recognition of connections, and a version of Dijkstra's algorithm in the (max,min)-semi-ring for off-line recognition of territories. For reasonable playouts, we describe how to select moves according to some probability distribution, and how to weigh results.

1 Introduction

The board game Havannah was invented by Christian Freeling in 1979 and published by Ravensburger in 1984. It is a two player, finite, zero sum game with complete information. Players alternately place stones on empty intersections on a hexagonal grid. Once placed, stones are not moved or removed. Havannah combines the themes of connection and territory, since it has three winning conditions: connect two corners, connect three edges, or surround territory.

Havannah partially derives from Hex in using a hexagonal grid and the theme of connection. The theme of territory links Havannah to Go. Still, Havannah keeps the "sudden death" theme of Hex: once a winning condition is reached, the game ends. Draws are possible (no winning condition and no valid move), but extremely rare.

The game's author claimed (2002) "Havannah is not programmable". Since then, there have been several attempts at writing Havannah computer players, but none of them produced convincing results. One immediate obstacle is that while each individual move is simple (place a stone), there is a large number of available moves (initially, hundreds), since all moves on empty intersections are valid. So, the situation is similar to that in computer go, where even the best available programs are easily out-played by medium strong amateurs.

But that is currently changing: the method of Monte Carlo game tree evaluation has been applied successfully in computer Go [GWMT06], and playing strength of top programs improved by about 10 kyu. The aim of the present paper is to describe efficient algorithms and data structures that can be used in applying the Monte Carlo framework to the game of Havannah.

After defining the game in Section 2, we briefly review Monte Carlo (MC) game tree search in Section 3. In the following sections, we discuss how to organize MC playouts: move selection by "weighted all-moves-as-first" in Section 4, abridged playouts in Section 5, and move selection according to a probability distribution in Section 6. Then, we turn to recognizing the winner of a playout: we describe an incremental ("on-line") algorithm for recognition of connections in Section 7 and an "off-line" algorithm for recognition of surrounded territory in Section 8. This algorithm uses Dijkstra's algorithm with weights in the (max,min)-semiring and is the main technical contribution of the present paper.

We have implemented these algorithms, and present some performance data in Section 9. The resulting program can be accessed from `http://dfa.imn.htwk-leipzig.de/havannah/`. It plays reasonably on small boards (size 4). For standard boards (size 8 and 10), the MC method does not seem applicable directly. We argue in Section 10 that it can be used for the analysis of sub-games. The combination of these analysis results is the subject of further study.

Acknowledgements. We are grateful to the following persons for helpful discussions on computer Havannah in general, and on preliminary versions of our algorithms in particular: Ingo Althöfer, Christian Freeling, Mirko Rahn, Karsten Weicker, and (anonymous) participants of the `littlegolem.net` forum.

2 The Game Havannah

For defining the rules of the game Havannah, we follow [Fra99] and assume familiarity with some basic concepts from graph theory. The board of side length s realizes the graph $H_s = (V_s, E_s)$ with

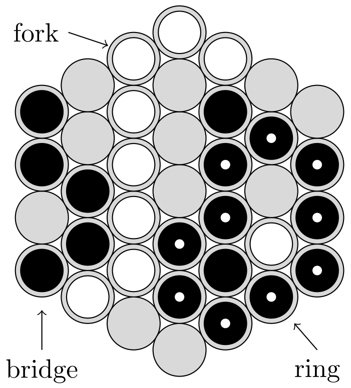
- vertices $V_s = \{x = (x_1, x_2, x_3) \mid -s < x_i < s, \sum x_i = 0\}$,
- edges $E_s = \{(x, y) \in V_s^2 \mid \operatorname{dist}(x, y) = 2\}$ where $\operatorname{dist}(x, y) = \sum |x_i - y_i|$.

In this graph, vertices of degree 3 or less are called *corner vertices*, vertices of degree 4 are called *side vertices*, and vertices of degree 6 are called *inner vertices*. Special cases are $s = 0$ (empty graph), $s = 1$ (one vertex which is a corner), and $s = 2$ (no side vertices).

For $s \geq 3$, the restriction of H_s to the side vertices has 6 connected components that are called *sides*. (We drop the subscript from H_s since s it will be clear from the context.)

A board *configuration* is a partial mapping from vertices to colours $b : V \hookrightarrow C$ where $C = \{\text{Black}, \text{White}\}$. For a configuration b, we call a set S of vertices a *chain* of colour c if $b(S) = \{c\}$ (all positions have colour c) and H restricted to S is connected. A board configuration is *won* for colour c if it contains a fork, bridge or ring of that colour, where a chain S of colour c is called

- *bridge* if it contains at least two corner vertices,
- *fork* if it intersects at least three different sides,
- *ring* if the removal of S from H creates a graph H' with at least one connected component that consists of inner vertices only. We say that the ring *captures* these vertices. Note that the captured vertices can have any colour, or none. In diagram, the positions belonging to the ring are marked.



The *move* relation $b_1 \rightarrow_{(p,c)} b_2$ holds if $p \notin \operatorname{dom} b_1$ and $b_2 = b_1 \cup \{(p,c)\}$, considering the mapping as a set of ordered pairs. Starting from the empty configuration, players move alternatingly until a won configuration is reached for either player (so the situation in the previous figure cannot occur in an actual game) or until there is no more valid move, resulting in a draw.

3 Monte Carlo Game Tree Search

We briefly review the Monte Carlo (MC) method [KS06] for estimating the game theoretic value of a position. This is applicable to the move relation $\rightarrow_{(p,c)}$ of any two person game.

The *state* of an MC evaluation is given by a prefix tree T of the full game tree F (their roots coincide and each path in T is a prefixes of a path in F).

An MC evaluation step ("playout") consists of

- choosing a path p in T ("in the tree"),
- choosing an extension q of p in F ("below the tree"),
- according to the result of $p \circ q$, updating annotations along p (that is, in T),
- extending T (by the first node of q)

Each node in T is annotated with numbers (w, n) where n is the number of playouts through this node so far, and w is the number winning playouts for the player that is to move in this node.

During playouts, in each node the most interesting move should be chosen. How this is done depends on whether the node is in T or not.

While playing in the tree, the standard UCT algorithm suggests to move to the child node with the highest *upper confidence bound* (UCB), which is computed as the win ratio's mean value plus standard deviation.

While the number of playouts is low, the deviation is large and results are unreliable. The method of "rapid action value estimation" [GS07] (RAVE) improves on this. It uses a map R that assigns to each pair (b, m) of node (in T) and possible move m, a pair of numbers (w', n'). If a playout chooses move m in position b' (on q), then for each node b on p, the information on the playout's result is recorded in $R(b, m)$. That is, the move m is considered "as first move" in position b. The move selection in the tree then uses a weighted combination of actual playout information and RAVE estimation. We describe in Sections 4 and 5 how to adapt this method for the “sudden death” feature of Havannah.

Below the tree, there is no prior information for move selection. So we turn to choosing moves randomly. If done literally, most of the playouts are actually meaningless (possible wins are not realized, leading to “false losses”; threats are not answered, leading to “false wins”). Some guidance for move generation is needed, but its implementation should be reasonably efficient. We describe in Section 6 how to implement move selection with reasonable probability distributions.

4 Weighted All Moves As First

As described in Section 3, the “all moves as first” heuristics is a means to extract more information from a playout (than just the won/lost/draw result).

Havannah has the feature (in common with Hex, but distinctive from Go) that the first occurence of a winning condition ends the whole game. Therefore, short playouts (that give a result after few moves) are more important than long playouts. For each playout, we compute a weight, where short playouts (immerdiate wins/losses) get heigher weight. When updating RAVE information in the move tables, we multiply the result of the playout (loss -1, draw 0, win 1) by the weight of the playout.

Note we favour “winning moves” no matter whether they were made by the player who is to move in the given node. This helps to play for a win, and helps to block winning moves made by the opponent. It is not clear whether this should be extended by the deprecation of “losing moves”.

It is a delicate matter to compute the weight. We use the reciprocal of the length of the playout, but in some cases this seems to lead to “short-sightedness” (it will play all local threats first and tends to forget about the global picture).

5 The Playout Horizon

We argued in Section 4 that short playouts are important, and give them large weights. Long playouts are unimportant, and a radical consequence is to ignore them altogether.

This can be realized as follows: for each node we maintain a value h as the “current playout horizon”. If a playout takes more than h moves, it stopped and discarded, and h is increased. If a playout ends (with win or loss) before h moves, then its value is used and then h is kept or decreased (even drastically: this playout’s length becomes the new horizon).

6 Move selection during playouts below the tree

For the selection of moves in playouts below the tree, we use some probability distribution that is computed from prior knowledge (in the tree) and updated by knowledge obtained during the playout.

We employ a data structure that realizes a map P from moves (keys) to weights (values) and provides efficient implementations of the following operations:

- to select randomly a key k with probability $P(k)/\sum\{P(k') \mid k' \in \operatorname{dom} P\}$,
- to insert, update and delete keys resp. their values.

When we start a playout below the tree, we load a probability distribution that corresponds to the RAVE values. In each step, we select a move according to the current distribution, and then remove it (it is no longer a valid move). For the neighbours of the selected move, we increase the probabilities in the distribution (drastically), since we want the playout to continue locally.

As an implementation, we work with a binary tree where each node n contains a key $K(n)$ and a value $V(n)$. Define $S(n)$ as the sum of the values of all nodes below n (including n). Then, selecting a node from a tree with root r can be done as follows: generate a random number x in the range $[0 \ldots S(r)]$ and find the unqiue node n such that for the sum l of valuess of nodes left of n in the pre-order enumeration we have $l < x \leq l + V(n)$. This can be realized in logarithmic time if the tree is balanced and each node is enriched with the $S(n)$ information.

```
**procedure** select (Tree $n$, number $x$)
    **if** $x < V(n)$
    **then** return $K(n)$
    **else if** $x < V(n) + S(n.\text{left})$
        **then** return select ($n$.left, $x - V(n)$)
        **else**  return select ($n$.right, $x - V(n) - S(n.\text{left})$)
```

In addition to the tree, we keep a (hash) map that maps keys to tree nodes. Doing so, we can drop any requirements on the ordering of the keys in the tree. This allows for easy (re)balancing. In particular, the tree can be "nearly balanced": starting from the root, each level l is full (contains 2^l nodes), and the last level is filled from the left. This is realized e.g. when using an array $a(1..m)$ for nodes, and the node at n has children at $2n$ and at $2n + 1$. We still have to keep the invariant that all $S(n)$ annotations are correct, so the operations of inserting, deleting and updating involve one or two walks to the root. This can be realized in logarithmic time. The interesting case is deletion: deleting the rightmost node (at index m) is easy (one walk to the root for adjusting total weights), and to delete any other node (at index $n < m$), we swap it with the rightmost one (thus, two walks: one from m and one from n).

7 Online Recognition of Bridges and Forks

By “online” we mean that the algorithm works incrementally: information on the current board position is kept in some data structure that is updated after each move. In fact we use two structures, one for each player. So we consider the recognition of bridges and forks for one each player separately.

Each players’ stones are seen as a set of connected subsets. We employ the standard idea of *Union/Find abstract data type* [CLR90]. On each move, a new singleton subset is created, which is then joined with the subsets corresponding to the adjacent stones of equal colour, if any.

```
procedure player $C$ moves on position $p$
    add new singleton set $S_p = \{p\}$ to $S$
    for each position $q$ adjacent to $p$
        if $S$ contains a set $S_q$ that contains $q$
        then unite the sets $S_p$ and $S_q$.
```

This can be implemented as *Disjoint Set Forest* with union-by-rank and path compression. Then the amortized run time of each operation grows slowly (inverse Ackermann), and is practically constant.

To check whether a bridge or fork is present, we enrich this data structure: for each subset of stones, we keep the set of corners and edges that it contains. Such sets can be represented efficiently as bit vectors. We need a bit width of 12 (six corners, six edges). On each “union” operation, the corresponding subsets are united as well. (The bitvectors in the corresponding roots of the disjoint set forests are combined with the “or” operation.) We use a lookup table (of size $2^{12} = 4096$) to check whether a winning condition was reached.

8 Offline Recognition of Rings

There does not seem to be an efficient on-line algorithm for recognition of rings, so we do the calculation off-line. Overall, this leads to the following procedure for an MC playout:

```
while (no bridge and no fork and horizon is not reached)
    select and play next move (see Section 6)
    check bridge/fork winning condition (see Section 7)
for each colour $C$
        find earliest ring for $C$, if any (this section)
```

The ring-find algorithm is executed for each colour C independently. When looking for rings in colour C, opponent’s stones are ignored, since it does not matter whether the positions surrounded by the ring are filled or not.

As input for ring-find, we use a mapping b from board positions to numbers where $b[p]$ is n if position p was played in move number n by player C, and $+\infty$ else.

For each position p on the board, the algorithm determines the move number $s[p]$ where this position was first surrounded by stones of C. If p never was surrounded, then $s[p] = +\infty$.

During the algorithm, we will also use a value $-\infty$, and the ordering on these values is $-\infty < 0 < 1 \ldots < +\infty$. The set $\mathbb{N} \cup \{-\infty, +\infty\}$ is a semi-ring with the operations of "maximum" for addition ($\oplus$) and "minimum" for multiplication ($\otimes$). The neutral element for $\oplus$ is $-\infty$, and the neutral element for $\otimes$ is $+\infty$.

We now assign weights to the directed edges of the Havannah graph: the weight $w[p, q]$ of edge (p, q) is $b[p]$ as defined above. We view w as a square matrix (row and column indexes are positions on the board). Matrix addition and multiplication are defined in the usual way (using the semi-ring operations on the elements). Since this semi-ring is idempotent, complete and ω-continuous, we can compute the reflexive and transitive closure of w by $w^* = w^0 \oplus w^1 \oplus w^2 \oplus \ldots$, cf. [Kui97].

Also, we use an initial weight vector v that assigns $+\infty$ to all positions on corners and edges, and $-\infty$ to all inner positions.

Proposition 1. $s = v \otimes w^*$.

Proof. By definition, $w^*[p, q]$ is the sum of the weights of all paths from p to q, where the weight of a path is the product of the weights of its edges. In particular, $(v \otimes w^*)[q]$ is the sum of the weights of all paths from a corner or edge position to q. Note that "sum" is "max" and "product" is "min". That is, for each path from the outside to q we compute the minimum of its edge weights, which is in fact the minimum over the numbers in the nodes on the path, excluding q. We then take the maximum over all these results. This is indeed what we want: if the minimum over a path is k, this means that at move k (and later), this path no longer connects q to the outside. The last connection to the outside is cut at the maximum over all those k. Thus indeed $s[q]$ gives the time at which q was captured. □

While the correctness proof was using paths, the implementation does not actually consider each path individually. Instead we compute the closure of the matrix w, applied to v, by Dijkstra's algorithm [Cle95].

Its input is matrix w as above. (Vector v will be computed implicitely.) The algorithm uses a priority queue M that realizes a partial mapping from positions to $\mathbb{N} \cup \{-\infty, +\infty\}$. For positions $p \notin \operatorname{dom} M$, we assume $M[q] = -\infty$. The output is the vector s (a partial map from positions to values that is empty initally).

```
for each position $p$ on corners and edges
        insert $M[p] = +\infty$
while $M$ is not empty
    remove an item $p$ with maximal weight $m$ from $M$
    set $s[p] = m$
    for each neighbour $q$ of $p$
        if $q \notin \operatorname{dom} s$
        then update $M[q] := M[q] \oplus (m \otimes w[p, q])$
```

The invariant of the main loop is that the values in M are a lower bound for the correct values, and the values in s are correct. This works since the semi-ring addition (max) is idempotent, and the ordering is natural: $x \geq y \iff x = x \oplus y$.

Finally, the minimal entry of s gives the earliest time of a capture. (If nothing was captured, all entries are $+\infty$.)

We can choose any reasonable implementation of the prioritiy queue abstract data type [CLR90]. For each edge of the graph, there is one update. For the Havannah board, the number of edges is linear in the number n of vertices (each has ≤ 6 neighbours). The total run time is $O(n \log n)$, so the amortized cost for a single move (in a full playout) is $\log n$. One drawback of this approach is that it always considers the whole board (abridged playouts do not reduce the total run time of the ring test).

9 Measurements

In this section, we present some run times for a test implementation of our algorithms. We used the Mono C# compiler version 124.55.0.0, the execution platform is Mono JIT compiler version 124455 under Debian GNU/Linux, running on (one core of) a 3 GHz Intel Xeon X5365 processor.

The following table gives average run time for one playout in milliseconds. In the weighted playouts (cf. Section 6), the weight distribution starts with unit weight on the center position, and after each move number n at position p, the neighbours of p get weight n^3. Bridge/fork checks (cf. Section 7) are done on-line (after each move), and the ring check (cf. Section 8) is done off-line (once, after last move). Full playouts fill every position on the board. The conclusion is that on board size 8, we can do 1000 playouts per second. Further speed improvement can be expected from playouts with restricted horizon (cf. Section 5).

board size	4	6	8	10
random full playout, not checking any winning condition	0.035	0.049	0.098	0.167
random full playout, with bridge/fork checking	0.15	0.37	0.67	1.15
random full playout, with bridge/fork/ring checking	0.41	1.00	1.9	3.2
weighted full playout, with b/f/r checking	0.56	1.38	2.6	4.5
weighted playout until b/f win, with ring checking	0.24	0.53	0.98	1.6

We cannot quantify the effects of the methods from Sections 4 and 5 on playing strength reliably. To our knowledge, there is no Havannah computer opponent available for comparison. We did some self-play experiments that confirmed that a program using weighted AMAF and a playout horizon, wins roughly 3/4 of its games against a program without these features, even when starting second. These experiments were done on a board of size 4. (Winning as second player on a small board is hard.) All UCT and RAVE parameters were equal for both programs. As the board size increases, the difference in strength becomes less obvious, basically because pure MC programs are missing strategic insight and play more and more random. This will be discussed in the following section.

10 Frames

For reasonable Havannah play on larger boards, plain MC game tree evaluation is not enough. The issue is “Monte Carlo Laziness” [Alt08]: on a large and nearly empty board, a plain MC game tree evaluation does not see any compelling line of play since all winning configurations are well behind its (noise) horizon. In short, MC is tactically strong, but strategically weak.

Programs need to know about “frames”, a basic concept of serious Havannah play. (The similar concept of “virtual connection” is known from the game of Hex [Bro00] and is being used in top Hex programs [Ans00].)

A *frame* is a set of positions occupied by one colour that are not necessarily connected, but whose connection cannot be prevented by the opponent. Typically, frames consist of “knight’s moves” (unbreakable connection between occupied positions) and “edge templates” (unbreakable connection to edges). The *width* of a frame is the number of moves that need to be added to make it into a chain. A *winning frame* is a frame whose underlying chain satisfies one of the three winning conditions (bridge, fork or ring).

A reasonable (human) game of Havannah can indeed by described as the fight for building a winning frame that is tighter (of lesser width) than the opponent’s. In particular, top human players agree that the fork is the strategic goal while bridge and ring are used as tactical threats.

To compute a stratig evaluation of the board, we therefore suggest to use “edge frames”: frames that are considered winning if they reach one specific edge e.

The value of a board configuration (for one player C and one edge e) can be described by a pair of numbers (z, w) meaning that it takes $z \in \mathbb{Z}$ moves to obtain a winning edge frame of width $w \in \mathbb{N}$, where $z < 0$ means that C’s opponent ignores $|z|$ moves, and $z > 0$ means that C can ignore $|z|$ opponent moves.

Of course these numbers are (“strategic”) idealizations as they will be influenced by local (“tactical”) battles. Monte Carlo game tree evaluation, using methods given in the present paper, can be applied to play out these battles and compute measures (z, n) w.r.t. each player C and edge e. Then these values can be combined, using methods from Combinatorial Game Theory, resulting in a value for the full board (w.r.t. the full winning condition). The implementation of this plan is left as future work.

11 Conclusion

In this paper we have shown efficient algorithms and data structures for fast playouts of Havannah games. These form the basis for Havannah computer players using the Monte Carlo approach.

We do not claim that the algorithms from this paper alone would be sufficient to obtain a program that plays reasonable moves on regular boards (size 8 or 10).

Indeed our test implementation plays nicely only on board size 4, but becomes more and more helpless on larger boards.

We discussed how our playout algorithms can be applied to answer questions about frames, thus allowing a higher level of Havannah "understanding" for computer programs.

We encourage other games researchers and programmers to take up the challenge of writing intelligent Havannah programs.

References

[Alt08] Ingo Althöfer. On the laziness of monte-carlo game tree search in non-tight situations. Technical report, Universität Jena, 2008. `http://www.althofer.de/mc-laziness.pdf`.

[Ans00] Vadim V. Anshelevich. The game of hex: An automatic theorem proving approach to game programming. In *AAAI/IAAI* [DBL00], pages 189–194.

[Bro00] Cameron Browne. *Hex Strategy*. A K Peters, 2000.

[Bro05] Cameron Browne. *Connection Games*. A K Peters, 2005.

[Brü93] Bernd Brügmann. Monte carlo go. Technical report, Max-Planck-Institute of Physics, Munich, 1993. `http://www.ideanest.com/vegos/MonteCarloGo.pdf`.

[Cle95] Kieran Clenaghan. Calculational graph algorithmics. Technical Report CS-R9518, Centrum voor Wiskunde en Informatica, 1995.

[CLR90] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[DBL00] *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*. AAAI Press / The MIT Press, 2000.

[Fra99] Frans Fraase. Formalisation of the game havannah. `http://www.iwriteiam.nl/Havannah.html`, 199?

[Fre98] Christian Freeling. Havannah (board game). Ravensburger, 198?

[FSS06] Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors. *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, volume 4212 of *Lecture Notes in Computer Science*. Springer, 2006.

[Gha07] Zoubin Ghahramani, editor. *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvalis, Oregon, USA, June 20-24, 2007*, volume 227 of *ACM International Conference Proceeding Series*. ACM, 2007.

[GS07] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In Ghahramani [Gha07], pages 273–280.

[GWMT06] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA, 2006.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Fürnkranz et al. [FSS06], pages 282–293.

[Kui97] Werner Kuich. Semirings and formal power series. In *Handbook of Formal Languages*, volume 1, pages 609–677. Springer, 1997.

[Saf08] Abdallah Saffidine. Utilisation d'uct au hex. `http://enslyon.free.fr/rapports/info/Abdallah_Saffidine_L3.pdf`, 2008.