EventHorizon: a Monte-Carlo Blackhole Player

Alan Ansell

March 3, 2019

1 Introduction

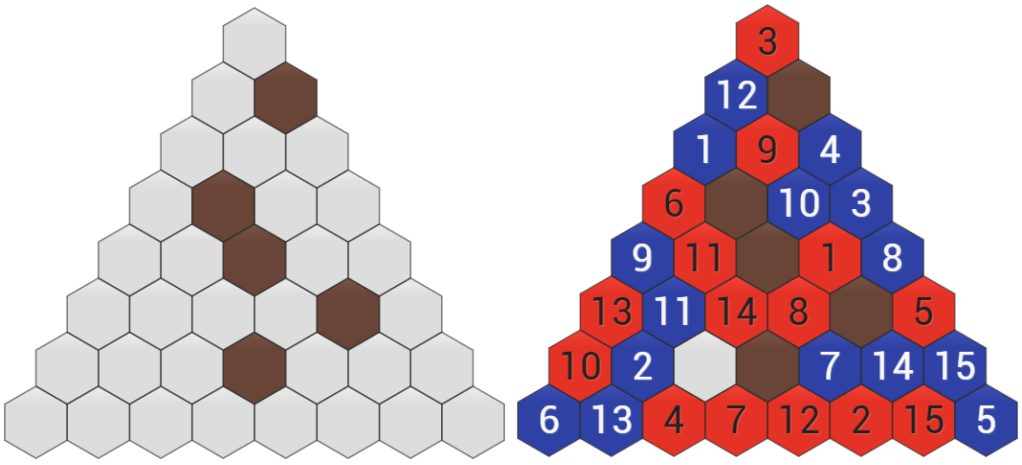
The game Blackhole was the subject of the 2018 edition of the CodeCup [1], an international programming contest. Competitors were required to write a program to play the game, and a tournament was then run to determine the strongest programs. This paper describes the algorithm used by EventHorizon, the 3rd placed program in the final competition, and subsequent modifications made to improve the program's playing strength.

I will first describe the rules of Blackhole and make a few observations about strategy. I will then describe UCT-RAVE, the search algorithm used in EventHorizon; AMAF priming, a modification to UCT-RAVE which was incorporated into EventHorizon after the CodeCup competition; and the domain specific knowledge exploited. Finally I will present some experimental results showing that after the modifications made since the competition, EventHorizon outperforms the winning entry of the CodeCup 2018.

2 Blackhole

2.1 Rules

Blackhole is played on a board consisting of 36 hexagonal cells, five of which are filled at random with a brown stone at the beginning of the game, as shown in Figure 1a. The two players, "blue" and "red," each start with 15 stones numbered 1 up to 15. They take turns placing a tile on an empty cell of the board. When all the stones have been placed, there is one remaining empty cell on the board, referred to as the "black hole." Each player attempts to maximise the sum of the values of their stones adjacent the black hole less the opponent's sum.



(a) A possible board state at the beginning of the game

(b) A possible board state at the end of the game

Figure 1: The Blackhole board

The scoring convention used in the CodeCup is as follows: let R be the sum of the values of red tiles adjacent the black hole, and B be the sum of values of blue tiles adjacent the black hole. Then red scores $75+R-B$ points and blue scores $75+B-R$ points. Note that the minimum and maximium scores a player can achieve are 0 and 150 respectively. For instance, in Figure 1b, $R = 4+7+14 = 25$ and $B = 2+11 = 13$, so red has won with a score of 87 to 63.

2.2 Notation

We can think of a game state as consisting of the following attributes:

- C, the set of all cells.
- $E \subset C$, the set of empty cells.
- $t \in \{\mathrm{r}, \mathrm{b}\}$, the player who is next to play.
- $H_r \subseteq \{1, 2, ..., 15\}$ and $H_b \subseteq \{-1, -2, ..., -15\}$, the set of stones not yet played by red and blue respectively.
- v_i, the "value" of cell i, $\forall i \in C$. $v_i = 0$ if $i \in E$ or cell i contains a brown stone. If cell i contains a red stone, then v_i is the value of that stone, and if cell i contains a blue stone, then v_i is the *negative* value of that stone.
- A_i is the set of cells adjacent to cell i, $\forall i \in C$.

We also define the *score* s_i of a cell i as the sum $\sum_{j \in A_i} v_j$ of its neighbours' values.

At the end of the game, E contains a single cell h, the black hole. According to the CodeCup system, red's final score is $75 + s_h$ and blue's is $75 - s_h$. However we will consider the final score simply as s_h. Thus a positive score denotes a win for red, a score of 0 a draw, and a negative score a win for blue.

2.3 Strategy

2.3.1 Reasonable and Unreasonable Moves

We will refer to a possible move in Blackhole as "unreasonable" if there exists an alternative move which we can prove is never worse and may in some circumstances be better than the move in consideration. We will now consider some circumstances in which we can deem a move unreasonable:

- Suppose the player to move t has decided to place a stone in an *isolated* cell i, i.e. $A_i \cap E = \emptyset$. Since playing in an isolated cell does not change the score of any other empty cell, it would be unreasonable for the player to place any of his stones in cell i except the least valuable remaining stone.
- Let I be the set of isolated empty cells. Since these cells are equivalent for all purposes other than their score, it would be unreasonable for player t to play in any of them other than the one whose score is least favourable for player t, i.e. minimal if t is red and maximal if t is blue.
- Suppose there are two adjacent cells c and d which are adjacent only to each other, i.e. $A_c = \{d\}$ and $A_d = \{c\}$. Playing in either cell would leave the other one isolated, so it could only be reasonable to play in the one with the less favourable score such that the more favourable one was left empty.

2.3.2 Targets and Dead Cells

The concept of a *target* enables us to make more sophisticated judgements about whether a move is reasonable or not. The target α is the minimum final score which red would be "happy" to achieve. We now define a new game *α-Blackhole*, which is identical to Blackhole except that it has only two outcomes: red wins if the final score is $\geq \alpha$, and blue wins otherwise. We call a position in a game of Blackhole *α-winnable* for red if the the result with perfect play of the corresponding position in α-Blackhole is a win for red, and we call it α-winnable for blue otherwise. Therefore the final score which would arise from perfect play in a given position is the greatest α for which the position is α-winnable for red.

We say that red *α-controls* an empty cell i if $s_i \geq \alpha$ and that blue α-controls i if $s_i < \alpha$. Note that any empty cell must be α-controlled by one of the players at any given time.

We call a cell i *α-dead* in favour of player p if p is guaranteed to α-control i in the final position. Note that an isolated cell is trivially α-dead in favour of whichever player α-controls it, since its score cannot change in the future.

We can determine whether a cell i is α-dead in favour of player p by finding the least favourable score $w_i^{(p)}$ that cell i can possibly have for player p in the final position and comparing it to α. This least favourable score occurs when all of the

empty cells adjacent to i are filled with the least favourable remaining stones:

$$\text{Let } S_i = \{B \subseteq H_b \cup H_r : |B| = |A_i \cap E|\}$$
$$w_i^{(r)} = s_i + \min_{B \in S_i} \Big\{ \sum_{j \in B} j \Big\}$$
$$w_i^{(b)} = s_i + \max_{B \in S_i} \Big\{ \sum_{j \in B} j \Big\}$$

Let $D_\alpha^{(p)}$ be the set of cells which are α-dead in favour of player p.

We can now make a powerful observation: if $|D_\alpha^{(p)}| > |H_q|$, the number of stones the opponent has left, then the position is α-winnable for player p. This is because player q does not have enough stones to fill all the cells which are α-dead in favour of player p. Hence, assuming player p avoids playing in any of the cells in $D_\alpha^{(p)}$, the black hole will be one of the cells in $D_\alpha^{(p)}$.

2.3.3 Stale Cells

Recall from Section 2.3.1 that it is only reasonable for player p to play a stone in an isolated cell i if i is the least favourable isolated cell for p and the stone is the least valuable stone available to p. The concept of a *stale* cell allows us to generalise this property of isolated cells to other cells. We say that a cell i is α-*stale* in favour of player p if i is α-dead in favour of player p and all the empty cells neighbouring i are α-dead in favour of either player. In this case, playing in cell i will not affect who will α-control any cell on the board in the final position, and so it is effectively isolated. Therefore in a game of α-Blackhole, it can only be reasonable for player p to place a stone in an α-stale cell i if the stone is player p's least valuable stone and i is the least favourable stale cell for p.

3 Monte-Carlo Tree Search and its Variants

3.0.1 Monte-Carlo Methods

Traditional search methods for game playing such as Minimax require a hand-crafted evaluation function which estimates the utility of a given position. This is not a problem for games such as Chess where there are many simple and effective evaluation heuristics. However in some games, notably Go, it has proved much more difficult to devise suitable heuristics. Blackhole is a fairly obscure game which has barely been played by humans, so it also has few readily available heuristics.

In the absence of effective heuristics, we must resort to the only ultimate source of game knowledge: the score in a terminal position. The fundamental principle of Monte-Carlo search is to perform many random "simulations" or "playouts." A simulation is a possible continuation of the game up until a terminal position is reached. The utility of a candidate move is estimated as the mean of the outcomes of simulations starting with that move.

3.0.2 Monte-Carlo Tree Search (MCTS)

MCTS [2] maintains a search tree in which nodes represent positions or *states* in the game and edges represent moves or *actions*. For each state we store a visit count $N(s)$, the number of times a simulation has visited s, and a value $Q(s)$, the mean outcome of all simulations which have visited s. We will also denote $Q(s,a)$ to be the mean simulation outcome when action a is performed in state s; that is $Q(s,a) = Q(t)$, where t is the state which arises when action a is performed in state s. Each MCTS simulation consists of four phases:

1. *Selection*: Starting at the root state s_0, MCTS applies a *tree policy* to select some sequence of actions leading to a leaf state s_l. The resulting path can be written as $s_0 a_0 a_1 s_1 ... a_{l-1} s_l$.
2. *Expansion*: All possible successor states of s_l are added to the search tree as its children.
3. *Playout*: One of the children of s_l is chosen to be the next state s_{l+1} of the simulation. A playout originating in state s_{l+1} is then carried out according to a *default policy*.
4. *Update*: The visit counts of states $s_0, ..., s_{l+1}$ are incremented and their values are updated according to the outcome of the playout.

3.0.3 Upper Confidence Bound for Trees (UCT)

A naive choice of tree policy would be to choose the next action in the simulation to be the action a of the current state s with the highest value $Q(s,a)$. This would work poorly because if the first playout from a leaf state resulted in a loss, it would be unlikely that that state would ever be visited again, even if a deeper analysis of the state would show it to be favourable. An effective tree policy must balance exploration and exploitation; the aforementioned greedy tree policy focuses entirely on exploitation and neglects exploration.

UCT [3] is a tree policy which encourages exploration by assigning each possible action a in the current state s a score $Q_*(s,a)$ calculated as

$$Q_*(s,a) = Q(s,a) + c_e\sqrt{\frac{\log N(s)}{N(s,a)}},$$

where c_e is an "exploration constant." The second term of this expression encourages the search to consider children which have been less visited frequently.

3.0.4 Rapid Action Value Estimate (RAVE)

One weakness of UCT is that when a position has not yet been explored, there is no better way of choosing which child to visit next other than random selection. RAVE [4] is a way of calculating a *prior* for unvisited moves. The fundamental assumption of RAVE is that moves which are good in a certain position are likely to be good in other similar positions. Let $\bar{N}(s,a)$ be the number of simulations in the subtree of state s in which action a has been chosen, and the *action value* $\bar{Q}(s,a)$ be the mean outcome of these simulations. RAVE assigns each candidate action a a score by taking a convex combination of its simulation value $Q(s,a)$ and its action value $\bar{Q}(s,a)$:

$$Q_*(s,a) = (1-\beta(s,a))Q(s,a) + \beta(s,a)\bar{Q}(s,a),$$

where $\beta(s,a)$ weights the relative importance of the RAVE prior and simulation results and is calculated as:

$$\frac{\bar{N}(s,a)}{N(s,a) + \bar{N}(s,a) + c_r N(s,a)\bar{N}(s,a)}$$

EventHorizon uses a hybrid of UCT and RAVE in which the final score assigned to a move is the RAVE score plus a UCT exploration bonus:

$$Q_*(s,a) = (1-\beta(s,a))Q(s,a) + \beta(s,a)\bar{Q}(s,a) + c_e\sqrt{\frac{\log N(s)}{N(s,a)}}.$$

EventHorizon also uses a RAVE "horizon," whereby the action count $\bar{N}(s,a)$ and value $\bar{Q}(s,a)$ are affected only when a is chosen in the subtree of s at a depth $\leq h$ for some horizon depth h. This is because positions at a depth greater than h are likely to be too dissimilar to s to contribute accurate estimates of the action value.

3.1 AMAF Priming

After the final CodeCup competition, I implemented a modification to RAVE which I called "AMAF priming." At the time I was not aware that others had tried similar modifications, but AMAF priming appears similar to "Generalized Rapid Action Value Estimation" (GRAVE) [5].

One drawback of RAVE is that it only exploits AMAF statistics from the subtree of the node at which selection is taking place. When the subtree is small, these statistics are based on a small sample size. It would be preferable to share AMAF statistics with related nodes with a higher visit count when the subtree of the current node is small. The idea of AMAF priming is to use the parent's AMAF statistics if the current node's visit count is less than a certain threshold. GRAVE on the other hand uses the AMAF statistics of the deepest ancestor node whose visit count is above a threshold.

4 EventHorizon's Domain Knowledge

4.1 Targets and Reasonable Moves

Rather than carrying out a single Monte-Carlo Tree Search, EventHorizon carries out one for each of several target values: that is, it carries out an MCTS for α-Blackhole for a range of α values. We will define $N^{(\alpha)}(s,a)$ and $Q^{(\alpha)}(s,a)$ to be the visit count and mean simulation value of action a in state s in the search tree for α-Blackhole.

In its search tree for α-Blackhole, EventHorizon only considers α-reasonable moves, as defined in Section 2.3. EventHorizon also utilises the observation that player p is certain to win with optimal play if the number of cells which are α-dead in p's favour exceeds the number of stones not yet placed by the opponent by terminating the search in such positions. These two pruning techniques entail an enormous reduction in the search space, especially later in the game. EventHorizon has a "solver" which uses dynamic programming with a move ordering heuristic and the above optimisations to attempt to solve positions after move 12 for optimal play. The solver is often successful in solving positions with 19 empty cells within 0.1 seconds despite there being $19! \times (9!)^2 \approx 1.6 \times 10^{28}$ legal continuations from such a position.

EventHorizon's move selection process (Algorithm 1) in a state s consists of two steps: first, selecting an appropriate target α_m, and second, selecting a good move by searching the tree for α_m-Blackhole more deeply. α_m is chosen to be the most optimistic α for which the MCTS indicates that $Q^{(\alpha)}(s) \geq \gamma_m$; or in other words, the most optimistic α which EventHorizon thinks it has a probability $\geq \gamma_m$ of achieving for some γ_m.

Algorithm 1 EventHorizon's move selection procedure

function SIMULATE($T^{(\alpha)}$, α)
  Augment MCTS tree $T^{(\alpha)}$ with a single simulation of α-Blackhole and update $N^{(\alpha)}$ and $Q^{(\alpha)}$ accordingly.
end function

function SELECT(s, α_{init}, t_1, t_2)
  $\alpha \leftarrow \alpha_{init}$
  initialise $T^{(\alpha)}$ to be an empty search tree rooted at s $\forall \alpha$
  repeat
    SIMULATE($T^{(\alpha)}$, α)
    if $Q^{(\alpha)}(s) \geq \gamma_i$ **then**
      $\alpha \leftarrow \alpha + 1$
    else if $Q^{(\alpha)}(s) \leq \gamma_d$ **then**
      $\alpha \leftarrow \alpha - 1$
    end if
  until t_1 seconds have elapsed

  $\alpha_m \leftarrow \max\{\alpha : Q^{(\alpha)}(s) \geq \gamma_m\}$
  repeat
    SIMULATE($T^{(\alpha_m)}$, α_m)
  until t_2 seconds have elapsed
  return $\text{argmax}_a N^{(\alpha_m)}(s, a)$
end function

4.2 Default Policy

Instead of choosing purely random moves during the playout phase, EventHorizon uses some heuristics to try and produce higher-quality simulations. Playout moves are chosen in two phases:

1. The cell to play in is chosen. Each empty cell is entered into a lottery, receiving a number of tickets equal to the number of non-α-dead neighbours it has, with a 1 ticket penalty if the cell itself is α-dead in favour of the player and a 1 ticket bonus if the cell is α-dead in favour of the opponent. The reasoning behind this is that the more neighbours a cell has, the more influential it is to place a stone in that cell, but dead cells are permanently under the control of one player, so they should be discounted. A ticket is drawn at random from the lottery and the cell with that ticket is selected.
2. The stone to use is chosen. If the chosen cell is α-stale, then the player's least valuable remaining stone is chosen. Otherwise, the player's remaining stones are entered into a lottery, receiving a number of tickets equal to the number on the stone. This is because it is usually best for a player to use their more valuable stones earlier in the game, since they are more influential when the mean number of empty neighbours per cell is higher.

5 Results

I will present the results of two versions of EventHorizon: the version which competed in the final CodeCup competition, and EventHorizon 5.0, a subsequent version with some bug fixes and AMAF priming implemented. The CodeCup version placed

third in the competition overall, but had a positive score against every individual opponent except *wormhole*, the first place winning program written by Abdessamad El Kasimi [6], which was made open source after the competition.

Version	Expected score
CodeCup	74.85
5.0 (no AMAF priming)	75.35
5.0 (AMAF priming)	75.54

Table 1: Performance of EventHorizon against wormhole.

Table 1 shows the performance of EventHorizon when playing against wormhole under conditions similar to CodeCup competitions. Each match consisted of 4,000 games, with both programs playing an equal number of games as red and blue. The score shown was the mean number of points scored out of 150 across the 4,000 games - a score of 75 would indicate that the two players were equal. The results show that the CodeCup version of EventHorizon was slightly weaker than wormhole, as expected on the basis of the competition results. The bug-fixed version was somewhat stronger than wormhole, and the AMAF priming improvement was worth an additional 0.19 points per game.

References

[1] Dutch National Olympiad in Informatics. *CodeCup 2018*. 2018. URL: `http://archive.codecup.nl/2018/32/competition_qcomp_e219.html`.

[2] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *5th International Conference on Computer and Games*. May 2006, pp. 72–83.

[3] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Machine Learning: ECML*. Sept. 2006, pp. 282–293.

[4] Sylvain Gelly and David Silver. "Monte-Carlo tree search and rapid action value estimation in computer Go". In: *Artificial Intelligence* 175.11 (July 2011), pp. 1856–1875.

[5] Tristan Cazenave. "Generalized rapid action value estimation". In: *IJCAI'15 Proceedings of the 24th International Conference on Artificial Intelligence*. July 2015, pp. 754–760.

[6] Abdessamad El Kasimi. *blackhole*. 2018. URL: `https://github.com/elkasimi/blackhole`.