Universiteit Leiden
The Netherlands

Opleiding Informatica

Determining game values of Hackenbush with Monte Carlo Tree Search

Esther Koene

Supervisors:
Walter Kosters & Luc Edixhoven

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 24, 2024

Abstract

The research described in this thesis aims to discover to what extent two variants of the Monte Carlo algorithm, namely the Pure Monte Carlo algorithm and the UCT algorithm, can determine the game values of game positions of the mathematical game Hackenbush. Additionally, Pure Monte Carlo is applied to determine game positions' outcome classes of a new Hackenbush variant: Multiple move Hackenbush.

To test the effectiveness of the two algorithm variants, they are used on various game positions with increasingly smaller positive game values. The variants are also used on more complex Hackenbush positions, like one whose game value is NP-complete to determine. In order to determine the game value, we present and apply the Monte Carlo Scale Method. Using this setup, we have determined that Monte Carlo Tree Search can differentiate game values with a precision of 1/1024 in Classic Hackenbush. Using Pure Monte Carlo, we have determined the outcome classes of some game positions of Two Move Hackenbush with various game values with 100% accuracy.

Contents

1 Introduction

Since its first use as an algorithm for artificially intelligent players in Go [KS06], Monte Carlo Tree Search (MCTS) has been used extensively to develop game-playing bots or solve sequential decision problems. Because of its simple yet effective properties, MCTS can be adapted to work well in different fields and for varying purposes. The possibilities are endless, and new variants can be employed widely, extending their original purpose. For these reasons, the algorithm is still relevant and actively researched.

Hackenbush is a simple two-player game that has been used to demonstrate concepts in combinatorial game theory [BCG04]. Its game positions are used to construct values of the surreal number system, making the game an interesting visualisation and research tool for mathematicians. In Classic Hackenbush, two players remove an edge of their colour each turn from a game field like the one shown in Figure 1. A player loses when they cannot move anymore. Different variants of Hackenbush exist, and we introduce a new variant: Multiple Move Hackenbush. In this variant, players have to remove multiple edges per turn. For example, in Two Move Hackenbush, a player has to remove two edges per turn. The game values of this new variant include so-called Nimbers, just like the game values of Red-Blue-Green Hackenbush. Interestingly, determining the game values of a type of position in Classic Hackenbush called *Redwood Furniture* has been proven to be NP-complete, meaning that the time needed to determine the game value grows exponentially with the number of edges [BCG04]. Improving methods to solve NP-complete problems is a whole field of research in itself, and MCTS or a variation thereof might prove to be a useful tool in this challenging task.

This bachelor thesis, supervised by Walter Kosters and Luc Edixhoven of LIACS, aims to apply MCTS to Hackenbush variants and report its effectivenes.

This thesis is structured as follows. In Section 2, both Hackenbush and Monte Carlo Tree Search are explained in more detail. Section 3 describes Multiple-move Hackenbush using some example game positions and their values. Then, Section 4 explains the methods used and experiments executed to acquire the results shown in Section 5. Section 6 reports the conclusion and mentions possible further research.

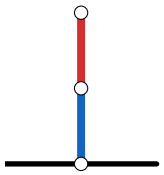


Figure 1: A Blue-Red Hackenbush game.

2 Background

The following sections will explain Hackenbush and Monte Carlo Tree Search in further detail.

2.1 Hackenbush

Hackenbush is a combinatorial game [ANW19]. It starts with a position like the one shown in Figure 1. Each turn, a player has to remove an edge of their colour, red or blue. We will refer to the players as Blue and Red. After a turn, any edges that are no longer connected to the ground (which is the black line in the figure) are removed as well. If a player cannot remove an edge, they lose.

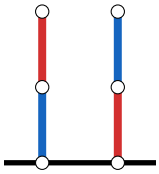


Figure 2: A Blue-Red Hackenbush game.

We are interested in the outcomes of different game positions, and we can categorise these positions into four outcome classes: the first player wins, the second player wins, Blue wins regardless of who moves first or Red wins regardless of who moves first (see Table 1). To illustrate: Figure 1 shows a position where Blue always wins, and Figure 2 shows a position where the second player wins.

Class	Name	Definition
$\mathcal{N}$	Fuzzy	The first player wins
$\mathcal{P}$	Zero	The second player wins
$\mathcal{L}$	Positive	Blue wins regardless of who moves first
$\mathcal{R}$	Negative	Red wins regardless of who moves first

Table 1: The four outcome classes of combinatorial games.

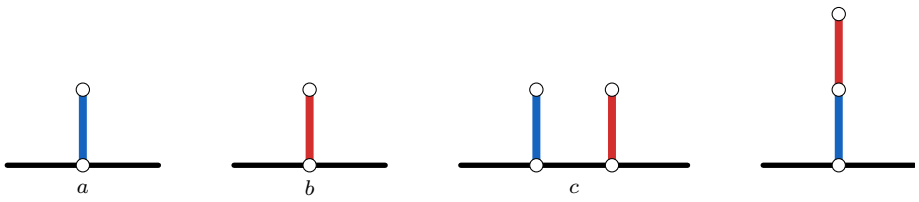


Figure 3: Simple Hackenbush positions with values 1, -1, 0 and $\frac{1}{2}$ from left to right.

We can give numerical values to game positions to quantify each player's advantage. Looking at Figure 3a for example, Left has a one move advantage, so the game value is 1 (outcome class $\mathcal{L}$). In Figure 3b, Red has a one move advantage, so the game value is -1 (outcome class $\mathcal{R}$). We can also define a game position G by its options $G = \{\mathcal{G}^L|\mathcal{G}^R\}$, where $\mathcal{G}^L$ and $\mathcal{G}^R$ are the set of options for Blue and Red, respectively. Position c in Figure 3 can then be described as $\{-1|1\}$: after Left moves, position b remains, while a remains if Red moves. This symmetrical position has game value 0 (outcome class $\mathcal{P}$). For the sets of options, we only consider the best ones. For example, if Red can either play to a position of value 1 or one of value -1, only the position with value

-1 will be considered. The game values of finite Blue-Red Hackenbush positions are *dyadic rationals*, rational numbers whose denominators are a power of 2 [ANW19]. To determine the value of a position $G = \{\mathcal{G}^L|\mathcal{G}^R\}$, we find the simplest number between $\mathcal{G}^L$ and $\mathcal{G}^R$. The definition of the simplest number n is as follows:

- If there are integer(s) between $\mathcal{G}^L$ and $\mathcal{G}^R$, then n is the one that is smallest in absolute value.
- Otherwise, n is the number of the form $i/2^j$ between $\mathcal{G}^L$ and $\mathcal{G}^R$ for which j is minimal.

For example, the value of $G = \{0|1\}$ is $\frac{1}{2}$ (game value of the position in Figure 3d), and the value of $G = \{3 \mid 7\frac{1}{2}\}$ is 4.
Different Hackenbush game positions can be added together to form a new game position. The game value of the new game position is the sum of the values of the game positions that it consists of. Reversely, game positions can be broken up into components when groups of edges are not interconnected. Then, the value of the total position can be determined by establishing the values of its components. To illustrate: the position in Figure 2 consists of two components: the left and the right stack of edges. The value of the left stack is $\frac{1}{2}$, and the value of the right stack is $-\frac{1}{2}$. Therefore, the value of the whole position is $\frac{1}{2} - \frac{1}{2} = 0$.
Blue-Red Hackenbush with an infinite number of edges has been used to construct ordinals. While the values of Blue-Red Hackenbush can be expressed with real numbers, some game values of other variants cannot. This is where surreal numbers come into play; Red-Blue-Green Hackenbush can be used to construct *star* $(*)$ and all other so-called *Nimbers* [ANW19], as described in the next section.

2.1.1 Nimbers

Figure 4 shows a game position of Red-Blue-Green Hackenbush with a single green edge. Either player can remove the green edge and win, so this position can be written as $\{0|0\}$. This is the combinatorial game value $*$ (pronounced *star*) [BCG04]. Its outcome class is first player wins, or $\mathcal{N}$.

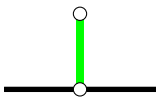


Figure 4: Simple Red-Blue-Green Hackenbush position with game value $*$.

If we add a second green edge on top of the first one, we get a position with value $\{0, *|0, *\}$: each player can play to either a zero or a $*$ position. The value of this situation has been dubbed $*2$ (pronounced *star two*). Continuing this trend, a stack of n green edges has value $*n$. Stacks of green edges are identical in game value to heaps in the game Nim: a Nim heap of size n has the *Nimber* $*n$.
Another way to describe a game position is with its *birthday*. The birthday of a game $G = \{\mathcal{G}^L \mid \mathcal{G}^R\}$ can be defined recursively as follows: it is the maximum birthday of any game in $\mathcal{G}^L \cup \mathcal{G}^R$ plus 1.

The base case is that if $\mathcal{G}^L = \mathcal{G}^R = \emptyset$, then the birthday of G is 0. Furthermore, a game is *born by day* n if its birthday is less than or equal to n. The only game born on day 0 is the game $0 \stackrel{\text{def}}{=} \{ \mid \}$. The four games born by day one are:

$$\begin{aligned} 0 &\stackrel{\text{def}}{=} \{ \mid \} \\ 1 &\stackrel{\text{def}}{=} \{0 \mid \} \\ -1 &\stackrel{\text{def}}{=} \{ \mid 0\} \\ * &\stackrel{\text{def}}{=} \{0 \mid 0\} \end{aligned}$$

2.1.2 Determining game values

There is a simple method to determine the game value of a single stalk of red and blue edges in Classic Hackenbush [ANW19]. First, we assign the value 1 to edges along the stalk until the first edge of the other colour is encountered. Then, we divide the value by 2 with each new edge. The sign of each edge depends on its colour. For example, the value of the stalk in Figure 5 is

$$1 + 1 - \frac{1}{2} + \frac{1}{4} + \frac{1}{8} - \frac{1}{16} + \frac{1}{32} = 1\frac{27}{32}$$

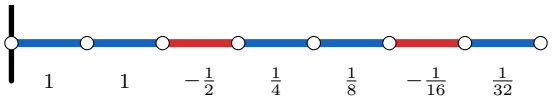


Figure 5: Value of a LR-Hackenbush stalk.

To determine the game value of any Hackenbush position, the simplest method is to analyse all the sub-games recursively. In this way, we find the value in the format $\{\ldots \mid \ldots\}$. Then, the definition for the simplest number is used to find the game value. Even though the method is simple, the number of subgames that need to be analysed can be enormous, growing exponentially with the number of edges in the position.

2.2 Redwood Furniture

Redwood Furniture forms a class of Hackenbush positions which have the following properties: the red edges do not touch the ground, and each blue edge (called a *foot*) touches the ground with one end and a unique red edge (called a *leg*) with the other end [BCG04]. The value of a piece of Redwood Furniture is $\frac{1}{2^n}$ for some $n = 0, 1, 2, \ldots$. Figure 6 shows a *Redwood Bed*. This is a piece of Redwood Furniture in which all red edges other than the legs each have just one end at the top of a leg. In this case, its value will be of the form $\frac{1}{2^n}T$, where T (called the Redwood Tree) is a subset of edges of the bed where removing any red edge would disconnect the picture. To determine the game values of positions of this class, one must determine the smallest Redwood Tree in the bed which contains all its legs. This problem is *NP-complete*, so the determination of the game values of Redwood Beds is *NP-complete*.

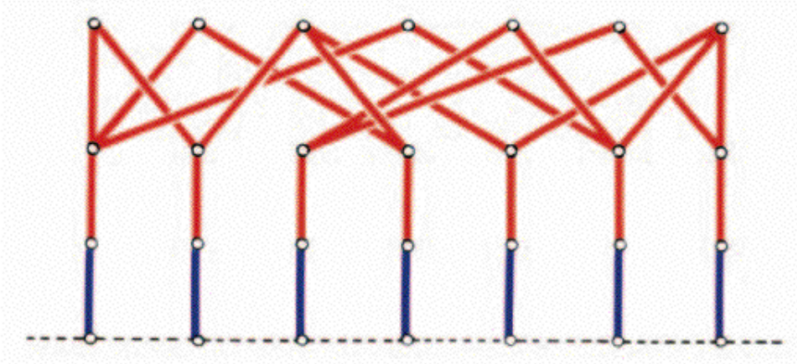


Figure 6: A Redwood Bed HACKENBUSH game position [BCG04].

2.3 Monte Carlo Tree Search

For games, *Monte Carlo Tree Search* (MCTS) has become a state-of-the-art technique [BPW+12]. In the context of games, the algorithm searches the game tree for optimal moves by balancing exploration and exploitation. Characteristic of the method is that it relies on repeated random sampling, using randomness to solve a deterministic problem. For more complex problems, applying MCTS successfully often requires modification or combination with other methods. The intent of research regarding MCTS often is to determine how the basic algorithm can be adjusted and enhanced to suit each specific situation or domain, and how variations and enhancements for one objective can be applied more widely.
The basic algorithm works as follows: within a predefined *computational budget*, a search tree is iteratively built. Then, the action that performed the best is returned. A game state is represented by a node in the tree, and the directed links between nodes are actions leading from state to state. Per iteration of the algorithm, four steps are applied:

1. *Selection*: starting at the root node, a *selection policy* is used to select child nodes at each level, descending through the tree, eventually reaching the most urgent expandable node.

2. *Expansion*: child nodes are added to the node reached in the selection phase. The child nodes correspond to game states reached by actions available in the selected node.

3. *Simulation*: to get the outcome of the new node(s), a complete random simulation of the game/problem is run. This is the "Monte Carlo" part of the algorithm.

4. *Backpropagation*: updates the statistics of the selected nodes by "backing up" the result of the simulation.

Figure 7 shows one iteration of the MCTS algorithm.

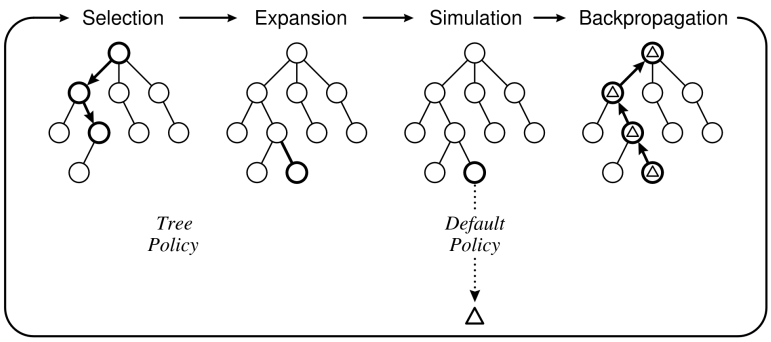


Figure 7: The four phases of Monte Carlo Tree Search [BPW+12].

2.3.1 The UCT Algorithm

How a search tree is built depends on which child nodes are selected during the selection phase. To maintain a proper balance between the exploration of actions that have not been tested well and the exploitation of the best actions found so far, a tree policy is chosen. The most common one used in MCTS is the *Upper Confidence Bounds applied for Trees* (UCT) [BPW+12]. This algorithm is efficient, simple and ensures that the growth of regret is within a constant factor of the best possible bound. The choice is modelled as an independent multi-armed bandit problem: the properties of each node are only partially known at the start of the iterations and become better known as more iterations are completed. The iterations consist of selecting one of multiple choices (i.e. arms). In this case, a child node j is selected to maximise

$$UCT = \overline{X}_j + C_p\sqrt{\frac{2\ \ln\ n}{n_j}}$$

where $\overline{X}_j$ is the average reward for child j (e.g. the win rate), $C_p > 0$ is constant, n is the visit count of the current (parent) node, and n_j is the visit count of child j. Generally, when $n_j = 0$, $UCT = \infty$, ensuring that nodes that have not yet been visited will have priority over all other options. The fact that every child node has a non-zero probability of selection is essential, given the random nature of playouts. Different types of play are explored because even children with a low reward are guaranteed to be chosen eventually (given that the computational budget does not run out before). Furthermore, the values of $\overline{X}_j$ are understood to be within $[0, 1]$ [BPW+12]. The term $\overline{X}_j$ encourages exploitation of child nodes with more promising rewards, while the term $\sqrt{\frac{2\ \ln\ n}{n_j}}$ encourages the exploration of less-visited nodes.
In UCT, C_p is the exploration term; it can be adjusted to increase or lower the amount of exploration that the algorithm allows. The value $\sqrt{2}$ has been proven effective for C_p, and is commonly used with rewards in the range $[0, 1]$. When using rewards outside this range, other values for C_p may be needed.

3 Multiple Move Hackenbush

In the multiple move variant of the game HACKENBUSH, the player has to choose multiple moves from their current move set each turn, which are then executed simultaneously. If there are fewer moves in the player's move set than the number specified before the game, the player loses. Firstly, we look at TWO MOVE HACKENBUSH. In this variant, each player has to execute two moves during their turn.

Figure 8 shows a few positions with a game value of 0 in TWO MOVE HACKENBUSH. Blue has the set of available edges $\{a\}$ in the first and third position, and the empty set in the middle position. Red has the set of available edges $\{1\}$ in the second and third position, and the empty set in the first position. Because the number of available edges is smaller than the required number of two moves, neither player can move. The game value is $\{\ |\ \}$, which is also expressed as 0. The outcome class of all these positions is $\mathcal{P}$: the first player loses.

In both game positions shown in Figure 9, Blue can move once and Red cannot move, so these positions have game value 1.

Two games of TWO MOVE HACKENBUSH are not additive: a single blue edge has value 0, but two blue edges next to each other have value 1 (and $0 + 0 \neq 1$).

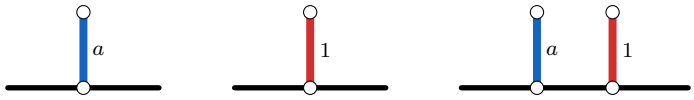


Figure 8: Simple HACKENBUSH positions with combinatorial game value 0 in TWO MOVE HACKENBUSH.

Figure 9: Simple HACKENBUSH positions with combinatorial game value 1 in TWO MOVE HACKENBUSH.

Figure 10 shows two positions where the next position has game value 0, regardless of whether Red or Blue starts. This can be written as $\{0|0\}$, which is the combinatorial game value $*$. Whoever starts is the winner, so this position is of class $\mathcal{N}$, or Fuzzy. In CLASSIC HACKENBUSH, such a position does not exist. As we've seen, in RED-BLUE-GREEN HACKENBUSH, the position in Figure 10 would be comparable to a single green edge (Figure 4). Figure 11 shows a game position that has value $*2$ in TWO MOVE HACKENBUSH.

Figure 10: Two HACKENBUSH positions with combinatorial game value $*$ in TWO MOVE HACKENBUSH.

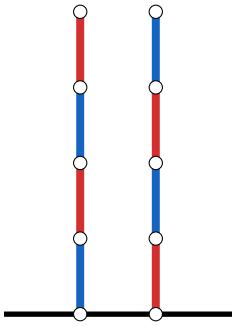


Figure 11: HACKENBUSH position with combinatorial game value $*2$ in TWO MOVE HACKENBUSH.

By definition, a game value G is *infinitesimal* if $-n < G < n$ for all positive numbers n. An example of such an infinitesimal value is $*$. It is incomparable with 0, and $* = -*$. Two more infinitesimal values are $\uparrow$ (pronounced *up*) and $\downarrow$ (pronounced *down*), which correspond to the game states $\{0|*\}$ and $\{*|0\}$ respectively. Both are infinitely small, but $\uparrow$ is positive, while $\downarrow$ is negative. Just like numbers, these infinitesimals can be added up. For example, the value of the game in Figure 12 is $\uparrow + *$, or $\uparrow *$ for short. In this position, Blue has two choices (considering only the most advantageous moves): either they play to the position 0, or to the position $*$. Because 0 and $*$ have the same birthday, both options are written in the game state: $\{0, *|0\}$. The outcome class of this position is $\mathcal{N}$.
In Figure 13, the most advantageous move for Blue is to remove the two edges in the stalk to the left, leaving a position with game value 1. Red has no other choice than to remove the two red edges, also leaving a position with game value 1. This means that the position is of the form $\{1|1\}$, which has the game value $1 + *$, or $1*$ for short.

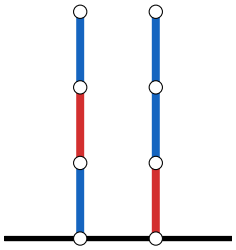


Figure 12: HACKENBUSH position with combinatorial game value $\uparrow *$ in TWO MOVE HACKENBUSH.

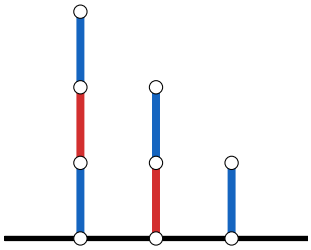


Figure 13: HACKENBUSH position with combinatorial game value $1*$ in TWO MOVE HACKENBUSH.

4 Method

In order to determine the outcome class of a game position, two Monte Carlo players are set up to play against each other. The experiment is done twice: once with Blue as the starting player and once with Red as the starting player. Then, the win rates are used to decide the outcome class: if the win rates for both starting players are 0%, the outcome class is $\mathcal{P}$ (second player wins). If the win rates for both players are 100%, the outcome class is $\mathcal{N}$ (first player wins). If the win rates for Blue and Red are 100% and 0% respectively, the game is classified as $\mathcal{L}$. The classification of the game position is $\mathcal{R}$ if the rates are inverted. If the win rates are between 0% and 100%, they are compared to a margin. If a win rate is lower than the margin, it represents a 0% win rate. If it is higher, it represents a 100% win rate.
As seen in Section 2.1, when two disjoint games of HACKENBUSH are added together to form a new position, the game values of the two original positions can be summed up to determine the game value of the new game position. This property, together with the determination of the outcome class, is used to determine the game value in a method called the HACKENBUSH Scale Method. This method is called the Scale Method because repeatedly adding edges to the position until the outcome class is $\mathcal{P}$ (which can be considered a balanced position) can be compared to determining an object's weight using a traditional scale with two dishes.

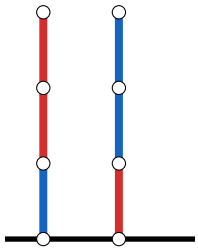


Figure 14: HACKENBUSH position with combinatorial game value 0 in CLASSIC HACKENBUSH, consisting of two compontents: a stalk with value 1/4 on the left, and a stalk with value −1/4 on the right.

Figure 14 shows an example of a game balanced using this method. The value of the component on the left is unknown, but the value of the component on the right (the so-called *weighing stalk*) is known. When the outcome of the game played by the Monte Carlo agents is $\mathcal{P}$, the game is balanced and we can infer that the unknown value is the negative of the value of the weighing stalk. The game value of this weighing stalk is known: it can be calculated using the properties of a HACKENBUSH stalk as explained in Section 2.1.2.
In the function MCGAME (see the pseudocode on the next page), a HACKENBUSH game is played by two agents. Each turn, the best move is determined using a variant of the Monte Carlo algorithm, and this move is carried out. In the weighing stalk, the algorithm only tests "smart" moves; we know that in a stalk of edges of one colour, removing the top edge is the best move. To increase the speed and win rate of the algorithm, only these moves are considered. If the starting player wins the game, the function returns `True`.
The Monte Carlo agents show imperfect play, causing Red to win games where Blue should theoretically win, and the other way around. To quantify the agents' success, whole games are repeated (the number of playouts), and their win rates are used. A margin is used to decide in favour of which player the game position actually was (or if the game was balanced). Through experimentation in the development phase, a margin of 0.4 has been proven successful. The results of an experiment that was done to choose the margin is shown in Figure 20 in Section B. This figure shows two graphs of the win rates of Blue when Blue started the game and Red when Red started a game, set out against the number of edges added to the game. The game value of these particular positions was −1/8, so when four edges were added (one blue edge and three red edges), the game was balanced. The first graph shows that for this number of edges, the win rate of Blue falls just below 0.4. The second graph shows that the win rate of Red rises to a little below 0.4 for the right number of added edges. Assuming that the positions used in this experiment represent other HACKENBUSH positions well, 0.4 is an effective margin.
To determine the actual game value of positions with 100% accuracy to verify the outcomes of the experiments, a brute force method that analyses all subgames recursively was used [Tom11]. The Scale Method has been implemented using a combination of the aforementioned code, an implementation of Monte Carlo Tree Search [Mic20] and own code.

noend 1 The HACKENBUSH Scale Method

```
function DETERMINEOUTCOMECLASS(GameState)
    for n = 1, ..., Playouts do
        if MCGAME(Blue) then
            BlueWins = BlueWins + 1
        if MCGAME(Red) then
            RedWins = Redwins + 1
    BlueWinRate ← BlueWins/Playouts
    RedWinRate ← RedWins/Playouts
    if BlueWinRate < Margin and RedWinrate < Margin then
        return 𝒫
    else if BlueWinRate > (1 − Margin) and RedWinRate > (1 − Margin) then
        return 𝒩
    else if BlueWinRate > (1 − Margin) then
        return ℒ
    else
        return ℛ

function DETERMINEGAMEVALUE(GameState)
    class ← DETERMINEOUTCOMECLASS(GameState)
    if class = 𝒫 then
        return 0
    else if class = 𝒩 then
        return *
    while class ≠ 𝒫 do
        if class = ℒ then
            add redEdge to GameState.WeighingStalk
            update gameValue
        else if class = ℛ then
            add blueEdge to GameState.WeighingStalk
            update gameValue
        class ← DETERMINEOUTCOMECLASS(GameState)
    return gameValue
```

4.1 Monte Carlo algorithm

The HACKENBUSH Scale Method has been implemented to be used with two different Monte Carlo methods: the Pure Monte Carlo algorithm and UCT (see Section 2.3).

4.1.1 Pure Monte Carlo algorithm

As explained in Section 2.3, the basic MCTS algorithm knows four phases. In the pure version, these four phases are utilised in the following way:

1. *Selection*: there is no selection policy in the sense that each available node is explored the same number of times.
2. *Expansion*: no expansion takes place, we go directly to the simulation phase.
3. *Simulation*: a random game is played a certain number of times for each node.
4. *Backpropagation*: the node with the best win rate is selected and its move is returned.

This version of the Monte Carlo algorithm is not an actual tree search; it does not look further than the moves that are immediately available.

4.1.2 UCT

In UCT, the four phases are implemented in the following way (see Section 2.3.1):

1. *Selection*: a node j is selected to maximise
$$UCT = \overline{X}_j + C_p\sqrt{\frac{2\ \ln\ n}{n_j}}.$$
2. *Expansion*: if the selected node has children that are not yet part of the tree, one of those is chosen randomly and added to the tree.
3. *Simulation*: from the new node's state, the game is played out randomly; each turn, a move is randomly selected and played, until a terminal state is reached.
4. *Backpropagation*: the value of the terminal state is backpropagated to all the nodes visited during this iteration.

The algorithm terminates and returns the best move that it found only once the computational budget has been reached. The best move corresponds to the child of the root with the highest visit count.

4.2 Experiments

In order to determine the performance of the Monte Carlo algorithms, experiments were set up with two variables: the number of times a game position was played out to determine the win rate, and the number of times a random simulation was run to determine the effectiveness of a move in the simulation phase of the algorithm. We call the first variable the playouts, and the second variable iterations. In this case, the number of iterations functioned as the computational budget. The performance of the algorithms was tested on various game positions of Red-Blue Hackenbush. Firstly, the precision of the algorithms in determining game values was tested by running the experiment on positions with dyadic rational numbers as their game values. Positions with values $1/2^\ell$ with $\ell \in [1, \ldots, 10]$ were used. As explained in Section 2.1.2, these positions consist of a single blue edge with i red edges stacked on top. For the number of playouts, the values $[100, 200, 400, 800]$ were used. For the number of iterations, the values $[100, 200, 600, 1200]$ were used. An additional experiment was done with a position valued $1/2048$. For this position, the two algorithm versions were tested with 100 playouts and 1200, 1300 and 1400 iterations.
Additionally, experiments were done to determine the values of Hackenbush positions with a more random nature. These positions are depicted in Figure 17 (position A), Figure 18 (positions B and C), and Figure 19 (positions D and E). These positions have negative values, whereas the positions in the aforementioned experiment all have positive values.
Experiments were run on the Redwood Bed position depicted in Figure 6, and on the Bluewood version of it: the same position, but with the colours inverted. The experiment was carried out using UCT, 400 playouts and iterations [600, 1200, 2400, 4800, 9600, 19200].
The function DetermineOutcomeClass (see Section 4) was tested on various Two Move Hackenbush positions. The values were: 0 of outcome class $\mathcal{P}$, $*$, $*2$, $\uparrow *$ and $\downarrow *$ of outcome class $\mathcal{N}$, $1/2$ and $1*$ of outcome class $\mathcal{L}$ and $-1/2$ and $-1*$ of outcome class $\mathcal{R}$.

5 Results

The tables on the following pages show the results of the experiments mentioned in the previous section. In each table, each cell contains a pair of numbers: the win rate of Blue when Blue started the game and the win rate of Red when Red started the game, respectively. When the content of a cell is red, the algorithm returned the wrong game value: the game value returned was either overestimated or underestimated. In the case of an overestimation, the table shows the returned value. In case of an underestimation (in which case the algorithm kept adding edges to the weighing stalk until a safeguard was triggered), the win rate at the point where the game was balanced and the algorithm should have returned the value is shown. In some cases, win rates exceeding the margin would cause a wrong edge to be added. In this scenario, the algorithm would also continue indefinitely, eventually triggering the safeguard. In the table, this is signified by "F" (for *Failed*) written in red.
When running experiments with positions with values $1/2$, $1/4$, $1/8$, and $1/16$, the algorithm returned the correct values with 100% precision. Regardless of the number of playouts and iterations, the returned win rates were (0, 0). Therefore, the results shown are from $1/32$ and smaller values.
Table 2 shows the results of the experiment ran on game positions with values $[1/32, 1/64, 1/128, 1/256, 1/512, 1/1024]$, with varying numbers of playouts and iterations, using the UCT algorithm.

Table 8 in Section B shows the results of the same experiments, but with the use of the Pure Monte Carlo algorithm.
Figure 15 shows the win rate of Blue as a function of the number of iterations that the Pure Monte Carlo algorithm was carried out with. The game value of the position used was 1/32, and it was balanced with the HACKENBUSH Scale Method. Figure 16 shows the win rate of Blue as a function of the number of iterations that the UCT algorithm was carried out with. This time, the game value was 1/128, again balanced with the HACKENBUSH Scale Method.

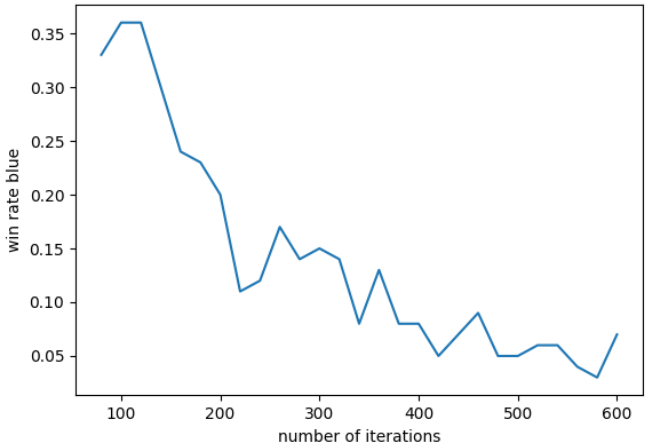


Figure 15: Graph of the win rate of Blue when Blue starts a game with value 1/32 balanced using the Pure Monte Carlo HACKENBUSH Scale Method, set against the number of iterations the algorithm has run.

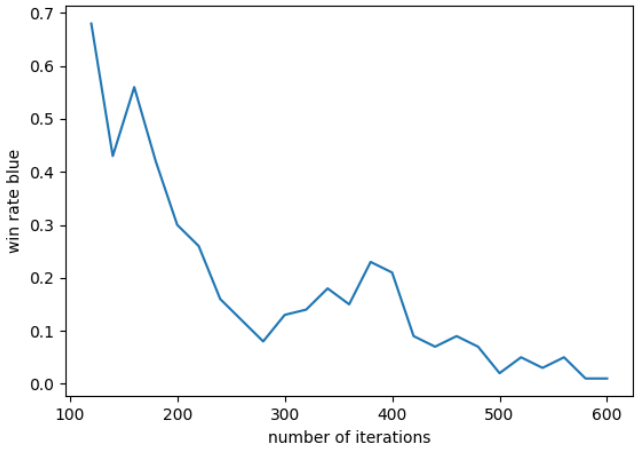


Figure 16: Graph of the win rate of Blue when Blue starts a game with value 1/128 balanced using the UCT HACKENBUSH Scale Method, set against the number of iterations the algorithm has run.

1/32	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	0.1, 0	0, 0	0, 0	0, 0	0, 0
200	0.08, 0	0.01, 0	0, 0	0, 0	0, 0
400	0.08, 0	0.01, 0	0, 0	0, 0	0, 0
800	0.10, 0	0, 0	0, 0	0, 0	0, 0

1/64	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	0.4, 0	0.09, 0	0, 0	0, 0	0, 0
200	0.4, 0	0.08, 0	0, 0	0, 0	0, 0
400	0.42, 0	0.11, 0	0, 0	0, 0	0, 0
800	0.44, 0	0.08, 0	0, 0	0, 0	0, 0

1/128	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	1/64	0.35, 0	0.19, 0	0, 0	0, 0
200	1/64	1/64	0.15, 0	0, 0	0, 0
400	1/64	1/64	0.16, 0	0.01, 0	0, 0
800	1/64	0.39, 0	0.17, 0	0.01, 0	0, 0

1/256	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	1/64	1/128	0.21, 0	0.23, 0	0.03, 0
200	1/64	1/128	1/128	0.17, 0	0.01, 0
400	1/64	0.71, 0	0.15, 0	0.22, 0	0.01, 0
800	1/64	0.66, 0	1/128	0.22, 0	0, 0

1/512	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	1/64	0.83, 0	1/256	1/256	0, 0
200	1/64	0.86, 0	1/256	0.24, 0	0.03, 0
400	1/128	0.80, 0	1/256	1/256	0.03, 0
800	1/64	0.85, 0	1/256	1/256	0.24, 0

1/1024	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	1/128	0.68, 0.11	1/512	1/512	0.37, 0
200	1/128	1/256	1/512	1/512	0.39, 0
400	1/128	1/256	1/512	1/512	0.34, 0
800	1/128	1/256	1/512	1/512	0.35, 0

Table 2: Results of the UCT HACKENBUSH Scale Method on a HACKENBUSH game position with various values. The numbers shown are the win rate of Blue when Blue starts and the win rate of Red when Red starts respectively. A fraction in red refers to an incorrect, prematurely returned value.

Table 3 shows the results of using the HACKENBUSH Scale Method with Pure Monte Carlo and UCT on a game position with game value 1/2048.

1/2048	**Pure Monte Carlo**	**UCT**
1200	1/1024	0.45, 0
1300	1/1024	0.5, 0
1400	1/1024	0.57, 0

Table 3: Results of the Pure Monte Carlo and UCT HACKENBUSH Scale Method on a HACKENBUSH game position with value 1/2048.

Table 4 shows the results of using the win rates of the Pure Monte Carlo algorithm with 100 iterations and 100 playouts to determine the outcome classes of various TWO MOVE HACKENBUSH positions. The win rates are shown as well. As can be read from the table, the determination of outcome classes of these positions is achieved with 100% accuracy.

	Actual outcome class	**Returned outcome class**	**Win rates**
0	$\mathcal{P}$	$\mathcal{P}$	0, 0
$*$	$\mathcal{N}$	$\mathcal{N}$	1, 1
$*2$	$\mathcal{N}$	$\mathcal{N}$	1, 1
$\uparrow *$	$\mathcal{N}$	$\mathcal{N}$	1, 1
$1/2$	$\mathcal{L}$	$\mathcal{L}$	1, 0
$-1/2$	$\mathcal{R}$	$\mathcal{R}$	0, 1
$1*$	$\mathcal{L}$	$\mathcal{L}$	1, 0
$-1*$	$\mathcal{R}$	$\mathcal{R}$	0, 1

Table 4: Results of the determination of outcome classes of various TWO MOVE HACKENBUSH positions, using the win rates of the Pure Monte Carlo algorithm.

Table 5 shows the time in seconds that different methods took to determine the value of a position with game value 1/512. For the two Monte Carlo variants, the variables 100 playouts and 600 iterations were used.

	Time in seconds
Determine recursively	0
Pure Monte Carlo	139
UCT	65

Table 5: Time in seconds needed to determine the game value of a position with value 1/512. For the Monte Carlo algorithms, 100 playouts and 600 iterations were used.

Table 6 shows the results of an experiment using UCT on a Redwood Bed position (see Figure 6) and an experiment on the same position, but with inverted colours. If the algorithm returned an incorrect value, the cells show that value in red and the win rates in black. If the safeguard of the algorithm got triggered, the cells show the win rates (in red) at a point where the calculated value got closest to the actual game value, and they show the closest value. The table on the left shows that the algorithm returned 1/4 with a budget of 600 iterations. This took 380 seconds. Using the brute-force method of determining the game value of the Redwood Bed recursively, the value was determined to be 1/32. This took 175 seconds.

	Value	**Win rate**
600	1/4	0.38, 0.35
1200	3/32	0.65, 0.19
2400	1/32	0.83, 0.07
4800	1/8	0.40, 0.31
9600	1/8	0.37, 0.31

	Value	**Win rate**
600	−23/32	0.60, 0.22
1200	−15/32	0.49, 0.33
2400	−9/32	0.69, 0.27
4800	−9/32	0.82, 0.20
9600	−1/8	0.35, 0.37

Table 6: Results of the determination of the game values of HACKENBUSH positions of the classes Redwood Bed (left) and Bluewood Bed (right) (see Figure 6). This table shows the returned value and the win rates when that value got returned, or the win rates at a point where the calculated game value got closest to the actual game value.

Table 7 shows the results of using the HACKENBUSH Scale Method with UCT to determine the values of several HACKENBUSH positions (see Figure 17, Figure 18 and Figure 19). For this experiment, 400 playouts and a varying number of iterations were used.

	Number of iterations		
	600	**1200**	**2400**
A	0.40, 0.29	0.44, 0.21	F
B	0.475, 0.14	0.28, 0.03	0.13, 0
C	0.01, 0.25	0, 0.26	0.01, 0.21
D	0.67, 0.21	0.54, 0.27	0.53, 0.3
E	F	F	F

Table 7: Results of the determination of the game values of several HACKENBUSH positions (Figure 17: position A, Figure 18: positions B and C, and Figure 19: positions D and E). Cells show the win rates of Blue when Blue started the game and Red when Red started the game at the moment that the game was balanced. An *F* indicates that a wrong edge got added to the weighing stalk and no game value got returned.

6 Conclusion and Further Research

To calculate the game values of HACKENBUSH positions, all subgames have to to be analysed in a recursive manner. Experiments have been done to see if the game values can be calculated using

the newly developed HACKENBUSH Scale Method, a method that utilises variants of the Monte Carlo Tree Search algorithm. From the results of these experiments, the following conclusions can be drawn.

Firstly, Table 2 shows a clear increase in accuracy with an increase in the number of iterations. For example, the results of an experiment on a position with game value 1/64 show that 100 iterations are not enough to accurately determine the game value. However, when running with 200 iterations and more, the algorithm consistently returned the right game value. Moreover, the win rate of Blue seems to decrease as more iterations are done. The results also show that the game value was increasingly hard to determine as the game values got smaller. Where the position with value 1/32 only needed 100 iterations to determine the value accurately, the position with value 1/1024 needed at least 1200 iterations. The aforementioned observations can also be made on these results: the accuracy of the algorithm increases with the number of iterations and decreases as game values shrink. From the decrease in accuracy, we can conclude that the Monte Carlo agents choose non optimal moves more often in games with smaller game values. One explanation for this is that games with smaller game values simply have more edges, which means that there are more options for moves, which means the probability of choosing a wrong move increases. Another explanation is that for games with small values, the actual advantage that one player has over the other is small. This means that even if a Monte Carlo agent plays quite well, the outcomes of individual games are sensitive to chance, making the win rates less representative for the outcome class. If the outcome class is determined incorrectly, the algorithm might return the wrong value. This becomes evident in the experiments on a game with value 1/1024: even when ran with 600 iterations, the algorithm returns 1/512 instead of 1/1024 (see Table 2).

As shown in Table 3, neither the Pure Monte Carlo nor the UCT version of the Scale Method returned the right game value for a position with value 1/2048, even when 1400 iterations were used. From these experiments, we can conclude that for the type of position experimented with, the accuracy of the HACKENBUSH Scale Method is 1/1024 when using a computational budget of 1400 iterations, both when using Pure Monte Carlo and UCT.

Figure 15 and Figure 16 show that the win rate of Blue decreased with the number of iterations that the Pure Monte Carlo algorithm and the UCT algorithm ran. For a balanced game, the expected win rate of Blue (and Red) is 0, so the decrease in win rate is actually an increase in accuracy. This result is to be expected: in UCT's case, more iterations means that it has more iterations to explore all possible moves and then exploit promising moves. More iterations also means that the effect of randomness of wins during the simulation phase decreases, which makes the scores of moves a more accurate representation of how good they actually are. Decreasing the effect of randomness might be even more relevant when using the Pure Monte Carlo algorithm, which explains the increase of accuracy with more iterations when using this algorithm as well.

The results in Table 6 show us that the Scale Method does not work on these particular Redwood Bed and Bluewood Bed positions. For the Redwood Bed, either the algorithm returned a value prematurely, or it did not return a value at all. The win rates of the experiment with 2400 iterations show that when the game was balanced in theory and the win rates should have been 0, Blue had an advantage and won most of the games. Apparently, the Monte Carlo agent is worse at playing Red than playing Blue in this position. One explanation for this is that the strategy for Red is more complicated than the strategy for Blue. Blue can remove the legs of the Bed, causing multiple red edges to disappear if they are still interconnected, while the strategy of Red is to minimize the number of interconnected parts.

In the case of the Bluewood Bed, wrong edges were added to the weighing stalk. The win rates did not get close to 0, and the algorithm continued until the safeguard was triggered. The results do show that an increase in number of iterations increased the accuracy as well: $-9/32$ is closer to $-1/32$ than $-23/32$ is. There seems to be a bias in the program causing Blue to win more often than Red, which explains the wrongly added edges. The fact that the colours were inverted for this experiment and the higher win rate of Blue persisted suggests that there is more to the bias than just a difference in strategy between the two colours. Further development of the method should include investigating this bias.
In Table 2, the win rate of Red when Red starts the game is always 0. A possible explanation for this is that this is because of the implementation of a "smart" move that can be made in the weighing stalk. When the value of a position is positive (which is the case for the positions used for the experiments done), the weighing stalk consists of one red edge with a number of blue edges stacked on top of it. With the smart move implemented, Blue always removes the highest edge of this stack. In the original position, which edge will be removed is determined by the algorithm. As we explained before, choosing the highest edge is an effective strategy, and this is reflected in the 0 win rate of Red.

In conclusion, the results show us that the values of various HACKENBUSH game positions can be approximated with a method using Monte Carlo Tree Search. However, there is no evidence that this method is quicker or more efficient than the brute force method of analysing all subgames recursively. Additionally, calculating the value recursively has the advantage of having perfect accuracy. The HACKENBUSH Scale Method does not seem to work for all HACKENBUSH positions. With the computational budget used in the experiments, stalks with a value of $1/1024$ were the limit, and the method could not accurately determine the value of a Redwood or Bluewood Bed. On the Redwood Bed, the method also took longer than the brute force method: it took 380 seconds where the brute force method took 175 seconds. When running the experiment on positions with a more random nature, some of the game position values could be accurately determined with differing numbers of iterations. This difference in iterations suggests that the computational budget needed to determine the game value is dependent on the complexity of the game position as a whole, not just on the size of its game value. The experiments were done on a limited number and limited types of positions, not fully representative of the many possibilities of HACKENBUSH positions. Lastly, the efficiency and speed of the Scale Method were not optimised. Therefore, the full potential of the method has not been realised. More research and development should be done to improve the method further and make it feasible in mathematical research.

A first boundary of accuracy has been set at $1/1024$ with the use of 1200 iterations, but additional research could be done to determine if the method can determine game values even more accurately when using more iterations or different versions of the Monte Carlo algorithm. In the Monte Carlo Scale Method, an alternative approach to the interpretation of the win rates could be tried as well: instead of observing their convergence to 0 or 1, the difference between the win rates could be used. If the difference between the win rates of the two players is statistically significant, this could be an effective indicator that the game position is not balanced yet and that more edges should be added. Additionally, experiments could be set up with different types of HACKENBUSH positions to investigate on which types of positions the method works well, and on which it does not. Theoretical research could be done to classify types of positions. It could, for example, be possible that there are

positions where one player needs a complicated strategy to win whereas the other player can win even with random moves. Redwood Beds could be of this type, and more experimentes should be done with them to determine whether it is possible to determine their values. Red-Blue-Green Hackenbush could also have positions worth looking into. If experiments with smaller values prove to need more iterations, efforts could be made to improve the efficiency of code used to implement the UCT algorithm, as the run time has proven to be quite high. One improvement for more complicated positions could be to break positions up into components and use the Scale Method to determine the values of the various sub-games. A hash table could be used to store values of components. This approach would be a combination of recursively analysing a game all the way through and the Scale Method.

For Two Move Hackenbush, the Pure Monte Carlo algorithm has been used to determine the outcome class of a game position. Future research could explore the possibility of using the Hackenbush Scale Method to determine the game values as well, combining the single move weighing stalk with the two move game. Due to additivity of games, the method could be used in a similar way. However, the weighing stalk should allow only one move, and the original game position should allow two, because Two Move Hackenbush positions are in itself not additive (see Section 3). It would be interesting to see if and how the Scale Method can approximate Star values.

References

[ANW19] Michael H. Albert, Richard J. Nowakowski, and David Wolfe. *Lessons in Play*. CRC Press, second edition, 2019.

[BCG04] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for your Mathematical Plays*. A.K. Peters, second edition, 2004.

[BPW+12] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[fil23] file-acomplaint. Hackenbush: Pocket edition. https://fi-le.itch.io/hackenbush, 2023. Version: 0.6. Accessed: 21-06-2024.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Proceeding Conference on Machine Learning*, pages 282–293. Springer, 2006.

[Mic20] Michael Bzms. Monte Carlo Tree Search c++ implementation & application to quoridor. https://github.com/michaelbzms/MonteCarloTreeSearch, 2020. Accessed: 20-06-2024.

[Tom11] Tom Davis. Hackenbush in c. http://www.geometer.org/puzzles/hack.c, 2011. Accessed: 04-02-2024.

A Additional figures

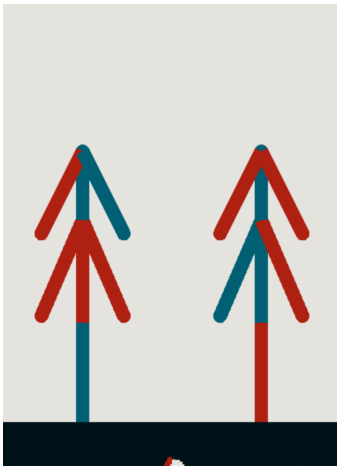
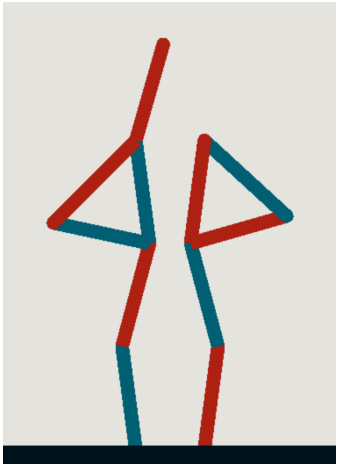
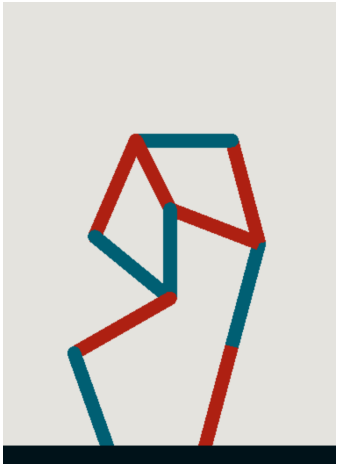


Figure 17: HACKENBUSH position A with game value $-3/16$ [fil23].

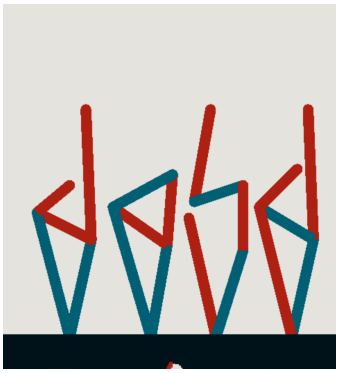


(a) HACKENBUSH position B.

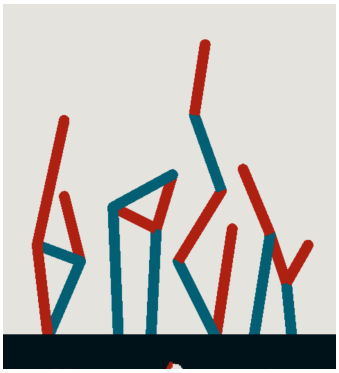


(b) HACKENBUSH position C.

Figure 18: Two HACKENBUSH positions with game value $-1/8$ [fil23].



(a) HACKENBUSH position D.



(b) HACKENBUSH position E.

Figure 19: Two HACKENBUSH positions with game value $-1/8$ [fil23].

B Additional results

1/32	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	0.31, 0	0.19, 0	0.06, 0	0.05, 0	0, 0
200	0.43, 0	0.18, 0	0.07, 0	0.04, 0	0, 0
400	0.34, 0	0.21, 0	0.18, 0	0.05, 0	0, 0
800	0.35, 0	0.18, 0	0.09, 0	0.04, 0	0, 0

1/64	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	0.54, 0	0.35, 0	0.18, 0	0.16, 0	0.02, 0
200	0.59, 0	0.39, 0	0.21, 0	0.09, 0	0.03, 0
400	0.57, 0	0.32, 0	0.15, 0	0.10, 0	0.02, 0
800	0.53, 0	0.35, 0	0.19, 0	0.12, 0	0.03, 0

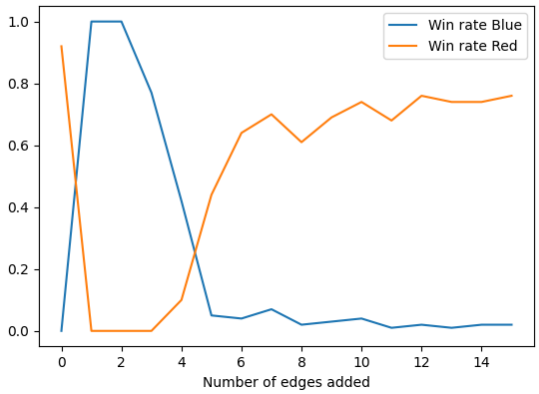
1/128	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	1/64	0.53, 0	0.33, 0	0.25, 0	0.06, 0
200	1/64	0.56, 0	0.32, 0	0.23, 0	0.07, 0
400	1/64	0.52, 0	0.26, 0	0.20, 0	0.06, 0
800	1/64	0.56, 0	0.30, 0	0.19, 0	0.06, 0

1/256	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	F	1/128	0.5, 0	0.34, 0	0.14, 0
200	F	1/128	0.47, 0	0.32, 0	0.14, 0
400	1/128	1/128	0.46, 0	0.34, 0	0.10, 0
800	1/128	1/128	0.48, 0	0.32, 0	0.15, 0

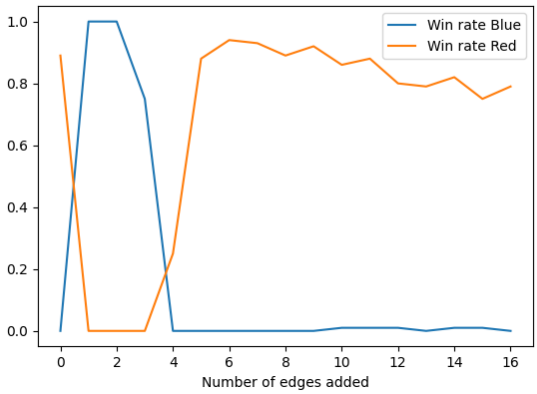
1/512	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	F	1/256	1/256	0.5, 0	0.12, 0
200	F	1/256	1/256	0.47, 0	0.2, 0
400	1/128	1/256	0.59, 0	0.45, 0	0.25, 0
800	F	1/256	1/256	0.47, 0	0.22, 0

1/1024	*Number of iterations*				
Number of playouts	**100**	**200**	**400**	**600**	**1200**
100	1/256	F	1/512	1/512	0.36, 0
200	F	F	1/512	0.64, 0	0.35, 0
400	1/256	F	1/512	1/512	0.36, 0
800	F	F	1/512	0.59, 0	0.35, 0

Table 8: Results of the Pure Monte Carlo Hackenbush Scale Method on a HACKENBUSH game position with various values. The numbers shown are the win rate of Blue when Blue starts and the win rate of Red when Red starts respectively.



(a)



(b)

Figure 20: Win rate of Blue when Blue started the game and Red when Red started the game, set out against the number of edges added to the game. Results of positions shown in Figure 18a and Figure 18b.