University of Alberta

Playing and Solving Havannah

by

Timo Ewalds

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Abstract

Havannah is a recent game that is interesting from an AI research perspective. Some of its properties, including virtual connections, frames, dead cells, draws and races to win, are explained. Monte Carlo Tree Search (MCTS) is well suited to play Havannah, but many improvements are possible. Several forms of heuristic knowledge in the tree show playing strength gains, and a change to the rules in the rollout policy significantly improves play on larger board sizes. Together, a greater than 80% winning rate, or 300 elo gain, is achieved on all board sizes over an already fairly strong player. This MCTS player is augmented with a few engineering improvements, such as threading, memory management and early draw detection, and then used to solve all 6 openings of size 4 Havannah, a game with a state space on the order of 6×10^{15} states. Castro, the implementation and test bed, is released open source.

Preface

I've never been very good or interested in playing board games, but I've always had a fascination with how to play them well. I started programming when I was 13 years old, and one of my first projects was to write an AI for tic-tac-toe. This is rather easy, as the game is tiny, but it was a good project for teaching me to program. A few years later I was introduced to mancala, and as a way to understand the game better, I decided to write a program to play it, and in the process, independently reinvented minimax and rediscovered that many games are zero-sum. My mancala program was never any good as I didn't know anything about alpha-beta and wikipedia hadn't been invented yet, but I have always had a greater interest in understanding how the mechanics of the game work than actually playing the game. Writing a strong program is a great challenge, and a very satisfying one if your program becomes a stronger player than you are yourself.

In late 2008 I was introduced to Pentago, an interesting game invented in 2005. It is a 2-player game played on a 6x6 board where each turn is composed of placing a stone and rotating a 3x3 quadrant with the goal of forming 5 in a row. After playing a few rounds and losing badly, I decided to figure out how to write a program to play it so that I could better understand the strategy and tactics. During a few rounds of play I devised a simple heuristic, which on its own is very weak, but when used with alpha-beta is quite strong. With some optimization, my program, Pentagod, became strong enough to easily crush me and my friends.

In early 2010, while taking a computer science course in computer game AI with Martin Mueller, I was tasked with writing a program to play Havannah. Basing my program, Castro, on my earlier work on Pentagod, my program became reasonably strong by program standards, but still quite weak by human standards. In fact, Christian Freeling, the creator of Havannah was so certain that programs would remain weak that he issued a challenge for €1000 to anyone who can beat him in only one in ten games on size 10 by 2012. I continued working on my program after the course finished, implementing techniques mentioned in class or used in other games, trying to use the theoretical properties used in the related game Hex, optimizing my code for pure efficiency and parallelism, and coming up with Havannah specific techniques. In September 2010 I went to Kanazawa Japan to compete in the Computer Games world championship and won 15 out of 16 games, winning the tournament. Soon after I attempted to solve size 4, a small version of the game, and succeeded in January 2011.

This thesis is the story of what it takes to write a strong Havannah player, and how this player was used as the basis of solving size 4 Havannah. Chapter 1 introduces some of the concepts and motivations for this thesis. Chapter 2 explains the required background knowledge for the algorithms used in the rest of thesis. Chapter 3 describes the rules of the game and introduces several properties of the game itself that make writing a program challenging, and a few that can be exploited to increase the playing strength. Chapter 4 explains how the general techniques were adapted to Havannah and introduces a few Havannah specific heuristics that together lead to a tournament level program. Chapter 5 explains how the player was used to solve size 4 Havannah and the extra techniques needed to accomplish this goal along with the solution to size 4 Havannah. Chapter 6 provides a summary, and describes possible future work.

Acknowledgements

I'd especially like to thank Colin Ophus, for playing so many games of Pentago and Havannah with me, and trying to deconstruct the games. His insights and enthusiasm helped me stay motivated and continually improve Castro. Also, I appreciate the thesis template and thesis advice.

Thank you Ryan Hayward, Martin Mueller and Jonathan Schaeffer, for your insights into game playing algorithms, how they apply to other games and possibly to Havannah, and for advising me on my thesis.

Thank you Marcin Ciura for your havannah.sty which made the Havannah diagrams easy, and the constant discussion of Havannah ideas.

Thanks to my family and friends, for their constant support as I worked on my masters.

Thank you Christian Freeling, for inventing such an interesting game.

Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation	Meaning
$\alpha\beta$	Alpha-Beta algorithm
AI	Artificial Intelligence
CAS	Compare And Swap
DAG	Directed Acyclic Graph
GC	Garbage Collection
LGR	Last Good Reply
LGRF	Last Good Reply with Forgetting
MCTS	Monte Carlo Tree Search
MPN	Most Proving Node
PNS	Proof Number Search
RAVE	Rapid Action Value Estimate
TT	Transposition Table
UCB	Upper Confidence Bounds
UCT	Upper Confidence bounds as applied to Trees
VC	Virtual Connection

1

Introduction and Contributions

1.1 Introduction

Artificial intelligence (AI) is an important and exciting field of research with the potential to fundamentally improve the way society functions. One of the earliest and more well-known sub-fields of AI research is games and puzzles. It was once commonly thought that once a computer could play Chess at a world championship level, it would be on par with human intelligence. Deep Blue, the Chess program created by IBM, accomplished world championship level play in 1997, using brute force search. While Chess-playing ability turned out to not be representative of general intelligence, the search techniques pioneered in Chess and similar games are undoubtedly effective at problem solving and are widely applicable to other domains. For AI researchers, the next goal after playing better than humans is to solve the game, in essence to play optimally. Several games, such as Connect 4 and Checkers have been solved, ensuring that a computer player cannot be defeated. Those who aren't working on optimal play are working on harder games than Chess. They are discovering new algorithms and heuristics that continually push the bounds of what computers can do.

Havannah is a board game invented in 1979 by Christian Freeling. The rules and properties of Havannah are described in detail in Chapter 3. While it is not a popular game, it is interesting from a game research perspective. It is a two player, zero-sum, perfect information game, like Chess, Go and Hex, and like Hex, it is a connection game. Unlike Chess, and like Go, however, it has no known strong heuristic for evaluating a position, making the classical techniques ineffective. Christian Freeling is so confident that computers cannot play Havannah well that in 2002 he placed a €1000 wager that no program could beat him in even one out of ten games on a size 10 board by 2012. This challenge makes it an interesting game for developing newer game playing techniques.

The goal of this thesis is to develop a program that plays strong Havannah on board sizes 4 through 10, and to use this player to solve all 6 openings of the size 4 board.

1.2 Contributions

Havannah is closely related to Hex, a similar game that has received significantly more attention over the years. Hex has several mathematical properties that allow a program to ignore certain moves, or to prove the outcome of a game many moves before the end of the game. Several of these properties are shown in Section 3.4 to not apply in Havannah, or to apply only in a limited sense. Unlike Hex, draws are possible in Havannah, and detecting these early are key to solving certain positions. A technique for detecting draws once no wins are possible is presented in Section 5.1.5.

All of the algorithms and ideas presented here were implemented in a program named Castro. Castro is written in C++ and has been released as open source at `https://github.com/tewalds/castro`. It includes an MCTS player and several solvers, along with several heuristics. Most of the testing was done using ParamLog, a distributed testing framework written for testing Castro. It has also been released as open source at `https://github.com/tewalds/ParamLog`.

With ParamLog, testing a large number of features becomes easy, so all the algorithms and heuristics were tested with multiple values on board sizes 4-10. This is a departure from previous work on Havannah which generally focused only on a single or a few board sizes.

Several knowledge heuristics were tested in Section 4.8, including maintaining virtual connections, local reply, locality, edge connectivity, group size and distance to win. Several of these haven't been tested in Havannah before.

Havannah's three winning conditions interact with MCTS in unusual ways, so four novel ring rule variations are introduced and tested in Section 4.9.4.

Testing the many knowledge heuristics and rollout policy features shows that a greater than 80% winning rate against an already fairly strong baseline can be achieved on all board sizes greater than size 4.

While proof backups have been used in MCTS before, they are shown to be particularly effective in a Havannah player in Section 4.6 when combined with a two-ply look-ahead. Chapter 5 builds on this work and adds threading, draw detection and memory management to solve size 4. The perfect-play solution to size 4 Havannah is presented in Section 5.2.3.

2

Game Playing Techniques

Most game playing programs build a game tree, and then chose the most promising move at the root of the tree. Many game playing algorithms exist, and they vary based on the order in which they explore the tree, the in-memory representation of the game tree, the evaluation method of leaf nodes, and how they back up the values to interior nodes.

2.1 Minimax

The minimax algorithm is the foundation of all game playing algorithms. The goal is the find the minimax value of a state or set of states, or equivalently of

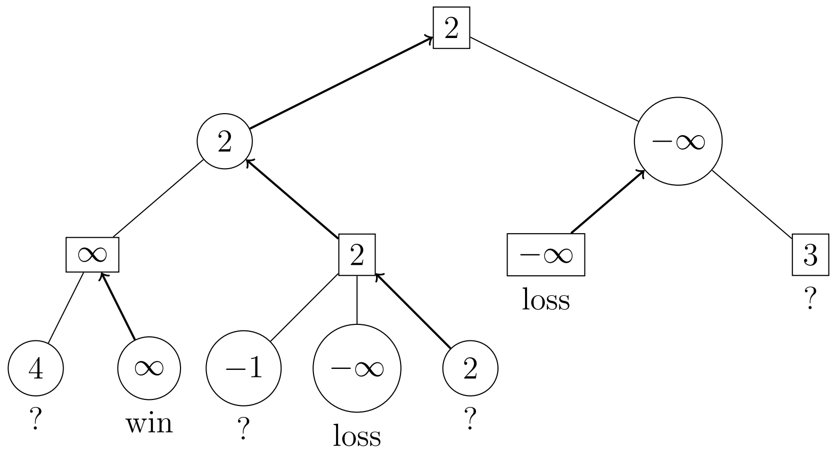


Figure 2.1: Minimax Tree, squares are MAX nodes, circles are MIN nodes

a set of moves, and then choose the move with the highest value. All values are from the perspective of the root player. The value of a node for the root player is the maximum of its children nodes, and the minimum for the opponent's children. The values represent the outcome of the game, or a heuristic estimate of the value of the position if the game outcome isn't known. This is shown in Figure 2.1 with the outcomes of terminal nodes represented with positive (win) or negative (loss) infinity, and non-terminal leaf nodes having heuristic values. The minimax value of this tree is 2. The pseudocode for a simple depth first search version is shown in Figure 2.2.

2.1.1 Negamax

Minimax uses values as taken from a fixed perspective of the root player. This complicates the code with having to minimize for one player and maximize for the other. Noting that $max(a, b) = -min(-a, -b)$, the duplication can be removed by negating the value each time we switch perspective. In this setup all values returned from an evaluation function are from the perspective of the player who is making the move. The pseudocode for this transformation is

```
int minimax(State state){
    if(state.terminal())
        return state.value();
    int value;
    if(state.player() == WHITE){
        value = -INF;
        foreach(state.successors as succ)
            value = max(value, minimax(succ));
    }else{
        value = INF;
        foreach(state.successors as succ)
            value = min(value, minimax(succ));
    }
    return value;
}
```

Figure 2.2: Minimax Pseudocode

shown in Figure 2.3.

Several algorithms shown later reference the negamax formulation.

2.2 Alpha-Beta

Alpha-beta ($\alpha\beta$) is a refinement of minimax, pruning parts of the game tree that cannot affect the minimax value of the root[1]. It maintains two values that bound the minimum value each player is guaranteed given the tree searched so far. When these bounds meet or cross, this is called a cut-off, and the remaining moves need not to be considered.

The pseudocode for alpha-beta, written in the negamax formulation, is shown in Figure 2.4. The initial values for alpha and beta are negative infinity and infinity respectively. It is a depth-first implementation that returns after a

```
int negamax(State state){
    if(state.terminal())
        return state.value();
    int value = -INF;
    foreach(state.successors as succ)
        value = max(value, -negamax(succ));
    return value;
}
```

Figure 2.3: Negamax Pseudocode

maximum depth is reached. If a terminal node is found, the true value is returned, otherwise a heuristic value is returned.

The runtime of alpha-beta depends on the branching factor b, search depth d, and the number of cut-offs. Minimax has a runtime of $O(b^d)$, as does alpha-beta if it has no cut-offs. If the true minimax value is found early, as would happen if moves are examined in decreasing order of their minimax value, many early cut-offs will occur, leading to a runtime of $O(b^{d/2})$, an exponential speedup. In general, the move ordering will not be optimal, so the runtime will be between these two extremes. In practice, high performance game-playing programs often perform within a constant of $O(b^{d/2})$.

2.2.1 Transposition Table

Transpositions can lead to an exponential blowup in the search space by allowing the search to investigate multiple paths to a single node (because most game trees are really game graphs). To minimize the number of transpositions reevaluated, alpha-beta search is usually enhanced with a Transposition Table (TT) [2]. After searching a subtree, the root of the subtree and the results of the search are stored in the TT. When a state is reached in the search, the TT is checked to see if the result has already been obtained. Transpositions

```
int alphabeta(State state, int depth, int alpha, int beta){
    if(state.terminal() || depth == 0)
        return state.value();
    int val = -infinity;
    foreach(state.successors as succ){
        val = max(val, -alphabeta(succ, depth-1, -beta, -alpha));
        alpha = max(alpha, val);
        if(alpha >= beta)
            break;
    }
    return val;
}
```

Figure 2.4: Alpha-beta Pseudocode, shown in the negamax formulation

are usually found by comparing hash values and indexing into a large table. Sometimes a hash table is used, but usually the number of nodes searched is too big to store in memory, so a simple replacement policy is used. The simplest is to use the hash value as an index into a large array of values, replacing the previous node that indexed to the same location.

In many games this leads to a large speedup as the number of nodes searched is decreased dramatically.

2.2.2 Iterative Deepening

The runtime of alpha-beta is exponential in the search depth, and the strength of a computer player is dependent on the search depth (usually the deeper the better). If the algorithm is stopped before completion, the best move may not have been explored at all, so a shallower search that finishes is likely better than a deeper search that doesn't. Thus we start with a shallow search, and run incrementally deeper searches as long as we still have time [2]. This is not a big waste of work since the majority of the runtime is spent at the deepest level anyway. Iterative deepening allows alpha-beta to act similar to a breadth-first search with the memory overhead of a depth-first search.

Iterative deepening, when combined with a transposition table, also gives better move ordering. A node's value from the previous iteration gives a more accurate estimate of the value of a node than a heuristic estimate without a search. As we saw in Section 2.2, better move ordering can lead to an exponential speedup, easily offsetting the overhead from searching the shallow depths multiple times.

2.2.3 History Heuristic

A good move ordering can lead to many cutoffs and an associated speed increase. The history heuristic [3] is a game-independent move ordering method that gives higher priority to moves that have a track record of leading to cutoffs elsewhere in the tree. If a particular move gives a cutoff, it's quite likely that it will also give a cutoff for all of its siblings and so should have a higher priority there. This assumes that similar moves in different parts of the tree are related.

2.3 Proof Number Search

Proof Number Search (PNS)[4] is a best-first search used to answer binary questions such as the outcome of a 2-player game starting from a given state. Being a binary outcome with the minimax property, it is well represented as an AND/OR tree when all values are from the perspective of the root player. AND nodes and OR nodes are analogous to MIN nodes and MAX nodes respectively in minimax. Each node in the tree can have one of three values:

Proven/Win, Disproven/Loss, or Unknown. All nodes store two numbers that show how close it is to being proven or disproven. The proof number (pn) is the minimum number of leaf nodes in the subtree that must be proven for the node to be proven. The disproof number (dn) is the minimum number of leaf nodes in the subtree that must be disproven for the node to be disproven. Some leaf nodes, if solved, will change the proof number of the root. Other leaf nodes, if solved, will change the disproof number of the root. Others, if solved, won't affect the proof or disproof numbers of the root. The Most Proving Nodes (MPN) are the intersection of the set that affect the proof number and the set that affect the disproof number at the root. Solving an MPN will definitely affect either the proof or disproof number of the root. Every tree is guaranteed to have at least one MPN. Proof Number search grows its tree by continually expanding an MPN.

Proof Number search can be split into 3 phases: descent, expansion, and update. The most proving node is found during the descent phase. It can be found by selecting the child with the minimum proof number when at an OR node and by selecting the child with the minimum disproof number when at an AND node. This is applied iteratively until a leaf node is reached. This leaf node is an MPN.

Once the most proving node n is found, it is expanded, initializing all non-terminal children with $n_i.pn = 1, n_i.dn = 1$, winning children with $n_i.pn = 0, n_i.dn = \infty$ and losing children with $n_i.pn = \infty, n_i.dn = 0$, where n_i refers to the i^{th} child of n.

After expansion, the proof and disproof numbers of all the ancestors of the most proving node must be updated using these formulas. For OR nodes:

$$n.pn = \min_{i=0}^{k} n_i.pn, \quad n.dn = \sum_{i=0}^{k} n_i.dn$$

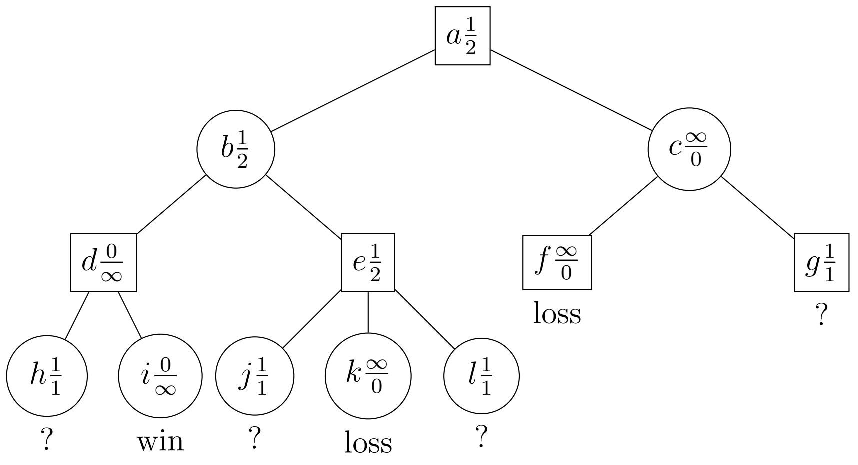


Figure 2.5: Proof Number Search Tree, squares are OR nodes, circles are AND nodes, proof numbers are on top, disproof numbers on the bottom, based on [5]

For AND nodes:

$$n.pn = \sum_{i=0}^{k} n_i.pn, \quad n.dn = \min_{i=0}^{k} n_i.dn$$

Note how this backs up a single win at an OR node as a win, or a single loss at an AND node as a loss. It also backs up all losses at an OR node as a loss, and all wins at an AND node as a win.

These three phases are repeated until the root is solved or the tree grows too big to be stored in memory. At the root r, if $r.pn = 0$ it is solved as a win, or if $r.dn = 0$ it is solved as a loss, otherwise it is still unknown.

Consider the tree in Figure 2.5. The most proving node is found by following the edges $a \rightarrow b \rightarrow e \rightarrow j$. If j has a child that is a win, it would be backed up as a win at j, leading to a win at e, and a win at b, giving the root player a winning move from the root. With $a.pn = 1$ at the root, only 1 node was needed to be proven as a win for the root to also be proven as a win. If both j and l were proven to be losses, then e would be a loss, leading b to also be a loss, and consequently the root to also be a loss. This is reflected in $a.dn = 2$ at the root. If, however, j has 1 non-terminal child m and no terminal children, m would have $m.pn = 1, m.dn = 1$ and would be the new MPN. If j has 2 non-terminal children and no terminal children, $j.pn = 2, j.dn = 1$, and l would be the new MPN.

This algorithm selects nodes based on the shape and value of the tree, using no domain or game specific heuristic. It is guided to parts of the tree where fewer options need to be proven. This results in it favouring slim parts of the tree, areas where there are few moves available, or where many moves are forced. In many games it is advantageous to have more moves available, or higher mobility, than your opponent. This often happens by forcing the opponent's moves. Proof Number search is very fast at solving these positions. In games or positions where the branching factor is constant or consistent, with few forced moves, Proof Number search approximates a slow breadth-first search, and thus isn't very fast.

Being a best-first search algorithm, the whole tree must be kept in memory, since any node could become an MPN and therefore be searched at any time. This makes it a memory-intensive search algorithm, with many of the variants attempting to reduce memory usage, allowing bigger problems to be solved.

One simple optimization is to stop the update phase once the proof and disproof numbers don't change. This often happens when siblings have the same value, causing a sibling to be the new MPN. A new search can be started from this node instead of from the root. A simple memory optimization is to remove and reuse the memory of subtrees under a proven or disproven node.

2.3.1 The Negamax Formulation

Just like minimax can be written in the negamax formulation, so too can proof number search. The Proof number at an OR node is the same as the Disproof

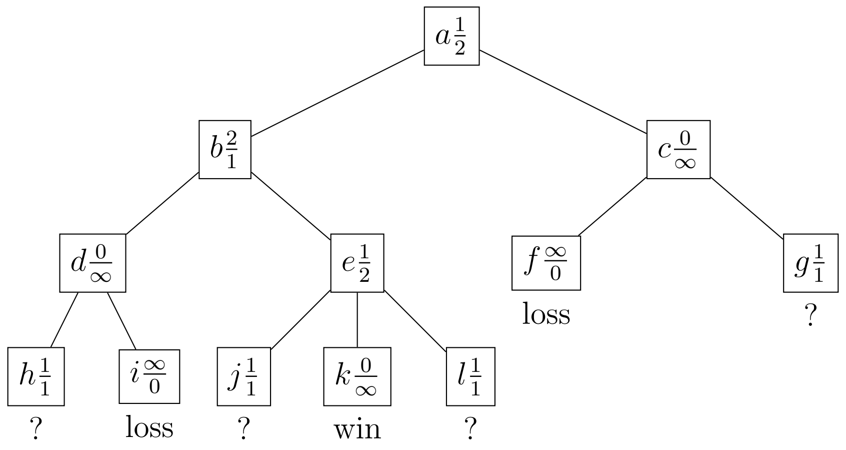


Figure 2.6: Proof Number Search Tree Using the Negamax Formulation, all nodes are OR nodes, ϕ is on top, δ is below, based on [5]

number at an AND node, and is named ϕ (phi). Similarly, the Proof number at an AND node is the same as the Disproof number at an OR node, and is named δ (delta). Instead of considering all nodes to be from one player's perspective, all nodes are considered to be from the player who is making the move at that node. This shift in perspective greatly simplifies the code.

Figure 2.6 shows the same tree as in Figure 2.5, except using the negamax formulation. Note how all nodes are now OR nodes, and the proof and disproof numbers are exchanged in the nodes that were previously AND nodes.

Given this shift in perspective, the descent and update formulas need to be corrected. The new descent move selection is always to choose the child with the minimum delta. The new update formulas are:

$$n.\phi = \min_{i=0}^{k} n_i.\delta, \quad n.\delta = \sum_{i=0}^{k} n_i.\phi$$

The pseudocode for Proof Number Search in the negamax formulation is shown in Figure 2.7. A State is the board state, and a Node is a node in the tree in memory.

```
int pns(State state){
    Node root = initnode(state);
    while(root.phi != 0 && root.delta != 0) search(root, state);
    return (root.phi == 0 ? PROVEN : DISPROVEN);
}
void search(Node node, State state){
    if(node.numchildren == 0){ //found MPN
        foreach(state.successors as succ)
            node.addchild(initnode(succ));
    }else{
        do{
            Node child = node.child_min_delta();
            search(child, state.move(child.move));
            bool changed = updatePD(node);
        }while(!changed && node.phi != 0 && node.delta != 0);
    }
}
Node initnode(State state){
    Node node; node.move = state.lastmove();
    if(state.win())        { node.phi = 0;   node.delta = INF; }
    else if(state.loss()){ node.phi = INF; node.delta = 0;   }
    else                   { node.phi = 1;   node.delta = 1;   }
    return node;
}
bool updatePD(Node node){
    int phi = INF, delta = 0;
    foreach(node.children as child){
        phi = min(phi, child.delta);
        delta = delta + child.phi;
    }
    bool changed = (node.phi != phi || node.delta != delta);
    node.phi = phi; node.delta = delta;
    return changed;
}
```

Figure 2.7: Proof Number Search Pseudocode, shown in the negamax formulation, with the optimization to not propagate up if no changes occur

2.3.2 Transposition Table

Proof number search uses an explicit tree which must be kept in memory, but the tree required is often bigger than available memory. One common approach to bounding the memory needed is to store the nodes in a transposition table instead of an explicit tree. This has the benefit of bounded memory as well as saving computation and memory on transpositions, at the cost of having to recompute nodes that are replaced in the transposition table. Even when a node needs to be recomputed, its children are often still in the transposition table, allowing for a quick recomputation. In many cases the transposition table can be several orders of magnitude smaller than would be needed to store the explicit tree.

2.4 Monte Carlo Tree Search

For games where a fast and effective evaluation function exists, alpha-beta search is likely to result in deep search and strong game play. Unfortunately a good heuristic is not known for many games including Go and Havannah. Monte Carlo Tree Search (MCTS) [6] is an algorithm for building and exploring a game tree that is based on statistics instead of a heuristic evaluation function. MCTS avoids using a heuristic by building its tree as guided by playing games of random move sequences. While a sequence of random moves by itself has a very low playing strength, in aggregate random games tend to favour the player that is in a better position.

MCTS consists of four phases [7] which together are called a simulation. The four phases, as shown in Figure 2.8, are:

Descent A path through the game tree from the root node down to a leaf node N is chosen. The path is chosen by recursively selecting a child by

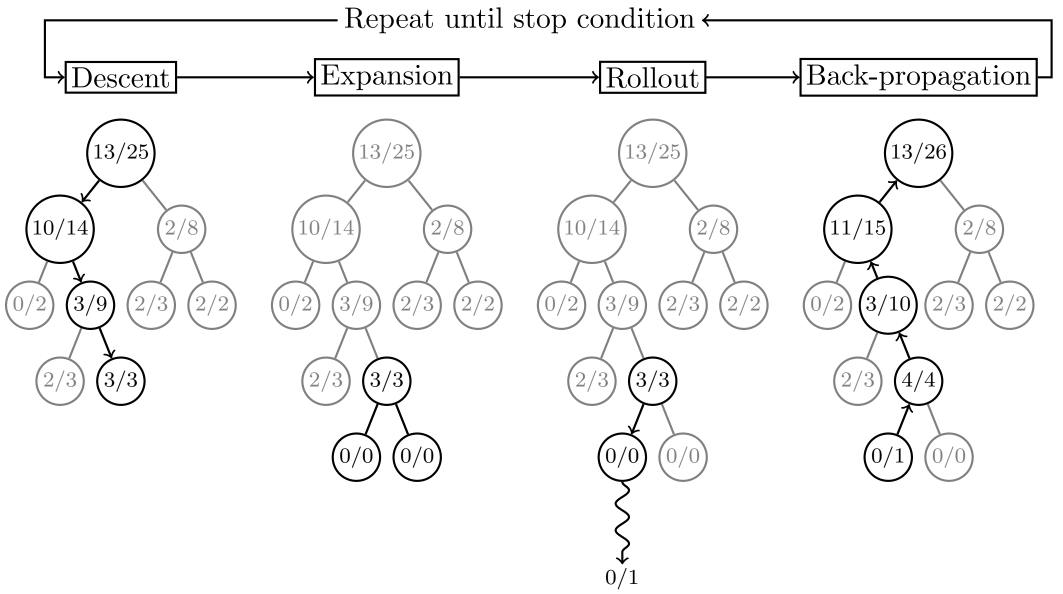


Figure 2.8: Four Phases of Monte Carlo Tree Search, together called a Simulation, shown in the negamax formulation with a minimum of 3 experience before expansion

applying some criteria (based on the current winning rate and possibly some heuristic knowledge) until a leaf node is found.

Expansion If the node N has enough experience from previous simulations, its children are expanded, increasing the size of the tree, otherwise this phase is skipped.

Rollout A random game, a sequence of random moves, is played from N through the newly expanded children to the end of the game.

Back-propagation The outcome of the rollout is propagated back to each node along the path to the root. The winning rate of the moves made by the player that won the rollout is increased while winning rate of the moves by the player that lost the rollout is decreased.

These four phases are repeated continually until a stopping condition is reached, such as running out of time or memory. Each simulation adds some experience to the tree, updating the expected chance of winning for the nodes it traverses. These winning rates are stored as the number of wins and the number of simulations through a node. For a given node n, $n.v$ is the winning rate and $n.n$ is the number of simulations.

Once a stopping condition has been reached, a move is chosen by some criteria. The four most common criteria are: most simulations, most wins, highest winning rate, and highest lower confidence bound on winning rate. Using the most simulations is the most conservative, but if a counter-move was found late in the game, it may still be the most simulated even if it doesn't have the highest winning rate. Using the most wins is a little less conservative and will favour a late new-comer if it has almost caught up. Use of the highest winning rate is quite risky since it may favour a move that has a very small subtree where a good counter move exists but hasn't been found yet. To deal with that a lower bound can be used, but a large confidence interval should be used to avoid choosing risky moves.

The pseudocode for MCTS is shown in Figure 2.9. A State is the board state, and a Node is a node in the tree in memory. This code glosses over a few important points, such as how the value of a node is computed, how nodes are initialized, and how random moves are chosen. Some common ways of implementing these details are explained in the next sections. UCT, RAVE and heuristic knowledge (described below) address the value of a node and node initialization. Rollout policy addresses how random moves are chosen.

2.4.1 UCT: Upper Confidence bounds as applied to Trees

The most common and most famous formula for the descent phase of MCTS is Upper Confidence bounds as applied to Trees (UCT) [8]. It derives from the Upper Confidence Bounds (UCB) formula, which is used on the multi-armed

```
Move mcts(State state){
    Node root = Node(state);
    while(!timeout)
        search(root, state);
    return root.bestchild();
}
int search(Node node, State state){
    //rollout
    if(node.numchildren == 0 && node.sims == 0){
        while(!state.terminal())
            state.randmove();
        return state.outcome(); //win = 1, draw = 0.5 or loss = 0
    }

    //expand
    if(node.numchildren == 0)
        foreach(state.successors as succ)
            node.addchild(Node(succ));

    //descent
    Node best = node.children.first();
    foreach(node.children as child)
        if(best.value() < child.value())
            best = child;

    int outcome = 1 - search(best, state.move(best.move));

    //back-propagate
    best.sims += 1;
    best.wins += outcome;
    return outcome;
}
```

Figure 2.9: Monte Carlo Tree Search Pseudocode, shown in the negamax formulation

bandit problem. UCB is used to balance exploitation and exploration when multiple options are available and each option returns a random distribution of reward. The amount of regret, i.e., the number of plays to non-optimal arms, should be minimized to maximize reward in the long term. UCT applies this idea to a tree of choices.

In the descent phase at node n, a child node must be chosen according to some criteria. UCT chooses the child node n_i that maximizes the value of:

$$n_i.v + c * \sqrt{\frac{\ln(n.n)}{n_i.n}} \tag{2.4.1}$$

where c is a tunable constant to balance the exploration rate. Intuitively, moves with high winning rate should be exploited more, but moves with a small number of simulations as compared to the parent should be explored to improve the confidence. This formula is guaranteed to converge to a best move given infinite time and memory.

2.4.2 RAVE: Rapid Action Value Estimate

In basic MCTS many thousands of simulations are usually run per second, but the information about which moves were made during the rollouts is unused. A win or a loss is composed of many moves which contribute to that outcome, and often good moves during a rollout are also good moves if made earlier during the rollout or descent phases. This is a similar to the reasoning behind the history heuristic. Thus, we can keep a winning rate for each move during the rollouts and use this to encourage exploration of moves that do well during rollouts. This winning rate is called the Rapid Action Value Estimate (RAVE) [9, 10]. RAVE experience is gathered more quickly than by pure experience alone, though it is less correlated to success, and so should be phased out as real experience is gained. For a given node n, $n.r$ is the RAVE winning rate and $n.m$ is the number of RAVE updates.

Usually RAVE experience and real experience are combined as a linear combination, starting as only RAVE experience and asymptotically approaching only real experience. This combination replaces $n_i.v$ in Equation 2.4.1:

$$\beta * n_i.v + (1 - \beta) * n_i.r \tag{2.4.2}$$

Several formulas for β have been proposed. The simplest two formulas for β are:

$$\beta = \frac{k}{k + n_i.n} \tag{2.4.3}$$

$$\beta = \sqrt{\frac{k}{k + 3 * n_i.n}} \tag{2.4.4}$$

both of which have a tunable constant k which represents the midpoint, the number of simulations needed for the RAVE experience and real experience to have equal weight.

David Silver computed an optimal formula for β under the assumption of independence of estimates [11]:

$$\beta = \frac{n_i.m}{n_i.n + n_i.m + 4 * n_i.n * n_i.m * b^2} \tag{2.4.5}$$

where b is a tunable RAVE bias value.

In practice, RAVE leads to a large increase in playing strength for games such as Go and Havannah where the assumption that a good move is also good if played earlier holds. The RAVE updates often lead to sufficiently large exploration that the constant in the UCT exploration term is set very low or even to 0, removing UCT exploration altogether.

2.4.3 Heuristic Knowledge

While UCT is guaranteed to converge given infinite time, game specific knowledge can encourage it to find good moves faster. When a node is expanded, its children all start with no experience, so the default policy is to choose between them randomly. The simulation is more representative of a good game, and leads to a better understanding of the minimax value, if it chooses a good move first. Eventually the best move will receive the majority of the simulations, and we'll do better if this is true right from the beginning. Each game has its own heuristics, and Havannah-specific ones are described in later chapters, but the way these heuristics are used is game independent.

The first way heuristic knowledge is used is to simply add fake experience to a node. Instead of initializing a node as $n_i.v = 0, n_i.n = 0$, good moves can be initialized with $n_i.v = a, n_i.v = b$, where a and b are tunable constants, which effectively means that this node has some amount of wins attributed to it before any simulations have gone through it. This has the effect of allowing the node to look good for the first while even if it is unlucky. The extra simulations will fade over time as the few extra wins becomes insignificant in the long run. Bad moves can similarly be initialized with fewer wins than simulations, effectively depressing its early winning rate. Depending on the implementation, this may encourage the first few simulations to avoid the good moves, due to their smaller confidence bounds compared to similar moves with the same high winning rate. This has the effect of making the grandparent move look bad. This knowledge could also be added as fake RAVE experience as well as, or instead of, actual experience.

Another way heuristic knowledge is used is to add a knowledge term to the value formula. This leaves the experience and confidence bounds alone, but gives a boost for the first few simulations to nodes with higher knowledge. This has the added benefit of being able to order the nodes by boost size. The knowledge term should fall off with increasing experience. Three suggested knowledge terms are:

$$\frac{n_i.k}{log(n_i.n)}, \quad \frac{n_i.k}{\sqrt{n_i.n}}, \quad \frac{n_i.k}{n_i.n}$$

where $n_i.k$ is the knowledge value for the node n_i.

2.4.4 Rollout Policy

The strength of MCTS is highly dependent on the average outcome of the rollouts being representative of the strength of the position. When a player who is in a good position has an easy defence to a devastating attack, but fails to defend, the outcome is not representative of the strength of the original position. Decreasing randomness by enforcing defences against devastating attacks can bias the outcome, but usually leads to higher quality and more representative games, leading to a stronger player. Most rollout policies used in real programs are game specific, but a few game independent ones are mentioned here.

Instead of pure random, a weighted random scheme can be used. Moves that have good experience in the tree can be selected with a higher probability to poor moves. This could be based on real experience, RAVE experience, pattern knowledge or heuristic knowledge as described in the Section 2.4.3.

The Last Good Reply [12, 13] scheme can be used, where the moves made by the player that won a rollout are saved for use in later rollouts when similar situations occur. When these moves fail to lead to a win in a later rollout, they may be removed from the list of replies.

All possible moves can be checked to see if they lead to an instant win if made, or an instant loss if made by the opponent. If a winning moves exists, it should be made, and if the opponent has a winning move, it should be blocked.

2.5 Summary

Several game playing and solving algorithms exist, but they're all based on minimax. Minimax chooses the move that minimizes the maximum outcome the opponent can achieve.

Alpha-beta is a refinement to minimax that prunes parts of the tree that can't affect the minimax value of the root. Transposition tables reduce the search space from a tree to a graph, reducing the search space. Iterative deepening, allows an early result to be returned, and combined with transposition tables, gives better move ordering allowing deeper searches. The history heuristic also improves move ordering.

Proof number search is an algorithm for solving the outcome of games. It maintains estimates of the difficulty of solving a subtree, preferring to solve easier parts of the tree. This leads to it preferring to explore forced moves and slim parts of the tree. A transposition table can be used to reduce the search space and solve problems that are bigger than physical memory.

Monte-Carlo Tree Search is a game playing algorithm that works well on problems where no good heuristic is known. It consists of four phases: descent, expansion, rollout and back-propagation. It chooses a leaf node, grows the tree, plays a random sequence of moves, and uses the outcome of this random game to bias the next descent. MCTS can be improved by choosing a good balance between exploration and exploitation. Gaining experience from the moves made within rollouts can be a big help, as can biasing the descent towards better moves based on heuristic knowledge. A rollout policy that leads to outcomes that are more representative of the true outcome is also useful.

3

Rules and Properties of Havannah

3.1 Rules of Havannah

Havannah is a connection game invented in 1979 by Christian Freeling. It is a two player, zero-sum, perfect information game played on a hexagonal board. Each turn a player places a stone on the board in alternating play. Stones are never moved nor removed after their initial placement. A *group* or *chain* is a set of connected stones of the same colour. The game ends when one of the players completes one of the three winning conditions which are shown in Figure 3.1:

- A *Bridge* is a group of stones that connects any 2 corners, for example

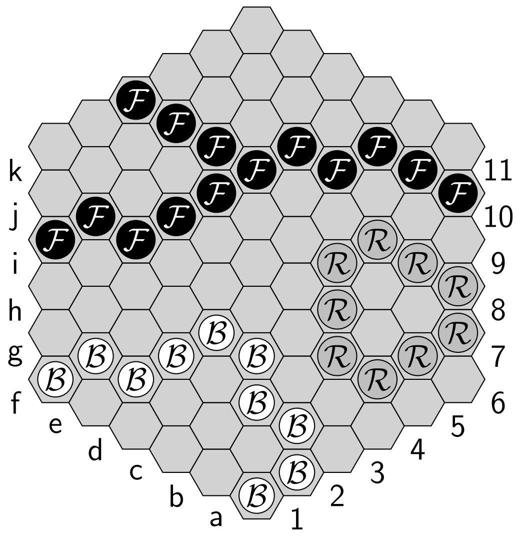


Figure 3.1: The Three Havannah Winning Conditions, as shown on a size 6 Havannah board

the stones labelled $\mathcal{B}$ in Figure 3.1.

- A *Fork* is a group of stones that connects any 3 edges (corners are not part of edges), for example the stones labelled $\mathcal{F}$ in Figure 3.1.
- A *Ring* is a group of stones that surround at least one cell (which can be empty or filled by either player), for example the stones labelled $\mathcal{R}$ in Figure 3.1.

The *size* of the board is defined as the number of cells along one edge, so the board in Figure 3.1 is size 6. A board of size n has $3n(n-1)+1 = 3n^2-3n+1$ cells, as listed in Table 3.1. Havannah can be played on any size board, but is usually played on boards ranging from size 4 to size 10. Stronger players prefer bigger boards, due to the larger component of strategy compared to the small boards where tactics dominate. In 2002, Christian Freeling offered €1000 for any program that beats him in just one in ten games on size 10 by 2012.

Havannah is played by a few thousand players around the world, primarily on Little Golem[1] and similar sites. It is also played by computer programs at the International Computer Games Association (ICGA) annual Computer Olympiads.[2]

3.2 Coordinate System

Several coordinate systems for specifying board locations exist. The one that will be used here was chosen because it has some nice mathematical properties[3] and because it is used in HavannahGui[4] and in the Little Golem[5] SGF files. An example board is shown in Figure 3.2a with each cell marked with its coordinate location. Figure 3.2b shows the same board as represented on a square grid. The empty points in the square grid are unused for the purposes of this representation. In the square representation connections are valid in the vertical, horizontal and $x = y$ directions, but not in the $x = -y$ direction. This square representation is often used to represent the board in memory. The size of the board is the number of cells along one short edge, or the radius of the board, not the diameter. Given this representation, the distance d between any two points (x_1, y_1) and (x_2, y_2) can be calculated as:

$$d = (|x_1 - x_2| + |y_1 - y_2| + |(x_1 - y_1) - (x_2 - y_2)|)/2$$

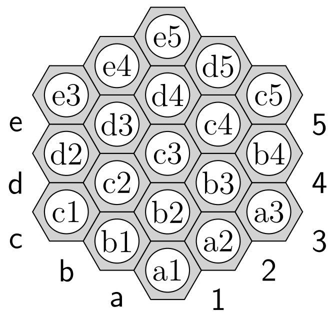
[1]http://littlegolem.net

[2]http://www.grappa.univ-lille3.fr/icga/

[3]http://www.iwriteiam.nl/Havannah.html

[4]http://mgame99.mg.funpic.de/havannah.php

[5]http://www.littlegolem.net



(a)

	1	2	3	4	5
a	a1	a2	a3		
b	b1	b2	b3	b4	
c	c1	c2	c3	c4	c5
d		d2	d3	d4	d5
e			e3	e4	e5

(b)

Figure 3.2: The Coordinate System (a) as drawn on a size 3 board. (b) as represented on a square grid.

3.3 State Space

A crude overestimation of the number of Havannah states is $T_0(n) = 3^n$ where n is the number of cells on the board. This includes many states that are unreachable purely based on the players having an uneven number of moves, such as all cells being played by player 1. A more accurate estimate is the sum of states where both players have made equal number of moves plus the sum of all states where player 1 has made one more move. This can be expressed as the formula:

$$T_1(n) = \sum_{i=0}^{(n-1)/2} \binom{n}{i} * \binom{n-i}{i} + \binom{n}{i+1} * \binom{n-i-1}{i}$$

These do not take symmetry or rotations into account, which gives approximately a 12 fold reduction in states: $T_2(n) = T_1(n)/12$. None of these approximations take the rules of the game into account, so includes positions where both players have winning formations or one player has multiple winning formations, but this is a much harder condition to approximate.

The state space complexity of Havannah is shown in Table 3.1, with various other board games listed for comparison. The Hex state space size is calcu-

Havannah	Cells	States (T_2)	Game	States	Ref
*Size 3	19	2×10^7	*Connect 4	10^{14}	[14]
*Size 4	37	6×10^{15}	*Checkers	10^{20}	[15]
Size 5	61	1×10^{27}	*Hex 8x8	10^{29}	[16] $T_1(8^2)/2$
Size 6	91	2×10^{41}	Go 9x9	10^{38}	[17]
Size 7	127	3×10^{58}	Chess	10^{46}	[18]
Size 8	169	3×10^{78}	Hex 11x11	10^{56}	$T_1(11^2)/2$
Size 9	217	2×10^{101}	Go 13x13	10^{80}	[17]
Size 10	271	1×10^{127}	Go 19x19	10^{171}	[17]

Table 3.1: State Space Complexity of Havannah. Other board games are shown for comparison, * means solved.

lated with the same formula except only two-fold symmetry. By comparison with the games that have previously been solved, size 3 Havannah should be trivial to solve, size 4 should be hard but possible with brute force, and size 5 may be possible but only if strong mathematical properties can be found to dramatically reduce the search space as is done in Hex.

3.4 Properties of Havannah

Havannah is considered hard for computers to play for several reasons, including the lack of a good heuristic evaluation function, few expert games, and a large state space complexity. Havannah is often compared to Hex, which is also a connection game, but few of the mathematical properties of Hex apply in Havannah. In this section some properties of Havannah are presented, especially in contrast to the better known properties of Hex.

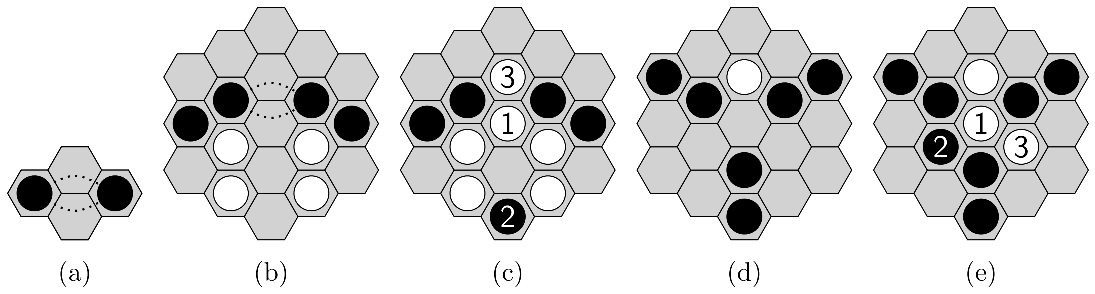


Figure 3.3: (a) A simple virtual connection. (b) Virtual connections are not guaranteed. (c) A threat can force a response other than maintaining the connection. (d) A false virtual connection between 3 groups can be broken, as shown in (e)

3.4.1 Virtual Connections

A virtual connection (VC) is a connection between two stones, or a stone and an edge, that can be completed even if the opponent makes the first move. A simple virtual connection is shown in Figure 3.3a. The two black stones are virtually connected, because if white plays in one of the two marked cells, black can complete the connection by playing in the other. In Hex, virtual connections are guaranteed, since there is no reason to not complete the connection, but this is not true in Havannah. In Figure 3.3b, black has a virtual connection between his two groups, but white can force a defence against a ring threat as shown in Figure 3.3c, which allows white to sever the black virtual connection. Such threats are rare in practice, but cannot be ignored.

Figure 3.3d shows a state where black's three groups intuitively look to be virtually connected, but aren't. If white plays in the center, black can choose which of his two top groups to connect to the bottom group, but he can't connect all three, as shown in Figure 3.3e. If white didn't have the first cell, then black could place there in move four, connecting all three groups.

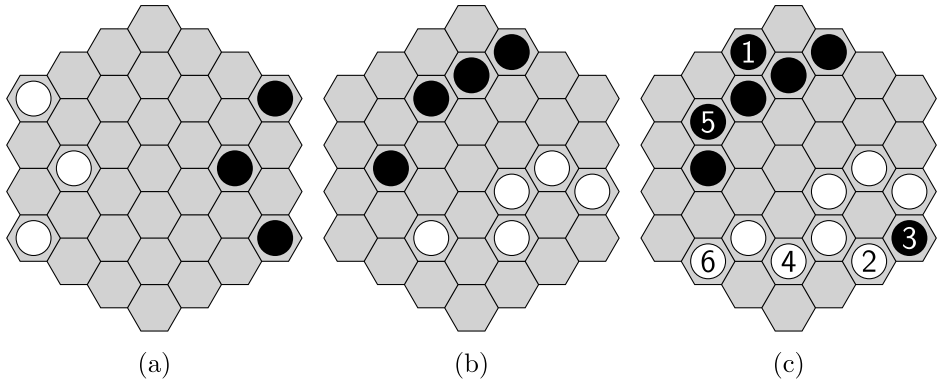


Figure 3.4: (a) Both players have a forced win in 2 moves. (b) Both players have a forced win in 3 moves. (c) Continuing from (b) with Black to move, White wins by threatening a faster ring connection.

3.4.2 Frame

A *frame* is a series of virtually connected stones or chains of stones that if connected would complete a winning condition. The length of the frame is the number of moves needed to complete the winning chain. Several frames are shown in Figure 3.4.

3.4.3 Simultaneous Forced Wins: Race to Win

In Hex, winning formations are mutually exclusive, so if one player has a forced win through virtual connections, he is guaranteed to win. This is because the winning conditions are side-to-side VCs which cross, so if one player has a side-to-side VC, the other player cannot also have one.

In Havannah winning formations are not mutually exclusive. Figure 3.1, though not a valid board configuration, shows three completed winning formations at the same time. While virtual connections can be broken, the formations needed to do so are often not present, in which case the virtual

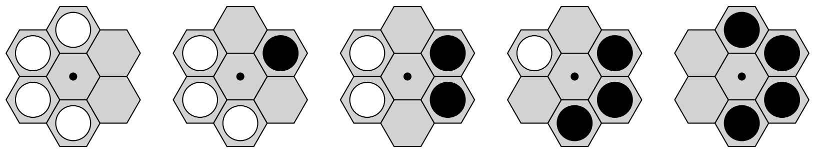


Figure 3.5: Hex-dead Cell Patterns. The cell marked with a dot cannot help either player form a winning connection in Hex.

connections are guaranteed. Figure 3.4a shows a situation where both players have a frame of length 2. This means they can force a win in two moves, so the first player to make a move wins. Figure 3.4b shows a situation where both players have a forced win in three moves, with black to move. One of white's moves threatens a faster ring victory, and although it is easily blocked, it gives white the move advantage and the win, as shown in Figure 3.4c.

3.4.4 Dead Cells

Strong Hex programs reduce the moves under consideration by avoiding playing in dead cells. Dead cells are cells that provably cannot affect the outcome of the game. They are dead because any chain of stones that passes through them already has a path through the existing stones. The five smallest Hex dead cell patterns are shown in Figure 3.5. Hex playing programs use these patterns to detect dead cells, then never consider playing in those cells, thereby reducing the branching factor.

Unfortunately the Hex-dead cell patterns don't apply in Havannah, because the Hex-dead cells can have an effect on the outcome of the game in Havannah. While they cannot affect a fork or a bridge win, they can still be part of a winning ring which surrounds one of the existing stones as illustrated in Figure 3.6. This idea can be used against all five of the smallest Hex-dead cell patterns in numerous ways.

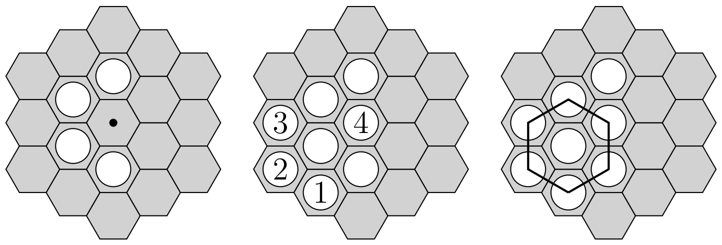


Figure 3.6: Hex-dead Cells Are Not Havannah-dead Cells. Starting with the Hex-dead cell pattern on the left, and adding stones 1-4 leads to the ring on the right which intersects the Hex-dead cell

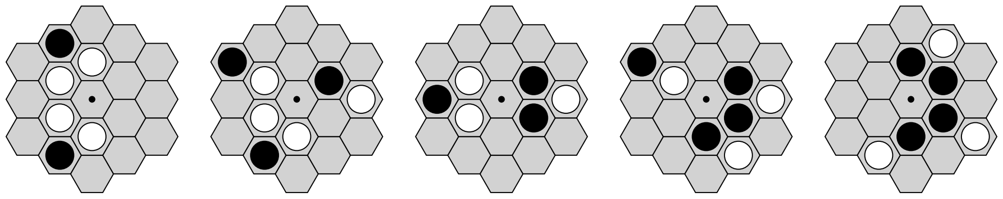


Figure 3.7: Some Havannah Dead Cell Patterns. The cells marked with the dot are dead in Havannah.

Note that if the ring is made bigger than a simple 6-ring, any ring that uses the dead cell would also work going around it. To create Havannah dead cell patterns, we add a stone of the opposing colour next to each existing stone to block the encircling 6-ring. A few examples are shown in Figure 3.7. Unfortunately these patterns are so rare that they aren't worth looking for explicitly.

3.4.5 Draws

Unlike Hex, where a filled board must have a winner, draws are possible in Havannah. Figure 3.8a shows a filled board of a game that ended in a draw. Backing up a few moves we see in Figure 3.8b that after move 31 no wins are possible even if one of the players were to pass all their remaining moves. The

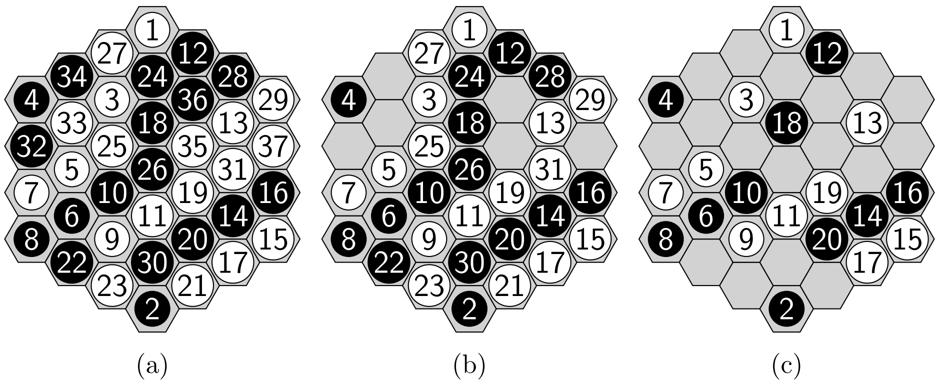


Figure 3.8: (a) Filled board ending as a draw. (b) After move 31 no wins are possible. (c) Proven draw after move 20.

game can be proven as a draw at least as early as move 20, shown in Figure 3.8c. Draws are possible on all board sizes above 3. They occur occasionally on size 4, but are rare on size 5 and above. Section 5.1.5 will describe a technique for detecting draws once no wins are possible.

3.5 Summary

Havannah is a recent game with fairly simple rules. It is played on a hex board, with the goal of connecting 2 corners or 3 edges, or forming a ring. It is generally played on board sizes 4 through 10. It is related to Hex, and shares many of the mathematical properties, but they are not as solid as in Hex. Virtual connections are formed the same way as in Hex, and are very important to Havannah strategy, but unlike Hex, they can be broken due to threats. Chains of virtual connections that, if completed, would form a win, are called frames. Multiple frames can co-exist on the board, and the player with the shortest frame will win unless an even shorter frame is constructed. Dead cells are cells can not affect the outcome of the game due to the pattern of stones that surround it, but the patterns are rare and more complex than the hex-dead cell patterns. Draws are rare but possible in Havannah, occurring when the board fills up without either player forming a winning formation.

4

Playing Havannah

These are early days for developing strong Havannah programs. The first published attempt at writing a program to play Havannah is a BSc thesis by Terry Rogers in 2004 [19]. In his thesis, he describes implementing several weak strategies, easily beaten by even the most amateur human player. Johan de Koning wrote an alpha-beta based Havannah program, named PZN, which competed in the 2010 ICGA Olympiad. It finished in last place because its heuristic function, which is quite similar to the distance to win heuristic shown in Section 4.8.6, is blind to rings, virtual connections and most threats, and is too slow for alpha-beta to reach sufficient depths to overcome these short-comings. The majority of attempts, and particularly published attempts, have been Monte-Carlo Tree Search (MCTS) based, primarily because MCTS does not need a heuristic evaluation function. This has also been influenced by MCTS's recent successes in the similar games of Go and Hex where MCTS has overtaken traditional techniques. Given this, MCTS is a good starting point for writing a Havannah player. This chapter will explore the performance of various search algorithms and enhancements as applied to Havannah using the Havannah program named Castro.

4.1 Castro

Castro is a strong Havannah program written by the author of this thesis, and released open source at `https://github.com/tewalds/castro`. It is written in C++, and includes an MCTS player and several solvers. It has a fast implementation of the rules, which are described in the Section 4.2. Castro speaks the GTP protocol,[1] and can be played against with HavannahGui.[2] The MCTS player is the main topic of discussion of the rest of this thesis, and is used both as a player and as a solver.

Castro includes an alpha-beta solver with iterative deepening and an optional transposition table. This alpha-beta solver is not useful as a player, as it lacks a heuristic evaluation of non-terminal nodes. It is the fastest of the solvers in terms of positions evaluated per second, but is rarely the fastest to solve non-trivial positions.

Castro includes three proof number search (PNS) solvers. The basic PNS solver is single-threaded and uses a tree data structure similar to the one described in Section 5.1.4. The second PNS solver is multi-threaded but uses the same tree data structure and includes garbage collection similar to Section 5.1.3. The third PNS solver is single-threaded but uses a transposition table

[1] Go Text Protocol: `http://www.lysator.liu.se/~gunnar/gtp/`

[2] `http://mgame99.mg.funpic.de/havannah.php`

instead of a tree. All three PNS solvers optionally use depth-first thresholds[20] as well as the $1 + \epsilon$ trick[21]. They also use a 2-ply lookahead, similar to Section 4.6. The tree based solvers can optionally use the distance to win heuristic described in Section 4.8.6, while the transposition table based solver can attempt to copy a proof to its siblings. The multi-threaded version could easily be turned into a player, but this has not been done yet.

4.2 Havannah Rules Implementation

The rules of Havannah are simple, and the three win conditions are easy to describe, but a fast implementation of the rules is not as obvious. This section will describe their implementation in Castro.

4.2.1 Fork and Bridge Connections

Both forks and bridges can be found in near $O(1)$ time using the union find algorithm. Each group includes a set of 12 bits, where each bit is associated with an edge or corner. Stones that are placed on a corner or edge set the associated bit on their group. As groups join, the bits for the groups are ORed together for the newly formed group. Once a group reaches two corners or three edges, that group forms a win condition.

4.2.2 Rings

Unlike forks and bridges, it is impractical to enumerate all the possible rings and wait for one of them to occur. Instead they can be detected with a limited search or with local patterns. Each of these approaches has advantages, so both are described here.

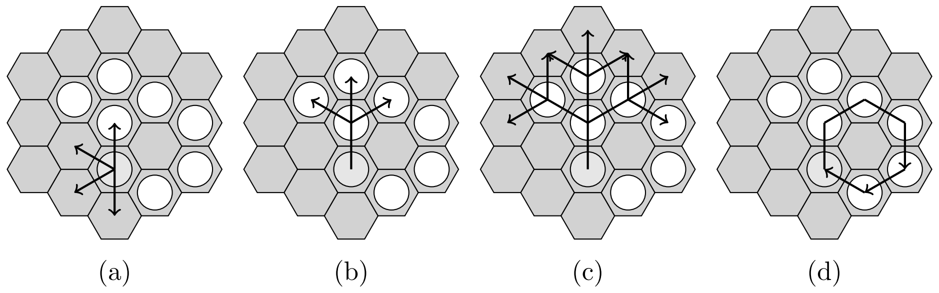


Figure 4.1: Search Ring Detection. Gray stone is the most recently placed and the start of the search. (a) Search after 1 step. (b) Search after 2 steps. (c) Search after 3 steps. (d) Search after 6 steps, ring found.

Search

The first approach, shown in Figure 4.1, is to start a search from the most recently placed stone. This search could be a recursive depth-first search or a breadth-first search. The recursive depth-first search was faster in testing, but the breadth-first variant is shown in Figure 4.1 for clarity. The search is only started if the group has at least six stones, and if the last stone joins one group of stones twice. From the starting stone, it searches in four adjacent directions (four is enough because any ring must start from one of the four directions even if it cycles back through the other two), continuing only in the forward direction to the next three stones. By avoiding sharp turns, the minimum cycle is 6 and any path back to the starting stone is a ring. This method could be quite slow when searching big groups but is fast in practice because the ring check can be skipped after most moves. It has the advantage that properties of the ring can be computed based on the stones that form it, as will be described in more detail in Section 4.9.4.

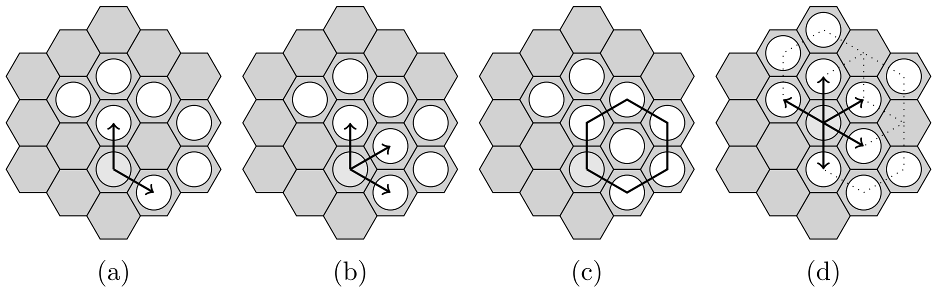


Figure 4.2: O(1) Ring Detection. Gray stone is the most recently placed. (a) Stone joins the white group twice with an empty stone between the two white stones, obviously a ring enclosing the empty stone. (b) Stone joins the white group three times, no empty stone, leads to (c) an extra check. (d) Worst case has 5 neighbours, and does 3 extra checks.

Patterns

Ring detection can be also be done in O(1) time, as shown in Figure 4.2, by using basic pattern detection. Rings occur when the most recently placed stone touches the same group twice with them being separated on both sides by empty space or the opponent's pieces. The only circumstance where that isn't true is a filled 6-ring, which can only happen in a small number of ways and is easy to detect. Figure 4.2a shows the common case where a stone joins a group twice and has empty space in the middle of the ring and on the opposite side. The surrounding 6 neighbours can be used as a pattern to lookup in a pattern database, which returns that it is a ring, isn't a ring, or that further checks are needed. If the center of the ring is filled, as is the case in Figure 4.2b, a check of the neighbours is enough to deduce that it might be a ring, but the remaining three cells must be checked to conclude it is a ring. The reason no bigger search is needed is because any bigger ring that passes through the newly placed stone would already be a ring passing through the existing stones and therefore would have been found earlier. The worst case, shown in Figure 4.2d is when the stone has 5 adjacent neighbours, in which case 3 extra checks for 6-rings are needed. This method is very fast in general, but needs to be done before joining the groups for the edge/corner checking. Doing the check before placing the stone likely means the conditions that allow the depth-first method to be skipped aren't known to be satisfied, meaning the check must be done for every move.

4.3 Testing Methodology

All testing was done using ParamLog,[3] an open source distributed testing framework written for this project. Tests were run on board sizes 4 through 10 with 1, 2, 5 and 10 seconds per move. The times are standardized to a machine that can complete 500,000 simulations on size 4 with basic UCT in 11 seconds, which is approximately equal to a Core 2 Duo 2.4ghz. This has the added effect of keeping results comparable across machines and in the face of optimizations. Only the results from the tests with 5 seconds are shown, because this has been common among other published papers about Havannah, but also because most test cases are fairly consistent between different amounts of time. All data points are based on several hundred games, which gives a maximum 95% confidence interval of ±5%, with many as low as as ±2%.

All tests are played relative to a baseline. UCT has strong guarantees about eventual optimality, so is the first baseline. After a few games, an exploration constant of 0.9 was chosen as a reasonable value to test against. RAVE has been shown by Teytaud to be strong in Havannah[22], and this finding will be confirmed in Section 4.4. For all tests thereafter, a RAVE player will be used as the baseline player. The RAVE baseline has no UCT exploration, keeps the tree between moves, and does 2-ply node expansion backups.

[3] https://github.com/tewalds/ParamLog

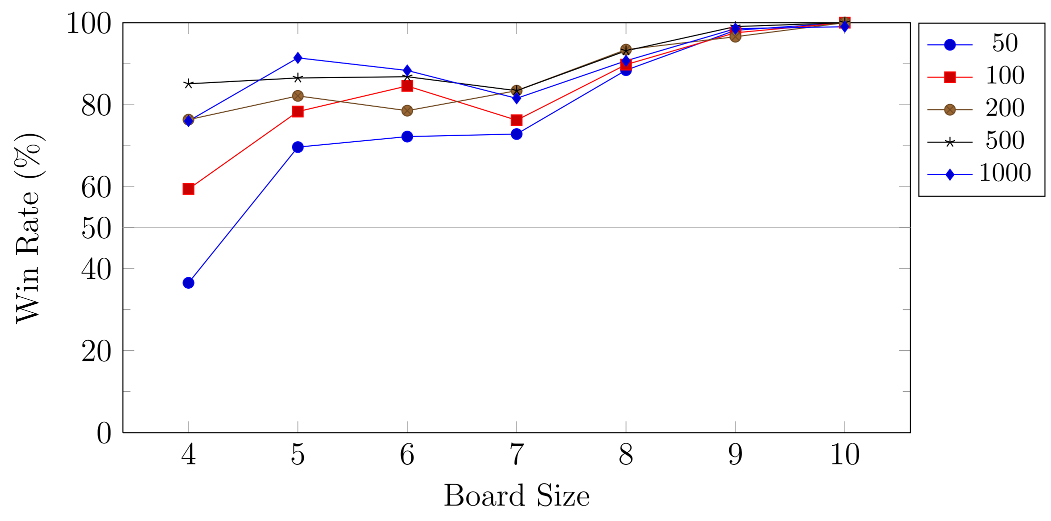


Figure 4.3: Rave vs UCT Baseline

4.4 RAVE

RAVE was introduced in Section 2.4.2 as a way to use the information from rollouts during future descents. Teytaud tested RAVE in Havannah in 2010[22], yielding a 100% winrate against UCT with a small number of simulations. Here we used the RAVE formula (2.4.3):

$$\beta = \frac{k}{k + n_i.n}$$

and test many values for k. Figure 4.3 shows the results. $k = 500$ yields a greater than 80% win rate on all board sizes, and so is chosen as the baseline for all future tests. This is equivalent to a 250 or more elo gain.

4.5 Keep Tree Between Moves

MCTS builds a large tree with real experience, and then makes a move. Most of the final move selection strategies will choose a large subtree usually consisting of more than 25% of the effort that went into this move choice, but often

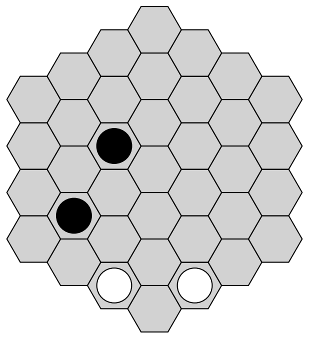


Figure 4.4: Early Position Solvable by MCTS in 1 Minute, white to play

consisting of as much as 95% of the work. This is real experience that would be duplicated after the next move if it was thrown away. The opponent's move is unknown at this point, but is frequently the expected reply, again often maintaining 25% or more of the tree. Keeping the remaining tree is a pure advantage with no downsides. Keeping the the tree has about a 53% winning rate against throwing away the tree after each move. This is not a huge improvement, but is easy, free and works on all board sizes.

4.6 Proof Backups

A game of Havannah ends when one of the winning conditions is met. This can happen with only a few stones on the board or with the board full of stones. In practice, the game rarely finishes with only a few stones, but frequently the tree includes positions where one player fails to respond correctly to a threat, leading the other player to win easily. MCTS is guided away from allowing these threats as the winning rate drops. By default it will continue trying this losing move as long as it looks better than the others, which could take quite a while. Late in the game, the game tree may be sufficiently small that the entire tree could be enumerated, but normal MCTS will take a long time to converge on the best move. If terminal nodes are marked, and winning

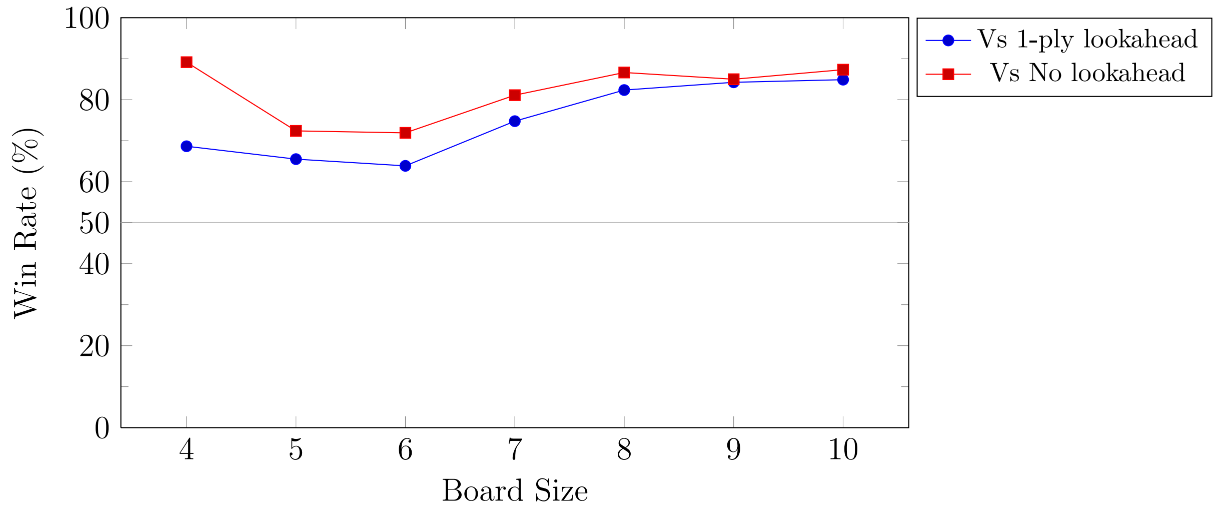


Figure 4.5: RAVE Baseline with 2-ply lookahead during expansion vs 1-ply lookahead and no lookahead

moves are backed up as losing moves for their parents and sets of losing moves are backed up as winning moves for their parents, full proof trees can be constructed, allowing faster detection of threats, and proving the outcome of moves late in the game. This has been suggested before, and was explored by Lorentz in 2010 with moderate success [23].

Terminal nodes are found in the expansion phase. If a winning move is found, the expansion stops and the parent node is marked as a win. If no winning moves are found, then threats that must be blocked are looked for. If the opponent has exactly one immediate threat, it must be blocked, so all other moves are discarded, and the simulation is continued, expanding that move's children. If the opponent has two or more immediate threats, only one could be blocked, and so the expansion is stopped and the parent is marked as a loss. These outcomes are then propagated as far up the tree as possible.

Lorentz showed a board position, shown here in Figure 4.4, which his program, Wanderer, solves in 45 minutes. Using RAVE and proof backups but a purely random rollout policy and no heuristic knowledge, Castro solves it in under a minute.

Figure 4.5 shows the results from using proof backups. It compares using 2-ply versus 1-ply and 0-ply lookahead. One ply means only checking for winning moves but not checking for the opponent's threats. Two ply means checking for winning moves and for the opponent's threats. Backing up 2-ply proofs gives a 75-90% winning rate on all board sizes, which is worth about 200-300 elo. At the olympiad, Castro solved the outcome of all 16 games about 15 moves before the end. Because it keeps the tree between moves, the remaining moves were played out of its precomputed proof tree.

4.7 Multiple Rollouts

Profiling was used to determine where Castro spends its time. Table 4.1 shows how time is used given 10 seconds to play on an empty board, on board sizes 5 and 10. When using UCT without RAVE, the most of time is spent in rollouts, followed by descent. Expansion and back-propagation, by contrast, take a relatively small amount of time. When using RAVE without exploration, descent takes the most time, with back propagation and rollouts taking most of the rest. Compared to using UCT, RAVE spends less time in rollouts, and more time in descent and back-propagation. While the optimal ratio of time is unknown, spending time evaluating nodes intuitively seems to be a better use of time than the bookkeeping associated with figuring out which nodes to evaluate. To alleviate this, multiple rollouts were run per simulation. This would lower the number of full simulations, but increase the number of rollouts. Given that rollouts are used to approximate the value of a given node, this should increase the accuracy of the experience in the tree, and hopefully the playing strength.

The experience from the rollouts are aggregated before the back-propagation

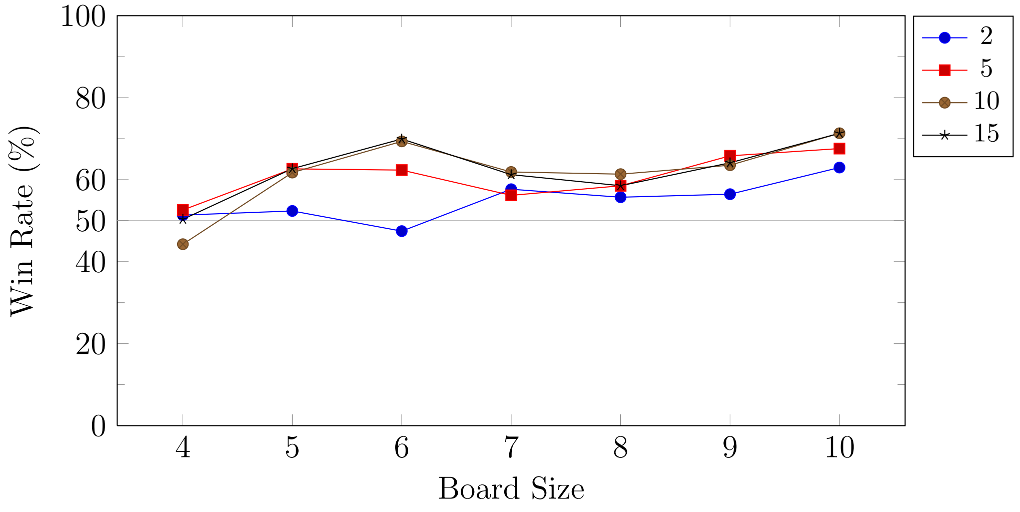


Figure 4.6: Multiple Rollouts per Simulation Against Baseline RAVE Player

phase so only a single update is needed for the RAVE and experience values of each node in the tree. Figure 4.6 shows the results when doing multiple rollouts per simulation, given the same amount of time. Multiple rollouts only has a small benefit on size 4, but is worth 50-100 elo on sizes 5 and up. Two rollouts has a mean winning rate of 55%, while 15 rollouts has a mean winning rate of 63%. For comparison, Tables 4.2 and 4.3 show the time used for 2 rollouts per simulation and 10 rollouts per simulation respectively. Doing 2 rollouts per simulation drops the number of simulations to two thirds, and doing 10 rollouts per simulation drops it to a fifth to a third as many simulations. This is a big drop in simulations, but the total number of rollouts increased more than enough to compensate. This has the added benefit of building a smaller, more accurate tree, reducing memory usage.

4.8 Heuristic Knowledge

Heuristic knowledge was described in section 2.4.3 as a way to bias the descent policy towards moves that are likely to be good before any experience has

	Iterations	Descent	Expansion	Rollout	Back-propagation
UCT size 5	296136	33.6%	12.0%	48.4%	5.9%
UCT size 10	102192	24.8%	11.3%	62.0%	2.0%
RAVE size 5	148713	45.1%	7.3%	22.2%	25.4%
RAVE size 10	41713	42.9%	4.8%	24.3%	28.0%

Table 4.1: Time Used by MCTS Phase with 1 Rollout per Simulation

	Iterations	Descent	Expansion	Rollout	Back-propagation
UCT size 5	201408	22.9%	7.0%	66.1%	4.0%
UCT size 10	66945	16.3%	3.5%	78.9%	1.3%
RAVE size 5	115704	34.7%	6.0%	32.6%	26.6%
RAVE size 10	31962	29.4%	4.0%	37.1%	29.6%

Table 4.2: Time Used by MCTS Phase Using 2 Rollouts per Simulation

	Iterations	Descent	Expansion	Rollout	Back-propagation
UCT size 5	58786	6.5%	1.4%	91.0%	1.1%
UCT size 10	16616	3.9%	0.4%	95.4%	0.3%
RAVE size 5	49131	14.8%	3.4%	65.6%	16.2%
RAVE size 10	12232	10.7%	1.9%	71.1%	16.3%

Table 4.3: Time Used by MCTS Phase Using 10 Rollouts per Simulation

been gained. Castro implements an extra additive knowledge term using the formula:

$$\frac{n_i.k}{\sqrt{n_i.n}}$$

This gives a boost to the node that is independent of the exploration bonus from UCT and independent of the β value for RAVE. Each heuristic has some small value, which is multiplied by a tuning constant. The different heuristic values are summed for the final knowledge value $n_i.k$ for that node.

4.8.1 Maintain Virtual Connections

Virtual connections (VC), as shown in Figure 3.3a, are a fast way of ensuring two groups can connect or that a group can connect to a remote part of the board. However, this is only true if they are maintained. If the opponent places a piece in one of the two empty cells, a response of playing in the other cell is usually a good move. Thus, a bonus should be given to maintaining these virtual connections.

Figure 4.7 shows the results of adding a knowledge bonus of 5, 10, 25, 50 or 100 for maintaining the VC. It seems to work very well on a few sizes with a winning rate as high as 70%, but in general it does not have a big effect on the larger board sizes. This may be because RAVE inherently finds these moves fairly quickly anyway, or because it focuses more on rings on the larger board sizes, in which case VCs aren't very important.

4.8.2 Locality

Large parts of the board are often empty and far from any existing stone, and playing there is rarely a good move. Giving a bonus to positions near existing stones is an intuitively good heuristic. We tested giving a bonus to being near

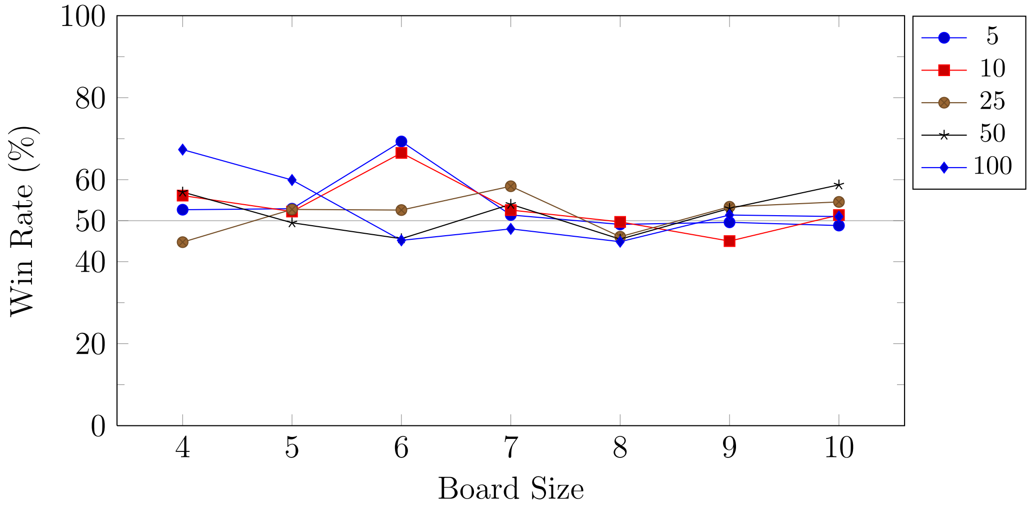


Figure 4.7: Maintain Virtual Connection Bonus Against Baseline RAVE Player

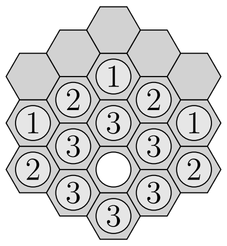


Figure 4.8: Points Given by Distance From an Existing Stone

any stone, and also to being near stones of your own colour. The bonus drops off with distance, with a bonus of 3 for a direct neighbour, 2 for forming a VC with an existing stone, and 1 for being distance 2 but not forming a VC. An example of the bonuses around a newly placed stone are shown in Figure 4.8.

First, giving a bonus to being near any stone regardless of colour was tested. The results are shown in Figure 4.9. While it seemed to help a little bit on bigger boards, in general it was not helpful.

Next, giving a bonus for playing near stones of your own colour was tested, as this has been shown to work before [23, 24]. The results are shown in Figure 4.10, which is much more encouraging. A value of 10 gives a winning ratio of

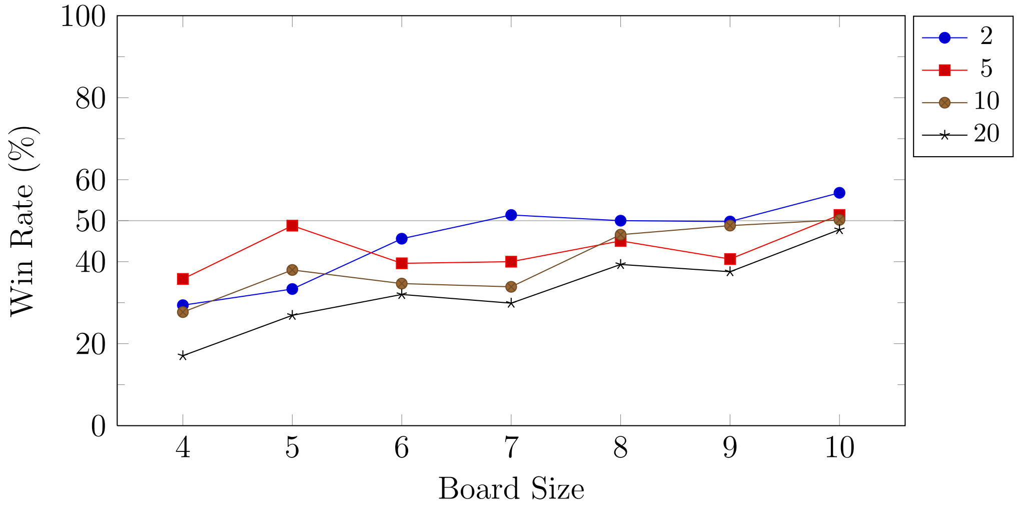


Figure 4.9: Locality Bonus for Playing Near Existing Stones of any Colour Against Baseline RAVE Player

65%, or about 100 elo on most board sizes.

4.8.3 Local Reply

Playing well in Havannah often means making defensive moves, replying to the opponent's last move. A bonus is given based on the distance to the opponent's last move. Direct neighbours get a bonus of 3, moves two away get a bonus of 2, and moves three away get a bonus of 1.

The results are shown in Figure 4.11, showing this heuristic to be not very helpful except on board size 4.

4.8.4 Edge Connectivity

Connecting to edges and corners is important for eventually winning the game, as most games end in a bridge or fork victory. Connecting to an edge is

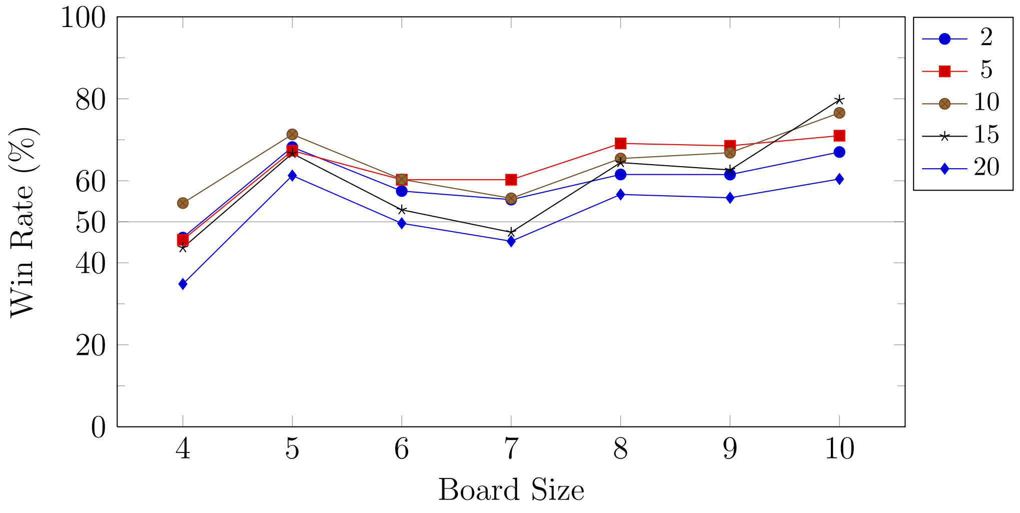


Figure 4.10: Locality Bonus for Playing Near Stones of the Same Colour Against Baseline RAVE Player

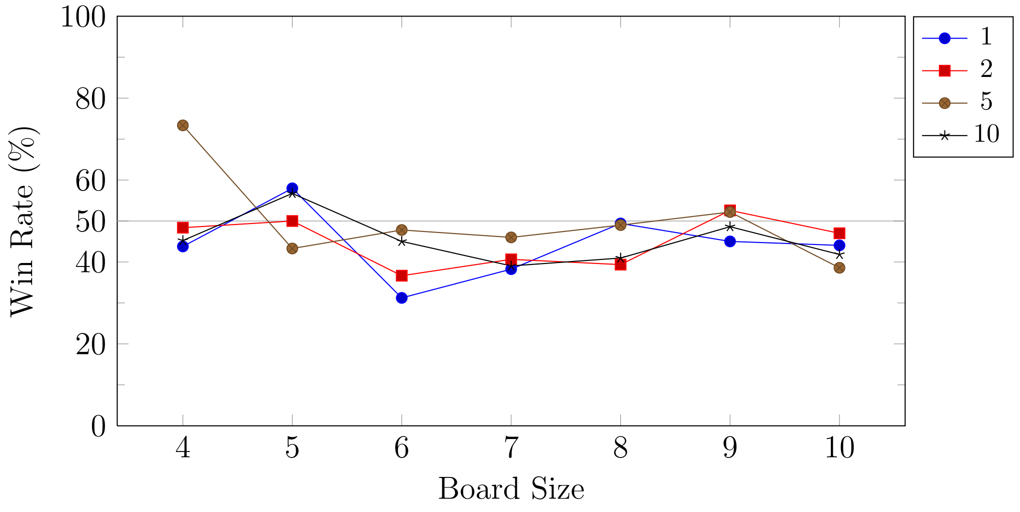


Figure 4.11: Local Reply Bonus for Playing Near the Opponent's Last Move Against Baseline RAVE Player

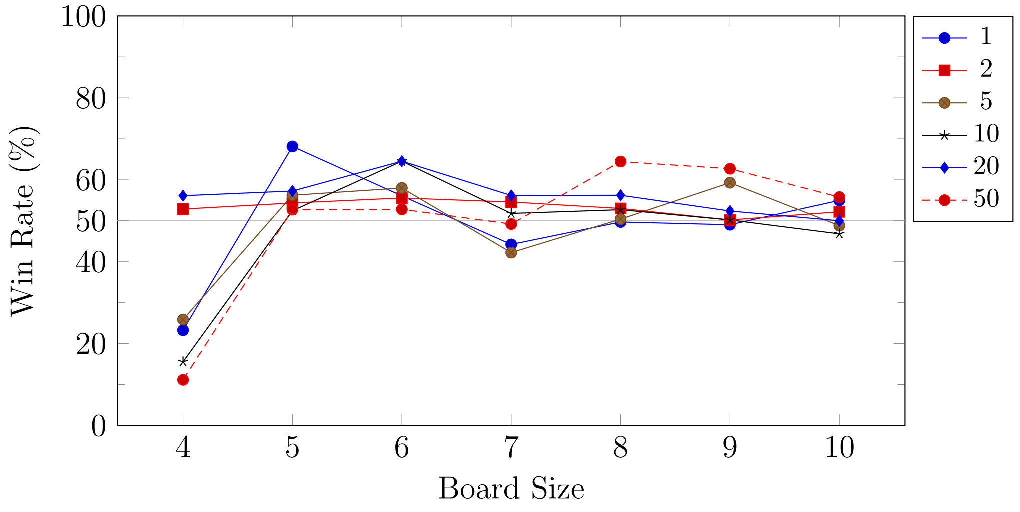


Figure 4.12: Edge Connectivity Bonus for Playing Near Groups that are Connected to an Edge or Corner Against Baseline RAVE Player

important, and extending groups that are already connected to an edge or corner is a good strategy for winning. The edges and corners that each group is connected to is already stored and updated for easy win detection. A bonus is given based on the number of edges or corners it is connected to. The maximum number of edges or corners a group could be connected to is 3, being 2 edges and 1 corner, so this gives a bonus between 0 and 3.

Figure 4.12 shows the results. While no single value is strong on all board sizes, some value is strong on each board size, giving a winning rate between 55% and 65%, or 50 to 100 elo.

4.8.5 Group Size

Winning formations must span large sections of the board, and tend to be big. Letting a large group get cut off and made useless is rarely a winning strategy. Connecting smaller groups into larger groups is often a good idea. All of these suggest playing near and forming big groups, so here a bonus is given based on

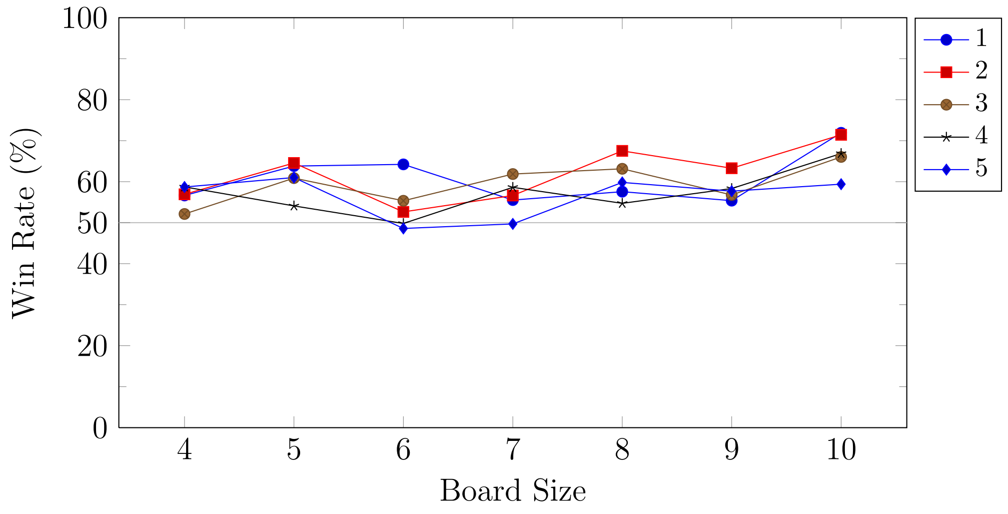


Figure 4.13: Group Size Bonus for Playing Near or Forming Big Groups Against Baseline RAVE Player

the size of the group. Placing a lone stone has no bonus. Joining two groups has a bonus equal to the sum of the sizes of the two groups. Joining a group has a bonus equal to the existing size of the group.

The results are shown in Figure 4.13. A winning rate over 60% can be achieved on all board sizes.

4.8.6 Distance to Win

Most Havannah games end in either a bridge or a fork victory. It is possible to approximate the minimum number of moves needed to achieve these win conditions by doing a flood fill from each corner and edge, and counting the distance to each cell from the starting point. The distance between two cells that are part of the same group is counted as zero, since no additional stones are needed to connect them. The distance between any other neighbouring cells is one. Once the distances to each corner and edge are computed, the number of moves needed to achieve a victory from that cell is the sum of the

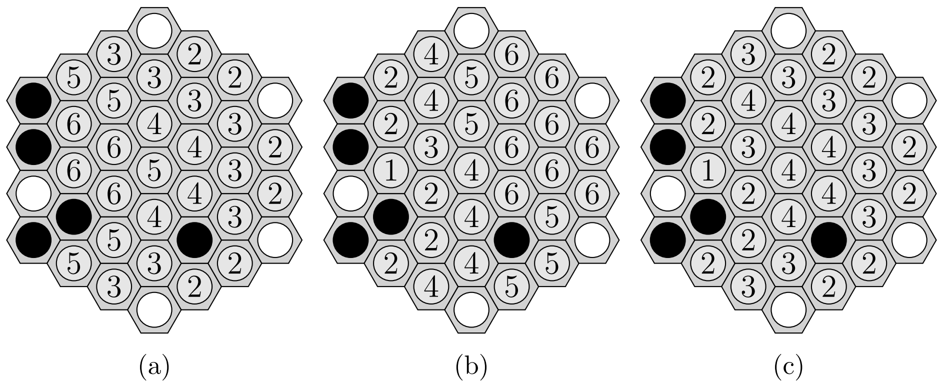


Figure 4.14: Distance to a Win for (a) White (b) Black (c) Minimum of White and Black

shortest two distances to corners or the sum of the shortest three distances to edges. This method does not consider ring victories at all. It can over-estimate the number of moves since it doesn't consider that the paths to the corners or edges may share part of the path. It does not consider any moves by the opponent, which could easily increase the distance by blocking an essential move, or even make the connection impossible. It does not take virtual connections into account. All that said, it is a reasonable estimate in many cases. It is slow to compute, as it requires 12 flood fills for each player, but in doing so computes the minimum distance for every cell on the board. An example is shown in Figure 4.14, showing the minimum distances for white, black, and the minimum of the two. Note how the immediate threat has a distance of 1. White has low distances on the right where it dominates, while black has low distances on the left where it dominates.

To turn this heuristic into usable knowledge, a small distance must give a larger bonus, while a large distance must give a small bonus. To accomplish this, the distance is subtracted from double of the board size, with a minimum value of zero.

First, using the minimum of the distances for the two players was tested, as in

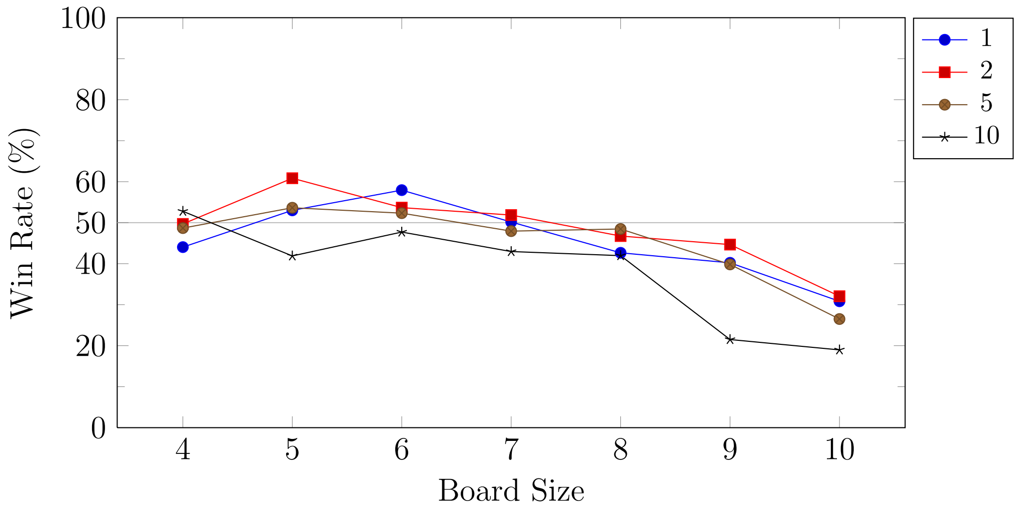


Figure 4.15: Minimum Distance to Win Bonus Against Baseline RAVE Player

Figure 4.14c. This takes both offensive and defensive positions into account. The results are shown in Figure 4.15. On smaller boards this method achieved a winning rate as high as 60%, though failed to achieve any success on the bigger boards.

Next, using only the distance for the current player was tested, as in Figures 4.14a and 4.14b. This is a much more offensive metric, ignoring the opponent's distance to win. The results are shown in Figure 4.16. This method achieves especially good results on small boards with a winning rate as high as 70%, but achieved positive results up to size 9.

4.9 Rollout Policy

The other main way to improve MCTS is by changing the move choice in rollouts so that the outcome of the rollouts is more representative of the real strength of the position. Usually this means using simple patterns or heuristics to make moves that are more sensible than choosing moves randomly.

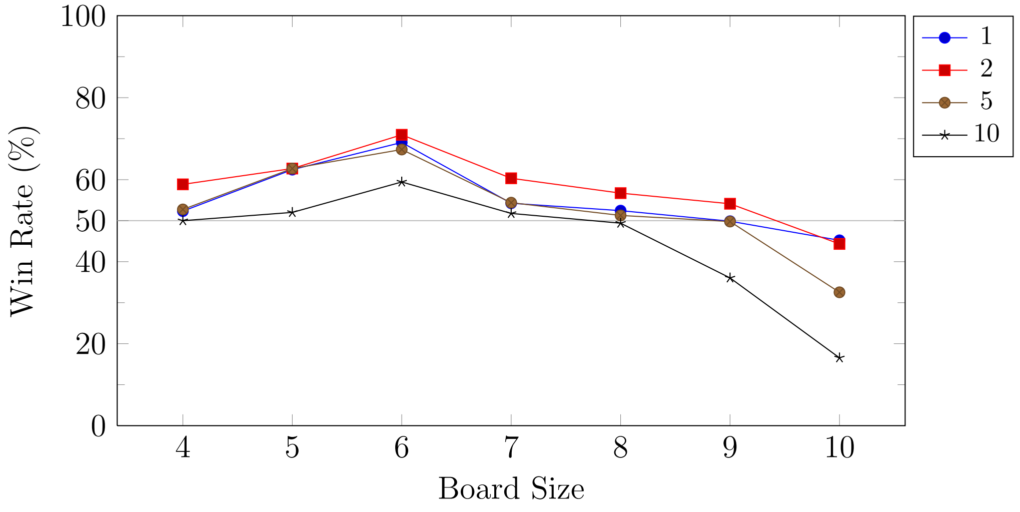


Figure 4.16: Own Minimum Distance to Win Bonus Against Baseline RAVE Player

4.9.1 Mate-in-one

If the opportunity to win with a single move (mate-in-one) exists, a smart player would make that move. Similarly, if the opponent has a threat to win in a single move, a smart player would block that move. Lorentz tested mate-in-one checks, and the defensive variant, to great success. Fossel also tested mate-in-one checks achieving a 59% win rate on size 5 with equal time[25]. Because checking for win conditions is a slow operation, both Lorentz and Fossel achieved the greatest success when limiting the checks to only the first few moves of the rollout, where they are most representative of the true board state.

Mate-in-one checks are very slow. The obvious way of implementing it is to check every empty cell for each player for whether a move there would complete a win condition. This is $O(n)$ of the number of cells, which makes rollouts $O(n^2)$. A simple optimization is to only do the win check if that cell is next to an existing stone, but this is still $O(n)$. A much better algorithm is to only do win checks on the empty cells next to the group that was last played, which is roughly $O(g)$ where g is the group size and $g \ll n$. This works because we are making the assertion that no mate-in-one moves exists before the last move, and so a new mate-in-one must be related to the last played stone. If the last played stone creates one new winning cell, we must block it. If it creates two new winning cells, it has a guaranteed win, and so we just end the rollout early. Castro does not have a facility to enumerate the empty cells surrounding a group, so instead it does a walk around the group. There must be a position around the cell that is not part of the same group, even if it is off the board. If this was not true, then it would be in the center of a ring, and the game would already be finished. It then follows the edge of the group, checking only whether placing an additional stone in the neighbouring empty cells would turn that group into a winning structure. The walk ends when it gets back to the first position searched. An additional untested optimization would be to only do this check if the last placed stone has a connection to an edge or corner, though this ignores ring threats.

We tested checking for mate-in-one moves after every move, and limiting the checks to only to the first N moves with N equal to the board size and double the board size. The results are shown in Figure 4.17. The best result is achieved exactly where Lorentz and Fossel reported their success, on size 5, but all other cases were inconclusive or quite negative. Discussions with other Havannah programmers suggested mate-in-one checks become more important as the time per move is increased, so size and size*2 were tested with 30 seconds per move, with the results shown in Figure 4.18. The results are better, with gains as high as 100 elo.

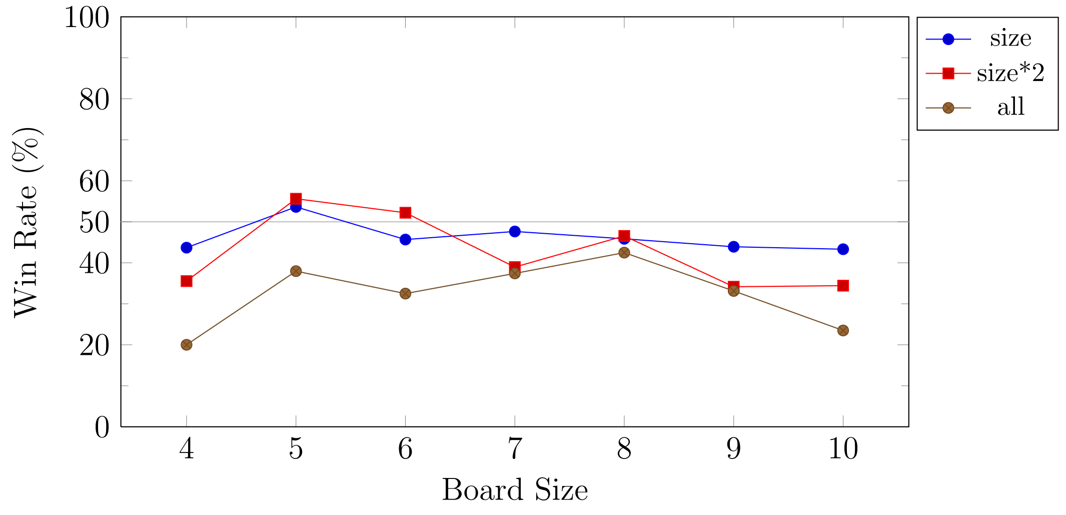


Figure 4.17: Mate-in-one Checking Against Baseline RAVE Player With 5 Seconds per Move

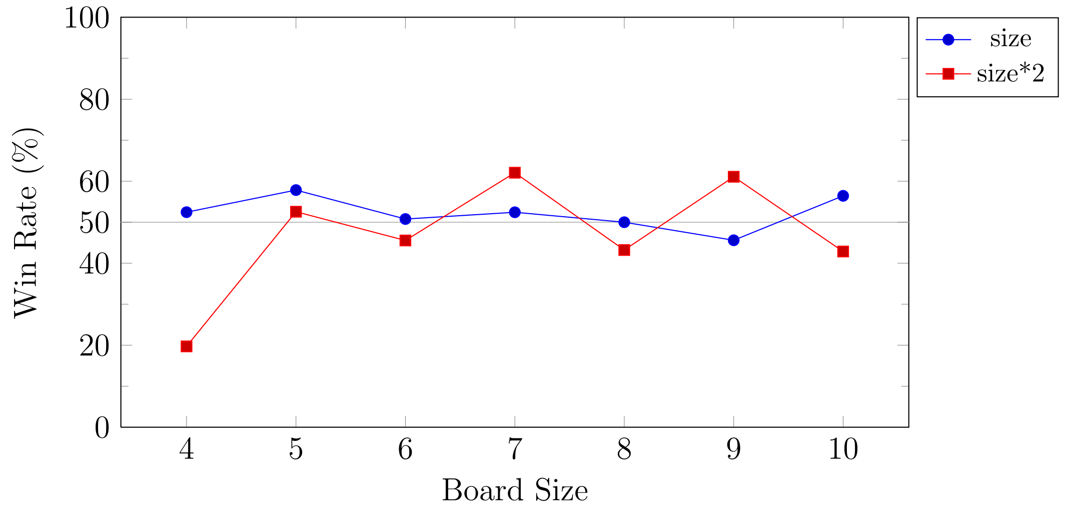


Figure 4.18: Mate-in-one Checking Against Baseline RAVE Player With 30 Seconds per Move

4.9.2 Maintain Virtual Connection

Maintaining virtual connections is important, as chains of virtual connections can form frames that, when completed, form a winning condition. If an opponent places a stone in one of the two parts of a VC, the response should be to play in the remaining empty position that used to form the VC. This can be implemented with a small state machine looking at the neighbours of the last placed stone, looking for the pattern of your stone, empty, your stone. If this pattern is found, the empty cell between your two stones is the correct response to maintain the virtual connection. Maintaining a connection to an edge is also important, so the pattern still works if either of your stones is off the edge of the board, as in your stone, empty, off the board. This pattern is checked after every move. If no pattern is found, then the default policy is used, resulting in a random move being chosen.

Looking for this pattern is extremely fast, causing no slowdown in terms of simulations per second. The time taken for the pattern lookup is offset by the approximately 10% shorter rollouts. The results are shown in Figure 4.19, showing a negligible or even negative result.

4.9.3 Last Good Reply

Last Good Reply (LGR) is a game-independent method of improving the moves in a rollout by taking advantage of local situations. It attempts to reuse good moves in later rollouts by saving all the replies by the winning player and forcing the reply in later simulations if the reply is valid.

LGR only saves the replies to individual moves. If a good reply is known to the opponent's last move, it is made instead of a random move. If that move is invalid, a random move is made instead.

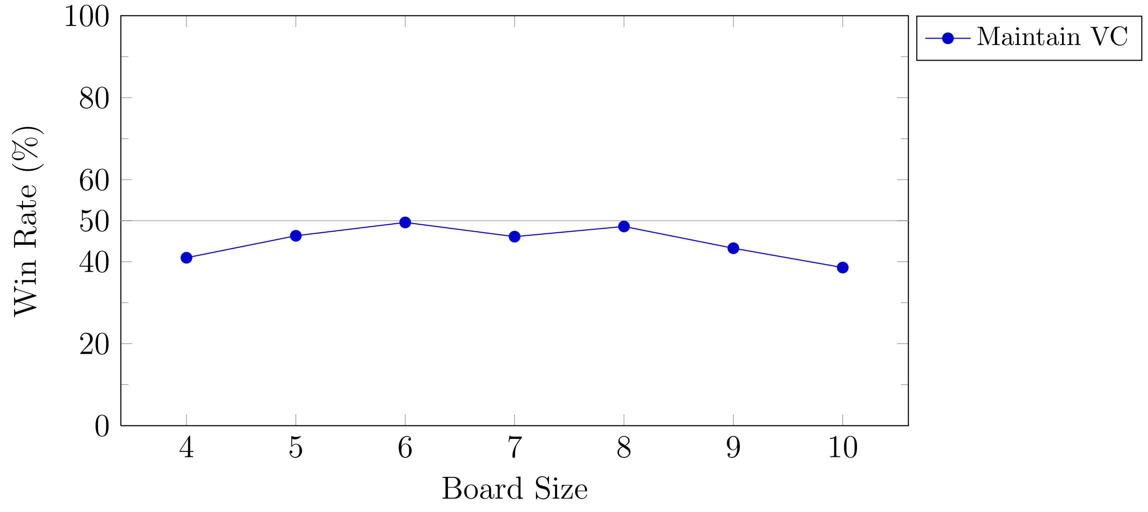


Figure 4.19: Maintain Virtual Connections in the Rollout Against Baseline RAVE Player

If a reply was played, but the outcome of the rollout is a loss, that may not be such a good move, and so it should be forgotten. This variant, called Last Good Reply with Forgetting (LGRF), removes the replies for all the moves made by the losing player. This increases the amount of randomness in the rollouts as compared to LGR.

Figure 4.20 shows the results. LGR only works on size 4, while LGRF is helpful on sizes 6 and up, leading to a 50-100 elo performance gain.

4.9.4 Ring Rule Variations

The three win conditions happen at different frequencies on different board sizes, as shown in Table 4.4. On size 4, a bridge is the most frequent win type followed by fork and the relatively infrequent ring victory. As the board size increases, it becomes harder for bridges and forks to form as they scale with the size of the board, but rings become relatively easier, because their size is independent of the board size and they are local formations. Given

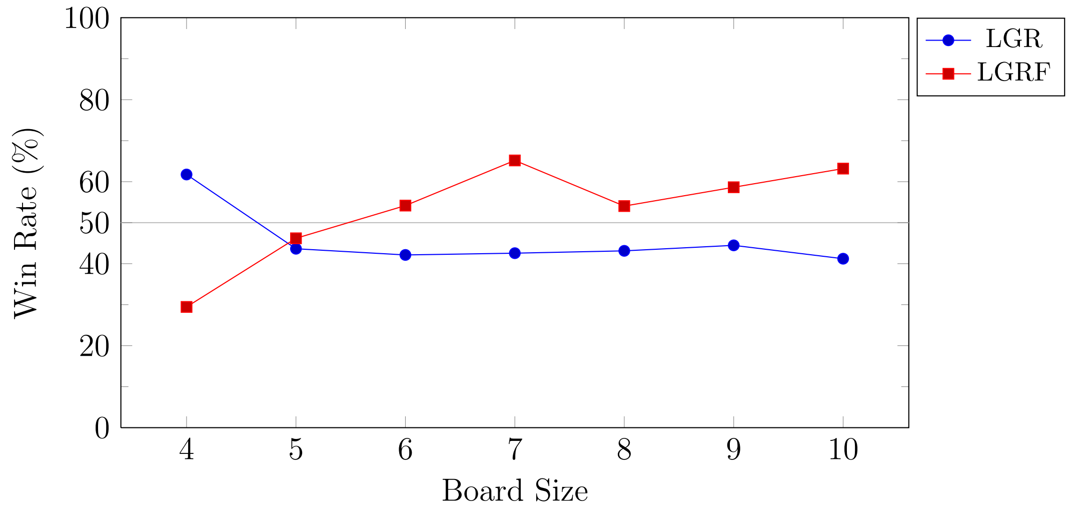


Figure 4.20: Last Good Reply Against Baseline RAVE Player

empty space on the board, a ring is likely to occur purely by making random moves. Rings are quite easy to block however, so while they happen frequently in random games, they are rare in real games. MCTS is very sensitive to the outcome of rollouts not being representative of the strength of the starting position, so this mismatch is a problem.

Four different solutions to this problem were explored. Instead of modifying where stones are placed, the rules of the game are modified. If a ring is formed, but the modified rules deem it to be a product of randomness instead of the product of good strategy, it is simply ignored and the rollout continues until a different winning condition occurs. Note that this only affects rings formed during rollouts. If the tree reaches a position with a ring, it is still considered a win regardless of the rule modifications considered here.

Ignore all rings

The simplest rule modification is to simply ignore all occurrences of rings during some fraction of the rollouts. At the beginning of the rollout a random

Size	Fork	Bridge	Ring
4	4267	5177	543
5	5111	2962	1926
6	4471	1691	3838
7	3536	1007	5457
8	2266	460	7274
9	1365	229	8406
10	796	126	9078

Table 4.4: Number of Wins of Each Type by Board Size Given 10000 Simulations

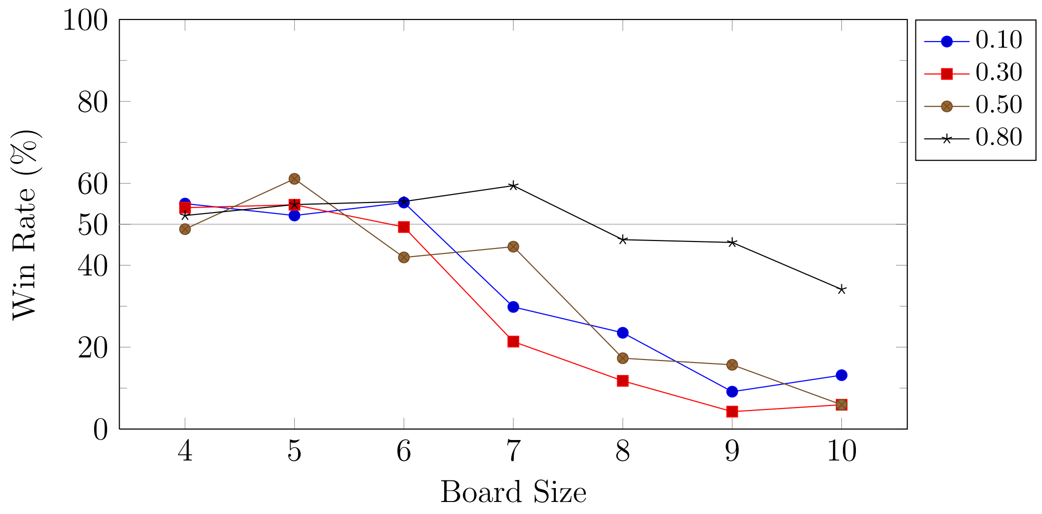


Figure 4.21: Ring Rule Ignore Rings Against Baseline RAVE Player

Size	Fork	Bridge	Ring
4	29.56	26.34	28.00
5	48.13	44.77	44.36
6	71.66	68.01	65.07
7	98.91	93.74	89.14
8	131.29	126.29	116.38
9	167.01	160.12	146.13
10	206.35	196.88	177.64

Table 4.5: Average Number of Moves in a Rollout Before Each Victory Type

number is generated and if it is below the configurable fraction, rings are checked, otherwise all rings are ignored.

The results are shown in Figure 4.21. There are minor successes on smaller boards, but bad results on bigger boards where rings are a problem. The bad results happen because it completely misses the opponent's real and faster ring threats, instead focusing on its own slower bridge and fork victories. By the time it notices the opponent's rings, it is too late to block them. This effect may be less pronounced against human opponents that are less likely to make ring threats, but it would introduce a new weakness that could be exploited.

Fixed depth

Table 4.5 shows the number of moves before each win condition is achieved, showing that rings occur the earliest, especially on big boards. This is expected due to their smaller size and being local properties. If instead of ignoring all rings, we allow rings for the first part of the rollout and ignore all rings after that cutoff, we may allow legitimate ring threats while ignoring rings that occur purely due to randomness. As the board size increases, this depth should also be increased, so instead of an absolute value, we use a fraction of the number

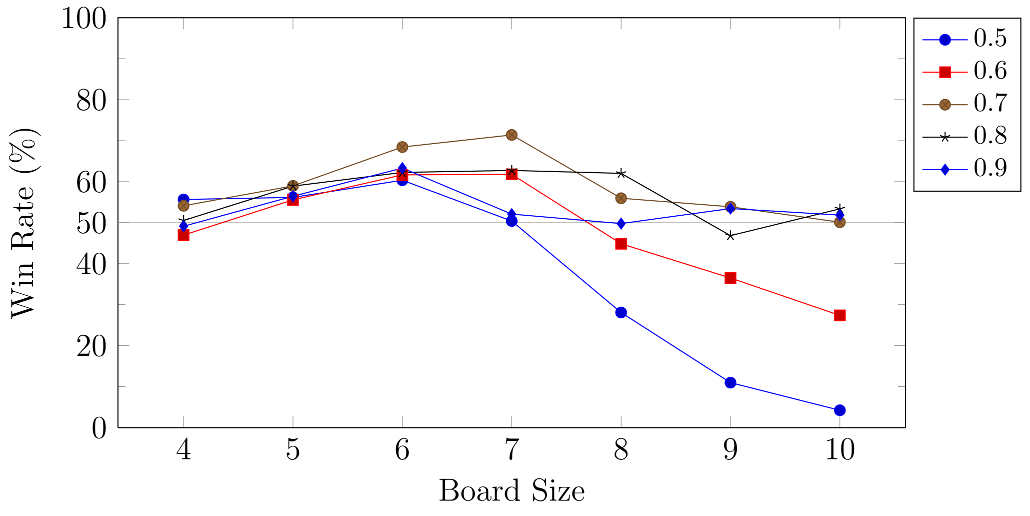


Figure 4.22: Ring Rule Fixed Depth Against Baseline RAVE Player

of empty cells on the board at the beginning of the rollout.

The results are shown in Figure 4.22, showing results as high as a 71% winning rate for only considering rings until 70% of the empty cells are played. The benefits on board sizes 8, 9 and 10 are still quite small though, where the problem is the worst.

Ring size

The most commonly occurring ring is the simple size 6 ring, which is easy to block. We may want to allow larger rings while still ignoring the easy to block 6 ring. Using the search-based ring check described in Section 4.2.2, we can ignore rings smaller than a minimum value. In this test we start by allowing all rings, and after every N moves, we increase the minimum size by 1 for variable N. After the first N this would imply only rings of size 7 or bigger, then 8 or bigger, etc. Here we set N as a fraction of the remaining empty cells.

The results are shown in Figure 4.23, again showing reasonable results on board

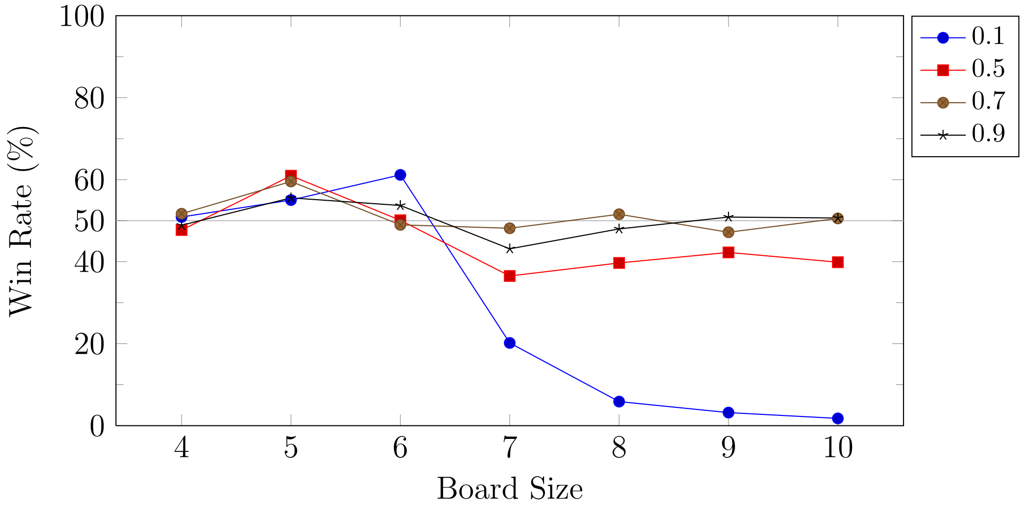


Figure 4.23: Ring Rule Ring Size Against Baseline RAVE Player

sizes 5 and 6 with a winning rate of 60%, but failing to show any improvement on larger board sizes. A fraction of 0.7 or 0.9 is so high that it never reaches a minimum size bigger than 7, though this is enough to ignore size 6 rings.

Permanent stones

If the problem is that random moves form rings in empty parts of the board, a solution may be to only consider rings that include stones that exist on the board before the rollout begins. Thus ring threats may be formed from existing stones, but not purely out of randomness with little relation to the stones already existing on the board. In this test, stones that exist before the rollout begins are marked as permanent stones, and any ring found during the rollout must have a minimum number of permanent stones to be considered a winning formation. Requiring between 1 and 5 permanent stones was tested, with the results shown in Figure 4.24. This shows a 50-100 elo gain on all board sizes. In human play it makes reasonable ring threats, but is no longer fixated on rings.

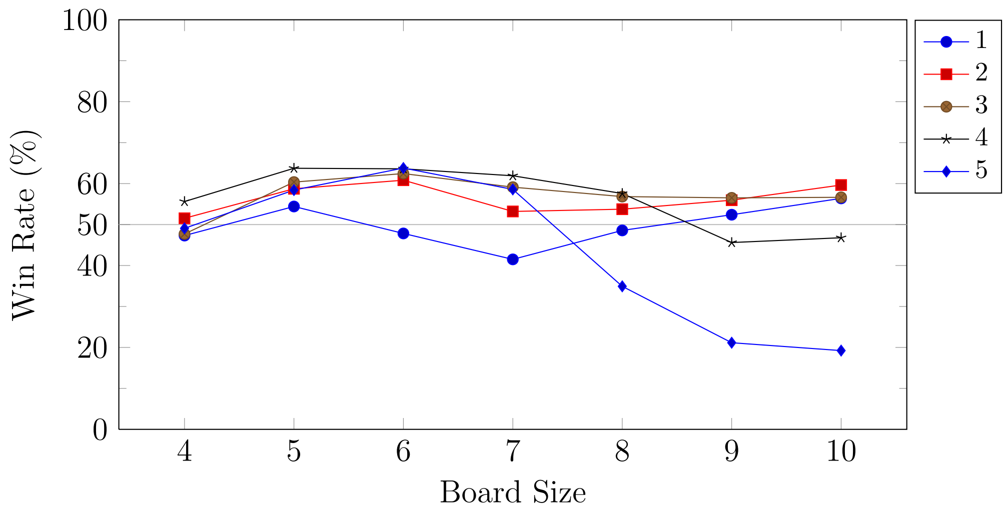


Figure 4.24: Ring Rule Permanent Stones Against Baseline RAVE Player

Size	Fork	Bridge	Ring
4	4212	5644	136
5	5986	3773	236
6	7051	2321	626
7	7444	2132	422
8	8283	1397	319
9	8635	929	436
10	8768	971	261

Table 4.6: Number of Wins of Each Type by Board Size Given 10000 Simulations When Only Counting Rings With Three or More Permanent Stones

The results of using this feature when requiring a minimum of 3 permanent stones are shown in Table 4.6, and show a stark contrast to Table 4.4 when not using this feature. The number of forks increases quite dramatically as the board size increases, especially relative to without this feature. This is expected due to the increasing amount of edges available, and the proportionately smaller increase in the number of stones needed to form a fork. The number of bridges decreases as the board size increases, as is expected since the number of corners is fixed but the distance between them is larger. Note how the decrease is not as dramatic as without this feature. The ring rate, however, drops dramatically compared to without this feature, and stays a fairly stable 2-6% of all wins across board sizes. This is a much more reasonable number when starting from an empty board, given that rings have almost no strategic value.

4.10 Combinations

To show that these features combine, all of the features that showed positive results were combined into a single test case, and then single features were removed. All of these test cases were then tested against the same RAVE baseline as all the other tests were tested against. The heuristic knowledge features and rollout policy features are shown in separate graphs for clarity, but the test case marked 'All' is identical on these two graphs.

The heuristic knowledge features that were included are:

- Maintain Virtual Connections (value 100)
- Connectivity (value 20)
- Locality (value 3)
- Local Reply (value 5)
- Distance (value 2)
- Group Size (value 2)

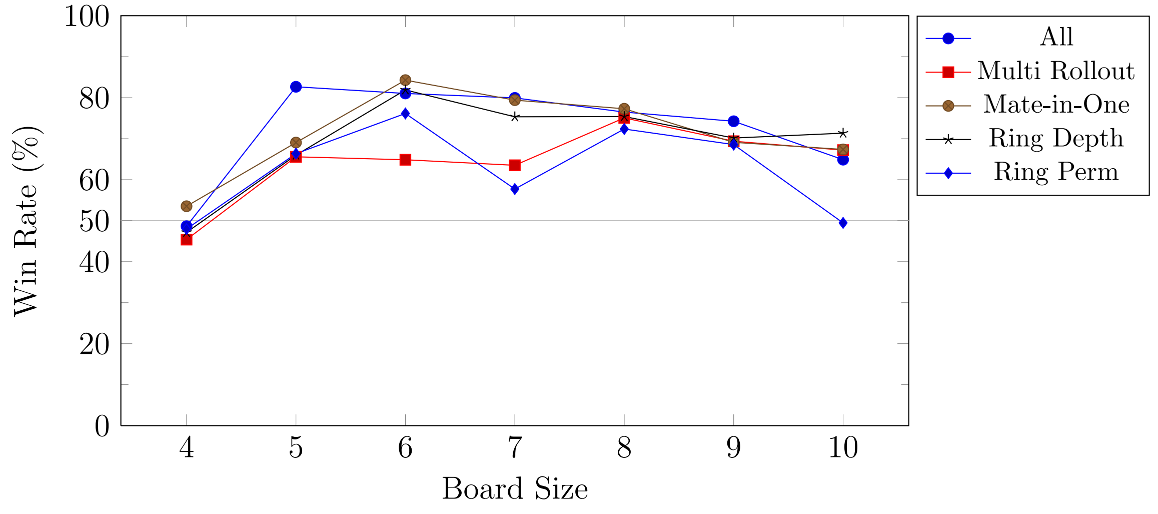


Figure 4.25: Rollout Modifications

The rollout policy features that were included are:

- Multiple rollouts (5 rollouts per simulation)
- Mate-in-one (Only for the first N moves where N is the board size)
- Ring Rule Depth (Only allow rings until 70% of empty cells are filled)
- Ring Rule Permanent Stones (Rings require 3 permanent stones)

The rollout policy related features are shown in Figure 4.25. Removing the multiple rollout feature causes a big drop in performance on board sizes 5-7, but little impact elsewhere. Removing mate-in-one checking or the ring depth causes a drop in performance on size 5, but no where else. The most dramatic difference is the removal of the permanent stones ring rule modification. Removing that feature causes a drop on board sizes 5-10, with particularly big drops on board sizes 5, 7 and 10. The permanent stone ring rule modification likely makes the heuristic knowledge features much more effective on the bigger board sizes. None of these features appear to have much affect on size 4.

The knowledge heuristic features are shown in Figure 4.26. Removing the

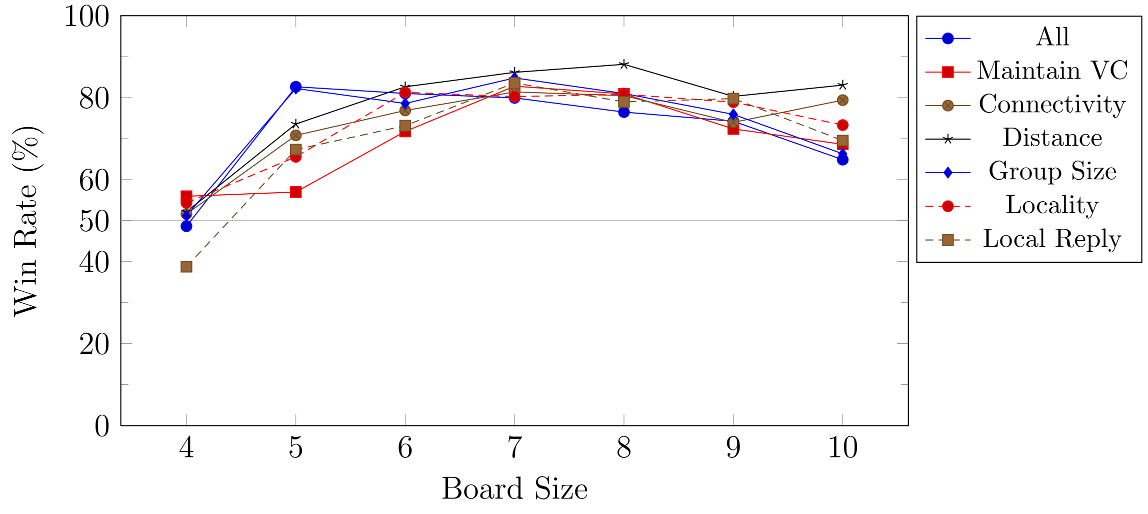


Figure 4.26: Knowledge Modifications

maintaining virtual connection knowledge causes a drop on sizes 5 and 6, but otherwise has little effect. Removing connectivity causes a drop on size 5, but improves size 10. Removing distance to win also causes a drop on size 5, but is an improvement on sizes 7-10. It is likely too slow of a computation, especially when locality is included. Group size appears to have no effect on any board size, likely due to the inclusion of locality. Locality is helpful on size 5, but is causes a bit of harm on sizes 8-10. Removing local reply is harmful on sizes 4-6, but has little effect on sizes 7-10.

Finding the smallest subset of features that would give optimal results is a very challenging task, as the parameter space is huge. Each of these features can be tuned to a large range of values. The most important thing to note here though, is simply how strong the combination of these features is relative to the RAVE baseline. The combination of all these features gives a greater than 80% winning rate, or about 300 elo, against the RAVE baseline on all board sizes except size 4.

5

Solving Havannah with MCTS

Section 4.6 showed that MCTS is capable of solving non-trivial positions in reasonable time. This chapter will show several improvements to MCTS in Castro that make it better suited to solving harder positions, as well as presenting the solutions to board sizes 2, 3 and 4.

5.1 Monte Carlo Tree Search Solving

Using MCTS to solve positions, as described in Section 4.6, would be enough to solve any position, given enough time and memory, but in a player the goal of the solver is mainly to avoid blunders, not necessarily to prove that the chosen move is optimal. When solving harder positions, more advanced techniques are needed to prove the outcome in reasonable time and memory. This section describes several techniques for reducing the search space, increasing the search speed and reducing memory requirements.

5.1.1 Symmetry

There are 37 cells on a size 4 board, but from the starting position only 6 of them are distinct. The rest are equivalent by symmetry, since the board has 6-fold rotational symmetry and 2-fold mirror symmetry. By storing a Zobrist hash [26] for each of the 12 possible board orientations and taking the minimum value as the representative hash, symmetries can be found and ignored. As stones are placed, the number of possible symmetries decreases dramatically. Symmetric moves are ignored for the first five ply at node expansion. After five ply, the cost of calculating the extra hash values and finding the unique moves becomes too expensive and so symmetry detection is turned off for all later moves.

Note that this does not find transpositions, only one-ply symmetries. Using a hash table of positions based on their Zobrist hash would turn the game tree into a directed acyclic graph (DAG), thereby dramatically reducing the search space, but could also lead to inaccuracies due to hash collisions.

5.1.2 Multi-threading

Solving a position usually takes significantly more effort than merely making a strong move. Therefore it is important to use as much computation power as possible. On today's multi-core machines, this means multi-threading.

Writing fast and thread-safe code is a challenge. In Castro, the control thread spawns a pool of player threads, but does not participate in the search. The player threads each follow a simple state machine that includes the states: Wait_Start, Wait_End, Running, Garbage_Collection and Cancelled. State updates are done using atomic compare-and-swap (CAS) machine instructions to ensure all state transitions are race-free, and to avoid the contention associated with locks. Barriers are used to pass control between the control thread and the player threads, as well as to decide which of the player threads will be used for garbage collection, as it is a single threaded procedure.

All updates to values in the MCTS tree are also updated with atomic instructions. Updating experience or RAVE values are done using atomic increment instructions. Adding children is done with CAS to update the value of the children pointer. If multiple threads attempt to create the same set of children only one will succeed and the others will instead do a rollout from the parent node. If multiple threads backup the outcome of a single node in the tree, a race condition related to early draw detection (described in Section 5.1.5) is possible, so if the value has changed unexpectedly, the backup is retried to ensure correctness.

All the player-threads use the same algorithm and the same parameters, so without any special handling would make the same choices as they descend the tree. To encourage the threads to explore different parts of the tree, virtual losses[27] are added as each thread descends the tree. A virtual loss is a loss that is added to the experience of a node during the descent phase, before the actual experience occurs. If the rollout results in a loss, this virtual loss is kept as real experience, but if the rollout results in a win, it is replaced with the true experience of a win. This makes the nodes that are currently being explored appear worse than they actually are, possibly worse than their siblings, thereby encouraging the other threads to explore the siblings instead. The virtual losses are added atomically as well.

Contention between threads must be minimized to maximize speed. One early source of contention was generating random numbers for the rollouts. The rand() function in C++ is very fast in a single threaded environment, but has a lock that limits thread scalability. To avoid this lock, the MTRand library[1] was modified to use only local state variables. One instance is used per thread. By having the structure local to each thread, no lock is needed and memory contention is minimized.

Pondering — thinking during the opponent's time — is a simple way of improving the strength of a player, and is easy to implement given the thread pool described above, but it also makes debugging long running solving attempts easier. Simply move to the starting position, then enable pondering. The player threads will continue solving in the background, while the control thread continues to respond to commands, making it possible to query the state of the player to see the status of the solving attempt. While this is not necessary for a solver, it was instrumental in determining why some of the openings were taking so long to solve.

5.1.3 Garbage Collection

When solving a non-trivial position, the size of the tree is likely to be exceed physical memory. For very hard problems it may be several orders of magnitude bigger. However, large portions of the tree are likely to be irrelevant at any given time so can be thrown away when the available memory is filled. If the deleted nodes are needed later, they can be recomputed. Various criteria for which part of the tree can be thrown away are possible, but the one used here is to discard the children of solved nodes, as well as the children of nodes that have fewer than N simulations, with a minimum N of 5. N is increased for the next round of garbage collection if less than half of the memory is freed, and decreased if more than half the memory is freed. This method worked

[1] http://bedaux.net/mtrand/

well as long as N remains reasonably small, say below 100, but as N grows large, the amount of recomputation increases, increasing the solving time.

5.1.4 Memory Management

Garbage collection, as described in Section 5.1.3, should be run any time the memory use exceeds the user specified memory limit, thereby freeing up enough memory to continue, but calculating memory use accurately is not trivial. Memory use is usually naively calculated as the number of nodes in the tree multiplied by the node size. Unfortunately this approach ignores the fact that malloc/free (or new/delete in C++) have some memory management overhead and tend to fragment memory over time. This fragmentation leaves pockets of permanently unusable memory, decreasing the usable available memory. With what amounts to a fixed node limit, extra memory is needed to compensate for the unusable memory, thereby exceeding the user specified memory limit.

During a game, fragmentation is unlikely to make a difference as the run time is short enough that the fragmentation is a small fraction of total memory. In a long running solver, however, fragmentation could use up to half of the available memory, likely leading to a severe performance hit as the system swaps memory to disk. An overhead as high as 30% was observed in practice. To avoid this, a compacting tree was implemented. It periodically rearranges the nodes in the tree to avoid fragmentation, while allowing the full memory to be used. Conceptually this is similar to the compacting garbage collectors in higher level languages like Java. Compacting the tree into a contiguous segment of the heap leaves a contiguous empty section of the heap, allowing a very fast allocation strategy to be used. It simply returns the pointer to the beginning of the empty segment, and moves the empty pointer forward by the amount that was allocated. This is more memory efficient than a normal malloc call, which uses fixed and inaccurate bucket sizes. It is also faster, as it is simply an atomic increment. Using this allocation strategy means every byte is accounted for, allowing strong upper bounds on memory limits. Several of the harder positions would not be solvable without this compacting tree, at least not without breaking the positions into smaller subtrees and solving them independently.

5.1.5 Early Draw Detection

Checking the game outcome at node expansion, and backing up wins, losses and draws as described in Section 4.6 is enough to solve any position, given sufficient time. Certain positions in Havannah lead to many draws, and can take prohibitively long to solve without more advanced draw detection. Figure 3.8b shows a board where no wins are possible after move 31 even if both players cooperate. Without draw detection this position will take $6! = 720$ simulations to enumerate and prove. In a game this is not important, since its win ratio approaches the correct value of a draw quickly, but this is not rigorous enough for a solver.

To show that a position is a draw, the three win conditions need to be checked to see if any wins of that type are possible. Fork and bridge wins can be detected with the heuristic described in Section 4.8.6: start a flood fill from each corner and edge for each player. If none of the empty cells can reach three edges or two corners for a player, then that player cannot form a fork or a bridge. One player being unable to form a fork or a bridge does not preclude the other player from doing so.

Potential rings can be detected by checking for encirclability. A group of stones that connects to an edge or corner cannot be encircled by the opponent. Any cell that is next to a group that connects to an edge or corner also cannot be encircled by the other player. If no cells can be encircled, then no rings are possible.

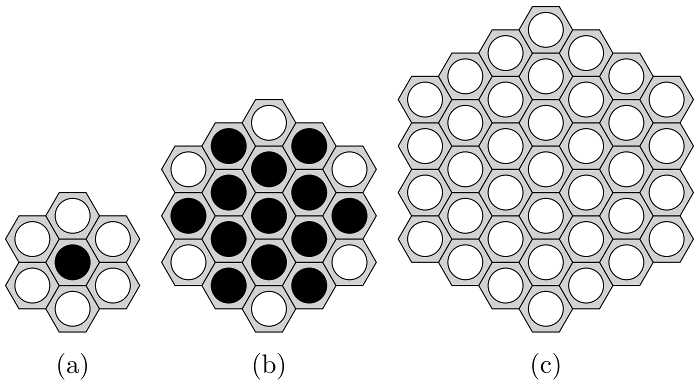


Figure 5.1: (a) Solution to size 2 (b) Solution to size 3 (c) Solution to size 4. The colour of a piece represents the winner if white makes the first move in that position. No openings lead to a draw.

If no forks, bridges or rings are possible for a player, then that player cannot win, and so should force a draw if possible. If both players' best outcome is a draw, then that position is a proven draw.

More advanced techniques of draw detection based on virtual connections could detect draws much earlier, possibly as early as move 20 in Figure 3.8c, but these techniques have not been explored. The speedup from the techniques described here may not be as large as 6! for all positions, but it is still at least an order of magnitude for most early draws.

5.2 Solution to Havannah Sizes 2, 3 and 4

The perfect play solutions to board sizes 2, 3 and 4 are shown in this section and in Figure 5.1 in particular. The colour of the piece represents the player that will win the game if white makes the first move on that cell. The subsections describe the proofs in more detail.

5.2.1 Size 2 Proof

Size 2 Havannah is a trivial game, with the solution shown in Figure 5.1a. It has 6 corners, no edges and a center. The corner opening is a win, since no reply blocks both neighbouring corners. The center opening is a loss, since it loses to any corner reply.

5.2.2 Size 3 Proof

Size 3 Havannah is more interesting than size 2, but is still simple enough that the solution, as shown in Figure 5.1b, could be derived by hand. It has been verified by 3 different solvers and was used as a benchmark when developing the solvers in Castro. Alpha-beta, proof number search and Monte-Carlo tree search all solve size 3 in under 100ms on commodity hardware. A proof tree as found by the MCTS player is shown in Figure 5.2. This is not a minimal proof tree. A minimal proof tree has a maximum depth of 10 moves.

5.2.3 Size 4 Proof

As shown in Table 3.1, size 4 has a state space that is 8 orders of magnitude bigger than size 3. The size 4 solution was computed by MCTS twice, first to show it was possible and to refine the method, then to confirm the proof and to save the proof tree. The two solutions produced different proof trees, but came to the same result, shown in Figure 5.1c. The proofs were calculated on a 10 machine cluster where each machine is an 8-core Xeon E5463 2.8 GHz with 32Gb of ram.

During the first solving attempt, each opening move was made, then the player was left to ponder until the solution was found. The a1, b2 and b3 openings completed in a single run but took up to a week each. The a2, c3 and d4

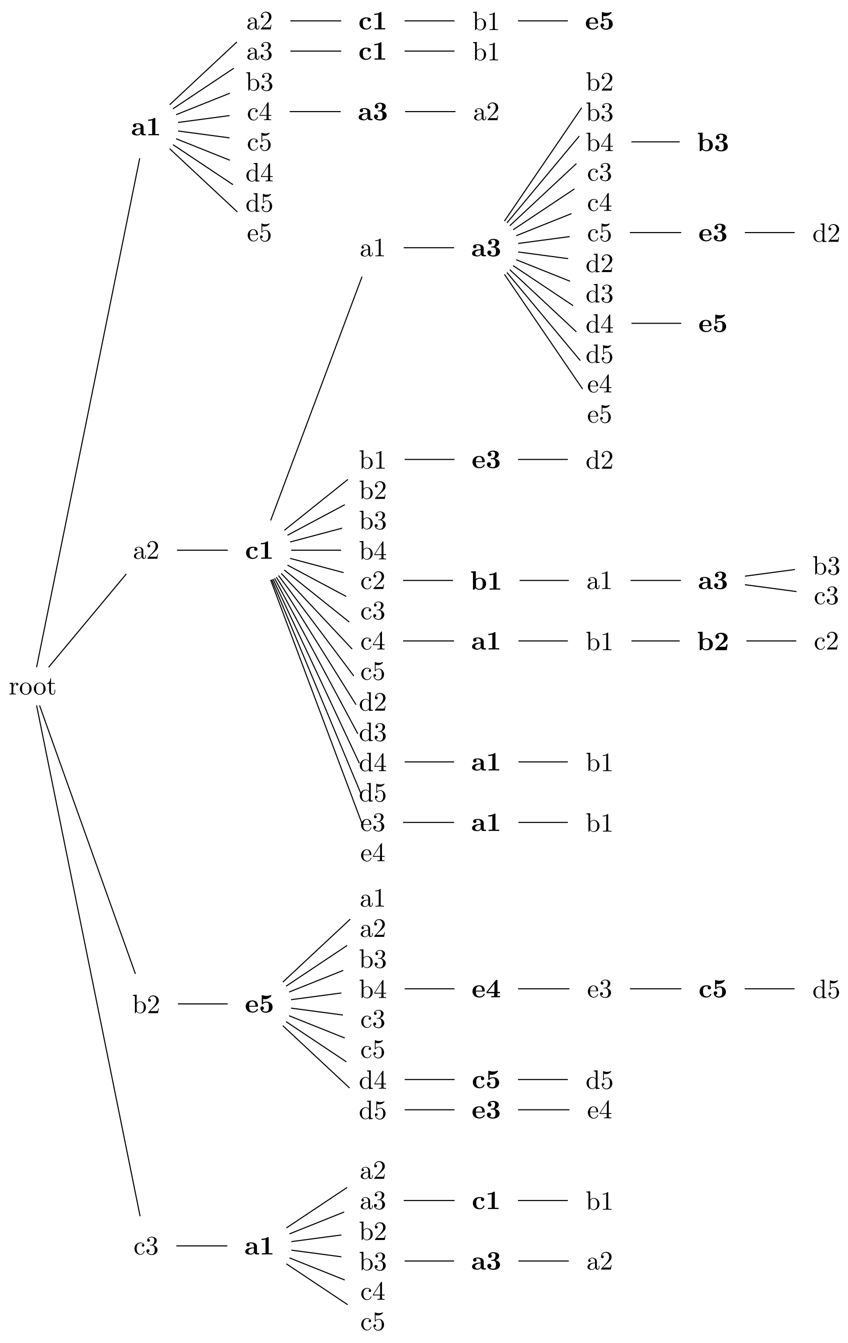


Figure 5.2: Proof Tree for Size 3 showing nodes that took more than 100 simulations to solve

openings had such big proof trees that their replies needed to be solved independently. The c3 and d4 openings often had upwards of 90% of the simulations ending in draws, which prompted the work on early draw detection. The proof trees for d4 were huge and led to so much memory fragmentation that the more advanced memory management was required. This solving attempt took several months to come to a reliable outcome, due to new features, bug fixing, parameter tuning and general trial and error. Basic logs of solved moves were kept, but were of little use to rebuild the proof tree.

A second solving attempt was done to produce a proof tree. The correct responses to the opening replies, as calculated in the first attempt, were used to speed up the computation, and the opening replies were computed independently for a2, b2, c3 and d4. This sped up computing the proofs for several reasons: several of the moves that looked strong had already been proven to be losses and could be ignored; the subtrees were smaller and so caused less recomputation of garbage-collected nodes; and the openings could be distributed over more machines. The proof trees for each opening were saved to sgf[2] files, which were later combined for the final proof. The complete proof required approximately 4×10^{11} simulations and took about a week across the 10 machines using all 80 cores. A queueing system was used to keep all machines consistently busy.

The proof trees for the 6 opening moves are shown in Figures 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8. The moves shown all took more than a minimum amount of simulations to solve, with the minimum value chosen to approximately fill the page, ranging from 10^8 simulations for a1 to 10^9 simulations for a2. Only the proof trees are shown, so a move that took more than the minimum amount to solve but was proven as a loss when a sibling was proven as a win is not shown. More detailed proof trees were recorded and are posted on the thesis website `http://havannah.ewalds.ca/`. All nodes that took at least 10,000

[2]Smart Game Format: `http://www.red-bean.com/sgf/`

simulations to solve are recorded in the posted proof trees.

A complete, independent confirmation of the proof has not been attempted, but the code is open source and more detailed proof trees are available for inspection. Several non-trivial problems have been independently solved by all the solvers included in Castro, all with the same result. The a1 opening has been confirmed by PNS with a 30gb transposition table, but this took upwards of 80 hours, about 10 times longer than MCTS on the same state. As the a1 opening is the easiest opening to prove, confirming the proofs on the other openings, without using the existing proof trees as a guide, would be prohibitively slow. The multi-threaded version of PNS, including the memory management, garbage collection and virtual loss improvements but without the transposition table also attempted the a1 opening on the same hardware, but failed to finish within several days.

PNS is faster than MCTS on many small problems, but is much slower at solving problems as large as the complete size 4. This is conjectured to be because PNS has little guidance other than the local threats and branching factor, while MCTS can deduce good moves from the result of its rollouts. When the tree has as little differentiating factors, as is true of the first several ply of size 4, the behaviour of PNS is quite similar to that of a breadth-first search. In essence, it gets lost. MCTS on the other hand starts differentiating good moves from bad moves quite early on, even when the structure of the tree is uniform. This makes MCTS a good candidate for solving hard states, while PNS is good for solving easier or less uniform states very quickly.

It is worth noting that size 4 took approximately 3×10^7 cpu seconds to solve, which is about 6×10^8 times more time than it takes to solve size 3. Referencing Table 3.1, the size 4 state space is about 3×10^8 times bigger, which is very similar to the difference in time to solve the two. This suggests that solving size 5 may well take about 10^{12} times more effort to solve than size 4, unless some clever mathematical properties can be exploited to shrink the state space.

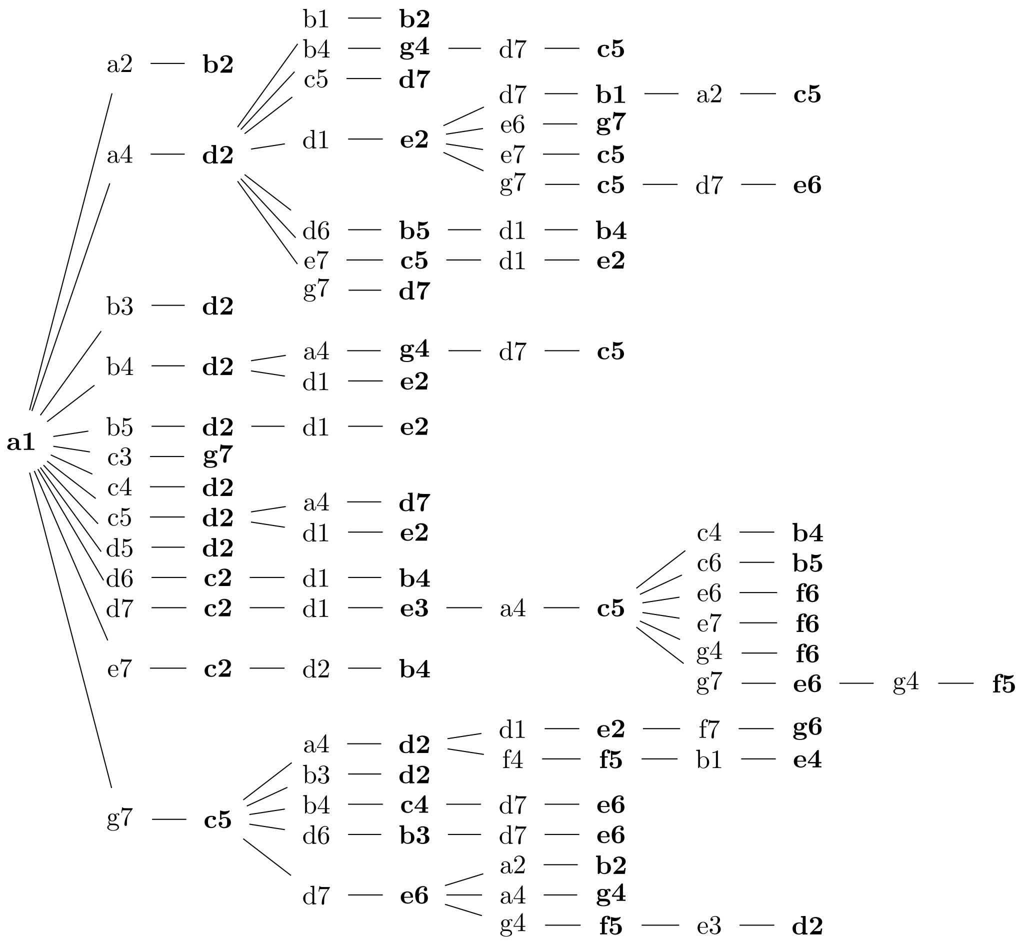


Figure 5.3: Proof Tree for the a1 opening on size 4 showing nodes that took more than 10^8 simulations to solve

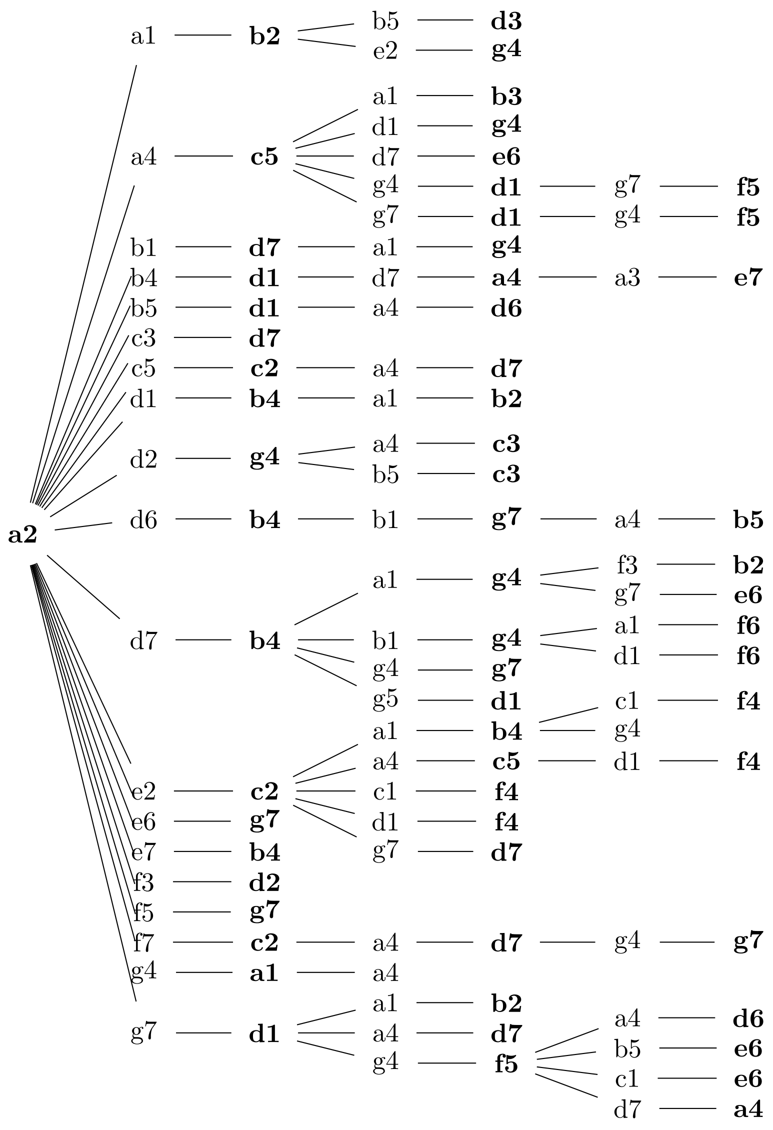


Figure 5.4: Proof Tree for the a2 opening on size 4 showing nodes that took more than 10^9 simulations to solve

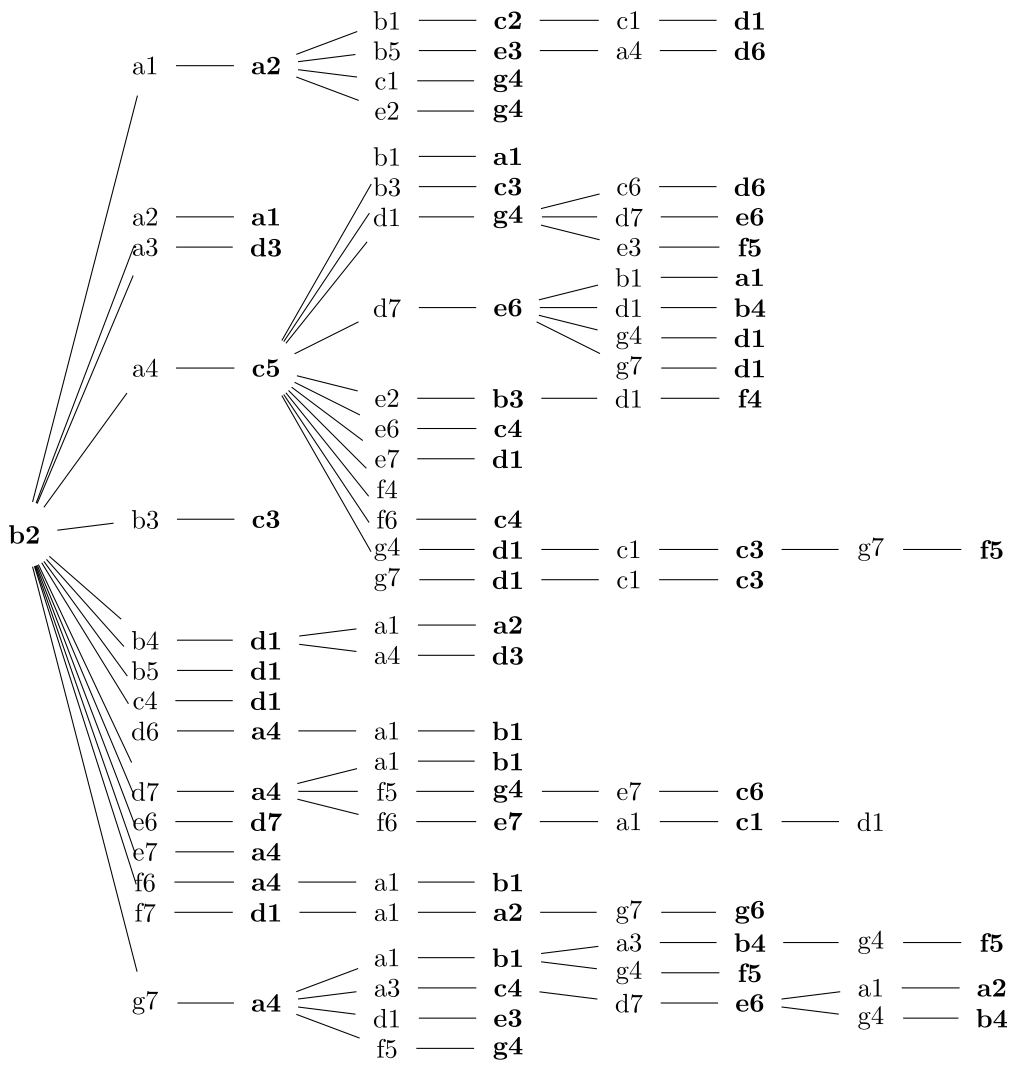


Figure 5.5: Proof Tree for the b2 opening on size 4 showing nodes that took more than 2.5×10^8 simulations to solve

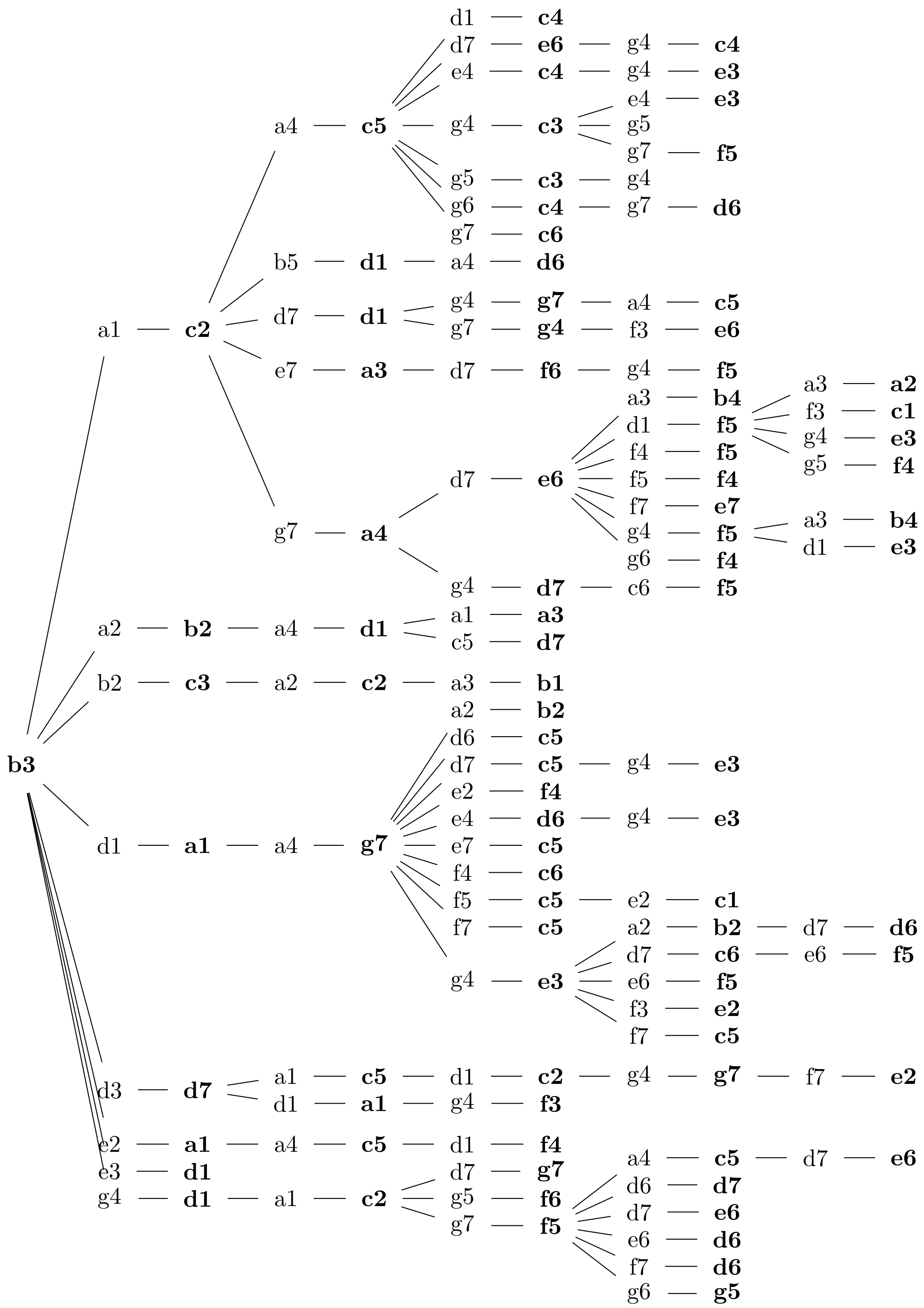


Figure 5.6: Proof Tree for the b3 opening on size 4 showing nodes that took more than 10^8 simulations to solve

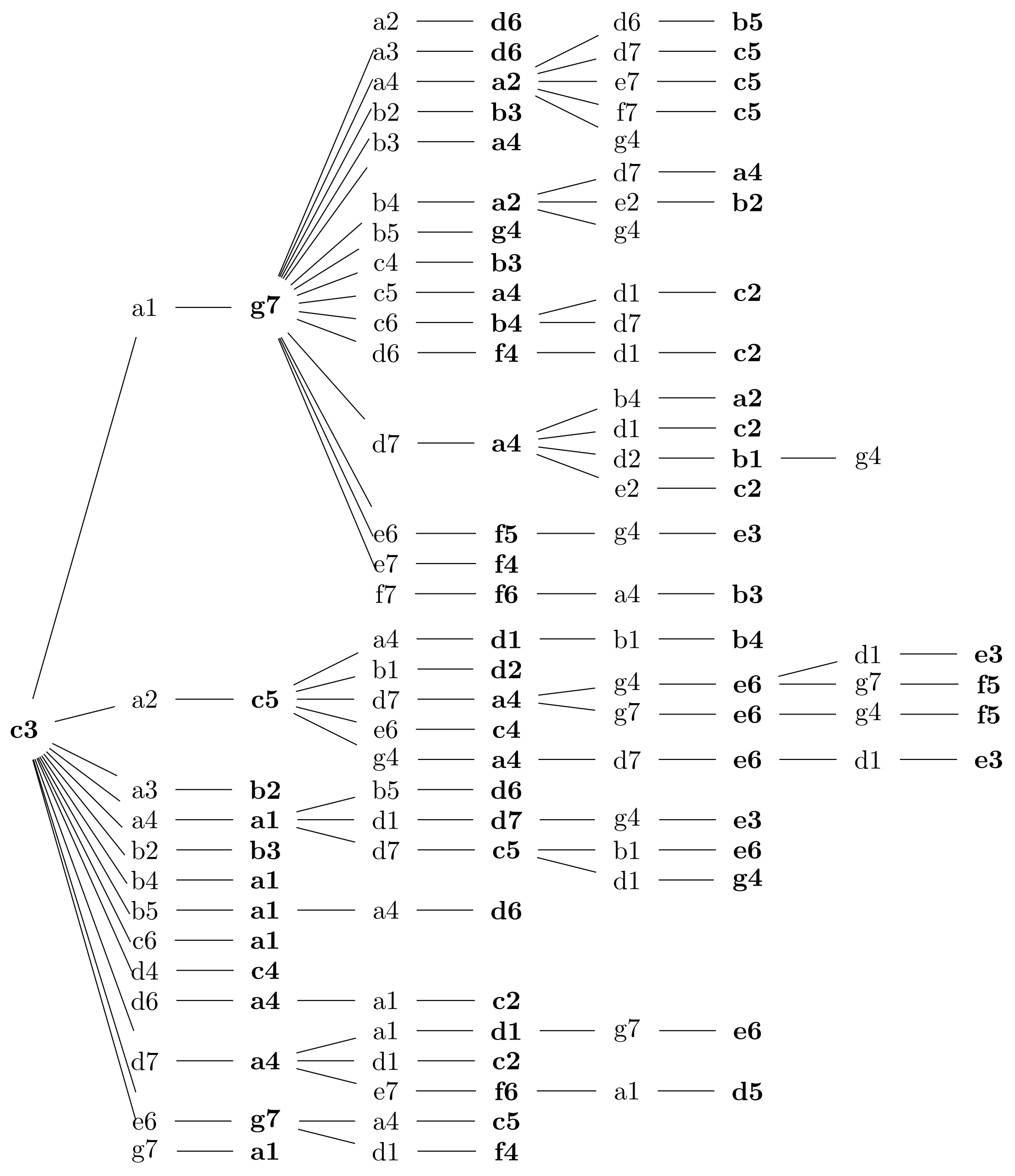


Figure 5.7: Proof Tree for the c3 opening on size 4 showing nodes that took more than 6×10^8 simulations to solve

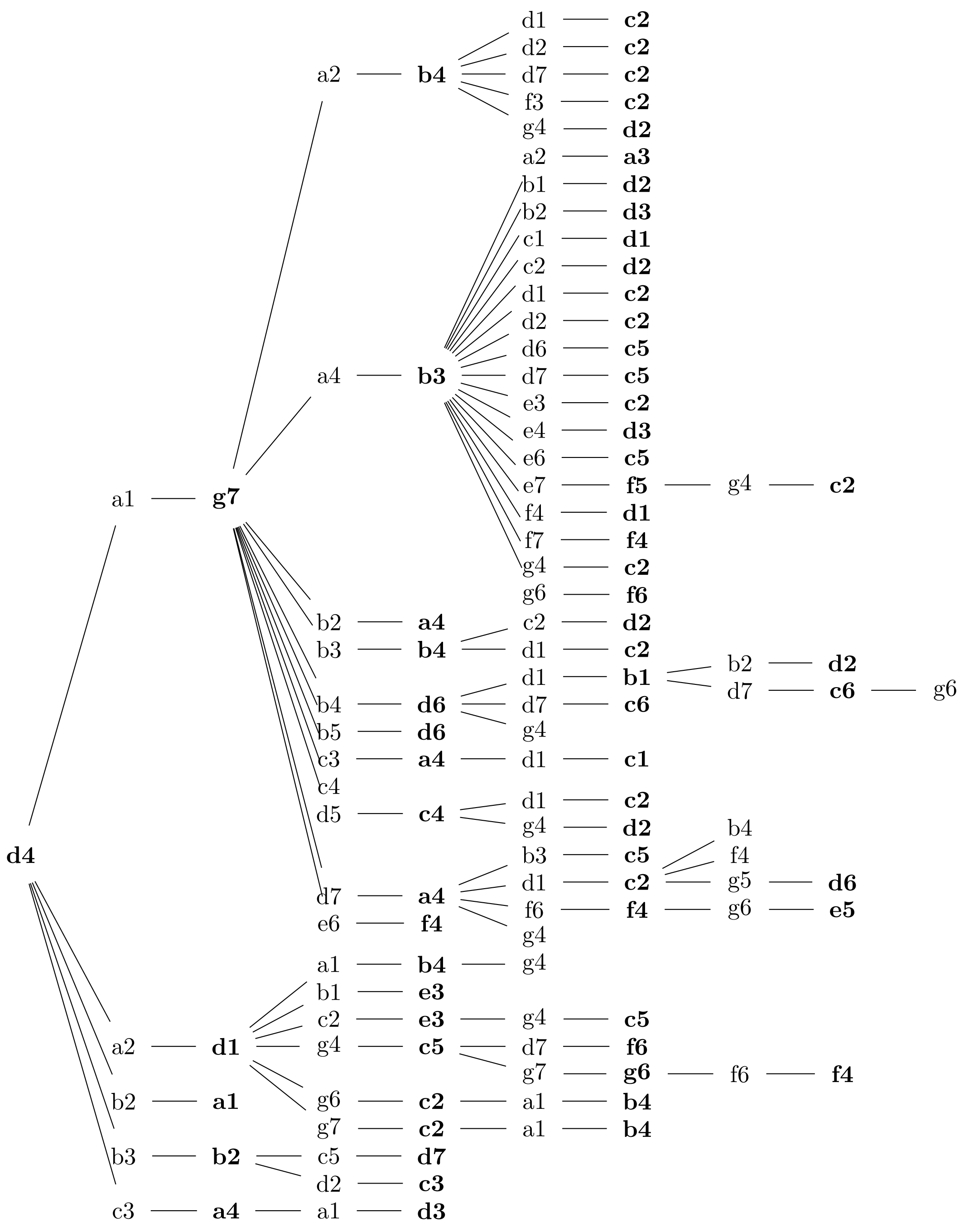


Figure 5.8: Proof Tree for the d4 opening on size 4 showing nodes that took more than 4×10^8 simulations to solve

6
Conclusions

6.1 Conclusions

Several properties of Havannah were introduced in Section 3.4. Virtual connections and frames, while important, are yet to be exploited to their full potential. Dead cells may show up in generalized pattern recognition, but are not worth searching for on their own. Draws, while rare, can be detected early, and this is shown to be important in solving size 4. These properties together show some of the challenges and future potential of Havannah playing programs.

Monte Carlo Tree Search is shown to play Havannah very well. Many improvements are possible to increase the playing strength, including heuristic knowledge to modify the move selection in the descent phase, as well as modifications to the rollout policy.

The three winning conditions are shown to interact poorly with MCTS on big board sizes, and four fixes are proposed. The permanent stone ring rule proved to work very well at fixing these interactions, and significantly improves play on big boards against both other programs and against humans.

Several heuristics showed significant improvements in playing strength on their own, some as high as a 75% winning rate or 200 elo. When combining all of the positive results, a winning rate of over 80%, or about 300 elo is achieved on all board sizes greater than size 4.

This player, which includes proof backups, was then multi-threaded, given draw detection and better memory management. This improved player was then used to solve all 6 openings of the size 4 board. Proof trees for all 6 openings are presented in Section 5.2.3.

Castro, the implementation and test bed for all these tests is released open source and is available at `https://github.com/tewalds/castro`. ParamLog, the distributed testing framework that was used to run most of these tests is also open source and available at `https://github.com/tewalds/ParamLog`.

Castro competed in the 2010 and 2011 ICGA Olympiads, taking first place both years. When playing on Little Golem, it routinely beats strong human players on board sizes 4-6. On larger board sizes, it still plays a strong game, often beating weak players, but usually losing to strong players. While significant progress towards human level play has been made, there is still a long ways to go.

6.2 Future Work

The state space estimates shown in Section 3.3 are quite simplistic, but a more accurate estimate could be found by generating random board states and checking how many of them are valid. This could show how many states include winning formations for one or both players and show how common draws are.

There is a long way to go before Havannah programs are comparable strength to strong human players on big boards, even after all the improvements presented here. Thankfully there are also many potential improvements left to try and to discover. Many ideas that have been tried in other games, such as Go, could be tried here. One of the large successes in Go is the use of patterns, both small and large, both as heuristic knowledge and to improve the rollout policy. The strength of these patterns can be learned based on strong human games, self-play or even on the fly from the results of rollouts. They could be applied deterministically or with a weighted random strategy like softmax.

The large-scale parameter tuning that was used to generate all the results in Chapter 4 works, and is an effective way of understanding why a particular feature works. It is very inefficient, however, in terms of examining the state space of all the different features to find optimal parameters. Using a machine learning technique like minorization-maximization[28], which computes an elo rating for each feature, could be much more efficient. This would help show which features are strong and which can be inferred from the other features.

Section 3.4.1 showed that virtual connections, and therefore frames, are breakable. Despite this, frames are the foundation human play. The first program to successfully make solid deductions based on frames, without paying a huge speed penalty, is likely to have a large advantage over all others. Even just using frames as a guide for a solver could be a large performance boost.

Using a hash table instead of a tree to reduce transpositions could significantly improve performance with longer times to play and on the smaller board sizes where transpositions are frequently reached. It does, however, present significant engineering challenges in terms of garbage collection and threading, and does introduce a tiny probability of inaccuracy in the form of hash collisions.

While proof number search wasn't tested extensively here, it could be interesting to try mixing MCTS and PNS. MCTS could be used to guide a PNS solver towards good moves, allowing for faster solving times. It could also be interesting to try a pure PNS player, which would have a very different playing style from MCTS.

Solving size 5 Havannah is likely a long ways off, but intermediate sizes with uneven edge lengths, such as $4 \times 4 \times 5$ and $4 \times 5 \times 5$, could be solved much sooner. It could be interesting to explore the properties of these odd sizes to figure out if they are interesting to play, or if they have similar problems as Hex's uneven board sizes where the second player has a trivial winning strategy.

A

Glossary

Here are general game playing concepts and definitions:

Anytime Algorithm An anytime algorithm can return an answer at any point of execution, but continues to run to provide a more accurate and potentially better answer.

Best-First Search In a best-first search nodes are explored in order of their heuristic value. Promising nodes are explored before less promising nodes. This is very memory intensive as all nodes explored to date must be kept in memory.

Branching Factor is the average number of moves available to each player. This depends on the rules of the game, board size, pieces in play and the stage of the game. This can be as low as one for forced moves, or very high, such as in the hundreds or thousands for Amazons or Arimaa.

Breadth-First Search In a breadth-first search, all nodes at a specific depth will be considered before any nodes at a deeper depth, in increasing depth. This can be very memory intensive as all nodes up to the specified depth are usually kept in memory.

Depth-First Search In a depth-first search (DFS), nodes are considered in a depth-first way. The full subtree of a node will be explored before any of its siblings will be explored. This is very memory efficient since it only needs to store the nodes along the path from the root to the current node, but leaves many nodes near the root unexplored for long periods of time.

Elo Rating The elo rating system is a measure of relative strength of players. Two players with equal elo ratings would expect to each win half of their games. A difference of 400 elo means the player with the higher rating is expected to win 90% of games between these two players. 100 elo translates to about a 65% winning rate.

Game Complexity The game complexity is the size of the state space, sometimes taking transpositions into account. This can either be the number of unique positions or the number of possible games.

Game Tree A game can be represented as a game tree. Each position in the game is a node, and each move is an edge in the graph connecting the position before the move to the position after the move. When there are multiple paths to a position, the position can be represented as separate nodes, leading to a tree, or combined as a single node, leading to a directed acyclic graph (DAG). Some games have loops, where a position can be reached multiple times in a single game, leading to a directed graph.

Hash Value A hash value is a representative number of a state used to detect transpositions. Transpositions all have the same hash value, but different states have different hash values. Often collisions are possible so two states that aren't a transposition have the same hash value, but this is very rare as large hash values (usually 64 bit unsigned integers) are used.

Heuristic A heuristic function takes a position and returns a value associated with the position. This value often represents the likelihood of winning from that position, but can also be just an abstract number that can be compared against other values to order nodes or moves.

History All moves leading from some starting position, usually the beginning of the game, to the current state.

Leaf Node A leaf node is any node that has no expanded children.

Minimax In 2-player games, each player attempts to win at the expense of the other player. To do so, each player attempts to minimize the opponent's gain while maximizing their own gain. To win, a player must have at least one winning move, but to lose all moves must be losing moves.

Minimax Backup Given a node N whose children all have known values, N's value is equal to the value of the most favourable child for the current player.

Minimax Value The value of a node given that both players play perfectly according to Minimax.

Move A move is a distinct action by one of the players leading from one state to another state. In games with multi-part moves, such as Amazons where each move consists of a movement plus shooting an arrow, the pair of actions would be considered a single move. There are usually multiple moves available from each state, but usually only one can be chosen per turn.

Node Value Each node has an outcome or expected outcome associated with it. Terminal positions have an exact value, whereas non-terminal positions have an expected or heuristic value.

Perfect Information A game has the property of perfect information when both players know the full state of the game.

Ply A move or turn by a single player. A search depth of 4-ply means looking at all possible paths that are 4 moves deep.

Ponder Thinking during your opponent's time.

Root Node The root node is the highest node in the tree. It has no parents, and there is exactly one of them in any tree.

State A state is a full description of a board position. It includes the locations of all the pieces, and any other relevant information. In Chess, the state would include whether each king can castle, and whether a pawn can capture *en passant*. In games where repeated moves are not allowed, the full game history may be included in the state. Occasionally a simplified version of the state is used when speed is more important than accuracy.

State Space The state space is the number of unique reachable states in the game.

Stochastic A game is stochastic if it has elements of randomness, such as dice rolls. Backgammon is stochastic while Chess, Hex and Havannah are not.

Terminal Position A terminal positions is a position where one of the players has won or it is a draw. They have an exact value such as win, loss or draw, or a score to show how much a player won by.

Transposition One state with multiple histories. If moves A-X-B leads to the same position as B-X-A, they are transpositions. They are the same state, so they will have the same minimax value (subject to history), and should not be searched twice.

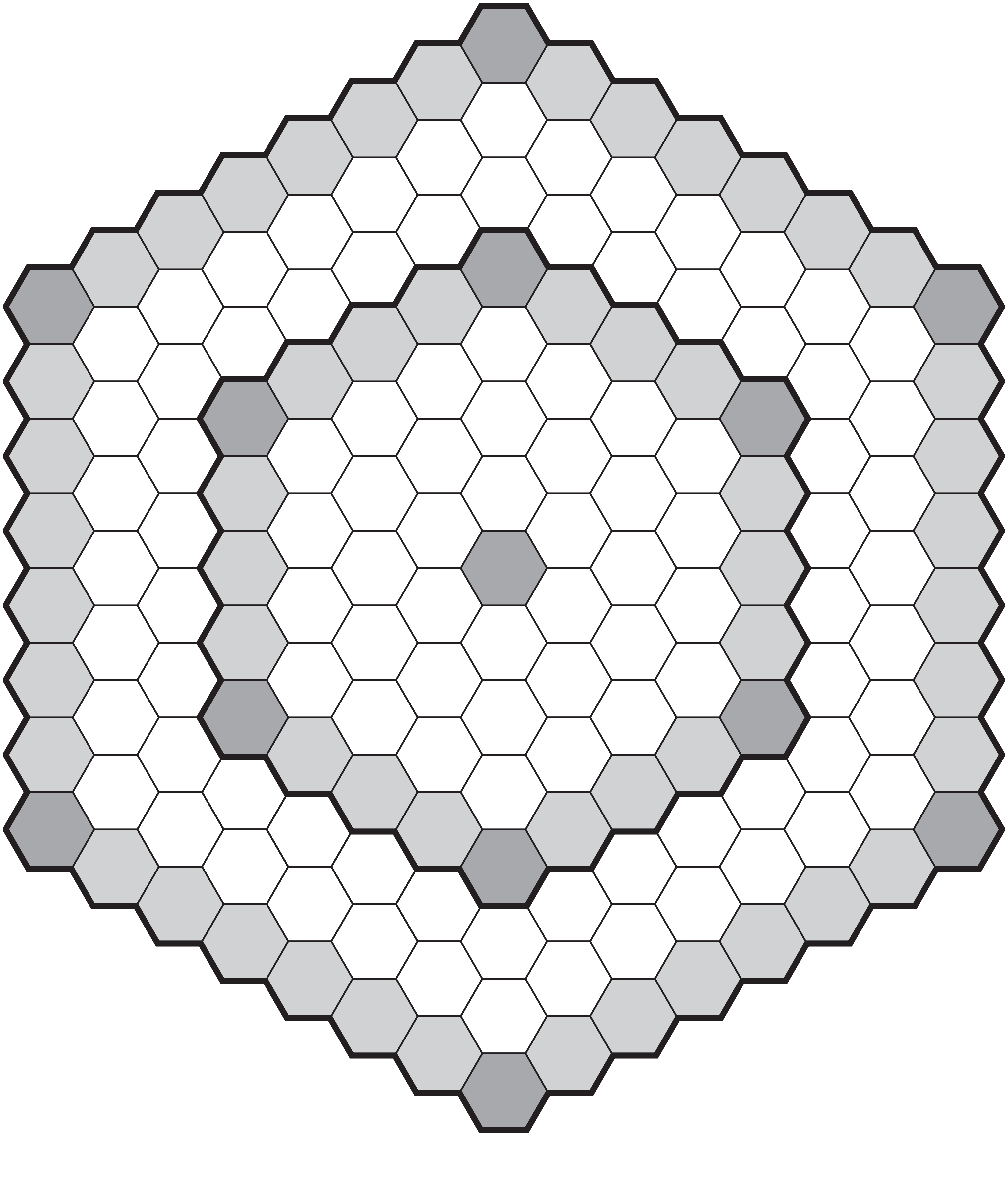
Zero Sum Games A zero sum game has the property that one player's gain is the other player's loss. A draw is still possible, but no move can help both players, so there is no incentive to cooperate.

Zobrist Hash A hash value that is built up incrementally by XORing a random string associated with each move against a previous hash value.

B

Playable Havannah Board

Below is a size 8 board with a size 5 board embedded in it. Size 5 is great for learning to play, while size 8 is played by more experienced players. It can be printed and played with small Go stones, or by marking the squares with different colour pens or pencils. Have fun playing!



References

[1] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[2] D.J. Slate and L.R. Atkin. Chess 4.5 The Northwestern University chess program. *Chess skill in Man and Machine*, pages 82–118, 1977.

[3] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11 (11):1203–1212, 1989.

[4] L.V. Allis, M. van der Meulen, and H.J. Van Den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

[5] M.H.M. Winands, J.W.H.M. Uiterwijk, and J. van den Herik. PDS-PN: A new proof-number search algorithm. *Computers and Games*, pages 61–74, 2003.

[6] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. *Computers and Games*, pages 72–83, 2007.

[7] GMJ Chaslot, M.H.M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy. Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, 4(3):343, 2008.

[8] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. *European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[9] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *24th International Conference on Machine learning*, pages 273–280. ACM, 2007.

[10] S. Gelly and D. Silver. Monte-Carlo tree search and rapid action value estima-

tion in computer Go. *Artificial Intelligence*, 2011.

[11] D. Silver. *Reinforcement learning and simulation-based search in computer Go*. PhD thesis, 2009.

[12] P. Drake. The last-good-reply policy for Monte-Carlo go. *ICGA Journal*, 32 (4):221–227, 2009.

[13] H. Baier and P. Drake. The power of forgetting: Improving the last-good-reply policy in monte-carlo go. *IEEE Transactions on Computational Intelligence and AI in Games*, (99):1–1, 2010.

[14] John Tromp. Number of connect-4 positions. URL http://homepages.cwi.nl/~tromp/c4/c4.html.

[15] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518, 2007.

[16] P. Henderson, B. Arneson, and R.B. Hayward. Solving 8×8 hex. In *21st International Joint Conference on Artificial Intelligence (IJCAI 09)(ed. C. Boutilier)*, pages 505–510, 2009.

[17] J. Tromp and G. Farneback. Combinatorics of Go. *Computers and Games*, pages 84–99, 2007.

[18] John Tromp. Number of chess diagrams and positions, 2010. URL http://homepages.cwi.nl/~tromp/chess/chess.html.

[19] T. Rogers. *A Program to Play Havannah*. B.Sc. thesis, University of Leeds, School of Computing, 2004.

[20] A. Nagai. Proof for the equivalence between some best-first algorithms and depth-first algorithms for and/or trees. *IEICE Transactions on Information and Systems*, 85(10):1645–1653, 2002.

[21] J. Pawlewicz and Ł. Lew. Improving depth-first pn-search: $1+\varepsilon$ trick. *Computers and Games*, pages 160–171, 2007.

[22] F. Teytaud and O. Teytaud. Creating an upper-confidence-tree program for Havannah. *Advances in Computer Games*, pages 65–74, 2010.

[23] R. Lorentz. Improving Monte–Carlo tree search in Havannah. *Computers and Games*, pages 105–115, 2011.

[24] J.A. Stankiewicz. *Knowledge-based Monte-Carlo Tree Search in Havannah.* PhD thesis, Maastricht University, 2011.

[25] J.D. Fossel. *Monte-Carlo Tree Search Applied to the Game of Havannah.* B.Sc. thesis, Maastricht University, Maastricht, The Netherlands, 2010.

[26] A.L. Zobrist. A new hashing method with application for game playing. *ICCA Journal*, 13(2):69–73, 1990.

[27] G. Chaslot, M. Winands, and H. van Den Herik. Parallel Monte-Carlo tree search. *Computers and Games*, pages 60–71, 2008.

[28] R. Coulom. Computing Elo ratings of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, 2007.