CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Havannah, a Monte Carlo Approach

A thesis submitted in partial fulfillment of the requirements

For the degree of Master of Science in Computer Science

By

Roberto Nahue

December 2014

The Thesis of Roberto Nahue is approved:

_____	_____
Professor Jeff Wiegley	Date
_____	_____
Professor John Noga	Date
_____	_____
Professor Richard Lorentz, Chair	Date

California State University, Northridge

ACKNOWLEDGEMENTS

Thanks to Professor Richard Lorentz for your constant support and your vast knowledge of computer game algorithms. You made it possible to bring Wanderer to life and you have helped me bring this thesis to completion.

Thanks to my family for their support and especially my daughter who gave me that last push to complete my work.

Table of Contents

Signature Page ii
Acknowledgements iii
List of Figures vi
List of Tables vii
Abstract viii
Chapter 1 1
Introduction 1
1.1 Havannah 1
1.1.1 Havannah Board Rules 2
1.2 Monte Carlo (MC) Algorithm 4
Chapter 2 9
Wanderer 9
2.1 Description 9
2.2 Classes 10
2.2.1 Point Class 10
2.2.2 BoardGame Class 12
2.2.3 Engine Class 12
2.2.4 MainEngine Class 12
Chapter 3 13
Wanderer Implementation Details 13
3.1 Board Initialization 13
3.2 Neighbors 14
3.3 Determining Winning Moves 17
3.3.1 Bridges 23
3.3.2 Forks 24
3.3.3 Rings 26
3.3.3.1 Simple Rings 27
3.3.3.2 Blobs 29
3.4 Monte Carlo Tree Search with UCT Implementation 31
Chapter 4 41
Testing Wanderer 41
Chapter 5 51
Future Considerations 51
5.1 Mate in One 52
5.2 MCTS Solver 53
5.3 Only One Move, Make it 56
5.4 Save MC Tree for Next Move 57
5.5 Biasing MCTS Node Selection 59
5.6 Multithreading 61
5.7 Weighted PCUF Algorithm 64
5.8 MCTS Node Building Efficiency 66
Chapter 6 67
Conclusion 67
References 69

Appendix A 71
Appendix B 72

List of Figures

Figure 1.1 – Havannah Board Base 10 2
Figure 1.2 – Havannah Board Winning Structures 3
Figure 1.3 – Basic MC Algorithm 5
Figure 1.4 – MCTS with UCT Variation 8
Figure 3.1 – Board Initialization on a 2-D Array 13
Figure 3.2 – Havannah Board Size 4 14
Figure 3.3 – Neighbors 15
Figure 3.4 – EGDES and CORNER Identification 16
Figure 3.5 – Chain Connections 19
Figure 3.6 – Chain Connections Tree Representation 20
Figure 3.7 – Two Different Chains 20
Figure 3.8 – Two Chains Connected 21
Figure 3.9 – Tree Depth Increases with Chain Connections 21
Figure 3.10 – Tree Depth Compression 22
Figure 3.11 – Two Chains Forming a Bridge 23
Figure 3.12 – Forks 25
Figure 3.13 – Rings 26
Figure 3.14 – Simple Rings 27
Figure 3.15 – Simple Rings Connection 28
Figure 3.16 – Simple Ring Connections (Continued) 28
Figure 3.17 – Blobs 30
Figure 3.18 – Blob Connections 30
Figure 3.19 – Blobs Processing 31
Figure 3.20 – Wanderer Implementation of the MC Tree 32
Figure 3.21 – MC Tree Child and Sibling References 33
Figure 3.22 – MC Tree – Root with Available Moves as Children 34
Figure 3.23 – MC Tree UCT Selection 36
Figure 3.24 – MC Tree Expansion 37
Figure 3.25 – MC Tree Results Propagation 37
Figure 3.26 – MC Tree Expanded 39
Figure 5.1 – Mate in One 52
Figure 5.2 – Children Creation From Leaf Node 54
Figure 5.3 – Win and Loss Tree Propagation 55
Figure 5.4 – Only One Move, Make it 56
Figure 5.5 – Save Tree for Next Move 58
Figure 5.6 – Joints and Adjacent Moves to Same Color Stones 60
Figure 5.7 – Fork Frame 60
Figure 5.8 – Leaf Parallelization 63
Figure 5.9 – Root Parallelization 64
Figure 5.10 – Weighted Union-Find Algorithm 65

List of Tables

Table 1.1 – UCT Formula 7
Table 4.1 – Human Player v. WandererMC 43
Table 4.2 – WandererMC as First Player 44
Table 4.3 – WandererUCT as First Player 45
Table 4.4 – WandererBase as First Player 46
Table 4.5 – WandererBase as Second Player 46
Table 4.6 – WandererUCT with Most Visits 47
Table 4.7 – WandererBase as First Player –Most Visits 47
Table 4.8 – WandererUCT as First Player – Most Visits 47
Table 4.9 – WandererUCT V as 50 48
Table 4.10 – WandererUCT V as 100 48
Table 4.11 – WandererUCT T as 5 48
Table 4.12 – WandererUCT K decreased 49
Table 4.13 – WandererUCT Improved 49
Table 4.14 – New Baseline v. WandererMC 50

ABSTRACT

Havannah, a Monte Carlo Approach

By

Roberto Nahue

Master of Science in Computer Science

This paper describes how the Monte Carlo Tree Search (MCTS) algorithm approach is used to implement a computer engine to play the board game known as Havannah, a game that has recently sparked interest in the artificial intelligence gaming community. A very simple approach using the MCTS is described with a slight variation to improve how memory is managed when building the Monte Carlo Tree. A few other improvements are also discussed as future considerations such as threading and memory management to further improve efficiency of how the Monte Carlo tree is generated. The Havannah engine described was given the name of Wanderer and it was one of two that participated in the 2009 Computer Olympiads held in Spain. Its source code has remained private only to be used as part of research conducted by the California State University, Northridge Computer Science department graduate students. Since its inception, the source code has gone through several iterations but this paper describes the

initial development which was aimed to check the feasibility of this algorithm with this type of board game. The MCTS has already been successfully used in different board games with the same style of play as the Havannah board game where connection of played points is required to win the game such as Go and Hex [8]. The computer engine that was developed to test the feasibility of the MCTS in Havannah was named Wanderer as it represents moves on the board that together form a destination. The programming language used was C++ as efficiency and optimization was required in order to fully test the computer engine's capabilities. Through much refining and optimization of the code an initial version of Wanderer's capabilities were tested during a programming competition held in Spain. Wanderer proved to be a strong competitor and took the winning spot proving that implementing this board game with a MCTS is feasible. Through testing and hands on experience playing with Wanderer, I can conclude that Havannah, just like other board games where similar algorithms have been used, was a perfect candidate for the Monte Carlo Tree Search algorithm.

Chapter 1

Introduction

1.1 Havannah

Havannah is a board game created by Christian Freeling, a Dutch Mathematician, in the year of 1979. Havannah is a connection game which is very simple for humans to play, as his inventor intended, but complicated for computers to program. Many reasons for this fact include Havannah not having material imbalance, there is no movement of pieces, no general direction as connections can start anywhere on the board and connect to any side, no captures and no promotions [8]. The inventor was so sure that no computer could be programmed to play this game well enough to beat him that in the 2002 edition of the Abstract Games magazine he claimed that on a board size with base 10 (the maximum board size allowed) no computer would be able to beat him once in 10 games in the following 10 years [2]. However, to prove Freeling wrong, in 2012, a match was held between Freeling and three of the strongest Havannah computer engines available that year, Lajkonik [3], Castro [4] and Wanderer in which computer programs did beat the inventor. Havannah is played on a hexagonal board, where each side is composed of hexagons. The number of hexagons on each side varies from four to ten which determines the size of the board. Moves on the board are given as a letter-number pair as if the board represented a coordinate system. The following figure shows a hexagonal Havannah board of size ten which is the biggest size board the game can be played on.

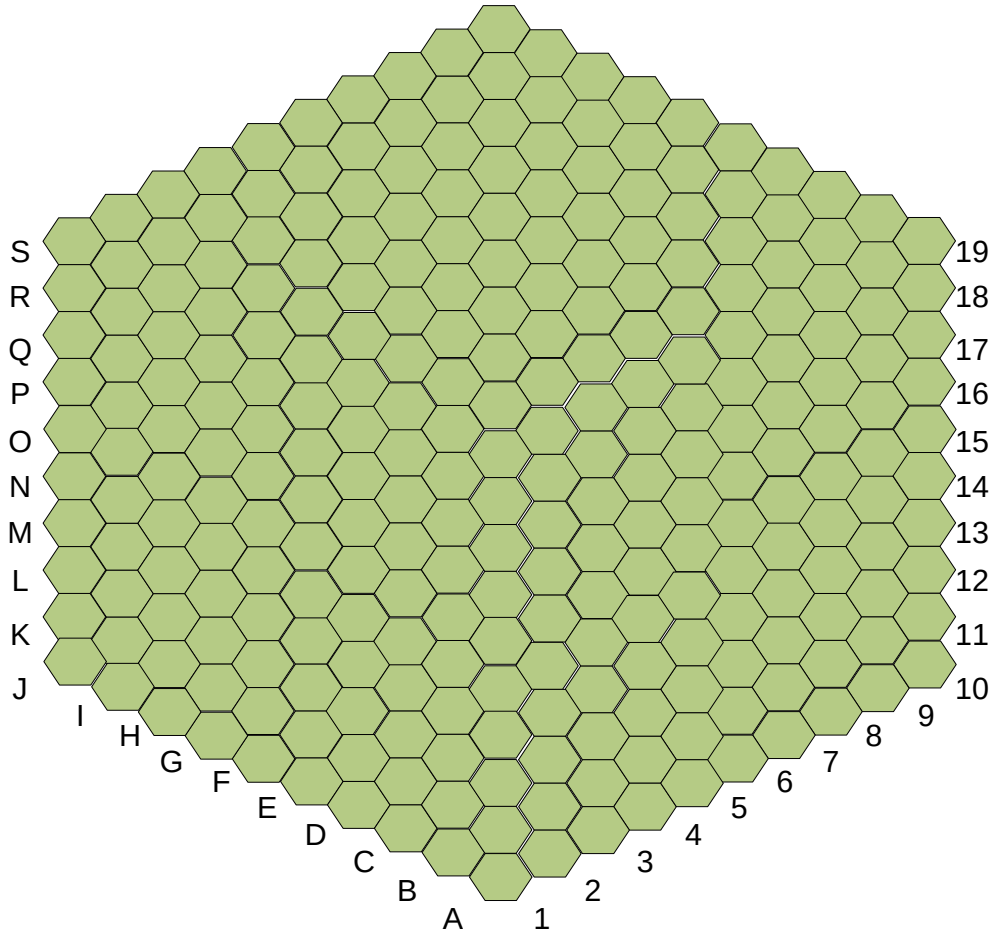

Figure 1.1 – Havannah Board Base 10

1.1.1 Havannah Board Rules

Havannah is a connection game played by two players in which each player, either color White or Black, tries to form any of the following three structures by placing stones on empty cells of the board:

- Ring
- Bridge
- Fork

A Ring is a connection of stones of the same color looping around one or more cells. The surrounded cells can either be empty, the same color as the current played point and/or the opposite color. A bridge is a connection of stones of the same color that connects any

two distinct corners of the board. Finally, a fork is a connection of stones of the same color that connect any three distinct sides of the board, not including corners. The following figure shows a representation of the winning structures that can be formed on a base ten Havannah board.

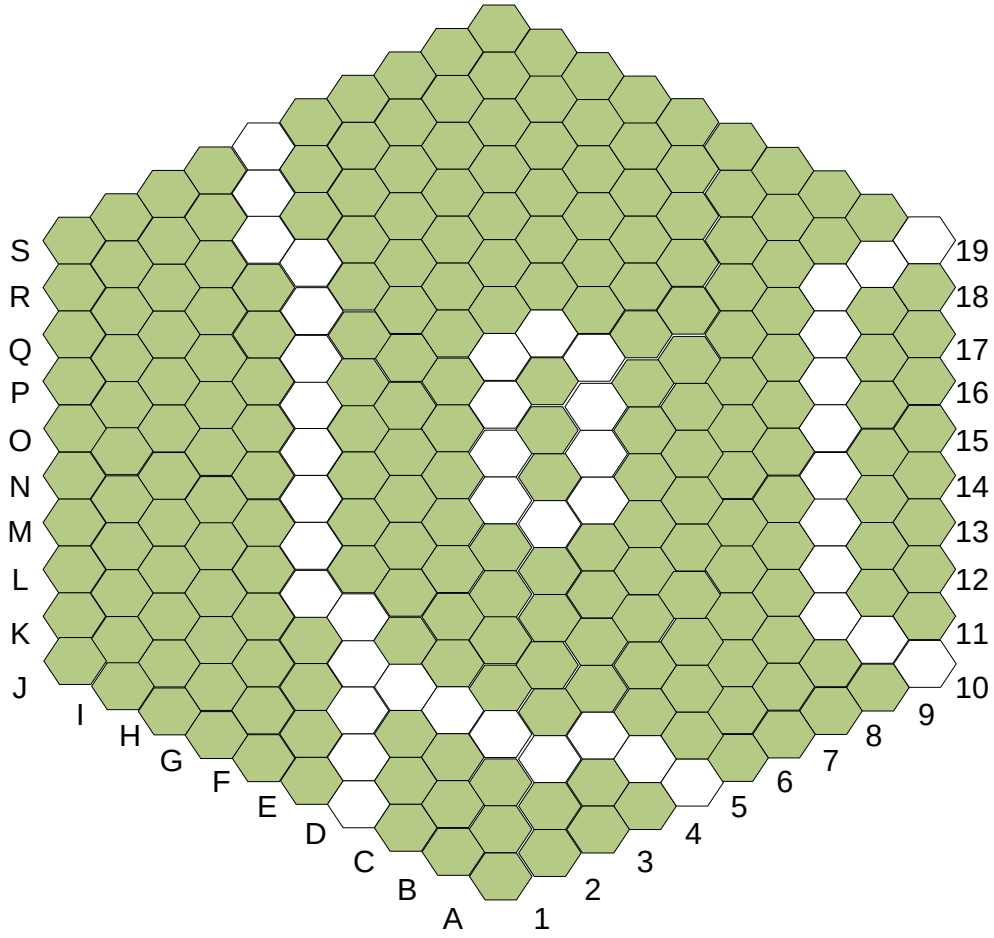

Figure 1.2 – Havannah Board Winning Structures

The game starts on an empty board where White places the first stone on the board and then each player alternates moves until one player forms one of the winning structures described. Since there is an advantage in starting the game, Havannah implements the concept of a swap where after the first move is made, the second player has a choice of either letting the move stand and take his turn with a black stone, or switching places with the white stone [11].

1.2 Monte Carlo (MC) Algorithm

As noted, the Havannah game is very simple to play where players keep placing stones on empty cells of the board until a player forms one of the winning structures as shown in *Figure 1.2 – Havannah Board Winning Structures*. A stone placed on the board has no value, or material imbalance, until it has actually created a winning structure. It is true that it can be estimated how close a player is to winning; however, individual stones on the board do not change value. A stone's value is only seen once connections are made that will lead to a winning configuration. Also, once a stone is placed on the board, it does not move. In addition, winning structures can be formed starting anywhere on the board and the state of a move does not change as the game progresses, meaning, once a stone is played, the stone's state remains the same throughout the entire game (stones can't change color for example except for when the swap is taken at the beginning of the game). So the concept of the game is quite simple and it takes good strategy to play the game well. There have been attempts at programming Havannah using search algorithms such as the Alpha-beta Search; however, these have yielded poor results [19]. There are a few reasons why Havannah is difficult to program. For example, there is no straightforward evaluation function that can determine if a move is useful or not. Also, since it is difficult to determine the usefulness of a move there is no clear or general rule for pruning bad moves. In addition, Havannah has a large branching factor. Its largest board game size contains 271 possible opening moves [19]. However, despite Havannah's complexity, it has been proven that with the use of MCTS algorithms, board games similar to Havannah, such as Go, are programmable and result in good players. The Monte Carlo Tree Search (MCTS) methodology is used to find optimal decisions for

a given domain, in our case the optimal next move in the game of Havannah which is done by randomly collecting samples and building a tree according to the results [12]. The Monte Carlo Algorithm in its simplest form is quite simple when applied to the game of Havannah. The tree that is built using straight Monte Carlo Search is only one level deep as opposed to the full scale tree built using a variation to the MCTS algorithm as explain later in this section. With MC in its simplest form, at any given move, a root note is created with its children being all the available moves for the current player. For every iteration of the algorithm, a random simulation will take place for each child of the root node. This simulation is essentially a random game to the end until either color forms any of the winning structures or neither color wins at all. Each child keeps track of the number of simulations that results in a won game for that child, namely, it keeps track of its winning rate. The following figure shows a representation of the basic concept of the Monte Carlo algorithm.

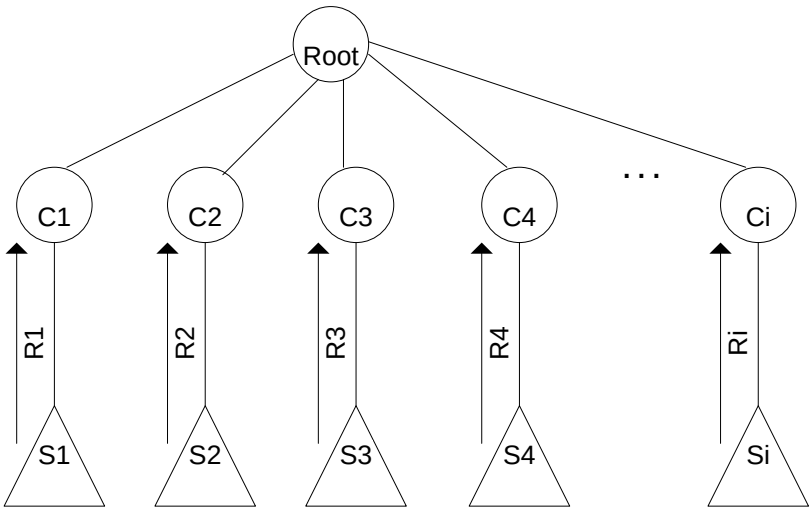

Figure 1.3 – Basic MC Algorithm

In the figure above, C1 through Ci represent all the available moves the current player can make at this time. Every child is then simulated randomly, Si, and then the result, Ri, is updated in the child node. Once enough simulations have been run, usually given by a

time constraint, and enough data collected, a child of the root node with the highest win rate or highest number of won games is selected as the optimal move and as the next move to be played. The basic concept of this algorithm, as shown above, usually yields good results, as it has been used to create world champion level play in board games such as Bridge and Scrabble [1]. However, the overall statistics will not converge to the best move [10] because random simulations are wasted on nodes that may result in losing games. For this reason, different variations have been applied to the Monte Carlo algorithm where a full scale tree is built and a search for the most promising move [12] is performed by looking at the entire tree.

In essence, the MCTS algorithm is composed of four steps [13] for every iteration the algorithm is performed. Instead of sequentially playing random simulations for each child, simulations are performed on a child based on a (1) selection strategy where nodes on the tree are selected until a leaf node is found. The node can then be either (2) randomly simulated or expanded following an (3) expansion strategy which adds children nodes below the selected node. After a simulation is performed, the result is then (4) propagated up the tree to a child of the root node that originated the simulation.

The selection strategy considered in this paper is called UCT, or Upper Confidence bounds for Trees [20]. A node in the tree is composed of two basic pieces of information, the number of wins for when this move develops into a winning structure and the number of visits or number of times the node has been selected using the UCT selection strategy. When the UCT variation is applied to the Monte Carlo Tree, the selection for the node is determined according to which has the highest UCT value as calculated by the following formula:

```
(child.wins/child.visits)+K*(sqrt((ln(parent.visits))/(child.visits)))
```

Table 1.1 – UCT Formula

The first half of the formula represents the win rate for a given path on the tree which is considered the exploitation of moves that yield winning playouts. The second part of the formula represents the exploration of moves with low simulations. K is a constant value which represents the exploration parameter, chosen empirically for the given domain [12]. Since the formula above relies on nodes being visited at least once (division by zero not allowed), child nodes that haven't been selected, or visited, yet are given a random UCT value which allows the random selection of moves as the tree is built. The following figure shows the concept of MCTS with the UCT variation. The first half of the figure shows C3 is selected for a random simulation and the second half shows C3 being expanded with children nodes. From this level down a random simulation is performed from node C2 and the result R2 of this simulation is then propagated upwards in the tree. When simulations are completed, given by a time or memory constraint, the move to play is the child of the root with the highest win value. See *Section 3.4* for details on implementation of MCTS with UCT.

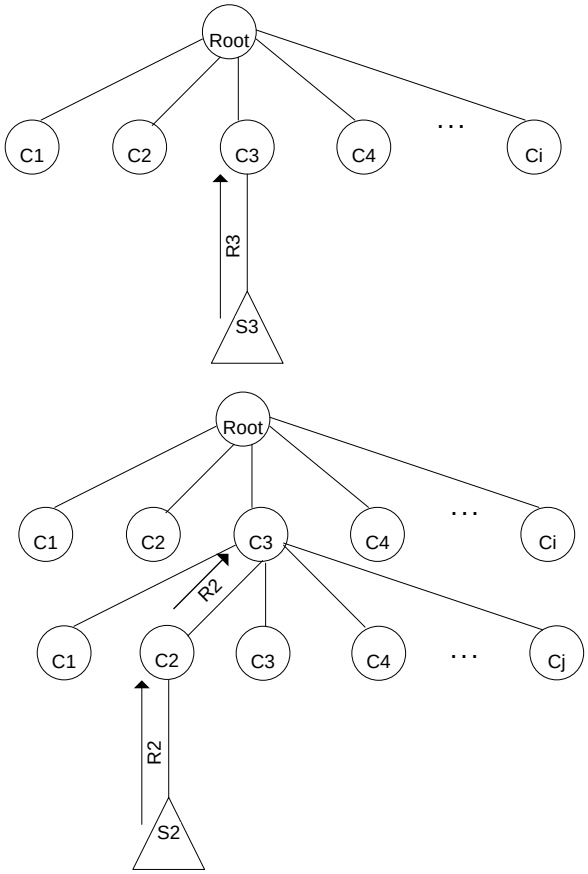

Figure 1.4 – MCTS with UCT Variation

Chapter 2

Wanderer

2.1 Description

Wanderer is a computer game engine developed to play the game of Havannah. Wanderer was implemented with the goal to prove the feasibility of the MCTS on this board game and to show actual game results instead of theoretical values. When Wanderer was first written, two main things were considered: it needed to be fast in checking for winning structures and it needed to be efficient in memory used. Since its conception, Wanderer has gone through different iterations in order to make it more efficient. The implementation details that will follow only consider the first iteration phase. The subsequent iterations and future implementations will be discussed in *Chapter 5 Future Considerations*.

In order to aid in making Wanderer fast, the language that was chosen to program Wanderer was C++ which allows for ease of use in memory allocation and de-allocation. Using C++ also presented a challenge since Wanderer was my first time working in a project of this scale in C++. Wanderer also implements a communication protocol called GTP v2 which is the standard protocol used in game competitions in games such as GO and Hex, both board games very similar to Havannah in which connections need to be made in order to win the game. The use of this protocol made it possible to test Wanderer at different levels since it allows for using third party graphical interfaces such as HGUI which allows the use of computer programs to either play against a human opponent or against another computer program, which in our case are two different versions of Wanderer. The capability of this tool that allows for two computer programs to play each

other is useful since it allows different versions of Wanderer to be tested. For example, the UCT constant used in the formula to determine the best child to select in the search tree shows how different values affect the win rate of Wanderer. Also, different thinking times, time allowed for Wanderer to build the search tree before making a move, can be tested to determine how time improves Wanderer's decision to make a move.

2.2 Classes

Wanderer was implemented with the following classes:

- Point.cpp
- BoardGame.cpp
- Engine.cpp
- MainEngine.cpp

2.2.1 Point Class

The Point class is the building blocks of Wanderer. A Point represents a move/position on the Havannah board. The board is represented as a 2-dimensional square array of Points where only some of the Point objects form the hexagonal board. This is accomplished by having valid and invalid Points where the valid Points are within the boundaries of the hex board. The name Point is used since the Havannah board is really an x-y coordinate system as implemented in Wanderer. The x coordinate maps directly to the letter on the left side of the board and the y coordinate maps to the number on the right side of the board (Refer to *Figure 3.1 – Board Initialization on a 2-D Array* for a visual representation of the board).

The following are the members that make up a Point object:

- posX {0-18} – the x coordinate of the point on the board.
- posY {0 - 18} – the y coordinate of the point on the board.
- isValid {true, false} – used to make up the subset of the points on the 2-dimensional square array to build the hexagonal Havannah board.
- curColor {NONE, WHITE, BLACK} – indicates a played move on the board.
- curPosition {EDGE = 4, CORNER = 3, MIDDLE = 6}- indicates where the point is located on the board.
- chain {0 - 255} – this is used to keep track of connected stones which is then used to determine wins during a game.
- wins {0 – (2^16 – 1)} – this is used to keep track of the number of wins a path on the MC tree yields. For a given simulation out of a given leaf node, if the simulation results in a win, then the wins variable for all nodes in the path matching the color of the leaf node will be updated (incremented by 1). But if the simulation is lost, then the opposite color nodes in the selected path are updated (wins variable is incremented by 1).
- visits {0 – (2^16 – 1)} – this is a counter for the number of times a node has been visited or selected in the path down the tree.
- parentNode {Point object reference} – a reference to another point on the board, used when connecting points on the board.
- child {Point object reference} – used during the building of the Monte Carlo tree and it represents a pointer to the list of child nodes.
- sibling {Point object reference} – also used during the building of the Monte

Carlo tree and it represents a pointer to another node. Siblings make up the child nodes to another node in the tree.

- neighbors {Point object reference array} – this represents a list of neighbor points surrounding a given point on the board.

2.2.2 BoardGame Class

This class is responsible for the initialization of the board. It also creates the neighbors for each point, a vector of references of surrounding points to a given point on the board. This is what determines the position for each point which is based on the number of neighbors a given point has. This class also offers the interface of the engine itself such as providing calls to set moves on the board, returning the next un-played point and checking for winning moves. In addition, it provides the calls for managing the MC tree such as creating children for a given node, playing random games off of leaf nodes or nodes without children and selecting the best path to follow in the tree by calculating the UCT value for children of a given node.

2.2.3 Engine Class

The Engine class is responsible for providing an interface to the GTP protocol with calls such as playing a move on the board, generating a random move and generating a UCT move. The Engine class is the intermediate class between the MainEngine and Boardgame classes.

2.2.4 MainEngine Class

The MainEngine class implements the GTP Protocol and it is the interface to the outside world. This is responsible for obtaining the parameters that will be used during a game such as setting the board size.

Chapter 3

Wanderer Implementation Details

3.1 Board Initialization

Wanderer's board initialization is done on an x-y coordinate system on a 2-dimensional array. Each x-y point corresponds to a stone being played on the board or an actual position on the board. For our purposes, we are going to be examining Wanderer's initialization on a size 4 board which translates to a 7x7 2-D array. Since not all points on the X,Y system are valid, during board initialization each point is marked as valid or invalid (each point is created with the flag set to invalid and it is only set to valid during initialization). Refer to the figure below.

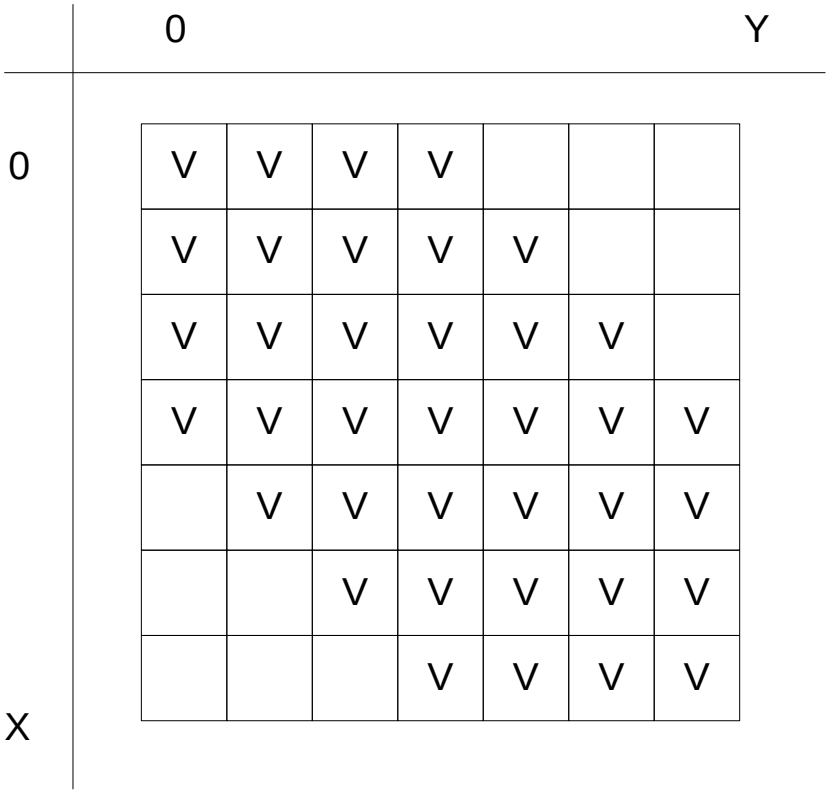

Figure 3.1 – Board Initialization on a 2-D Array

For simplicity, moving forward, the graphs will be using an actual mockup of how the board looks when played. So, the following is what the above 2-D translates to when looking at the actual board; where point A1 maps to point (0,0) on the 2D array and point C2 maps to point (2,1). Refer to the following figure below.

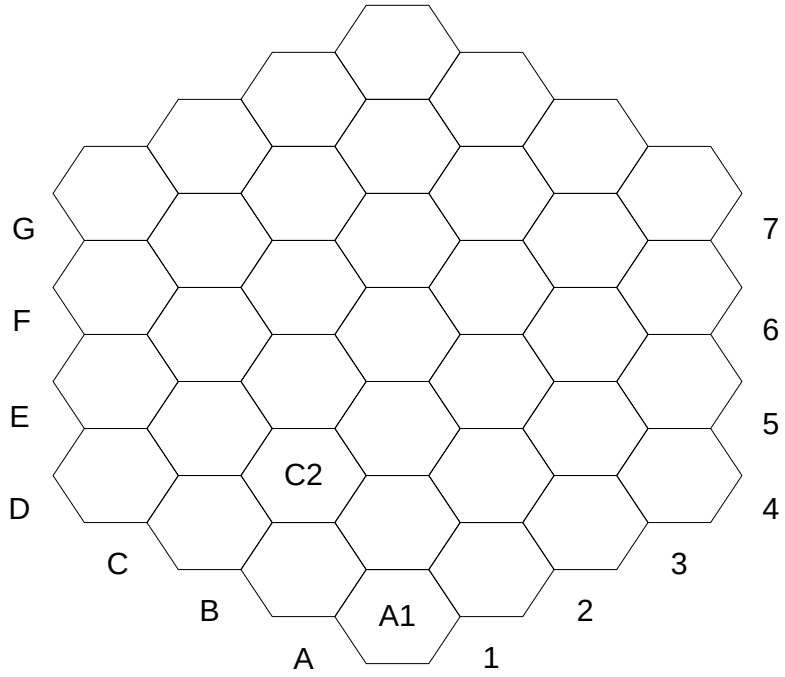

Figure 3.2 – Havannah Board Size 4

3.2 Neighbors

As pictured in *Figure 3.3 – Neighbors*, every point on the board is surrounded by other points. During initialization, each point will contain a list of these surrounding points, called neighbors. The concept of neighbors aids in determining connections. When a move is made, placed on the board, its neighboring points are checked to determine if any neighbor matches the color of the move that was just played. So neighbors make it easy to check surrounding points by traversing a list of these points instead of using the coordinate system (using the point's coordinates and then adding/subtracting from each axis to get to a surrounding point – a tedious and slow process if done in real time (during

the actual game execution) especially when not all points on a board are valid). Neighbors are added to each point in a certain order during the first initialization pass, and then these are updated to make it possible for the algorithm that checks for rings to work. See *Section 3.3.3 Rings*.

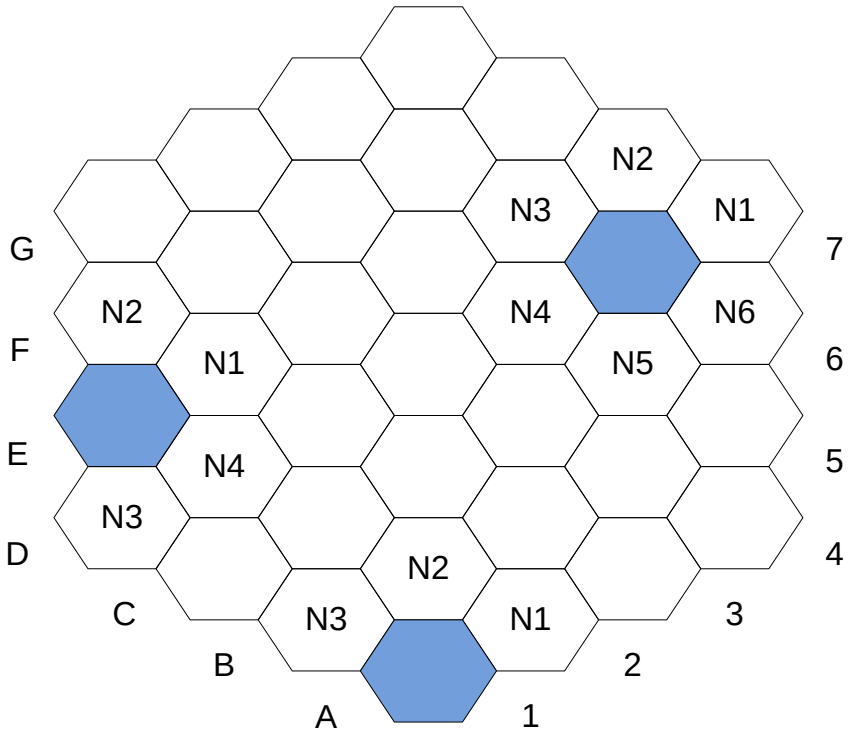

Figure 3.3 – Neighbors

A point can be one of three kinds, an EDGE, a CORNER or MIDDLE. This is determined, as seen above, by the number of neighbors the point has which is important as it really determines where the point is located on the board. Knowing the position of a point is important when determining if two or more corners are connected (bridge) or if three or more edges are connected (fork). Refer to *Appendix A: Setting Point's Neighbors* to see exactly how the x-y coordinate system is used to set each point's neighbors by using the current point's x-y position.

Neighbors are added to a vector of references to Point objects and these are added counter-clockwise. If we take a MIDDLE point, for example, as in *Figure 3.3 – Neighbors*, neighbors are added starting with the North-East point and going counter-clockwise to the North neighbor, North-West neighbor, South-West neighbor, South neighbor and finally the South-East neighbor. The fashion in which neighbors are added is important because it will play a role when determining Rings (*Section 3.3.3 Rings*). Once points have been marked as EDGE, CORNER and MIDDLE, the next step is to give EDGE points and CORNER points an ID. There are six possible corners and six possible edges for any given board size. Giving EDGE and CORNER points an ID, makes it possible to recognize winning structures, like forks and bridges (See *Section 3.2.3 Determining Winning Moves*) that require distinct EDGES in the case of forks and distinct CORNERs in the case of bridges. For example, three EDGE Points can form a fork, only if each Point is on different sides. The next diagram – *Figure 3.4 – EDGES and CORNERS* Identification – illustrates how Points are identified on the board.

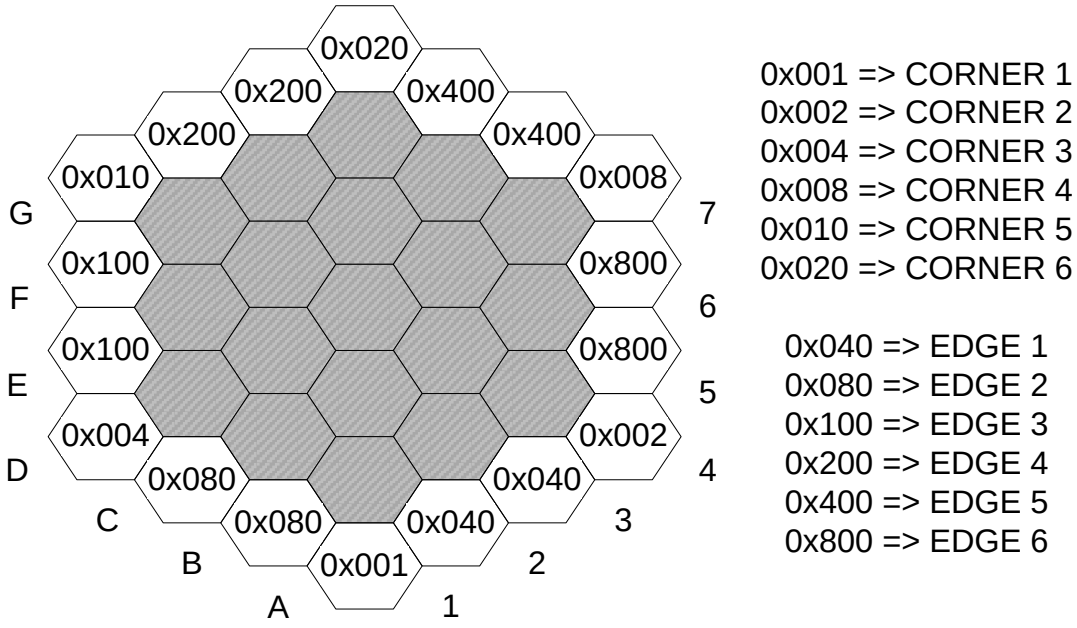

Figure 3.4 – EGDES and CORNER Identification

The point identification is a system used to easily find if a connection has produced a winning move. As seen in the diagram above, the id of a point is represented by bits. Every point on the Havannah board contains a variable called chain represented as an integer value. As points are connected together, the chain variables of the Points (or the parent of the connected Points) are OR'ed together. The resulting chain value will determine if the Point structure has CORNERS or EDGES in it. This chain value is divided into two parts where bits 0 through 5 represent CORNERs 1 through 6 and bits 6 through 11 represent EDGEs 1 through 6. For example, CORNER 1's chain value is 1; CORNER 2's chain value is 2; CORNER 3's chain value is 4 and so on. The following sections go into detail about how games are won and explain the algorithm used by Wanderer to determine Bridges, Forks and Rings and how the use of the chain variable is implemented.

3.3 Determining Winning Moves

A game can be won when a player can complete any of the following connection structures:

- Bridge
- Fork
- Ring

As noted in *Figure 1.2 – Havannah Board Winning Structures*, a Ring is a connected chain of points surrounding at least one point regardless of its color (including no color at all). A Bridge is formed when at least two corners are connected by a chain of points. Lastly, a Fork is a chain of points connecting at least three sides. As previously seen in *Figure 3.4 – EDGES and CORNER Identification*, CORNERs are not part of EDGES and

do not count when connecting Forks. When moves are placed on the board, its surrounding neighbors are looked at to determine if a connection to an existing chain of Points can be made. Neighbors are looked at counter-clockwise. Connections are made sequentially only to neighbors of the same color as the move that was just played. A connection means that the played point now forms part of the neighbor's connected points by sharing the same identifier. Because of this, in addition to points being connected to neighbors of the same color, they are only connected if the neighbor does not share the same identifier (this is only the case when the point that was just played has more than one same color neighbor). Wanderer implements a Path-Compression Union-Find (PCUF) algorithm when making point connections. Each point contains a reference to another point called a parent reference. When Points are played on the board, the parent reference is Null. As points are connected together on the board, a chain will be built with all points in the chain pointing to the same parent point known as the root node point. This makes it so each chain will always have a common identifier, the Point reference that every Point in the chain will refer to as the root Point. So when a point is played on the board and a neighbor is of the same color, the root of each point (current played point and neighbor of same color) is identified. If the root points are different then the connection can be made by setting the current played point's root point reference to the neighbor's root point. The following figure shows the connection that is made between points with different root points.

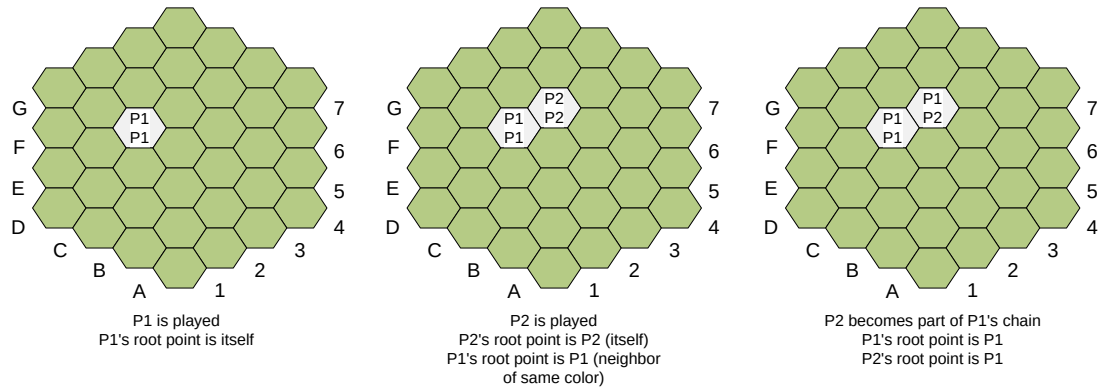

Figure 3.5 – Chain Connections

The purpose of the PCUF algorithm is to optimize the connection of Points on the board. Since each point has a common identifier, the root point, and the Points in the chain are connected in a tree like structure it takes O(log n) steps to obtain the reference to the root point for a given Point in a chain, where n represents the total number of points in the connected chain. For the current Wanderer's implementation of the PCUF algorithm, only a few steps are needed to reach the root point because the tree will balance itself as the root point is found for connected points [9]. Also, there are two players on the board and each player is not only trying to connect points but also trying to prevent the opposite player from making connections which makes the number of possible chains even smaller. For testing purposes and to show that the number of steps to reach the root node is a small value, Wanderer was simulated on a board size ten with 10 seconds per move. With an average of 162566 random simulations per move, with 30 moves to complete a winning structure, the highest number of steps to reach a root point was 10. This shows that the trees created when points are connected are not too deep. Also, as mentioned before, the tree is balanced as connections are made. For the same test, the average number of steps to reach the root point was only 1.5 steps.

The following figure is a tree representation of the implementation of the PCUF algorithm with moves on the Havannah board.

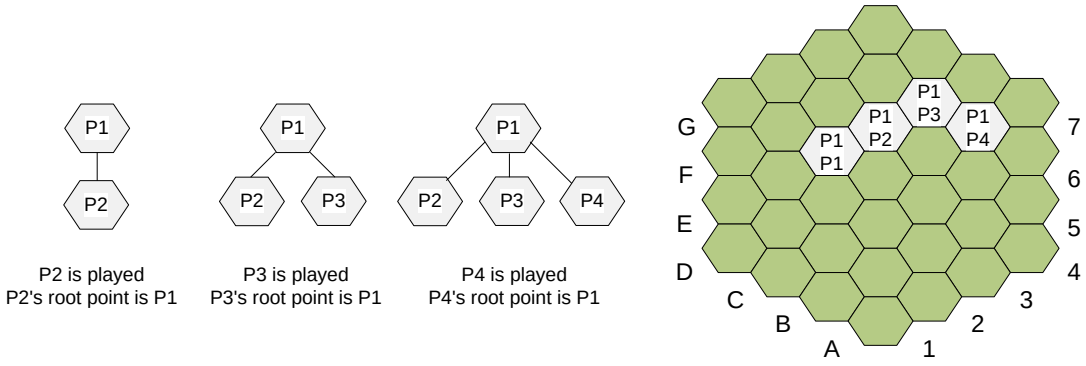

Figure 3.6 – Chain Connections Tree Representation

In the example above, every move played shares the same root and the number of steps to get to the root node is two since to find the root node, the current point's parent's parent also has to be looked at. In addition, the depth of the tree which is the factor driving the number of steps needed to get to the root, is balanced with path compression. This path compression is accomplished by setting each point's parent to point to the root node when the chain of connected points is recursively traversed. The following figures show the path compression in a tree representation along with its equivalent Havannah board.

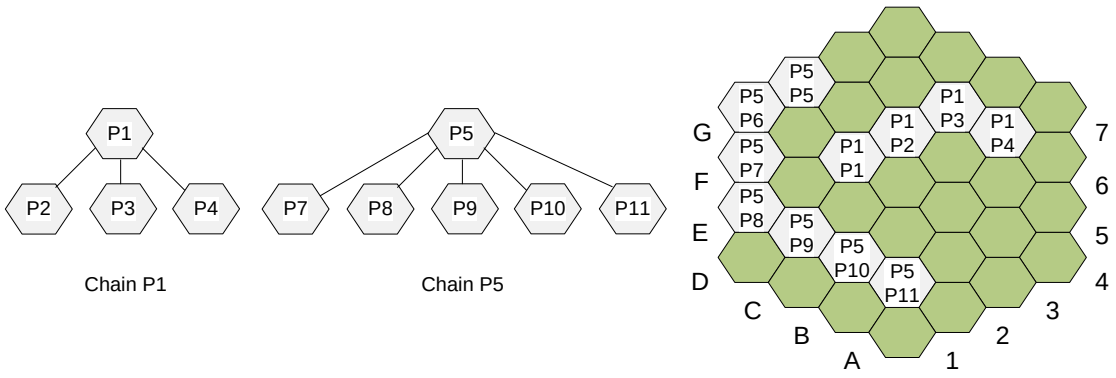

Figure 3.7 – Two Different Chains

The above figure represents two different chain of points and each chain is identified by a single root point, chain P1 has P1 as the root point and chain P5 has P5 as the root point.

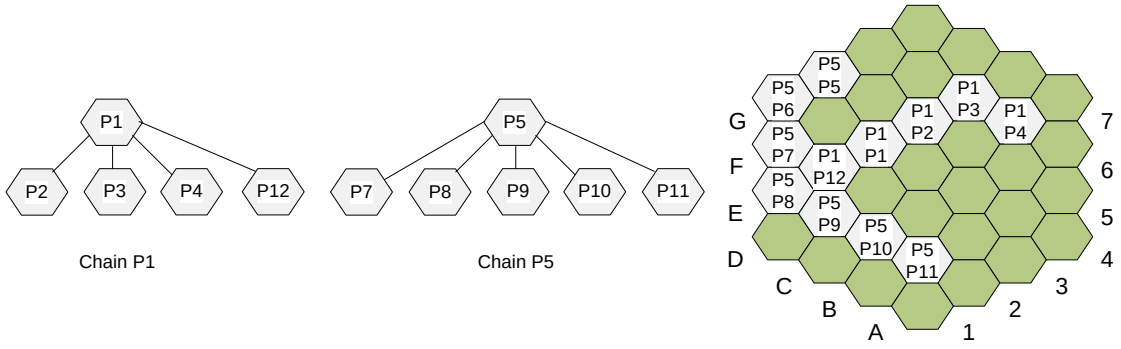

Figure 3.8 – Two Chains Connected

P12 is played and as *Section 3.2 Neighbors* explain, P1 is the first neighbor of the same color as P12 to be looked at (North-East point is looked at first – except when dealing with Rings – See Section 3.3.3 Rings). P1's chain's root point is P1 so P12's root point is P1. So P12 is now one more child of P1 as seen in the first tree above.

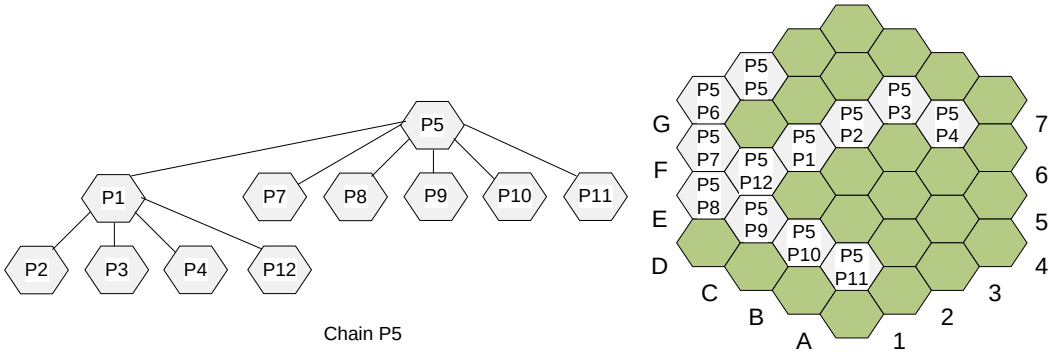

Figure 3.9 – Tree Depth Increases with Chain Connections

Moving counter-clockwise, the next neighbor to P12 is P7. P12's root point is P1 and P7's root point is P5. The algorithm always joins the current point's root to the neighbor's root so P12's root point which is P1 is now connected to P5 and then the entire chain has P5 as its root point. One thing to note is that the depth of the tree was incremented by one, therefore, if a point is played next to a child of P1, it will take three steps to get to its root point.

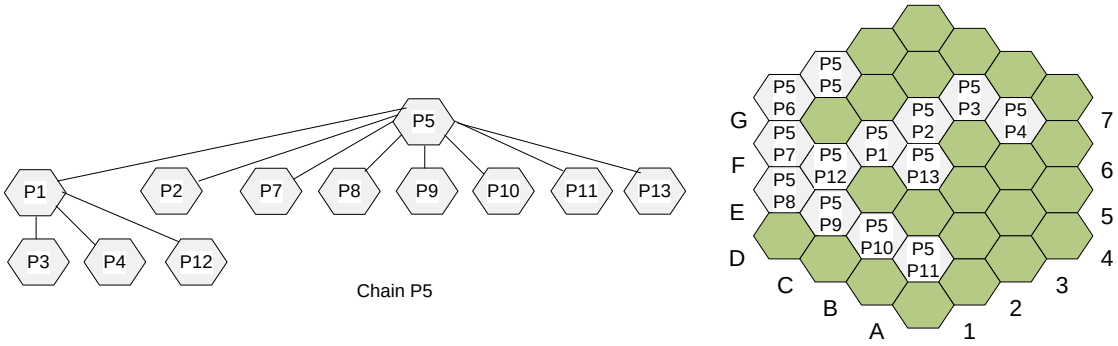

Figure 3.10 – Tree Depth Compression

In the above figure, P13 is played next to a child of P5. With the Path Compression algorithm, the searching of the root point of P2 in this case is done recursively, and as the recursion unwinds, each point traversed in the search is connected to the root point. For the above example, P2 is now pointing to P5 as its root point. So as the tree's depth increases, it also gets compressed when making connections to the leaves.

The current implementation of the Union-Find algorithm can be further optimized by adding a weight to a chain of points which would indicate the depth of the tree. Currently, the current played point always joins its surrounding neighbor's chains. If we add a weight to each chain, then we would add the less weighted chain to the more weighted chain. This would ensure that the tree is always balanced. As future improvements, adding this capability and through testing it would be interesting to see if the added optimization really adds significant value to Wanderer's speed.

The source code in *Appendix B: Using Point's Root for Union-Find Algorithm* shows the implementation of the Union-Find algorithm. Note that only relevant parts to the Union-Find algorithm are shown in the appendix. As part of the same source code, Bridges, Forks and Rings are also checked which are denoted with comments. The code for checking Bridges, Forks and Rings will be covered in the sections to follow.

3.3.1 Bridges

A bridge is formed when at least two distinct CORNERs are connected together as seen in *Figure 1.2 – Havannah Board Winning Structures*. Wanderer's implementation for detection of Bridges is quite fast as it only takes two steps to check if a Bridge has been found after Points have been connected using the PCUF algorithm described in previous sections. As mentioned before, each Point object contains a variable called chain. The chain variable is an integer where the first six bit positions identify the points as CORNERs 1 through 6. When a point is first played, it will search through its neighbor vector and it will join with points of the same color if the root points are different. When a point or chain of points is connected the chain variable of the roots are OR'ed together. The resulting chain, in the case of Bridges, will yield a value showing if two or more corners have been connected. In the figure below, each chain is connected to a corner and a move into position C5 is a winning move since it will result in a connection of two or more CORNERs.

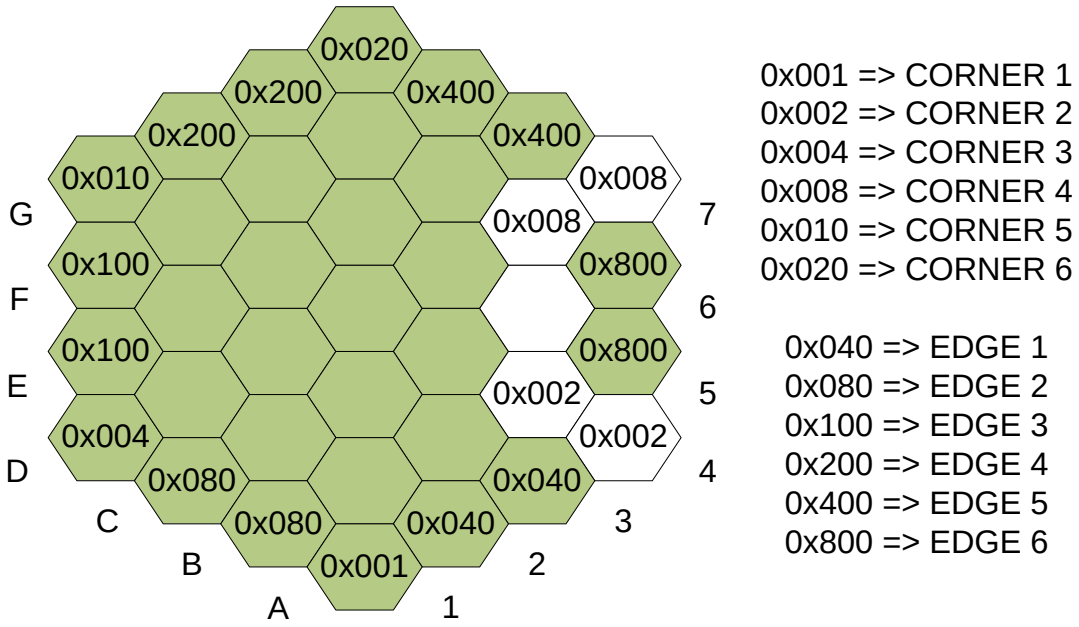

Figure 3.11 – Two Chains Forming a Bridge

In the above figure, when C5 is moved, the resulting OR'ed chain yields a value of 0x00A. This value is one of multiple combinations which will indicate that two or more corners have been connected together. Instead of creating multiple if statements to check for these winning combinations, Wanderer has been optimized to make the check of a Bridge O(1). During initialization of the board, a boolean array is created where the index in the array represents the OR'ed chain value and the value in the array represents whether a bridge has been found. In the example above, when C5 was moved, the resulting chain value of the root point in the connected structure is 0x00A. So the bridge Boolean array at index 10 (0x00A) is initialized to true, indicating a bridge was found.

3.3.2 Forks

A Fork is formed when at least three distinct EDGE's are connected together as seen in *Figure 1.2 – Havannah Board Winning Structures*. In the same manner as with Bridges, Wanderer takes two steps to check for Forks. The same variable chain is used, but EDGEs 1 through 6 are represented by the upper six bits in the chain variable.
When the chain variable is OR'ed it will yield a value which will represent if a Fork is found. The figure below shows an example of a Fork being connected.

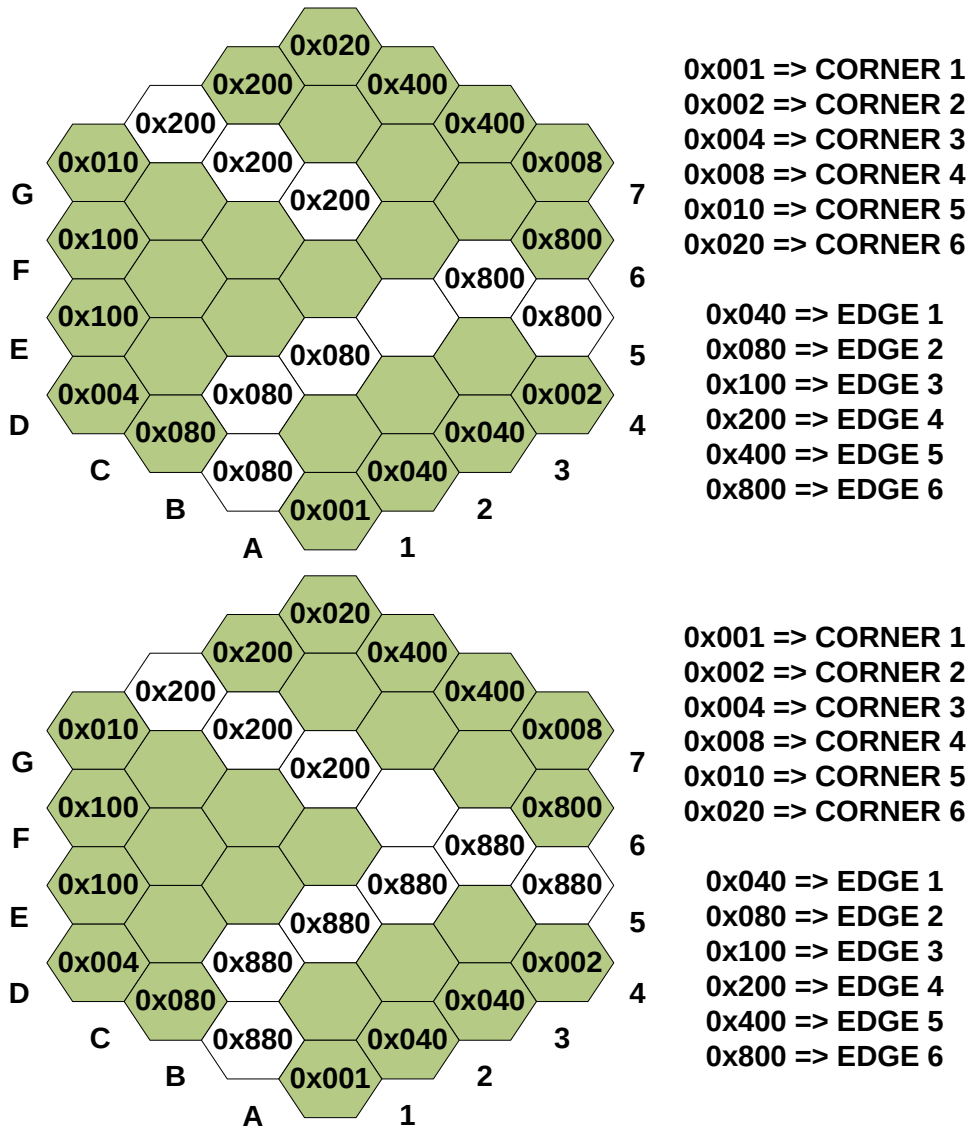

Figure 3.12 – Forks

In the first part of the above figure, we can see two EDGE chains being connected with a move to C4. The OR'ing of the two chains can be seen in the second half of the figure where the OR'ed chains show the new value of 0x880. This specific value only denotes that two EDGEs have been connected together and therefore does not yield a winning move. The second half of the figure shows D5 as the next move which results in the connection of three distinct EDGEs with an OR'ed value of 0xA80. In a similar fashion as with Bridges, during initialization, a Boolean array is created for Forks where the array

index represents the OR'ed chain, shifted 6 bits to the right, and the value indexed shows if a Fork has been formed. Checking for Forks in this manner also yields an O(1) check for Forks. In the case above the Fork Boolean array at index 42 (0xA80 >> 6) is initialized to true.

3.3.3 Rings

The creation of a Ring is the third way a game can be won in Havannah. *Figure 1.2 – Havannah Board Winning Structures* shows a simple representation of a Ring. As seen, a Ring is a chain or connection of points that surround at least one point regardless of its/their color. This is where the concept of a Blob is introduced in Wanderer's implementation. First, simple Rings, those surrounding at least an empty cell or point of the opposite color, are checked followed by Blobs, Rings surrounding a cell of the same color. The following figure shows different examples of Rings.

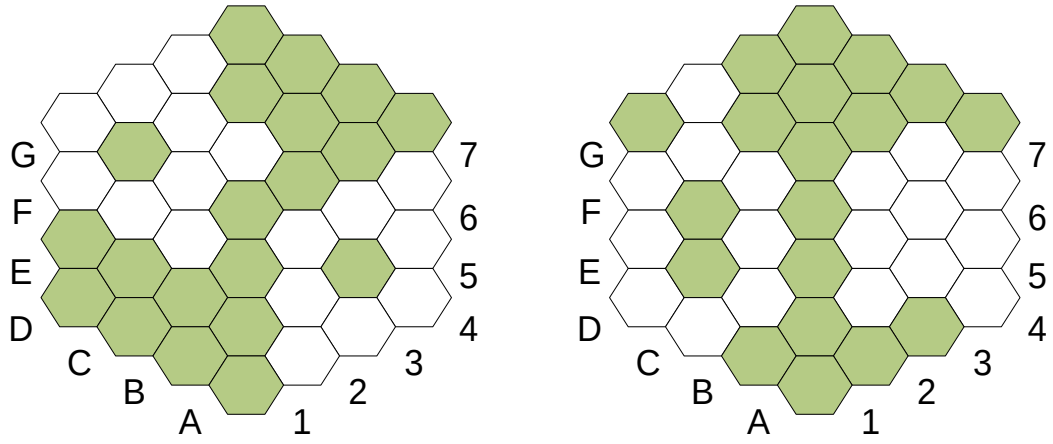

Figure 3.13 – Rings

3.3.3.1 Simple Rings

Simple Rings are those chain connections that surround at least one empty cell or a point of the opposite color. The next figure below shows different setups for Simple Rings.

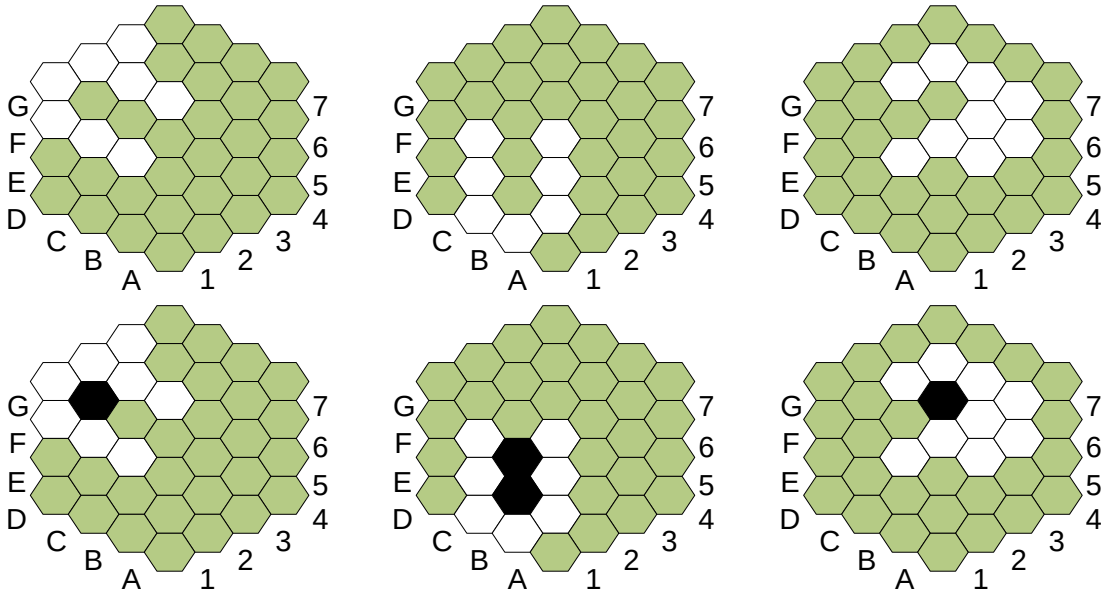

Figure 3.14 – Simple Rings

As noted in the above examples, a White move into position E4 forms a simple ring since it completes the structure that surrounds at least an empty cell or a cell of the opposite color. Wanderer's implementation for checking Simple Rings is different from how Bridges and Forks are checked. Looking at the second column in *Figure 3.14 – Simple Rings* above, when move E4 is played, E3 is the first neighbor of the same color (neighbors are checked counter-clockwise starting with the neighbor on the North-East side) the move played in E4 will be joined with. After the first connection to the neighbor in E3 is made, E4 points to E3's root point making them part of the same structure. Then, when the next neighbor of the same color is looked at, D4, the connection will result in a Simple Ring. The following set of figures shows graphically the algorithm used by Wanderer to detect simple Rings after move E4 is played.

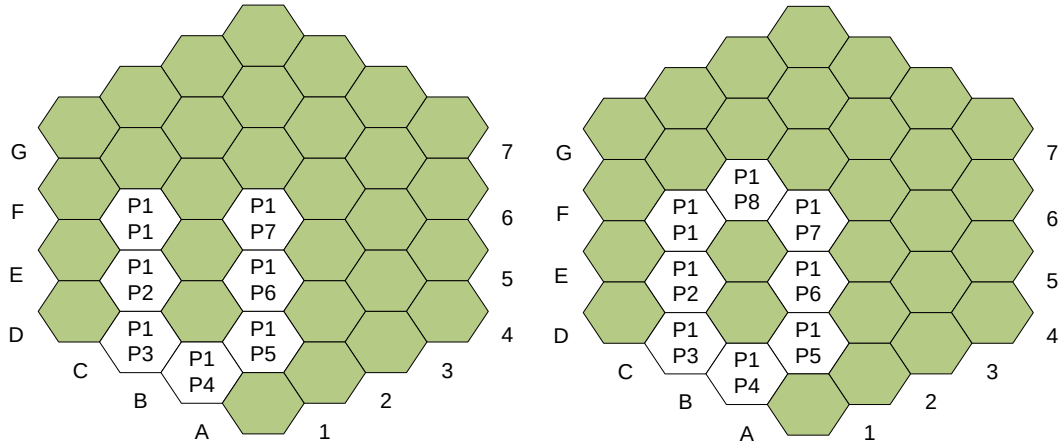

Figure 3.15 – Simple Rings Connection

In the figure above, point P8 closes the loop in chain P1 when moved into position E4. When point P8 is played, before it joins with P1's chain (as noted before, the played point always joins the neighbor's root point) it will traverse the remaining neighbors, in this case, D3 and D4.

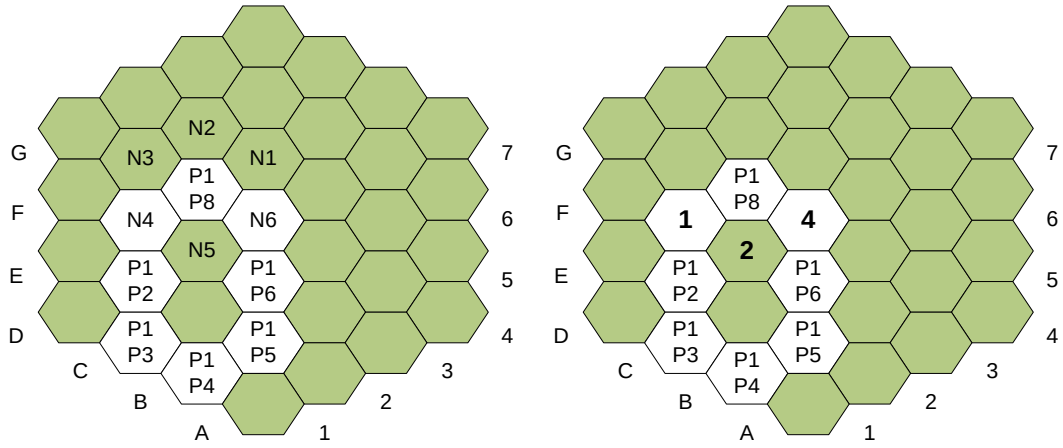

Figure 3.16 – Simple Ring Connections (Continued)

As seen above, neighbors N1, N2 and N3 are not of the same color as point P8 and therefore the algorithm for checking for simple Rings is not started yet. The second diagram in the above figure shows what happens after point P8's neighbor, E3, is of the same color. After a neighbor of the same color is found, E3, the algorithm marks it as the starting point as there is the possibility that a following neighbor of the same color may

also point to the same root point as E3 which is the case with the point in position D4. The algorithm starts out with setting a variable, loopSum, to 1 when E3 is found to be of the same color as P8. Following the remaining neighbors, the loopSum variable will be updated as follows. Bit 2 of loopSum will be set to 0 since D3 is not of the same color as P8. Bit 3 of loopSum will be set to 1 since D4 is of the same color and it shares the same root point as E3. In the above example, the final loopSum value is 0b101. Bits 1 and 3 signify that a point joined a point in each side that belonged to the same structure. Bit 2 means there was a point of the opposite color or an empty cell separating the structure. Visually, we conclude that move E4 closes the loop on a Simple Ring. Just like Bridges and Forks, there are multiple combinations that would create a simple Ring. Therefore, during initialization a Boolean array is also created for Simple Rings which delineates all of these possible scenarios. The index in the Boolean array represents the final loopSum value and the indexed value will show if a Simple Ring was detected or not.

3.3.3.2 Blobs

Blobs are a subset of Ring structures. A blob is really a group of connected Points forming a Ring around stones of the same color. Similar to Bridges, Forks and Simple Rings, during initialization a Boolean array is created for Blobs. The index in the Boolean array also represents the final loopSum value and the indexed value will show if a blob of points has been found and further processing needs to be made to determine if an actual Ring was formed. Blobs are only checked when the loopSum result returns a value that shows a cluster of points or Blob of points that could potentially lead to a connection of a Ring, hence the use of the same variable when checking the Blob Boolean array.
The following figures show the algorithm that is used to check for Blobs.

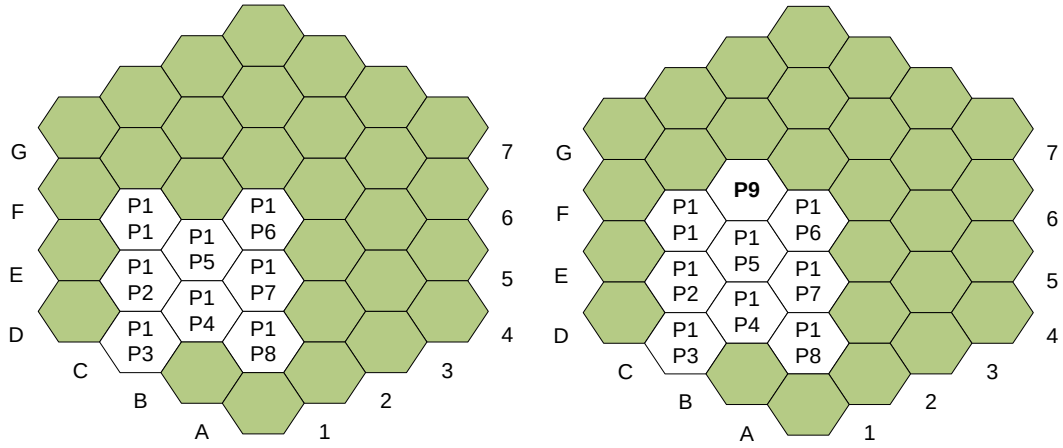

Figure 3.17 – Blobs

In the above figure, point P9 is played into position E4. We can clearly see that this move has created a Ring surrounding point P5 in position D3. Starting at point P1, the Simple Ring algorithm is started as the following figure shows.

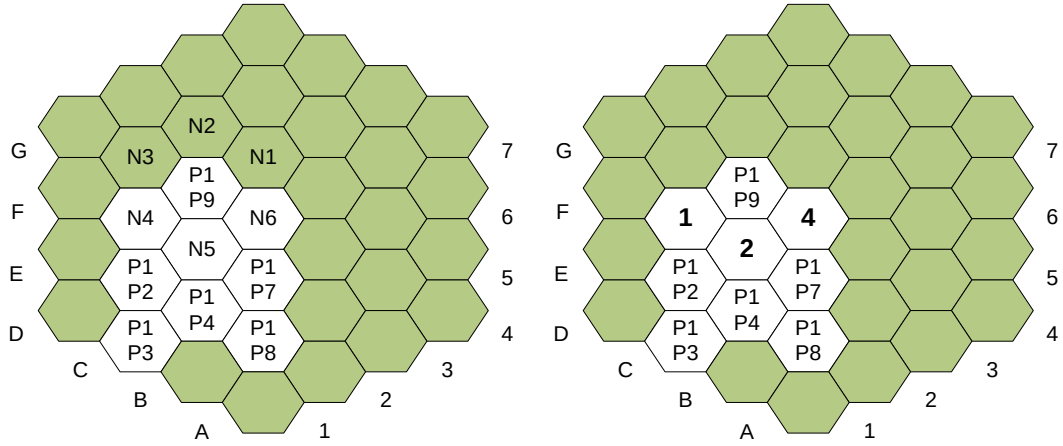

Figure 3.18 – Blob Connections

Variable loopSum is initialized to 1 with neighbor N4 in position E3. However, this time, the final loopSum result yields a value of 0b0111. This value shows that point P9 has connected several points belonging to the same structure with P1as the root point, however this may or may not result in a Ring. If the Blob Boolean array at index loopSum returns true it means that there is a possibility a blob of points has been found.

Then the blob algorithm is run to check that the potential connection of points forms an actual Ring.

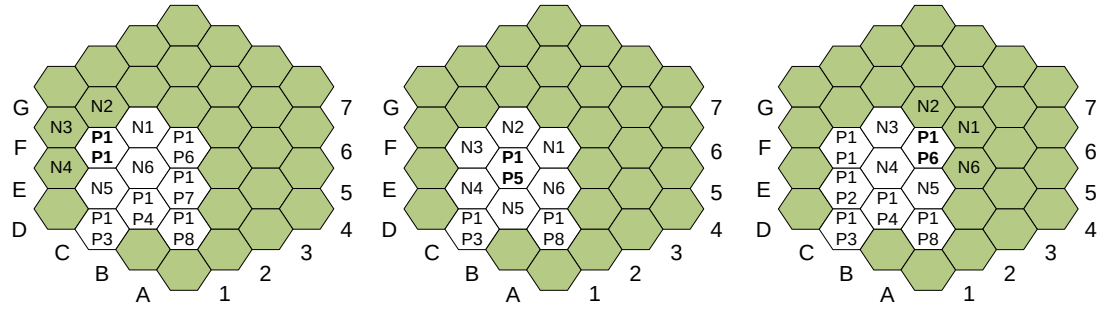

Figure 3.19 – Blobs Processing

As noted above, the three different figures show the processing of Blobs. The neighbors of the same color as the current played point into position E3 are looked at. The neighbors of each neighbor of the same color as E3 are checked to see if all surrounding points are of the same color as E3. If this is the case, as it is in the second part of *Figure 3.19 – Blobs Processing*, an actual Ring has been found when neighbors of P5 in position D3 are all of the same color.

3.4 Monte Carlo Tree Search with UCT Implementation

Section 1.2 gives an overview of the basic Monte Carlo algorithm and also introduces the concept of UCT in the MC tree to improve the selection process in obtaining better moves. Wanderer implements the MC Tree Search with UCT by reusing the class Point. Points in the context of the MC Tree are referred to as nodes moving forward. *Chapter 5 Future Considerations* describes an alternative to reusing this Point class in order to make Wanderer memory efficient as the Point class contains more variables than are needed to build the tree.

The members of Point used to implement the MC tree within Wanderer are as follows:

- wins – keeps track of the number of wins out of this node.

- visits – keeps track of the number of times the node has been selected either randomly or by calculating the best UCT value.
- child – a reference to a child node. Only a single reference is used to have access to all children of a node
- sibling – a reference to another node used to form children of a parent node.

Figure 1.3 – Basic MCTS Algorithm and *Figure 1.4 – MCTS with UCT Variation* shows each node with multiple children, with a reference downwards to each child node. However, in Wanderer, a node has multiple children by pointing to a single child node, and then this child node will reference sibling nodes. In the building of the MC Tree, child and sibling references are used to point downwards in the Tree and the propagating of the statistics in the nodes going up the tree is done recursively and therefore a parent reference from a child node is not needed. It is by adding sibling references that more children are added to a parent node. The following figure illustrates how Wanderer adds children to a parent node when building the Monte Carlo tree.

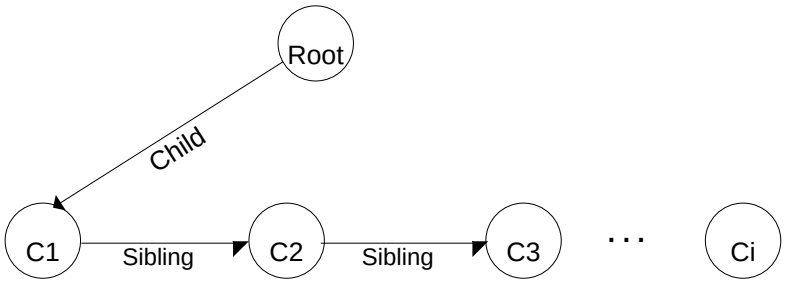

Figure 3.20 – Wanderer Implementation of the MC Tree

The above was done in order to simplify the algorithm and also to make the building of the tree more efficient since each node will at most have only two references, a child and/or a sibling instead of a node having multiple references for children. See below.

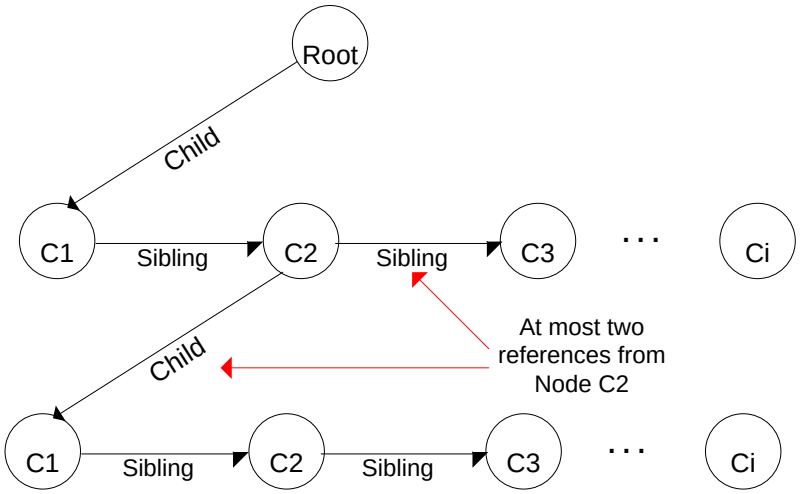

Figure 3.21 – MC Tree Child and Sibling References

Wanderer's building of the Monte Carlo Tree starts when it is its turn to make the next move on the Havannah board. The following explains the building of the Tree using the UCT variation assuming the game still has plenty of moves before it is completed. A few improvements are considered when generating the MC tree, which are discussed in *Chapter5 Future Considerations*. As a first step, the root of the tree is created and set to the opposite color followed by the creation of children matching the color of the current move for Wanderer to play. For memory efficiency and speed in building the tree, nodes are taken from a list of already created Point objects, called availableNodes. If the availableNodes list is empty, then a new Point object is created and added to the tree. After Wanderer decides which move to make, all the nodes in the MC tree are returned to the availableNodes list making it possible for Wanderer to reuse them when it is its turn to play again.

So generating the first few moves in the beginning of the game will create the first few Point objects but later on as the game progresses, nodes will be pulled from the availableNodes list saving time in creating Point objects every time the MC tree needs to

be expanded (creating of children), considering that every move, including the initial moves in the game are generated using the MC UCT algorithm.

The children of the root node, or any node for that matter, are all the possible available moves at the current state of the game. The following shows the initial MC Tree created for the current state of the board size 4, shown below.

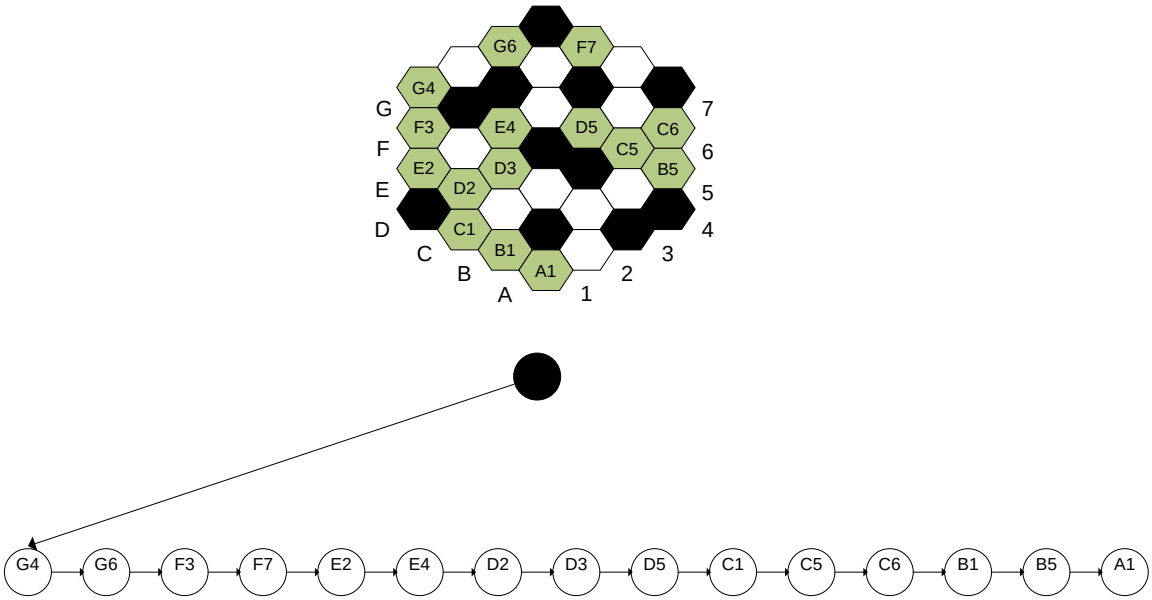

Figure 3.22 – MC Tree – Root with Available Moves as Children

As noted above, the children of the root are all available moves that haven't been played yet by either color. Instead of traversing the entire board for empty cells every time children need to be created, Wanderer maintains a list of available moves to play which are actual references to Points on the board. This makes it fast and efficient to create children since a search for available moves is not performed. The order in which Point references are added to this list is sequential and is done during initialization. As mentioned before, in *Section 1.2 – Monte Carlo (MC) Algorithm*, many simulations to the end of the game will be performed before Wanderer makes its move. These simulations are performed on the current board state and because of this, when a simulation is started,

the board state and the list of available moves are backed up and then later restored once the simulation is complete. This is done so that every simulation is started with the same state of the current board.

After children for the root node have been created and the board and list of available moves have been backed up, a branch of the tree, one of the root's children has to be chosen to start the simulation from. For this, the UCT formula is applied to all children of the root and the child with the best UCT value will be selected to continue with the simulation. The formula shown in *Section 1.2 Monte Carlo (MC) Algorithm* can only be applied to nodes that have been UCT selected, or visited, at least once before because the formula divides by the node's number of visits and this has to be a non-zero value. Some implementations of Monte Carlo with UCT do not apply the UCT formula until the node has been visited T times [13] (See *Chapter 4 Testing Wanderer* for exploration of this parameter). At this level of the tree, the visits variable for all children is zero, therefore one of the child nodes will be selected randomly. This random selection was touched on in *Section 1.2 Monte Carlo Algorithm* and Wanderer implements it as follows. Since Wanderer does not have knowledge that the children haven't been visited before it will traverse each child and if the number of visits is greater than T, then it will calculate the UCT value for the node. For children with visits less than T, Wanderer calculates a random UCT value using a random number generator. This random value given is high as compared to the value calculated using the UCT formula to force Wanderer to visit each child at least once. Note that the purpose of the UCT formula is to have a balance between exploitation and exploration. In order to obtain the optimal move, Wanderer needs to select the best child node based on previously collected knowledge, exploitation,

but at the same time, Wanderer needs to explore sub-optimal moves to obtain more knowledge about them (exploration)[15]. So in order for the UCT formula to work there needs to be knowledge about every possible move available to Wanderer and therefore a reason for setting a high random value. After all children have been checked, Wanderer then selects the child with the highest UCT value, updating the number of visits of the selected node. If the selected child is considered a leaf node because it has no children then a random simulation to the end of the game is performed. This child node will be updated with the result of the random simulation, incrementing the node's wins variable by 1 if the simulation results in a win and doing nothing if the result is not in favor of the child node. The following figure represents the initial build of the tree using the UCT best path selection algorithm.

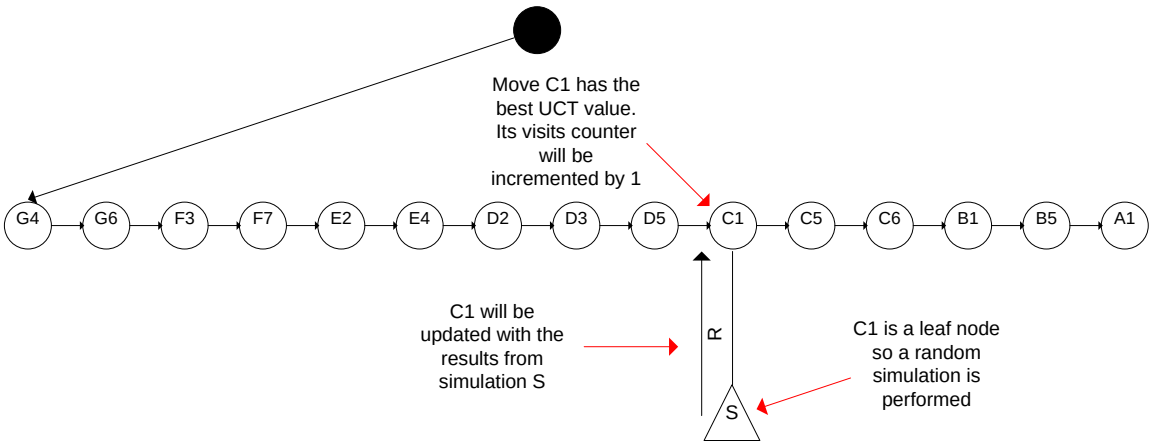

Figure 3.23 – MC Tree UCT Selection

Random simulations are played out on a leaf node at most V times. This V constant value determines when the Monte Carlo Tree is expanded. *Chapter 4 Testing Wanderer* shows different values of V and how varying this constant changes how the MCTS UCT behaves. The concept of expansion means creating children for a leaf node after the parent move has been selected. It is important to note that the children at the next level in the tree are of the opposite color of the current player, in our case Black. In the figure

below, the tree shows a node ready to be expanded after it has been visited V times.

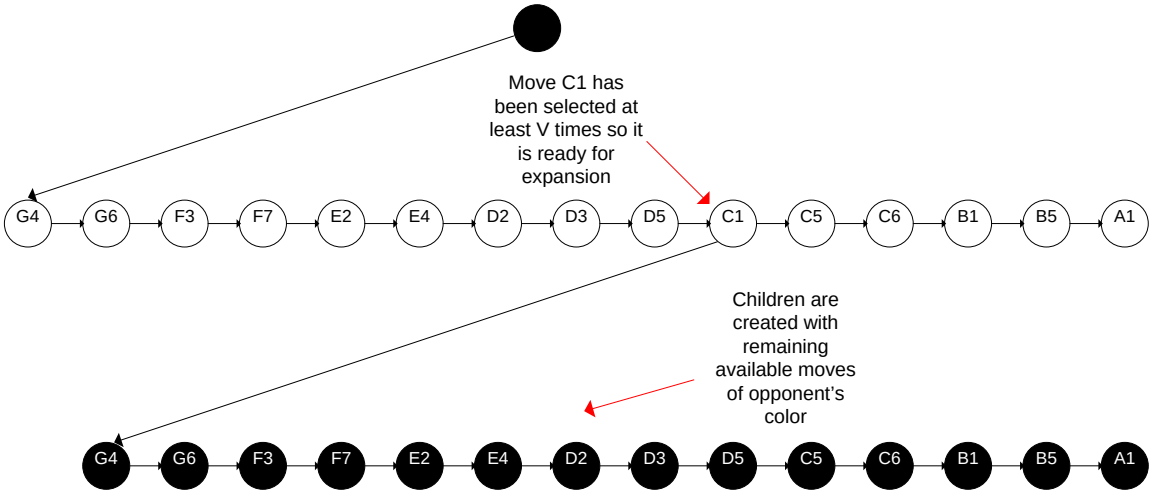

Figure 3.24 – MC Tree Expansion

After expansion is completed and since no child has been visited yet, a child node is randomly selected for a random simulation in the same way C1 was selected.

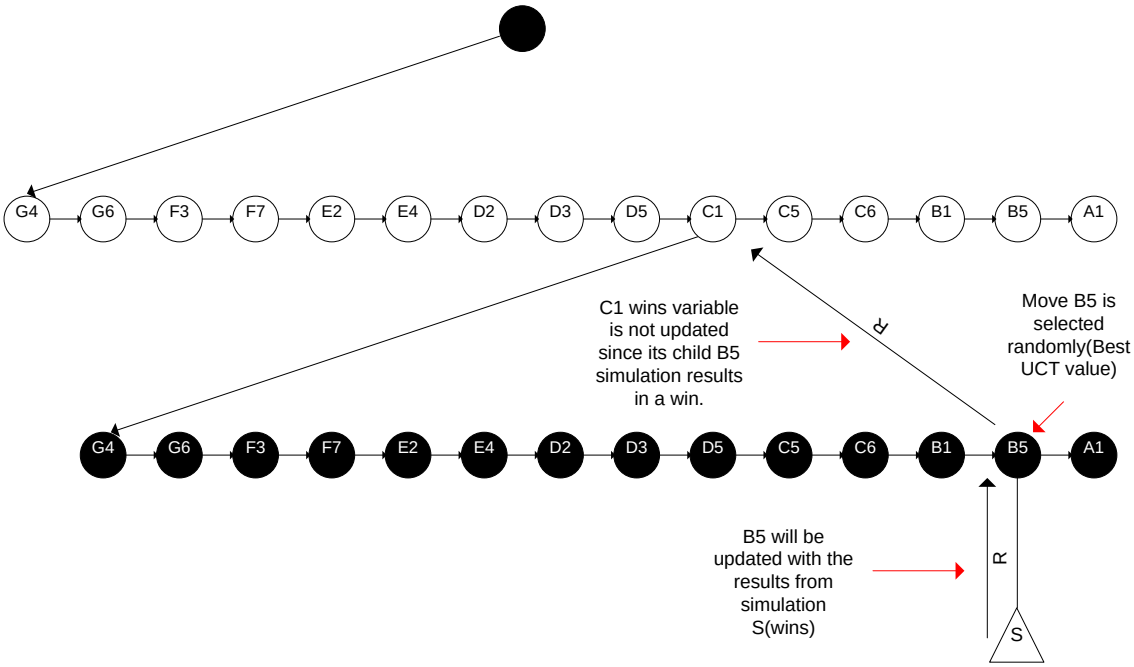

Figure 3.25 – MC Tree Results Propagation

As seen above, when move B5 is selected, it results in a win for Black after the random simulation is finished. The node B5 is updated with the number of wins being

incremented by 1. The number of visits was also updated at the moment it was chosen. This result will be propagated up the tree to the children of the root, however, the node C1 wins variable does not increment because its opponent on the lower level won the simulation resulting in a losing game for C1.

So as the tree develops and nodes are expanded, a path will be chosen by executing the UCT algorithm on the children of a given node. In the beginning, as noted, the path is chosen randomly, but as the tree grows and nodes are visited, the actual UCT formula will be applied. It is the combination of random moves and UCT selected moves what makes it possible to search for more important moves more often than bad moves [10]. The figure below shows the tree expanded to different levels and shows how the node's statistics are updated once a leaf node is randomly simulated, just like the previous figure.

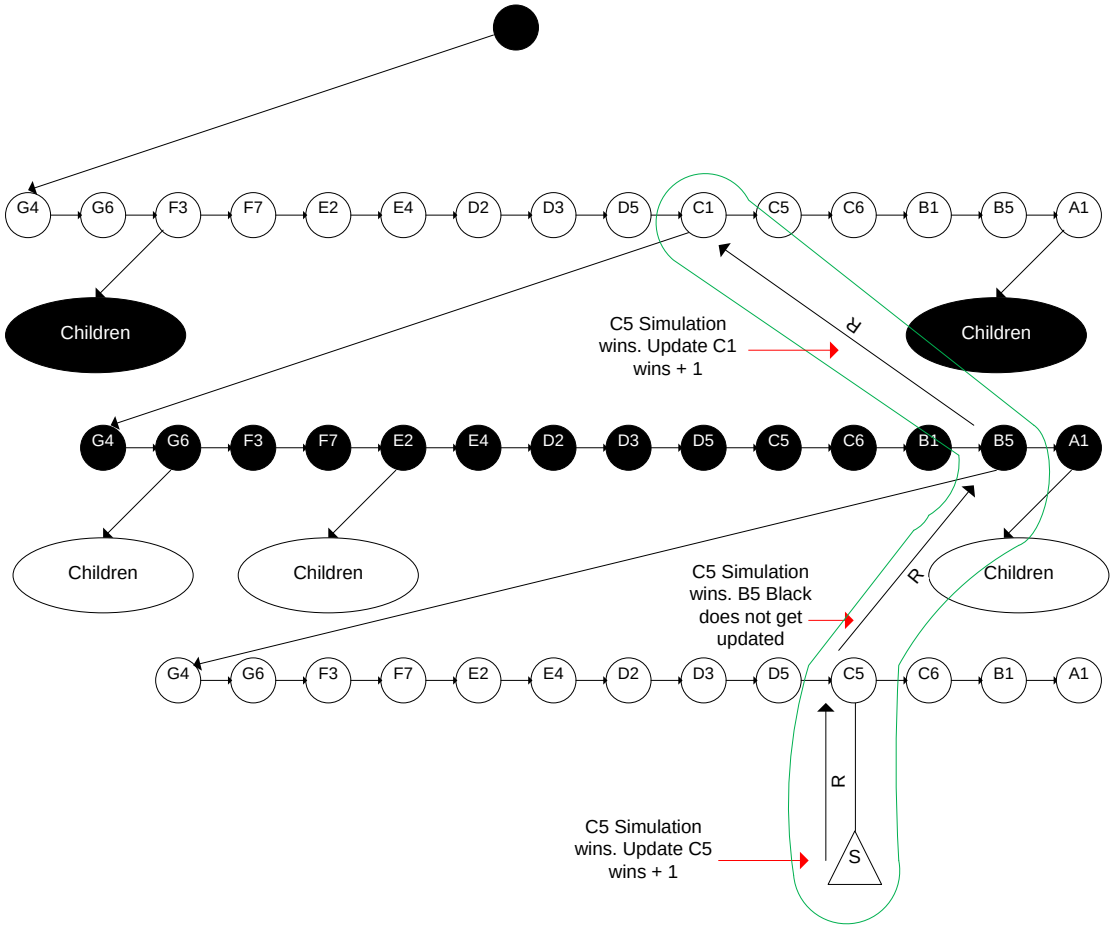

Figure 3.26 – MC Tree Expanded

The MC Tree will be generated and randomly played until the allowed time per move expires or the allowed memory has been exhausted. The longer the allowed time per move, the better the selection the MCTS algorithm yields because more random simulations are performed and the selection tree is bigger which yields more statistical data for Wanderer to base its next move on. Finally, the move that Wanderer selects is one of the child nodes of the root of the Monte Carlo Tree. This selection varies in different implementations of the Monte Carlo Tree Search with UCT, as is the case with some Computer GO programs [13]. There are four possible selections, (1) Max Child, where the node with the highest win rate is selected, (2) Robust Child, where the node with the highest number of visits is selected, (3) Robust-Max Child, where the node with the

highest win rate and number of visits is selected and (4) Secure Child, the node that maximizes the quantity ($w + A/\sqrt{n}$), where w is the win rate of the node, A is a constant parameter and n is the number of visits of the node. Wanderer implements the selection of the Robust Child; however, *Chapter 4 Testing Wanderer* tests the different variations of child selection and shows some results.

Chapter 4 Testing Wanderer also shows the power of the MCTS UCT Algorithm by giving statistics of two different versions of Wanderer, one implementing straight Monte Carlo and another version implementing Monte Carlo Search Tree with UCT. The same section will also show how different constant values used in the UCT formula and Visits before expanding play a role in Wanderer's move selection.

Chapter 4

Testing Wanderer

This section discusses how Wanderer was tested and shows statistics of how it performs against a version of Wanderer implementing the basic MC Search algorithm previously mentioned in *Section 1.2 Monte Carlo (MC) Algorithm.* For our purposes, this version of Wanderer is referred to as WandererMC and the version described in this paper referred to as WandererUCT. The statistics shown will be based on several hundred games played on different board sizes. This section will also cover the different variation points described in this paper such as the constant K used in the UCT formula, constant V which determines how many random simulations are performed before expanding the Monte Carlo UCT Tree and constant T which is used to decide how many times a node is given a random value before the UCT formula is applied. Two more variations are also tested which deals with the effect of the result returned by a random simulation when a tie is found and the final move made by WandererUCT, returning the child with the highest win rate as opposed to returning the child with the most visits. These variations are tested against the same version of WandererUCT with its parameters updated. Wanderer implements a communication protocol called GTP which is used in other board games such as Hex and GO. GTP stands for GO Text Protocol and it was mainly created for computer Go programs to be used during tournaments as well as for testing. This protocol dictates a range of commands Wanderer responds to and it is this protocol that makes it easier to test Wanderer. This protocol is also a standard in the Computer Olympiads which is the main reason for implementing it with Wanderer as it is here where Wanderer's playing strength is really tested against other Havannah playing engines.

The following is a subset of the GTP commands that Wanderer implements (for more details regarding the use of each command, refer to [11]):

- protocol_version – the version of the GTP protocol Wanderer implements. Current version of the protocol implement is version 2.
- name – the name of the engine, Wanderer.
- version – the version of Wanderer, version 1 as discussed in this paper.
- list_commands – the list of GTP commands supported by Wanderer.
- quit – terminates the session with Wanderer and the connection is closed.
- boardsize – changes the size of the board Wanderer plays on.
- clear_board – resets the state of the board to the initial configuration.
- play – sets a move on the board. Wanderer receives this command to make the move on the board by the opponent player.
- genmove – this command is used to tell Wanderer to generate the next move.

During initial implementation, the GTP protocol was used with the open source program called HGUI discussed in the littlegolem.com forums website [6]. This is a graphical interface that allows a human player to play against Wanderer. It also allows two different versions of Wanderer to play against each other.

For the results obtained in the remainder of this section, a shell script was created to have two different versions of Wanderer play each other. The shell script sets up the time each version has before a move has to be presented, the number of games to play and it keeps track of the number of wins each version has.

The following table is aimed to show the playing strength of WandererMC against a novice human player, the author of this thesis. It also shows the first move advantage a player in Havannah has in a small board. The layout of this table is the progression of multiple games played varying the time allowed for WandererMC to make a move to see how the playing strength is affected. Note: (w) represents which player is first.

Game #	Board Size	Human	WandererMC	ttt	Ties are wins on random simulations
1-6	4	6(w)	0	30s	No
7-14	4	0	7(w)	30s	No
15	4	0	0(w)	30s	No
16-17	4	2(w)	0	60s	No
18	4	0	0(w)	60s	No
19	4	1(w)	0	300s	No
21-22	4	0	2(w)	120s	Yes
23	4	1(w)	0	120	Yes

Table 4.1 – Human Player v. WandererMC

In the table above, games 1-6 show the first player has an advantage and with simple strategy it was easy to beat WandererMC. Games 7-14 show WandererMC has the upper hand when going first. However, with basic strategy, tie games were achieved as noted in game 15. Increasing the time for WandererMC has no effect in the results and its playing strength against a novice player is minimal. Games 21-23 show the difference in WandererMC playing strength when random simulations result in a win for tie games. WandereMC still loses to basic strategy when Human goes first but it is now impossible

for a novice player to tie the game. Through some testing, WandererUCT is also vulnerable to the same strategy on a size 4 board and it loses every time it plays second. However, WandererUCT's playing strength is noticed when playing first as a novice player is not able to tie the game. Tests for human v. WandererUCT were only done on a small board as obtaining results for bigger boards would be time consuming. Therefore, the following tables show statistics of two different versions of Wanderer playing against each other.

The following table shows the results of testing WandererUCT against WandererMC when having WandererMC go first and WandererMC has more ttt time on different board sizes. Also, WandererUCT returns the child with the highest win rate as its next optimal move.

Board size	WMC	ttt(s)	Wins on ties?	WUCT	ttt(s)	K	V	T	Wins on ties?
4	12%	60	No	88%	30	.5	25	1	Yes
5	16%	60	No	84%	30	.5	25	1	Yes
6	20%	30	No	80%	15	.5	25	1	Yes
7	38%	30	No	62%	15	.5	25	1	Yes
9	32%	20	No	68%	10	.5	25	1	Yes

Table 4.2 – WandererMC as First Player

The following table shows the results of testing WandererUCT against WandererMC when having WandererUCT go first and WandererMC still has more ttt time on different board sizes. Also, WandererUCT returns the child with the highest win rate as its next optimal move.

Board size	WMC	ttt(s)	Wins on ties?	WUCT	ttt(s)	K	V	T	Wins on ties?
5	5%	20	No	95%	10	.5	25	1	Yes
6	11%	20	No	89%	10	.5	25	1	Yes
7	28%	20	No	72%	10	.5	25	1	Yes
9	26%	20	No	74%	10	.5	25	1	Yes

Table 4.3 – WandererUCT as First Player

Table 4.2 and *Table 4.3* both show that Wanderer implemented with the UCT algorithm is a much stronger player than using a pure Monte Carlo search. It is also noted on the above tables how changing WandererUCT to go first affects its winning rate by increasing it and therefore showing the impact of the first move advantage.

The following tables explore the variations used in WandererUCT so far as described in the above tables when playing against WandererMC. The goal is to see the effect the parameters have in Wanderer's playing strength. As a final test, the tuned ~~up~~ version of WandererUCT will be used to test it against WandererMC once again to see the effects on its winning rate. For this purpose, all variations will be tested against the version that was tested in the previous tables (WandererUCT tested against WandererMC), and this version is referred to in the following tables as WandererBase while the version in which variations are made is referred to as WandererUCT. All tests from this point forward are tested with 10 seconds per move and only boards with size 5 and 7 are tested. WandererBase is also set to return the child with the best win rate (calculated by child.wins/child.visits).

Board size	WBase	WUCT	K	V	T	Wins on ties?	Best Child?
5	82%	18%	.25	1	1	Yes	Highest Winrate
7	59%	41%	.25	1	1	Yes	Highest Winrate

Table 4.4 – WandererBase as First Player

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	53%	44%	3	.25	1	1	Yes	Highest Winrate
7	45%	55%	0	.25	1	1	Yes	Highest Winrate

Table 4.5 – WandererBase as Second Player

Both tables above show the effect of setting V to 1. This means that the tree will be expanded after the node selected has been randomly simulated only once. However, as noted in *Table 4.4*, WandererUCT is no match for WandererBase on the smaller board size 5; however, it does play a bit better on board size 7 but still not able to beat WandererBase. In *Table 4.5*, even when WandererUCT plays first, as noted, on the smaller board it does not play better. On the bigger size board it is able to beat WandererBase but not by much. So the above parameters are do not make Wanderer a stronger player than the baseline.

The next tables show how varying the V constant a bit and also changing the returned child move to the child with the most visits affects Wanderer's playing strength. For the next tests, the advantage is given to WandererBase by having it play first since this is the version used as a gauge to improve WandererUCT.

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	74%	25%	1	.25	3	1	Yes	Most Visits
7	71%	29%	0	.25	3	1	Yes	Most Visits

Table 4.6 – WandererUCT with Most Visits

Surprisingly, it seems as if selecting the child with the most visits as the move to play has a big impact on how Wanderer plays. It is surprising because most implementations of the algorithm tend to use the child with the most visits as the optimal next move. To discard the possibility that returning the child with the most visits is the variation lowering WandererUCT's winning percentage, the following tables are set so WandererBase and WandererUCT have the same variations except for returning the child with the highest win rate over the child with the most visits.

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	60%	39%	1	.5	25	1	Yes	Most Visits
7	60%	40%	0	.5	25	1	Yes	Most Visits

Table 4.7 – WandererBase as First Player –Most Visits

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	44%	56%	0	.5	25	1	Yes	Most Visits
7	49%	51%	0	.5	25	1	Yes	Most Visits

Table 4.8 – WandererUCT as First Player – Most Visits

Based on *Table 4.7* and *Table 4.8*, it is concluded that choosing the child with the highest win rate makes Wanderer a better player than choosing the child with the highest number of visits as the winning percentage for WandererBase is higher than WandererUCT.

The following two tables explore the effect on Wanderer by increasing the V constant to 50 and then to 100 which means nodes will be randomly simulated quite a few times before expanding. WandererBase will play first and the goal is to decrease WandererBase winning percentage.

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	62%	37%	1	.5	50	1	Yes	Highest Winrate
7	59%	41%	0	.5	50	1	Yes	Highest Winrate

Table 4.9 – WandererUCT V as 50

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	61%	39%	0	.5	100	1	Yes	Highest Winrate
7	64%	36%	0	.5	100	1	Yes	Highest Winrate

Table 4.10 – WandererUCT V as 100

Increasing the V constant to a high number does not show improvement as far as playing better when going second. However, the winning percentages are fairly close on board size 5. But it is a bit higher in the bigger board.

Let's try changing V, T and K in the following two tables and see the effect of these in WandererUCT. Most importantly, the effect of changing T is tested below.

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	75%	25%	0	.5	5	5	Yes	Highest Winrate
7	72%	28%	0	.5	5	5	Yes	Highest Winrate

Table 4.11 – WandererUCT T as 5

The setting of T to 5 means delaying the use of the UCT formula 5 times. We are increasing the number of times nodes are selected randomly and it is noted that this

brings down the winning rate of WandererUCT. This is a good indication that the using the actual UCT formula early on over random selection is preferred.

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	90%	9%	1	.2	1	1	Yes	Highest Winrate
7	82%	18%	0	.2	1	1	Yes	Highest Winrate

Table 4.12 – WandererUCT K decreased

Table 4.12 above, shows the effect of decreasing the exploration constant. It negatively impacts the playing strength of WandererUCT and this is more noticeable in the smaller board. The next table explores parameters one at a time from the baseline to closely monitor WandererUCT's playing strength.

Board size	WBase	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	79%	21%	0	.8	25	1	Yes	Highest Winrate
7	55%	45%	0	.8	25	1	Yes	Highest Winrate
5	63%	36%	1	.5	15	1	Yes	Highest Winrate
7	55%	45%	0	.5	15	1	Yes	Highest Winrate
5	90%	10%	0	1.2	5	1	Yes	Highest Winrate
7	82%	18%	0	1.2	5	1	Yes	Highest Winrate
5	61%	38%	1	.5	5	1	Yes	Highest Winrate
7	48%	52%	0	.5	5	1	Yes	Highest Winrate

Table 4.13 – WandererUCT Improved

The above table, *Table 4.13*, shows an improvement over the WandererBase in its last row. The following table shows the results of playing this new baseline against WandererMC on board size 5 and 7 and this time WandererUCT goes first.

Board size	WMC	WUCT	Draws	K	V	T	Wins on ties?	Best Child?
5	5%	95%	0	.5	5	1	Yes	Highest Winrate
7	19%	81%	0	.5	5	1	Yes	Highest Winrate

Table 4.14 – New Baseline v. WandererMC

Table 4.14 shows the winning percentage on board size 5 remains the same; however, there is a 9% increase in the winning rate for WandererUCT on the bigger board. The above test has shown that with tuning of Wanderer's parameters a stronger player can be achieved.

Chapter 5

Future Considerations

As mentioned in previous sections, this paper was based on the first iteration of Wanderer. Since it was first created, Wanderer has implemented different improvements over the past few years [18]. Some of these improvements are discussed here in this section since these have been implemented by other people different than me. In addition, future work is also discussed. At the end of this section, a table of results is shown for the current version of Wanderer as of today against the initial version of Wanderer explained in this paper.

The following is a list of improvements that have already been implemented in Wanderer:

- Mate in One
- MCTS Solver
- Only One Move, Make it
- Save MC Tree for Next Move
- Improved Simulation Strategies
- Multithreading

The following are considerations for a future implementation of Wanderer:

- Weighted PCUF Algorithm
- MCTS Node Building Efficiency

Furthermore, [19] discusses improvements done on another computer program that plays Havannah. It would be interesting to apply some of these improvements to Wanderer and test their capabilities to see if any or all add strength to the level of play Wanderer has achieved.

5.1 Mate in One

This improvement is mostly designed to have Wanderer respond quicker when there exists a single move out of all the available moves that can win the game. This is especially useful when testing Wanderer with a few hundred games. If Wanderer is given 5 minutes to make a move, but the winning move can be checked right away, then Wanderer will not create the MC Tree but just make the move. The example below illustrates this scenario.

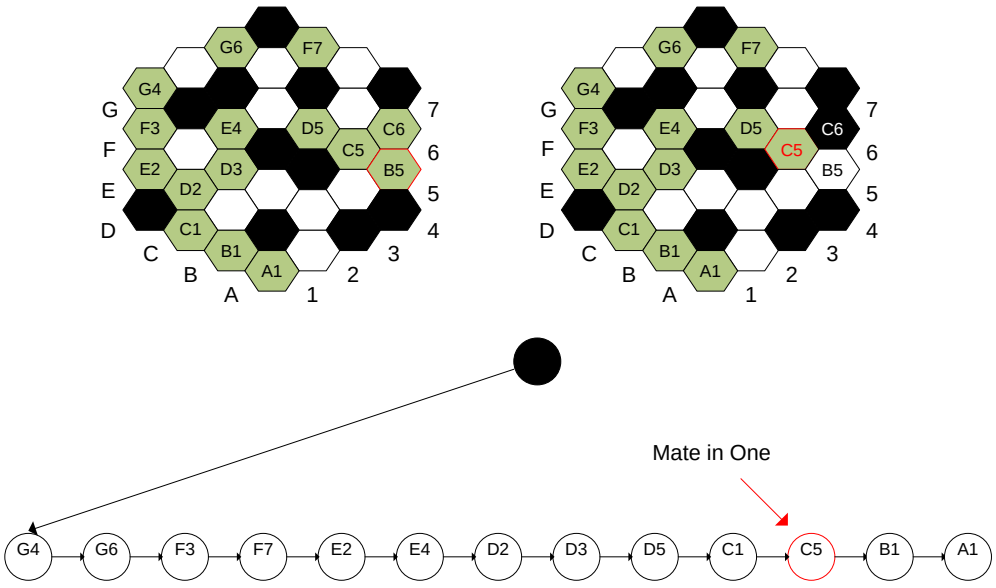

Figure 5.1 – Mate in One

In the figure above, in the first Havannah board, either move B5, C5 or move C6 sets up Wanderer to win the game in the next move (Wanderer plays White). With either move, no matter what Black selects, Wanderer wins the game. In this case, move B5 is selected by Wanderer followed by Black's move into C6. Now that it is Wanderer's turn to make a move, a partial tree will be created starting with the root node and all the available moves as children, as depicted in the third part of the figure. Each child of the root will be checked to see if any makes a winning structure. If it does, then it will be selected as the next move to be played by Wanderer and there is no need to keep building up the tree.

5.2 MCTS Solver

As discussed before, Wanderer's knowledge which will be used for node selection in later iterations of the algorithm is built from performing many random simulations that originate from leaf nodes. One of Wanderer's improvements is the implementation of a modified MCTS algorithm where three of the four steps described in *Section 1.2* have been modified to implement a Solver [16]. The idea of a Solver is to set nodes as terminal nodes in the MCTS tree so that when these are reached during the algorithm iteration no further random simulations need to be performed since the value of the node is already determined. [16] explores the idea that nodes can be set to either $+\infty$ for nodes that yield a won terminal position and $-\infty$ for nodes with a lost terminal position and then these values are propagated up the tree. During the selection phase, when these nodes are selected, the simulation can end at the terminal node and time that would have been spent performing the random simulation is saved.

In Wanderer's implementation of the MCTS Solver variation, each node in the Tree is set to a status of None, Win, Loss or Draw [18]. These values are set in the following manner. As discussed before, children of a node are all the available remaining moves belonging to the opponent after the node move has been played on the board. For the MCTS Solver variation, when a leaf node has been reached and during the node creation, the algorithm checks if every available child loses on the next immediate move. If it does, then this child is not added to the list of children of the node. See below for a graphical representation of this scenario.

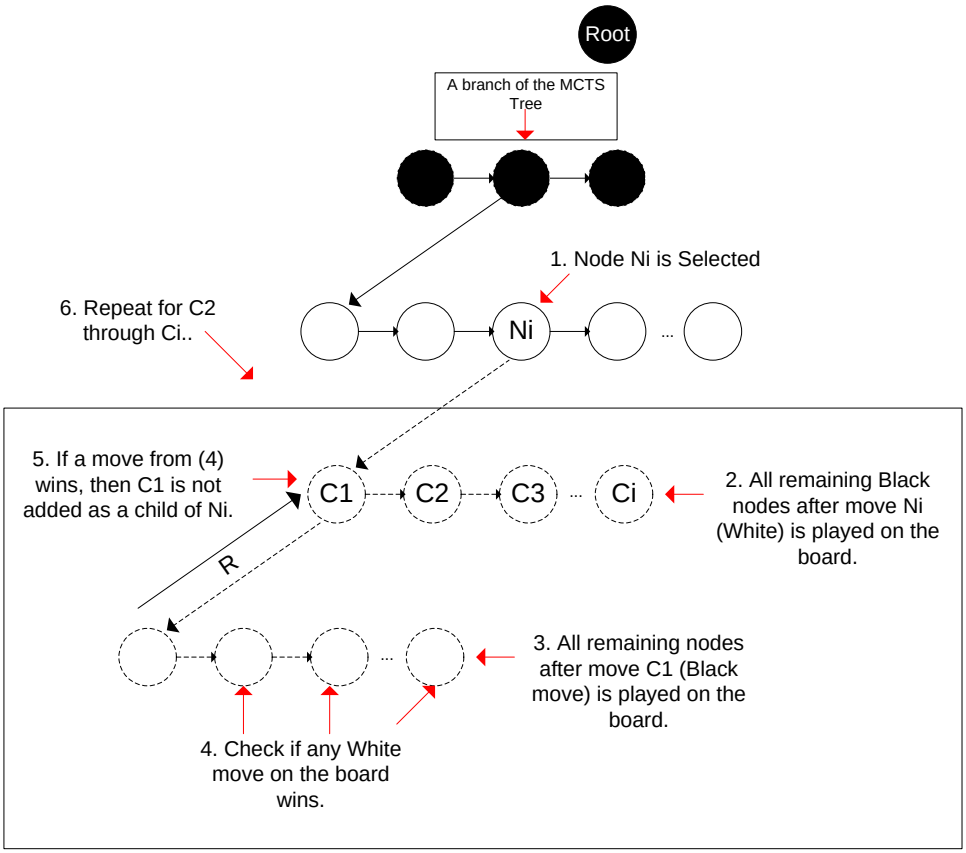

Figure 5.2 – Children Creation From Leaf Node

In the above figure, if step 6 is completed and no child is added as children of node Ni, then this node's status can be set to Win since every child is a losing move. Because this child status is a Win, its parent node is set to a Loss. Also, if after a leaf node has been reached and there are no more remaining moves, then the status of the node is set to Draw. This status is used during the Selection step of the MCTS where nodes are also updated. When it is time to determine which node to follow down the tree, all children are checked to see if all of their statuses are set to Loss or Draw. If the statuses of all children are set to Loss, then the parent node's status can be set to Win. If the statuses are a combination of Loss and Draw statuses then the parent node's status is set as a Draw. Statuses are then propagated up the tree and if the status of the Root node is determined, then the corresponding move can be played and there is no need to wait for the

time/memory allowed to expire. The following shows a representation of how nodes are updated with Win and Loss statuses and how these statuses are propagated up the tree. A node's status is set to Win if all children's statuses are set to Loss and set to Loss if at least one child's status is set to Win.

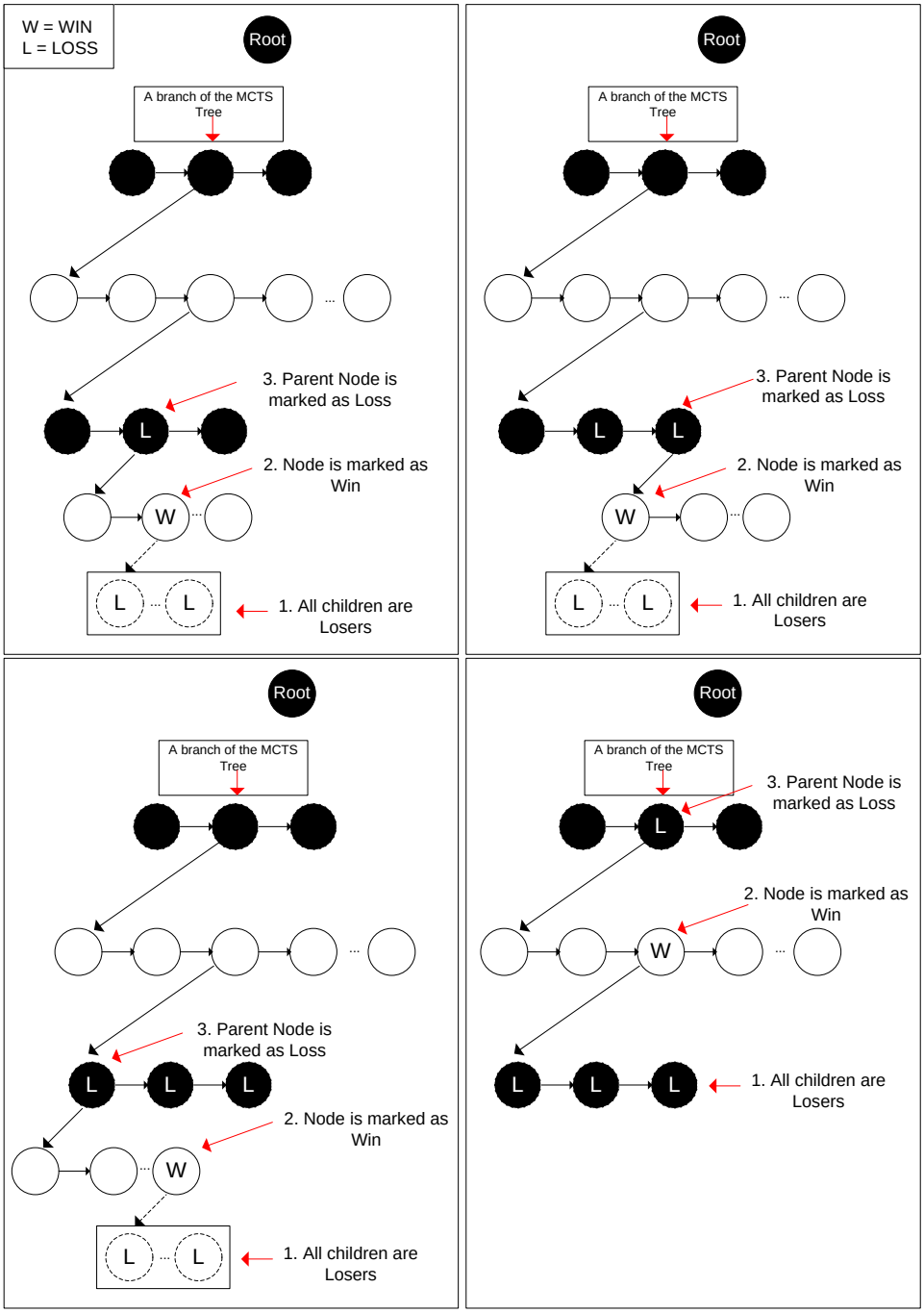

Figure 5.3 – Win and Loss Tree Propagation

5.3 Only One Move, Make it

As discussed in the previous section, *Section 5.2 MCTS Solver*, not all children are added to a node. As we saw, no children added means the node's status is a Win. There are also cases where only one move is added and therefore it is the only move that can be made. The figure below shows a case where only one child is added to a node.

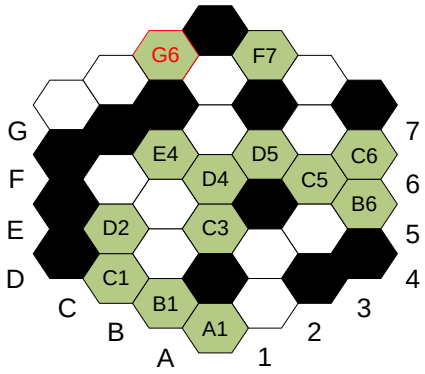

Figure 5.4 – Only One Move, Make it

As seen above, the only possible move for Wanderer to make is position at G6 since every other move results in Black winning the game immediately on the next move. The partial tree created from the above board game results in a node with a single child as every other move results in Wanderer loosing. Wanderer's design with this improvement is that if a node has a single child, then that child is selected as the next move (in this specific example) saving time with a selection process as it is not needed anymore for this situation.

5.4 Save MC Tree for Next Move

It was mentioned before that the more statistical data obtained through simulations Wanderer has to work with, the better the selection for the next move to play is. When it is Wanderer's turn to make a move, it will generate a tree and play random simulations until the time to make the move expires or memory has been exhausted. At this time, one of the children of the root of the tree with the highest win rate will be chosen as the next move. After the move is selected, the entire tree is discarded, and a brand new MCTS tree will have to be created for the next Wanderer's turn. However, this doesn't have to be the case. When a move is finally chosen, the tree can be saved so that it is available for the next move. The part of the tree that is useful for the next move is the branch under the node that is selected by Wanderer, more specifically, the branch that corresponds to the move selected by the opponent player. The root for the new tree for the current move becomes the opponent's move node and then the rest of the tree can be discarded (returned to memory). The following figure shows a representation of this improvement. C1 is the move with the highest win rate selected by Wanderer. The tree is saved until the opponent selects its move to play which in this case is B5. This is the branch that will be useful to keep for the next move and therefore B5 is set as the new root and every node not belonging to the saved branch is returned to memory.

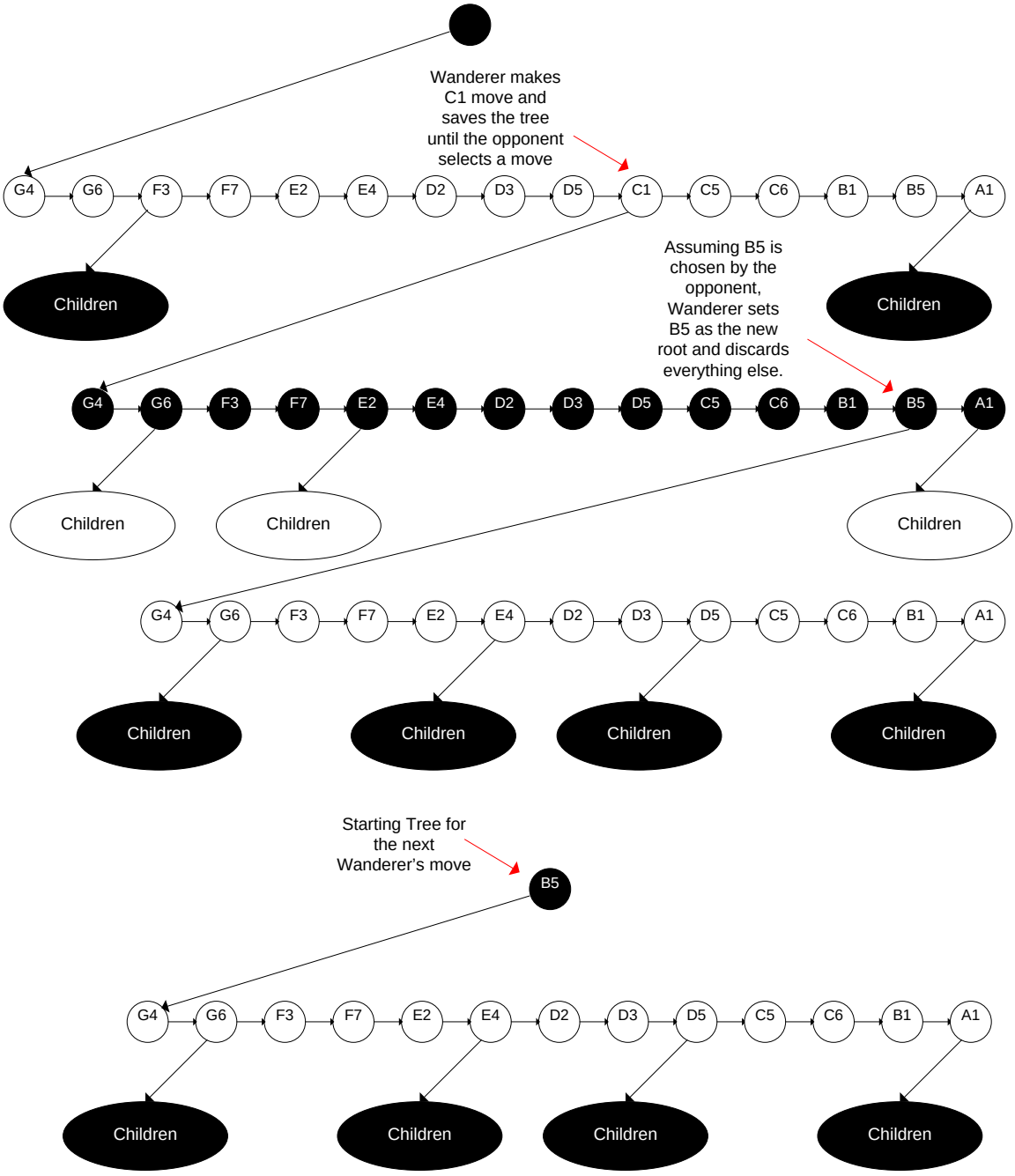

Figure 5.5 – Save Tree for Next Move

5.5 Biasing MCTS Node Selection

The strength of Monte Carlo Tree Search relies on the knowledge it obtains from the random simulations. It is this knowledge combined with UCT node selection that allows us to exploit winning positions on the tree and at the same time exploring other nodes that may lead to winning positions. As the tree is built and new nodes added as children, they are initialized as new nodes since there is no prior knowledge about them. New nodes meaning the win count and the visit count for that node is set to 0. However, a move's quality can be determined by its position relative to its neighbors and therefore it makes sense to guide Wanderer's selection process towards those moves [18]. One way of biasing this selection is to set the win count and visit count of the nodes that seem more promising to something other than 0. Setting the win count and visit count to high values makes the UCT selection to be directed towards these nodes. There are two types of moves described in [18] that on average are better than other moves not in these two categories. These are 'joint' and 'adjacent' moves. Joint moves are those that are two cells away from a stone of the same color and adjacent moves are those that are neighbors to a stone of the same color. In the figure below, White's move F6 is a joint move to stones on positions G5, E4, and D5. A1 is also a joint move to stones in positions C2 and B3. E7 is an example of an adjacent move to stone in position D6.

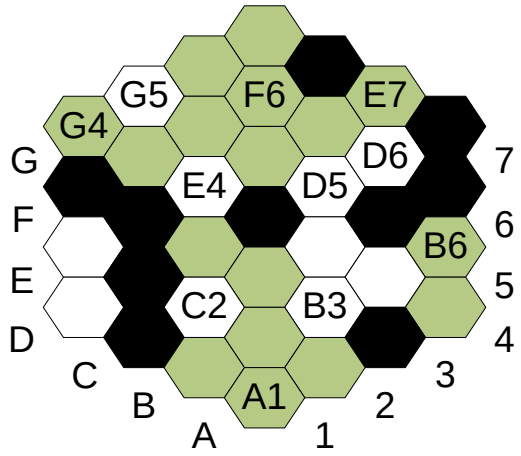

Figure 5.6 – Joints and Adjacent Moves to Same Color Stones

The purpose of the game is to make connections to form one of the winning structures described in *Section 3.3* so it makes sense that making moves that are close to already placed stones of the same color are beneficial. Joint moves, for example, actually are implementing a playing strategy described by the creator of the game, that of frames [19]. Forming frames in Havannah is an excellent strategy for playing the game as it is not required to form full connection from the start of the game but framing the structures is enough to have an advantage over the opponent. The figure below shows an example of a frame that will turn into a fork.

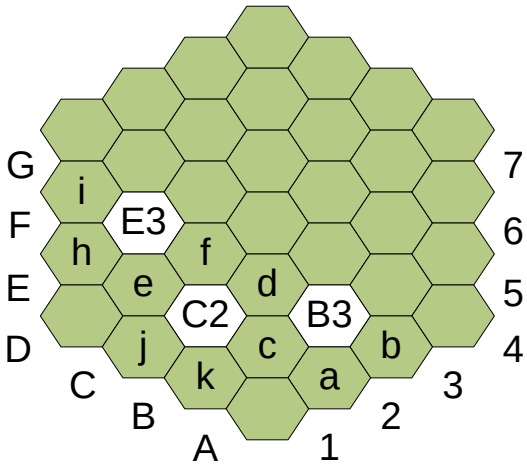

Figure 5.7 – Fork Frame

It can be noted that stones on position E3, C2 and B3 are joint moves of each other and in order to form a Fork structure, White can play in either a or b cells, adjacent moves to B3, to connect B3 to an edge. If Black plays a position then White plays b position and still makes the connection. The same holds true for connecting C2 to B3 by playing either c or d, connection C2 to a different edge by selection j or k, connecting C2 to E3 by playing either e or f cells and finally yielding a Fork structure by connecting E3 to an edge by playing either i or h cells. So we can immediately see the advantage of biasing the selection towards both these type of moves. Through some testing and trial and error this method actually proves to increase the playing strength of Wanderer [18] during the selection phase of the MCTS algorithm.

5.6 Multithreading

The concept of multithreading Wanderer makes sense since the more time that is spent collecting data, the better the selection for moves is [17]. There are different ways to design Wanderer to use threads for processing of the MC Tree. Two of these are discussed in this section, (1) Leaf Parallelization and (2) Root Parallelization. There is one more way discussed in [17], Tree Parallelization, for a Computer GO Program using MCTS; however, the authors found the best results with Root Parallelization and this is what is currently implemented with the new version of Wanderer (as of 2014). Leaf Parallelization seems like a good alternative to Root Parallelization and the purpose of this section is to explore the idea of using it to test against the current version of Wanderer to see if it produces better results. Parallelism at the leaf level is simple and it is implemented in the following way. The concept is to spawn different threads, as many as would be allowed by hardware. It is recommended to run a thread per available core [17], once a leaf node has been UCT selected. Each thread would then run a random simulation

to the end and once all threads are finished, the results would be updated in the node that spawned the threads and then upwards through the tree. This method allows for independent threads to be run without knowing previous explorations and therefore access to shared memory is not required. The node spawning the threads would collect data as each thread is finished. Implementation of this method is simple but it does present two main problems [17]. In order to collect all data, the node spawning the threads has to wait for all threads to finish and the fact that data is not shared presents a problem in itself. The time the node takes to collect all data is the time of the longest running thread. And since data is not shared, if most finished threads already determined the node is not a good one, the node spawning the threads still has to wait for the longest running thread to finish, wasting valuable processing time [17]. Some of these issues with this method may be mitigated with implementation of other improvements mentioned in this paper. Adding the ability to stop threads in the middle of processing based on threads that already finished may minimize the impact of longer running threads. Allowing the sharing of data to determine the state of the threads that are still running does complicate the algorithm, but it allows for knowledge to be known before terminating long running threads. On the other hand, simulations in Havannah are very fast; implementing this method to allow the early termination of threads based on data collected from already finished threads may be more time consuming than waiting for all threads to finish. Data needs to be collected to test this method in both way to determine if the drawbacks outweigh its benefits. Below is a simple representation of what Leaf Parallelization looks like in a tree.

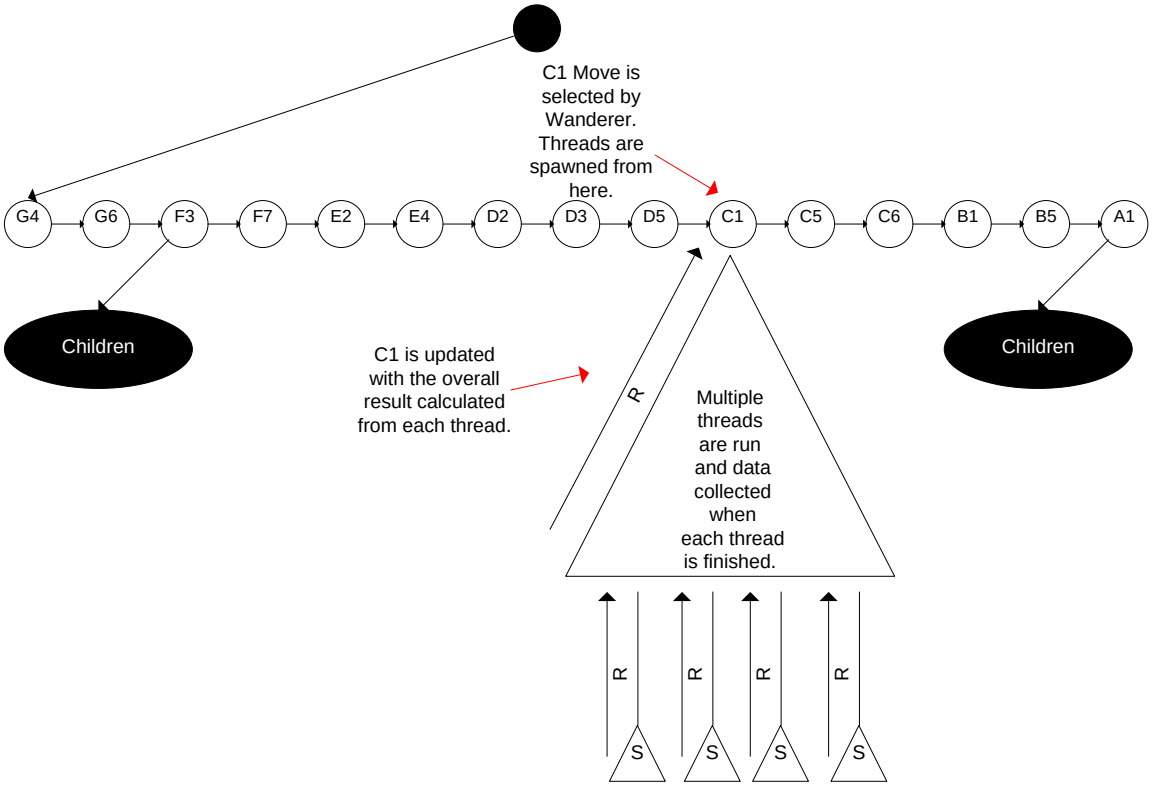

Figure 5.8 – Leaf Parallelization

Root Parallelization, the method implemented by Wanderer, consists of generating an entire MCTS UCT tree per spawned thread [17]. Information does not need to be shared between trees and each thread runs until the time/memory constraint expires. Each tree is a clone copy of each other and when it is time to return the move the information of all children is combined and then the child with the best combined results is selected. The following figure depicts this method.

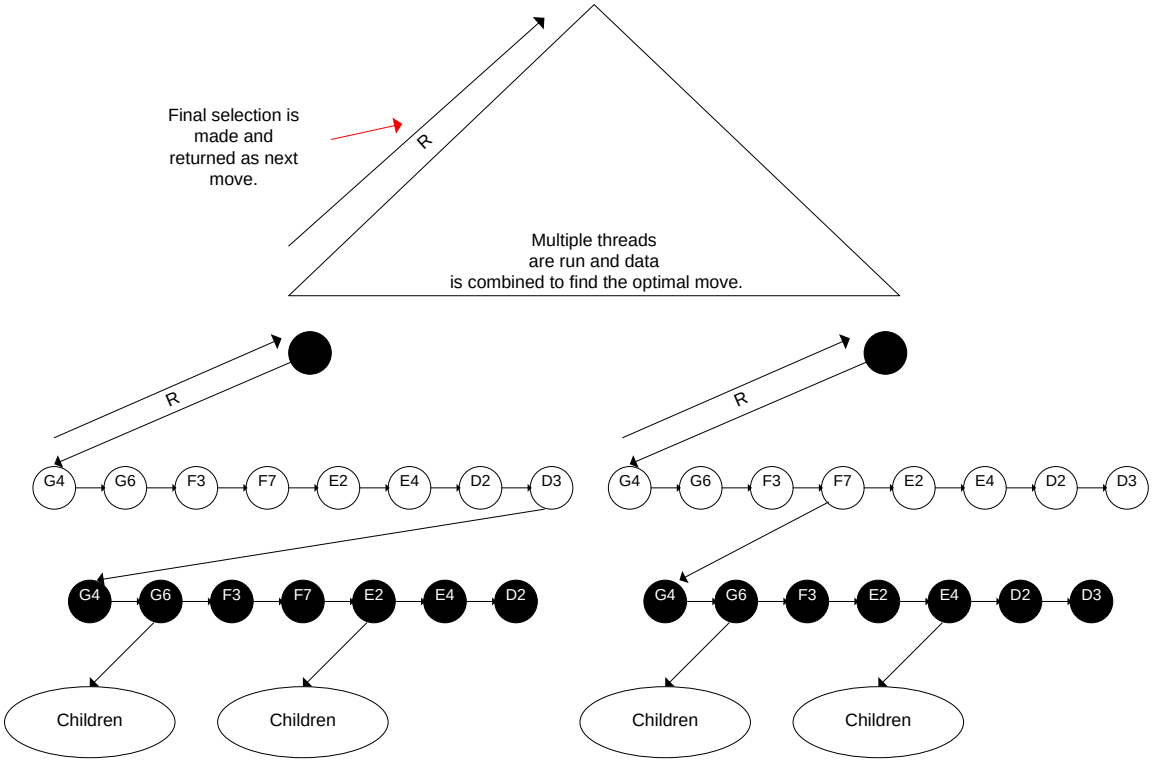

Figure 5.9 – Root Parallelization

5.7 Weighted PCUF Algorithm

The current implementation of Wanderer as per this paper always joins the currently played point to its surrounding neighbor points. This causes the chain trees to be unbalanced as there is the possibility to join smaller trees to taller trees. For our game domain, trees being unbalanced do not represent a problem since as it was demonstrated in *Section 3.3*, the tallest tree was found to be ten levels deep for the biggest board size. Also, the tree is compressed every time the root node of a chain is obtained minimizing the possibility for very unbalanced trees. The same test to obtain the maximum levels in the tree also showed that the average number of levels was only 1.5. However, this section is added to test the possibility that any improvement can be made by weighing the trees before joining them together. The concept is simple as it only means that instead of always joining the current played point to its neighbor points, a check would be made to

see which tree is taller and then join the smaller tree to the taller tree. The following figure shows a representation of this improvement where P1 is the move that was just played and R1 and R2 are the surrounding neighbors. Recall from *Section 3.2 Neighbors* that neighbors are added starting from the North-East side and proceeding counter clock-wise.

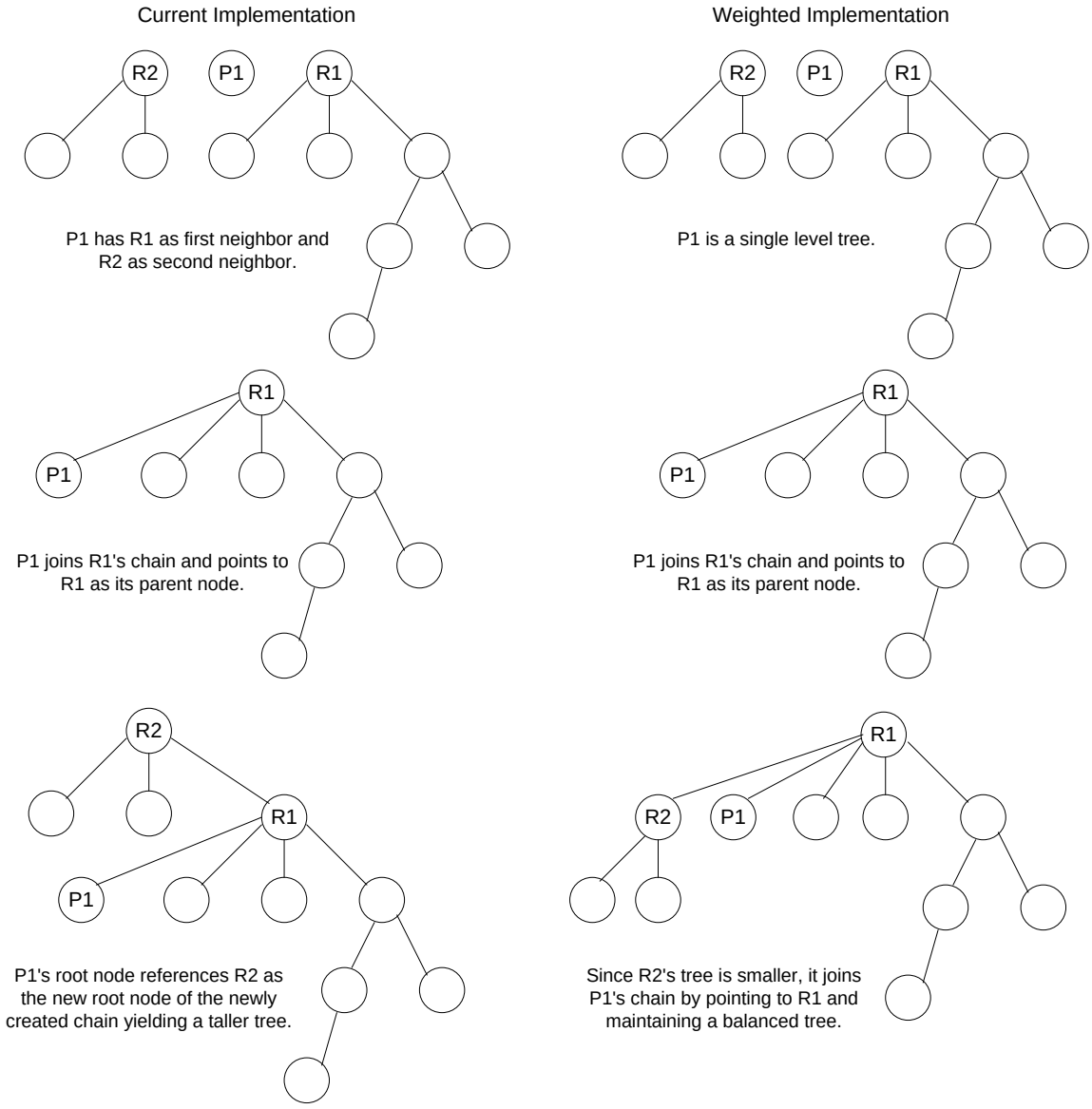

Figure 5.10 – Weighted Union-Find Algorithm

5.8 MCTS Node Building Efficiency

Sometimes, memory usage is at a premium, especially when multithreading Wanderer. With the initial implementation of Wanderer, it can be costly to create a MCTS tree as it was mentioned before, the class Point is reused to create nodes of the tree. Nodes on the tree use only but a subset of the variables associated with an actual Point when used on the board. One way to improve on this issue is to create a class that only has as members those needed variables for a tree node. This alone would cut down the amount of memory space used by Wanderer. Further improvements on this idea can be made by updating some of the variable types used in the different classes implemented in Wanderer. On average, on a board size base 10, with 10 seconds per move, the average number of nodes created in a tree before the optimal move is selected is 131 876. Each node, when using the Point class is approximately 37 bytes with the current implementation. So for an average tree, Wanderer occupies 4 879 412 bytes of the available memory. Now, if we were to create a class for a tree node, the size in bytes would only be 19 if we choose variable sizes carefully, for example, the current declaration of the x-coordinate is an integer; however, a byte will suffice. So for an average size tree, with the new class the number of bytes is only 2 505 644 bytes. Immediately, a savings of half the amount currently used is seen. Controlling the efficiency of memory used by Wanderer adds the possibility of porting Wanderer to different platforms such as mobile devices without sacrificing its computational strengths due to lack of memory.

Chapter 6

Conclusion

This thesis has shown that Havannah is another candidate for using the Monte Carlo algorithm approach especially when implementing it as a Tree along with the UCT variant. This thesis described then engine called Wanderer as a test bed for the algorithm. A basic Monte Carlo Search engine was tested against Wanderer and it was shown that its playing strength was far superior. With some fine tuning of the parameters in Wanderer, it was also shown that its playing strength was improved. It was mentioned that the aim of this paper was to test the feasibility of applying the MCTS UCT algorithm to the game of Havannah. It was also mentioned that this paper only describes the initial development of Wanderer which was started in 2009. Wanderer was one of the first engines for the game of Havannah and it made its debut in the Computer Olympiads held in Spain in 2009. At the tournament, there was one other engine that played against Wanderer but Wanderer took the winning spot among the two. At this initial state, Wanderer is still a weak player against a human player however. Since 2009, Wanderer has been updated with many improvements as described in Chapter 5 that make it a very strong player even when playing against other humans as shown in [18]. It is true that the best results are obtained in small boards; however, there is still more improvements that can be added to Wanderer to make its playing strength even better.

Since Wanderer was one of the first at the Computer Olympiads, it sparked so much interest that many joined in the implementation of new computer engines playing Havannah. More and more research has been performed in this playing platform since 2009 because of that first match at the Computer Olympiads. Even though Wanderer in

its initial state does not result in a strong player against a human player, it was shown that programming Havannah is feasible and with further refinements beyond the scope of this paper it proves to be a strong competitor against human players.

References

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton, “A survey of Monte Carlo Tree Search Methods,” in Trans. Comput. Intell. And AI in Games, Vol 4 No 1, March 2012.

[2] Chess Programming Wiki. (2014, Oct 1) “Havannah” [Online]. Available: https://chessprogramming.wikispaces.com/Havannah

[3] ICGA Tournaments. (2014, Oct 1) “Wanderer” [Online]. Available: http://www.grappa.univ- lille3.fr/icga/program.php?id=605

[4] ICGA. (2014, Oct 1) “Havannah” [Online]. Available: http://www.grappa.univ-lille3.fr/icga/program.php?id=618

[5] ICGA Tournaments. (2014, Oct 1) “Castro” [Online]. Available: http://ticc.uvt.nl/icga/cg2010results/Havannah.html

[6] Gunnar Farneback. (2014, Oct 1) “Specification of the Go Text Protocol, version 2” [Online]. Available: http://www.lysator.liu.se/~gunnar/gtp/gtp2-spec-draft2.pdf

[7] Little Golem. (2014, Oct 1) “HavannahGUI” [Online]. Available: http://www.littlegolem.net/jsp/forum/topic2.jsp?forum=50&topic=350

[8] Mindsports. (2014, Oct 1) “About Havannah” [Online]. Available: http://www.mindsports.nl/index.php/arena/havannah/48-about-havannah

[9] Princeton PPT Presentation. (2014, Oct 1) “Union-Find Algorithms” [Online]. Available: https://www.cs.princeton.edu/~rs/AlgsDS07/01UnionFind.pdf

[10] Sensei’s Library. (2014, Oct 1) “UCT for Monte Carlo Go” [Online]. Available: http://senseis.xmp.net/?UCT

[11] Wikipedia. (2014, Oct 1) “Pie Rule” [Online]. Available: http://en.wikipedia.org/wiki/Pie_rule

[12] Wikipedia. (2014, Oct 1) “Monte Carlo Tree Search” [Online]. Available: http://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[13] Guillaume M.J-B. Chaslot, Mark H.M. Winands, H. Jaap Van Den Herik, Jos W.H.M. Uiterwijk and Bruno Bousy. “Progressive Strategies for Monte Carlo Tree Search” [Online]. Available: https://dke.maastrichtuniversity.nl/m.winands/documents/pMCTS.pdf. June 6, 2008

[14] Sylvain Gelly, Yizao Wang, R´emi Munos, Olivier Teytaud. “Modification of UCT with Patterns in Monte-Carlo Go”. [Research Report] RR-6062, 2006. <inria-00117266v3>

[15] Yizao Wang, Sylvain Gelly. (2014, Nov 18) “Modification of UCT and Sequence-like Simulations for Monte-Carlo Go”. [Online]. Available: http://dept.stat.lsa.umich.edu/~yizwang/publications/wang07modifications.pdf

[16] Mark H.M. Winands, Yngvi Bjornsson, Jahn-Takeshi Saito. (2014, Nov 18) “Monte-Carlo Tree Search Solver”. [Online]. Available: http://www.ru.is/~yngvi/pdf/WinandsBS08.pdf

[17] Guillaume M.J-B. Chaslot, Mark H.M. Winands and H. Jaap Van Den Herik. (2014, Nov 18) “Parallel Monte-Carlo Tree Search”. [Online]. Available:

https://dke.maastrichtuniversity.nl/m.winands/documents/multithreadedMCTS2.pdf

[18] Richard Lorentz. “Improving Monte-Carlo in Havannah”. Experiments with Monte-Carlo Tree Search in the Game of Havannah, ICGA Journal, Vol 34 No 3, pg 140 – 149, 2011.

[19] J.A. Stankiewicz. (2014, Nov 18) “Knowledge-based Monte-Carlo Tree Search in Havannah” [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/msc/Stankiewicz_thesis.pdf

[20] Kocsis, L., Szepesvari, C. “Bandit based Monte-Carlo Planning”, In 15th European Conference on Machine Learning (ECML) pg 283 – 293, 2006.

Appendix A

Setting Point's Neighbors

```
//*****************************************************************//

//addValidNeighbor() function determines if a point is valid or

//within the valid range on the playing board

void Boardgame::setNeighbors( int x, int y )
{
    //declare a holder for this points
    vector<Point*> neighbors;
    //add NE neighbor
    addValidNeighbor( x + 1, y, neighbors );
    //add N neighbor
    addValidNeighbor( x + 1, y + 1, neighbors );
    //add NW neighbor
    addValidNeighbor( x, y + 1, neighbors );
    //add SW neighbor
    addValidNeighbor( x - 1, y, neighbors );
    //add S neighbor
    addValidNeighbor( x - 1, y - 1, neighbors );
    //add SE neighbor
    addValidNeighbor( x, y - 1, neighbors );

    //passes the vector of points to set each point's neighbors
    board[ x ][ y ].setNeighbors( neighbors );
}
//*****************************************************************//
```

Appendix B

Using Point's Root for Union-Find Algorithm

```
//*************************************************************//
Point* BoardGame::findRootPoint( Point & point )
{

    //if the parent is NULL then the point is the root point
    if ( point.parentNode == NULL )
    {
        return & point;
    }

    //find the root point of the parent point
    Point *parent = findRootPoint( *point.parentNode );

    //set each point searched in the chain to point to the root point
    point.parentNode = parent;
    return parent;
}

bool BoardGame::updateChains( Point & point )
{
    Point * neighborRoot;
    Point * myRoot;


    //visit neighbors
    for ( int i = 0; i < point.neighborCount; i++ )
    {

        //if the neighbor is of the same color as the played point
        if ( point.neighbors[ i ] ->getColor() == point.getColor() )

        {

            //get the neighbor's and the current played point's root
            neighborRoot = findRootPoint( *point.neighbors[ i ] );
            myRoot = findRootPoint( point );

            //if roots are not the same then join chains
            if ( myRoot != neighborRoot )  {
                //check for Rings
                myRoot->parentNode = neighborRoot;

                //OR chains together and check for Bridges and Forks
            }
        }
    }

}

//*************************************************************//
```