AN ABSTRACT OF THE THESIS OF

Jason Robert Suptic for the Master of Science

in Mathematics presented 07 June 2017

Title: Non-Losing Strategies for *Castellan*

Abstract approved: ____________________________

Non-losing strategies in games ensure a player can play in a manner in which, though they may not win, they do not lose. This thesis explores non-losing strategies for a restricted version of *Castellan*. *Castellan* is a game where each player attempts to best the other by using walls connected to towers to enclose regions with the most towers bordering them.

The question was narrowed to games with a rectangular layout. Research was conducted using a program written to enumerate games of relatively small size, ones having fewer than five unit regions. After observing outcomes of the computer program, conjectures were formed and lemmas proved.

In the end, it was found that with the restricted rules, the player to place the first piece has a non-losing strategy for games where the rows and columns both have an odd number of tower locations, or where exclusively the rows or towers have an odd number of tower locations. Lemmas and corollaries that are proved are used to support this fact.

Keywords: Game Theory, Graph Theory, *Castellan*, Zero Sum

NON-LOSING STRATEGIES FOR *CASTELLAN*

A Thesis

Presented to the Department of Mathematics

EMPORIA STATE UNIVERSITY

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Jason Robert Suptic

June 2017

Approved by the Chair of the Mathematics Department

Committee Member

Committee Member

Committee Chair

Approved by the Dean of the Graduate School and Distance Education

ACKNOWLEDGMENTS

I want to thank my wife Grace, who has put up with me disappearing for hours on end to work on this thesis and still being willing to play *Castellan* with me when I come home in order to test conjectures. She is able to listen to me explain my work and give honest feedback. I also thank Steven Kenny for taking the time to read through and offer advice on corrections. I would like to thank the *Panera* restaurant and the workers there for their supportive words while I worked for hours at a table in the back. Ryan Frier helped me with the formatting layout of the thesis and was there to listen to me when I felt like I couldn't do it. I would also like to thank Andy Graulund the programmer of Graph Theory JS, a graph visualization tool that was used along with MS Paint to create nearly all the images. Trevor Mahoney added valuable code edits and suggestions, without which the code would not have been nearly as much use as it was. I also thank my thesis board members (who I hope I did not turn into bored members). Finally, I would like to thank Dr. Chad Wiley and Dr. Thomas Mahoney, my advisors, for their invaluable input. Further, I appreciate their willingness to meet at odd times and on weekends.

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

A game is a conflict where two or more players make choices that involve gains or losses [1] [2]. Myriad gains exist and could be monetary, political, or points. *Castellan* is a game with the latter being of importance. During play *Castellan* pits two players against each other, with each attempting to enclose regions with walls and towers. Once a region is enclosed, a keep is placed inside to indicate which player took the region. Points are scored not by the number of regions a player encloses but by the number of towers adjacent to those regions. During the game, a player not only must concentrate on regions they want to enclose but also must concentrate on how to avoid giving regions up to the other opponent.

Reading the terms Player 1 and Player 2 over and again could be not only boring but also confusing. Thus, Albus will be considered to be the player we are in favor of, and Bellatrix will be the adversary we wish to destroy, or at least lay with in equal ruin. These names were chosen as a tribute to characters from JK Rowling's *Harry Potter*.

A non-losing strategy for Albus must ensure that for any move Bellatrix makes, he has a move that leads to a final game position where Bellatrix does not win. Given a proposed strategy, if there is even a single move Bellatrix may make that Albus does not have a move that ensures the game continues down a non-losing path, then the strategy itself is not a non-losing strategy.

Mathematical theory is used throughout this thesis to develop terminology and conjectures about *Castellan.* Conjectures are then proved using the definitions and other common mathematical concepts so that they become lemmas and corollaries. While finding examples of strategies for small game configurations can be interesting, the main goal was to find non-losing strategies for configurations that are as general as possible.

1.1 Rules

To make analysis of the game easier, we study a restriction of the official *Castellan.* The original rules are found in appendix A, as opposed to describing and then redefining them here. Traditionally *Castellan* is played using four pieces: towers, short walls, long walls (which we exclude), and keeps (Figure 1.1.1). For this thesis, the focus is limited to towers, short walls, and keeps. Thus, when the term *wall* is used, it is understood to be a short wall.

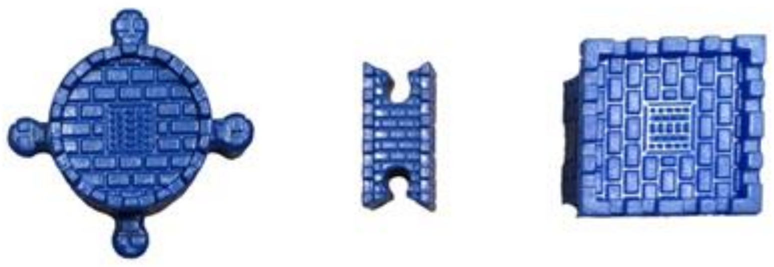


Figure 1.1.1 Pieces of play from left to right: Tower, Short Wall, Keep (photo by Jason Suptic)

A *tower* is a circular game piece with four small nubs placed on the outer edge of the piece at 90° intervals. Each nub may have attached to it exactly one wall. *Walls* are linear game pieces that can be connected at each end to a tower. Thus, a wall may connect to at most two towers. Further, a wall may only connect to a tower, and the converse is true.

Each player's goal is to use towers and walls to enclose courtyards, which will be referred to as regions or enclosed regions, to secure their keeps. A *keep* is a small square game piece used as a marker inside each enclosed region to indicate which player enclosed that region. Each keep is either red or blue, and exactly one keep is placed in each enclosed region. For simplification of images and diagrams, the letters ‘A’ and ‘B’ will be used inside enclosed regions to signify either Albus or Bellatrix enclosed the region.

Gameplay is turn based and alternates between Albus and Bellatrix. As a convention, we assume that Albus always takes the first turn. During each player’s turn they may place one of three combinations of pieces: a wall, a tower, or one of each. Notice that placing a keep is not a move, rather a way to indicate a player has taken a region. During play, and after the first turn, any piece placed must be attached to one already played. Once a player’s pieces and keeps are placed for their turn, play moves to their opponent. Play continues in this manner until either player has no moves left at the *beginning* of their turn. As such, *Castellan* is a rather simple game at its core.

Pieces may be played only inside predetermined boundaries. The only piece whose color matters is the keep, as a keep indicates which player took a specific region. Each player is allotted as many towers as there are tower locations and as many walls as there are wall locations within the boundary of the game board. After a player encloses a region, there may be wall or tower positions inside that region that can be played. In this case, only the player who enclosed the region may place walls and towers inside; doing so can create more enclosed regions, increasing that player’s score.

After gameplay ends, scores are calculated. The score of an enclosed region is equal to the number of towers that border that region. A player’s total score is the sum of the scores of their enclosed regions. Note that a tower may provide up to four points, one point for each region it borders. From this, it follows that a single tower may provide points to one or both players. Whomever has the highest score wins the game.

Based on how we have defined the rules, each player knows exactly what the other player can place on each turn and where they can place it. A game with this type of knowledge is known as a game of perfect information [3].

A singular challenge is presented to each player, enclose regions in a manner to ensure they have as many or more towers bordering their enclosed regions than the other player. Thus, *Castellan* lends itself to analysis of strategies that maximize a player's score or similarly minimize their opponent's score.

1.2 Images and Figures

Images created with GraphJS [4] contain two basic components, circles and lines. We use them to represent the walls and towers used in *Castellan*. Below is an example of a position where Albus has enclosed one region and the other unit region is partially enclosed.

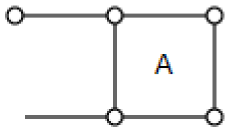


Figure 1.2.1 Wall Without Tower

Sometimes it will be necessary to create images that convey positions where pieces already placed and legal locations with no piece yet placed are displayed. To indicate locations where a tower can be legally placed, but has not been yet, a filled in circle will be used. A dashed line will be used to indicate a legal wall location where a wall has yet to be placed.

2 Definitions

While the rules above describe gameplay, there is still a need to concisely define the pieces for play and other terms that will be utilized for analysis. First, to help simplify analysis, restrictions are placed on where pieces may be played. The *Game board* refers to the underlying set of all locations in which a tower or wall can be placed during game play. Since a keep is only a place holder, when the number of locations is discussed, only places where a tower or a wall may be placed on the game board are considered in the count.

Since the pieces placed during a player's turn are of a specific size and are connected at $90°$ angles, we can view the game board as a Cartesian grid of points labeled with pairs of integers (k, j) such that. When thought of in this manner, tower locations are found at points of the form $(2k, 2j)$. When we let $(0, 0)$ be the bottom left corner of the game board, then wall locations are found at points of the form $(2k, 2j + 1)$ or $(2k + 1, 2j)$. Region locations, where keeps are placed, are the locations that have the form $(2k + 1, 2j + 1)$. A *complete game* is a game in which a piece has been placed in every location on the game board.

Position of a game means the currently placed pieces and how they are configured. In this manner, the position may be represented as a graph. A *connected* position is one in which given any two pieces played, a path exists between them. At the end of each players turn, the position must be connected.

Describing the "size" of the gameboard is also useful by giving a way to discuss locations in an absolute manner. As such, *height* is defined as the number of rows with a tower location, and *width* is defined as the number of columns with a tower location. Discussing the size of a rectangular game board leads to using the terms (k, j)-*game* or (k, j)-*game board* is to indicate a board of height k and width j; these terms are interchangeable. Figure 2.1 is of a $(2,3)$-game board, with height 2 and width 3.

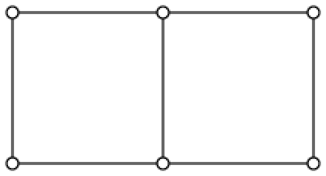


Figure 2.1 A (2, 3)-game board with all locations played

Saturated means that no matter what move the current player makes, there will be a unit region enclosable at the beginning of the next player's turn. Figures 2.2.a and 2.2.c are both of saturated positions. Assuming 2.2.b has only one piece left to be played to finish the game, then it is not a saturated position. This is because a saturated position indicates a region may be taken on the next turn. However, in 2.2.b there will be no more positions to take after the next piece is placed and no more turns.

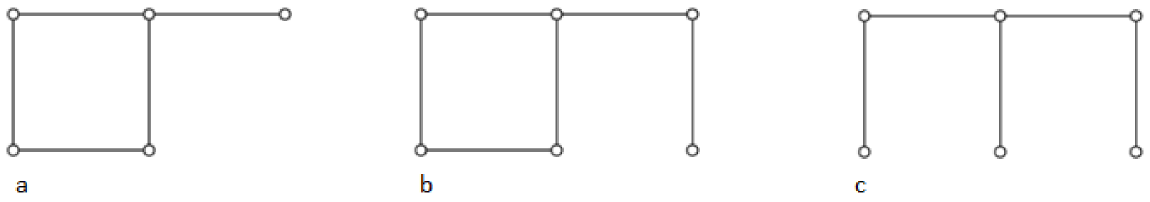


Figure 2.2 A saturated position (a), a non-saturated position (b), and a saturated position (c)

For $k \in \mathbb{Z}^+$ a *k-bucket* is a configuration of $k + 2$ towers and $k + 1$ walls such that two vertical walls are connected by contiguous horizontal walls, and no other walls are in the area between the vertical walls. The letter k indicates the number of unit regions between the two vertical walls. Figure 2.3 below is of a 2-bucket and a 3-bucket. These shapes occurred commonly when running the code on a $(2, n)$-game board and are only described for such boards. A bucket may have the open side facing up or down. However, motivation for the term was from the positions in Figure 2.4.

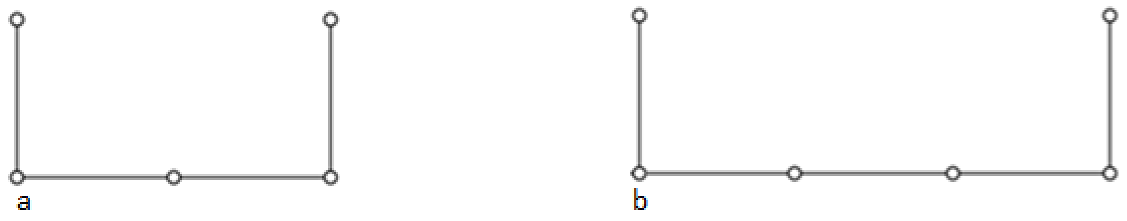


Figure 2.3 A 2-Bucket (a) and 3-Bucket (b)

A wall in *Castellan* need not be terminated on both ends by a tower. As such, a missing tower is not an accidental omission; rather, it represents that no tower was placed.

As previously mentioned, a game board lends itself to Cartesian representation. Also, some of the fundamentals of graph theory are useful to describe certain game positions. A *graph*, in the context of traditional graph theory, consists of a set of *edges* along with a set of *vertices*, "and a relation that associates with each edge two vertices (not necessarily distinct) called its *endpoints* [5]." In *Castellan,* the analogous pieces for edges and vertices are *walls* and *towers* respectively. A major difference is that in *Castellan* a wall may be incident zero, one or two towers. A wall may be incident zero towers only on the first turn, and only if the wall is the only piece played. These situations are shown below in Figure 2.4.

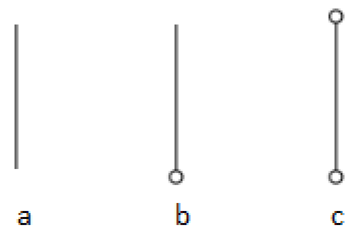


Figure 2.4 A wall incident zero towers (a), incident one tower (b), incident two towers (c)

A *path* is a finite sequence of towers such that consecutive towers are joined by a wall. A path may only visit each tower once, except perhaps the tower the path began on, in which case it is called a *closed path* or *cycle*. Note that a k-bucket is a path with $k+2$ towers and $k+1$ walls.

Figure 2.5 has labeled towers to make demonstrating each of the definitions in this paragraph clear. When two tower labels are listed next to each other, it indicates they have an edge between them and that we are moving from the first one listed to the other. Thus, (a, b, c) is a path, and (a, b, e, d, a) is a cycle.

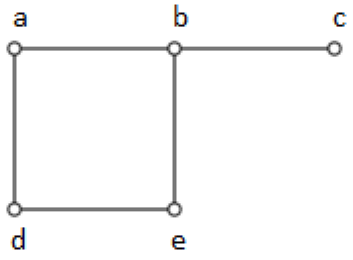


Figure 2.5

A *unit square* or *unit region* consists of locations on the game board that can be enclosed by a cycle made using exactly 4 towers and exactly 4 walls. As such, the only piece that may be placed inside a unit region is a keep. The cycle (a, b, e, d, a) in figure 2.5 is an enclosed unit region.

An *enclosed region* is the set of unit regions interior to a cycle. The size of an enclosed region, denoted $|\mathcal{R}|$, refers to the number of unit squares contained in the region. Let $\mathcal{R}_a$ represent the union of all regions enclosed by Albus and $\mathcal{R}_b$ those enclosed by Bellatrix. Thus $|\mathcal{R}_a|$ and $|\mathcal{R}_b|$ represent the size of all regions enclosed by Albus and Bellatrix respectively. It is also important to have language that allows us to discuss distinct types of enclosed regions. A *minimal region* is an enclosed region that does not contain any smaller enclosed regions. The *outer cycle* is the cycle that encloses all unit regions on the game board. *Interior locations* are those locations on the game board that are not located on the outer cycle. The *unbounded region* refers to the region outside the outer cycle. A *filled region* is an enclosed region in which every unit region inside is enclosed. It is sometimes necessary to address the locations where the only legal piece to be played is a keep; we shall call these locations *region locations.*

In Figure 2.6 there are multiple enclosed regions. The size of all enclosed regions is $|\mathcal{R}| = 5$. The size of the region labeled B is $|\mathcal{R}| = 2$ and it is a minimal region. Even though the region made of C and D together is the same height and width as region B, it is not a minimal region as it contains two unit regions.

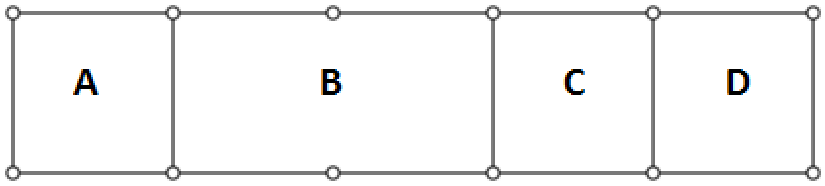


Figure 2.6

Definitions given above should be sufficient to begin discussing the game. Going forward, terms will be defined as necessary.

3 Methods

Mathematical proof is the primary method used within this thesis. However, computer programming and implementation of the minimax algorithm were also crucial in the development of lemmas. Python code was developed that can enumerate an entire game tree for relatively simple and small game boards – those with a rectangular layout that contain fewer than 8 unit regions.

For those readers who are interested in the code used for initial analysis, it is found in Appendix A. Many iterations were realized before a semi-final version was used. Semi-final is used to indicate that while it works, it may not run optimally. Also, it indicates that the code was tested for the limited scope of pieces allowed in the simplified version of the game presented.

3.1 Game Trees

A game in which the possible moves for each player's turn are finite and definite (not random), has outcomes that can be easily represented using a game tree. A game tree is one way to visualize the outcomes of a particular game (Figure 3.1.1). However, as a game grows in size or number of moves, a full game tree may be impractical.

Each node represents a game state. Thus, the children of a node represent game states that may be reached after one player's turn from the game state of the node one level above them. The root of a game tree represents the initial game state. In *Castellan*, the root would represent an empty game board. Children directly below the root represent the game states that are possible after Albus' first turn. The children on the next level represent possible game states after Bellatrix's first turn, and so on [6]. Figure 3.1.1 is a partial game tree of the game tic-tac-toe. Notice that at each tree level, the player whose turn it is alternates. Each level of a game tree represents the game states that are possible from the previous player's moves.

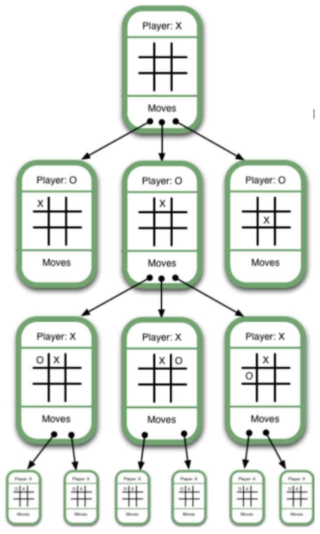


Figure 3.1.1 [7]

Branching factor refers to the number of distinct possible moves per turn, not including equivalent moves. For example, in the tree of Figure 3.1.1, an x in the upper left corner is equivalent to an x in any other corner. As such the play with an x in a corner only adds one to the branching factor. To give contrast of branching factor size, note that tic-tac-toe has a branching factor of approximately 3, whereas chess' branching factor is closer to 35 [8]. Let b represent the branching factor of a game and let d represent the depth of the game tree for the minimax algorithm to search. In this case, the number of positions for the algorithm to examine is b^d [6]. To solve tic-tac-toe (assuming symmetry is ignored) this is a relatively small number 3^9. In chess, there are considerably many more levels to go to beyond the ninth move depending on which branch is examined. Even at the ninth level there are already about 78-trillion moves to examine (again without regards being paid to symmetry and using the simplified calculation presented above).

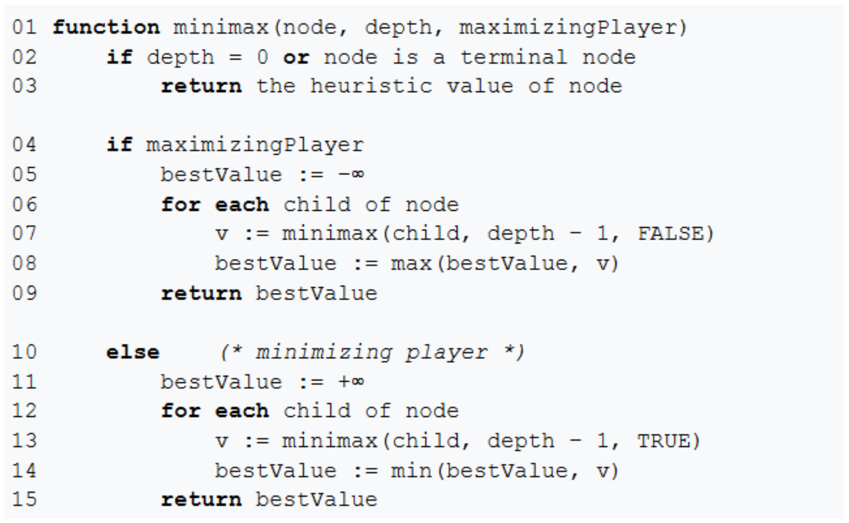
Castellan has a small branching factor on a small game board. The $(2,2)$-game board has a branching factor of about 3 with regards to symmetry and a max depth of 8.

However, as unit regions are added, the game quickly grows to a size that cannot be analyzed in a reasonable time. As such, the program created to analyze *Castellan* was used to gather information about smaller games. That data was then examined for patterns and conjectures were formed based on those patterns.

3.2 Minimax Algorithm

Minimax, or more formally the minimax algorithm, is the algorithm implemented by the code used for this thesis. Minimax explores the game tree and decides which branch leads to the best outcome for both players. This is done by alternating the interest of a player at each level of the tree. Minimax is a viable means of research in games of perfect information (games where both players know all the possible outcomes of each turn), where each player's interest is directly opposite the other [7].

Minimax gives the ability to find outcomes of a game board assuming both players are playing optimally. The algorithm returns a sequence of optimal moves for both players, but this sequence is not necessarily unique. Thus, while the end game value returned by the sequence of moves from the program will match that of optimal play, the moves may differ. Some description and an example of the minimax algorithm are warranted and given below. The pseudocode below has been given line numbers to make it easier to discuss what is happening where.



```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node

    if maximizingPlayer
        bestValue := −∞
        for each child of node
            v := minimax(child, depth − 1, FALSE)
            bestValue := max(bestValue, v)
        return bestValue

    else    (* minimizing player *)
        bestValue := +∞
        for each child of node
            v := minimax(child, depth − 1, TRUE)
            bestValue := min(bestValue, v)
        return bestValue
```

Figure 3.2.1 [9]

Minimax is a recursive function. To begin, the minimax function must know which node the game is currently on, what the max depth is to traverse, and which player is starting. In the pseudocode above, the maximizing player is the one starting. During each call to minimax, the depth variable is decremented by one. Without the decrementing call, the function would not end until the entire tree is enumerated.

Minimax is only called one time from outside of the function. However, the function is recursive. Each time Minimax is called by itself, it checks to see if either the desired depth has been reached or if the current node is a leaf. If either of these conditions is met, the algorithm returns the game value at that node. If neither of the conditions are met, then the algorithm calls itself for either the maximizing or minimizing player, and this value is switched inside the call in each of the for loops.

Notice that the initial best value for the maximizing player is negative infinity, and the initial best value for the minimizing player is positive infinity. In each case this represents the worst-case scenario for the players. This ensures that each player will never want to return their initial value, instead always preferring to find a better value.

Once the better value is found, it is passed to the next round of evaluation to the level of nodes the other player will choose values from. These values are passed upward in the game tree until the value of the root (which is the value of the game) can be computed. This game value represents the score that is the best-case scenario for both players, given both players are playing optimally.

When analyzing *Castellan* using the minimax algorithm, Albus is assumed to always be the maximizing player. Note, we do not set a depth limit in our code. As such we explore all branches of the game and ensure the score returned is the final score of the game. This ensures there could not be a better solution deeper in the game tree. Since an optimal strategy is desired for Albus, more information is needed than just the game value. Along with the best value for each node, we store a move that achieves this value. In the final output, we return a sequence of optimal moves for each player. Scores and lists of moves from different games were then able to be analyzed and searched for similarities in patterns of play.

Minimax searches the game tree depth first to either a predetermined node level, or if no level is given, it will run until the entire game tree is enumerated. In either case the bottom level nodes or leaves are assigned a minimax value. This value is calculated subtracting the minimizing player's (Bellatrix's) score from the maximizing player's (Albus') score at that node. Thus, a positive game value corresponds to Albus winning, and a negative game value corresponds to Bellatrix winning, and a game value of 0 means the players tie. Though it is necessary to know the minimax values at the given node level to analyze a complete game tree, it is not sufficient. For example, perhaps each branch of a game tree ends in two nodes, one with a positive minimax value and one with a negative value, then we need to know which one is optimal.

Four images follow that explore the game tree of an example game. Again, the goal of the maximizing player is to maximize the game value on their turns, and the goal of the minimizing player is to minimize the game value. At each node on the player's current level, they will pick the value from one level deeper that most benefits them. Once they have done this for all nodes at the current level, they pass the game to the other player, who does the same.

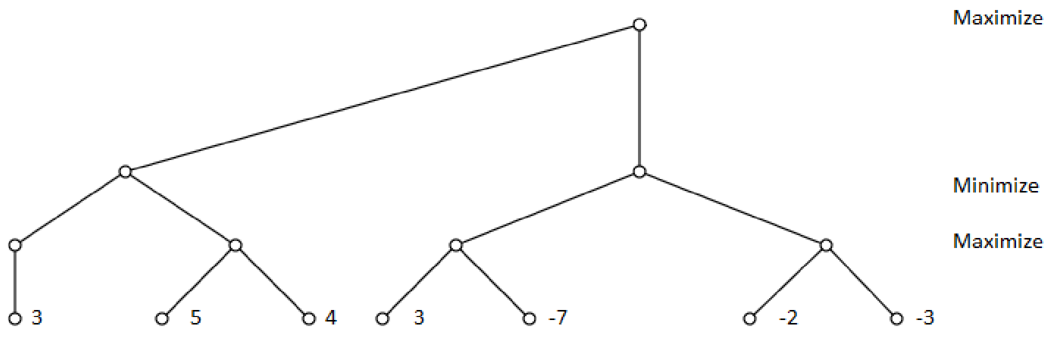


Figure 3.2.2

Figure 3.2.2 represents the initial game tree. For this example, arbitrary game values have been assigned to each end state. In an actual game, these values would represent the anvalue of the game at that point. At this node level, the maximizing player wants to choose the largest game values from each of the leaves.

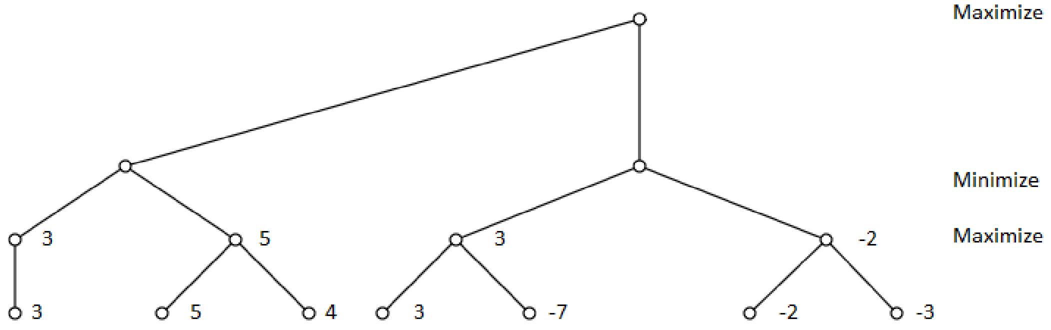


Figure 3.2.3

Next it is the minimizing player's turn (Figure 3.2.4). They want to choose the smallest value from each branch. Thus, they will choose 3 on the left branch and -2 on the right.

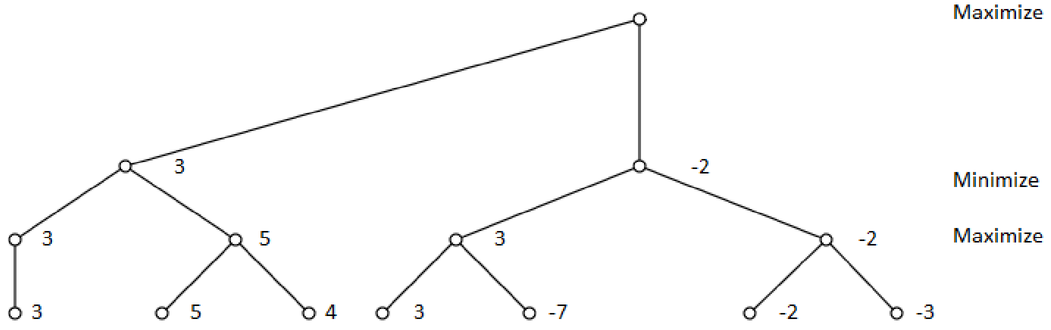


Figure 3.2.4

Finally, the maximizing player wants to choose the largest game value from the branches. As such, they will choose 3 and be the winner in this example game (Figure 3.2.5).

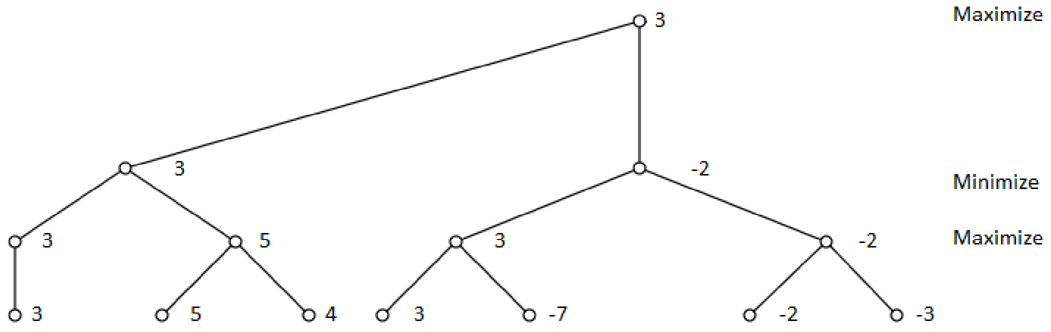


Figure 3.2.5

The minimax algorithm computes the value of the game that is achieved when both players make optimal moves. The actual winner of the game, however, is determined by the rules of the game being examined. For instance, if the rules of the example game above stated that a score of less than 5 indicates a win for the minimizing player, then the minimizing player would win.

4 Goal of Analysis

Prior to introducing strategies and proofs, a clear definition of the problem, as well as some observations, are warranted. A non-losing method of play will, for analysis, result in a game in which Albus does not lose. Notice that this does not imply Albus wins, just that he at least ties. Recall that given the restricted set of rules, *Castellan* is a zero-sum game of perfect knowledge. This fact helps to simplify analysis as it ensures that at each position Albus can calculate every possible move available to Bellatrix.

For purposes of analysis scores discussed will not be those held by Albus and Bellatrix, instead the game value will be used. The *game value* is calculated by taking Albus’ score minus Bellatrix’ score. Thus, the game value can be negative, zero, or positive. A negative score indicates Albus has lost, a value of zero indicates a tie, and a positive game value indicates a win for Albus. As such *non-losing* means a game value of zero or greater. For example, if Albus has a score of 10 and Bellatrix a score of 6, then the game value would be 4 and Albus has won. If instead, Bellatrix scored 10 and Albus 6, then the game value would be -4 and Bellatrix will win.

To have a starting point to form conjectures, it is useful to know how some smaller games will turn out. To do this, the minimax algorithm and *Python* programming language were utilized, which were discussed in sections 1.4 and 1.5 respectively. Initially *C++* was going to be the language of choice. However, *C++* can be more cumbersome to write. While using *Python* helped to condense the code, there were time and memory tradeoffs [10]. These tradeoffs left computer analysis of boards with more than eight unit squares not possible. Given these limitations, we use the program to find and test propositions.

5 Lemmas and Strategies

Methods described to this point were used to find possible patterns in playing *Castellan*. Below are lemmas and methods of play that were found. The lemmas that are included are not all directly related to the explanation or proof of the methods found to be non-losing. However, they have applications in situations that can arise in game play that does not have as many restrictions, such as playing on a non-rectangular board. Further, many of them may be of use in future research to finding more general methods of play.

5.1 Lemmas

Ensuring the results found through analysis are correct relies on certain fundamental facts. Lemmas, corollaries, and comments presented in this section will act as the fundamentals. Given a strong set of basic proofs, it is possible to expand to more generic cases. Unless otherwise stated, there is one universal assumption made for the lemmas:

during each player's turn they must place a wall, a tower, or one of each, in a manner that ensures the position is connected at the beginning of the other player's turn.

Game boards become more complex to analyze based not only on size but also on the shape of the game board. As such we begin by analyzing the simplest board that can yield a score. The smallest board that yields a score is the $(2,2)$-game board. The board consists of a single unit region and can be completely solved by the following lemma.

Lemma 1 (Ternary Lemma). Suppose the game board is configured such that once all locations have a wall or tower placed, the pieces form a single cycle. Let L represent the number of open locations currently on the game board. A player has a winning strategy, *iff* $L \not\equiv 0 \; mod \; 3$ at the beginning of their turn.

Proof. We prove the statement by induction on L. For the basis consider the cases where $L < 3$. These cases are considered because a player may play at most two pieces per turn.

When $L = 0$ at the beginning of a player's turn, the other player has placed the final piece to enclose the only region. Thus, the current player has lost and has no winning strategy. When $L = 1$ at the beginning of a player's turn, placing the final piece will enclose the only available region, giving the current player a score equal to the number of towers in the cycle. As such, the current player has a winning strategy, namely to place one piece. Suppose $L = 2$ at the beginning of a player's turn. Thus, the current position of the game forms a path. Since the final position will be a cycle and since every position of the gameboard must be connected, then the last two locations must be adjacent. Since locations on the gameboard alternate between a tower and a wall, then one location is a tower location and the other a wall. Since a player may place a wall and a tower on their turn, then the current player can place the last two pieces and enclose the only region available. Therefore, when $L = 2$ at the beginning of a players turn, that player has a winning strategy, namely to place the last two pieces. By these cases, the statement holds when $L < 3$.

For induction, let $L > 2$ at the beginning of a player's turn. For the induction hypothesis, suppose that for any $M < L$, the player about to move wins *iff* $M \not\equiv 0 \; mod \; 3$.

Since a player must play one or two pieces each turn, we examine the cases where $L = M + 1$ and $L = M + 2$. To prove the statement in the forward direct, we look to prove the contrapositive. For the first case, suppose $M \equiv 0 \; mod \; 3$ and that $M = L - 1$. Thus, $L - 1 \equiv 0 \; mod \; 3$ and it follows that $L \equiv 1 \; mod \; 3$. As such the next player has a non-losing strategy by the inductive hypothesis, which indicates the current player does not have a non-losing strategy. Next, let $M \equiv 0 \; mod \; 3$, further suppose $M = L - 2$. Therefore, $L - 2 \equiv 0 \; mod \; 3$ and thus $L \equiv 2 \; mod \; 3$. Again, the next player to move has a non-losing strategy by the induction hypothesis, again indicating the player about to move does not have a non-losing strategy.

We now prove the other direction by using the original statement. Let $M \not\equiv 0 \; mod \; 3$, and $M = L - 1$. Thus, $L = M + 1 \equiv 0 \; mod \; 3$. This fact along with the inductive hypothesis indicates the next player does not have a non-losing strategy, and the player about to move does – namely they should play one piece. Finally, let $M \not\equiv 0 \; mod \; 3$ and $M = L - 2$. We now have, $L = M + 2 \equiv 0 \; mod \; 3$. By this and using the inductive hypothesis, the next player does not have a non-losing strategy, and the player who is about to move does – they should play a tower and a wall. By the above cases the original statement holds. ■

With respect to our two players, what this says is that Albus always has a winning strategy when the game board consists of only a single unit region. His strategy is to ensure Bellatrix begins each of her turns in the $(2,2)$-game with $L \equiv 0 \; mod \; 3$.

Though basic, this result must not be underestimated. Not only does the ternary lemma allow us to solve the game with only one unit square available, it allows us to solve individual unit squares on a given game board. Thus, we have a way of solving certain positions of other game boards, which will be explored more in our analysis.

Corollary 1 (2-bucket Corollary). WLOG if Albus passes a position to Bellatrix in which the only open locations to place pieces are part of a 2-bucket, then Albus can take at least one unit region on his next turn.

Proof. If Bellatrix's turn begins with a position such that the only pieces she may place are located in a 2-bucket, then she must place at least a wall. This follows because the walls in a two bucket are all terminated with a tower. In a 2-bucket, there are 4 pieces that may be placed: 3 walls and 1 tower. Each unit region requires 2 walls and a tower to enclose, and the region composed of the two unit regions requires the same. Thus, Albus can enclose at least one unit region on his first turn after passing Bellatrix a 2-bucket. ■

Lemma 2 (Region Score Lemma). If R is an enclosed region, then the maximum score R can contribute to the player who enclosed it is $4|R|$.

Proof. First note that adding a wall or a tower to R does not decrease a player's score. This follows from the fact that adding a wall or tower either encloses a region or it does not. If a new region is not enclosed, then it will not decrease the current player's score because placing a piece cannot open an enclosed region. If a new region is enclosed, then the player's score will increase. Thus, the maximum score that R can contribute to the player who encloses it is attained by filling R with all possible towers and walls, creating a filled region made of unit regions. Note $|\mathcal{R}_A|$ is equal to the number of unit regions, and a unit region has a value of 4 in terms of score. Further, a unit region cannot be subdivided and this provides the maximum score for the area of the game board it covers then the maximum score for R is $4|R|$. ■

Corollary 2 (Filled Game Corollary). If at the end of the game all regions have all interior pieces played, then WLOG if $|\mathcal{R}_A| > |\mathcal{R}_B|$, then Albus wins. If $|\mathcal{R}_A| = |\mathcal{R}_B|$, then the game ends in a tie.

Proof. Playing all interior pieces in a region attains the maximum score of that region by the Region Score Lemma. Therefore, since each region has all interior pieces played, then the result follows directly. ■

One important question that needs addressed, at least for the $(2, n)$-board, is: can we restrict our analysis to only unit square regions and collections thereof? That is, do we need to concern ourselves with rectangular regions that are not complete?

Lemma 3 (Unit Region Lemma). Given a $(2, n)$-game board, under optimal play the largest region that will be enclosed by Albus or Bellatrix is a unit region.

Proof. WLOG, suppose that Albus is in the position to be able to enclose a region that is larger than a unit region. There are three positions we must consider for the completion of a larger region: Albus must place a wall to complete the cycle, Albus must place a tower to complete the cycle, or Albus must place a wall and a tower to complete the cycle.

First note that we may restrict our attention to regions with two unit regions inside of them. This is because if the larger region to enclose was three unit regions, then Bellatrix would have been able to enclose a unit region instead of giving Albus the opportunity to enclose the larger region. To show this, suppose Albus can enclose the maximum cycle on a $(2, k)$-game board where $k > 3$. Since Albus can enclose the maximum cycle, there are either 1 or 2 open locations on the maximum cycle. Thus, Bellatrix had $2, 3$ or 4 open locations on the outer cycle at the beginning of her turn. We need only examine the position with 4 open locations. In the maximum cycle, there are $4k$ locations total. Since the position must be connected, we examine two cases: when all empty locations are WLOG on the right most unit region of the position, and when the empty locations are all on the same horizontal side of the position. In the first case, the left most unit region will have 2 open locations to complete it at the beginning of Bellatrix's turn, thus she could complete a unit region. In the other case, since $k > 3$, the number of locations on one horizontal side of the board would be greater than or equal to 7. Thus, when there are 4 open locations on that side, there are at least 3 locations with a piece played. Thus, there is at least one wall placed on that side, and it must be one either the right most or left most unit region. Since the other horizontal side would have all pieces placed, then Bellatrix could play a tower and a wall to complete a unit region. Thus, we may assume we are playing on a $(2, 3)$-game board or a $(2, 3)$-game sub board.

Let L represent the number of locations on the game board which do not yet have a piece placed. WLOG, suppose for contradiction that Albus can enclose the maximum cycle. Thus, since one location is the interior one, Albus begins his turn with $L \in \{2,3\}$. This implies that Bellatrix's previous turn began with $L \in \{3,4,5\}$. We will examine a case for each possibility.

Suppose $L = 3$ at the beginning of Bellatrix's turn. Since one of the open locations must be the interior wall, there are two open locations on the cycle that Albus is attempting to complete, and they must be adjacent. As such, one location is a tower, and one is a wall. However, if this were the position, then Bellatrix would have a strategy to take the region, namely that she could place the last tower and wall piece on the cycle, thus enclosing the larger region. Thus, if Albus were playing optimally, he would not have passed play to Bellatrix with this position.

Suppose $L = 4$ at the beginning of Bellatrix's turn. Since one of the open locations must be the interior wall, then there are either two wall locations and one tower location available, or one wall location and two tower locations available (Figure 4.1). Given any of the positions in Figure 4.1, we have by the ternary lemma if the center location was not available for play then Albus has a winning strategy. However, since the center location is available for play, then Bellatrix has a strategy to not lose. The strategy is to place only the interior wall this turn.

Suppose $L = 5$ at the beginning of Bellatrix's turn. Thus, the position Albus is left with is described by either the $L = 3$ or $L = 4$ case.

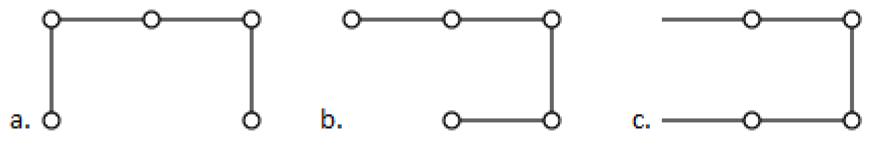


Figure 5.1.1

From the above, if both players are playing without the intention of losing, neither will complete the maximum cycle. Thus, given a $(2, n)$-game board, the largest region that will be enclosed is a unit region. ■

Corollary 3 (Odd Unit Region Game). If the $(2, n)$-game board has an odd number of unit regions and Albus has a non-losing strategy, then Albus has a winning strategy.

Proof. Suppose a game board has an odd number of unit regions. Thus, we can represent the number of unit regions as $2k + 1$ where $k \in \mathbb{Z}$. By the unit region lemma, the largest space to be enclosed will be a unit region. For contradiction, suppose the game ends in a tie. Since the game ends in a tie, the total number of unit regions can be written as $2j$, where j is the number of unit regions each player claimed. Thus, the number of unit regions is even, which is a contradiction. ■

In the next section, the desire is not only to prove the strategies to be true but also to describe the non-losing strategy that Albus can use against Bellatrix. Though the fact that a strategy exists is interesting, the point is to explain that strategy in a manner that can be easily remembered and implemented.

5.2 Strategies

Strategy 1 (Odd by Odd and Odd by Even Rectangular Board Strategy).

Consider the (m, n)-game where m is odd. Let $(0, 0)$ denote the center most location of the board. After setting this as the origin point, the game board can be thought of as a Cartesian plane, giving us four quadrants. Defining the game board in this manner allows us to use $180°$ rotational symmetry about the origin. When n is odd, $(0, 0)$ is a tower and the corners of the board are at $(\pm(m - 1), \pm(n - 1))$; when n is even $(0, 0)$ is a wall and the corners of the board are at $(\pm(m - 1), \pm(n - 1))$.

Suppose the game board has $(2k + 1, 2j + 1)$ rows and columns with tower locations respectively, or $(2k + 1, 2j)$ rows and columns with tower locations respectively with

$k, j \in \mathbb{N}$. Further, suppose that if a player places a wall and a tower during their turn, those pieces must not only be connected to the current position but also to each other. Albus' non-losing strategy is to place the piece in the origin location on his first turn. Viewing the game board as a Cartesian plane with origin $(0, 0)$, and letting the locations of Bellatrix's pieces placed be denoted (x, y), then after Albus' first turn he should play the same pieces as Bellatrix, but at location $(-x, -y)$.

Proof. We begin by addressing the legality of the play. First, how do we know the piece has not been placed yet? Since Albus' first move was to place the origin piece we know that the origin may not be played by Bellatrix. The origin is the only location where $(x, y) = (-x, -y)$. Suppose Bellatrix places a piece where the location at $(-x, -y)$ has a piece already placed. This contradicts Albus' method of play and thus cannot happen. Thus, any piece that Bellatrix places at (x, y) will have an empty location at $(-x, -y)$.

Next, we ask, how do we know the location Albus is to play on lies on the game board? This is quite simple. Since the game board is divided into four equal sized quadrants, then if the location Albus is to place a piece on is outside of the bounds of the game board, then so too was the location Bellatrix played on. Thus, the play is legal.

Now that we know the locations where Albus will place pieces are legal, we address how we know the game will not be a loss for Albus. First note, since we know the game has four symmetric quadrants about the origin, then there are an odd number of exclusively either tower or wall locations. Further, from this symmetry we know that by Albus taking the origin on his first play, the number of both tower and wall locations open are even on Bellatrix's first turn. This, coupled with the fact that each player must play a wall, a tower, or one of each on each of their turns, and if they place one of each, the pieces must be connected, gives us that there are an even number of both walls and towers left at the beginning of Bellatrix's last turn. Thus, Albus will always be the one to place the last piece on the game board. Further, from Albus' method of play, and due to the symmetric nature of the game board, for each region that Bellatrix encloses, Albus will enclose a region of equal size. Albus need not worry about Bellatrix taking the regions he set up to be taken by mirroring due to the rotational symmetry, restriction of number of pieces a player gets, and the requirement that pieces a player places need to be connected both to each other and the current position. Since each player gets a wall and a tower at most each turn, then two regions that are symmetric by rotation may not be enclosed by one player, as each region would require the same type of piece. Since the number of unit regions is even, it then follows that the game will end with Albus' score equal to Bellatrix's score, a tie. Thus, by Albus placing the origin piece on his first turn, and then placing each piece after that at $(-x,-y)$ when Bellatrix's piece is at (x,y), Albus has a non-losing strategy on the $(2k+1,2j+1)$-game board and WLOG the $(2k+1,2j)$-game board. ■

To begin work on more general game boards, the $(2,4)$-game board was examined. Though the code produced a win for Albus, a specific strategy and proof thereof was desired.

Strategy 2 ($(2,4)$ Winning Strategy). In the $(2,4)$-game board, there are three unit regions. As such, one player will win. This follows by the fact that there are four outcomes: a player may complete the maximum cycle, one player may enclose one unit region and the other may enclose a region composed of two unit regions, one player may enclose all three unit regions, or one player may enclose a unit region and the other player may enclose two unit regions. For ease of discussion the unit regions will be referred to as R_1, R_2, R_3, to denote the left most, center, and right most unit regions of the $(2, 4)$-game respectively.

On Albus' first turn he plays the left vertical wall and the upper left tower of R_1. After Bellatrix's first turn, Albus can ensure the game is in one of the two positions in Figure 5.2.1.

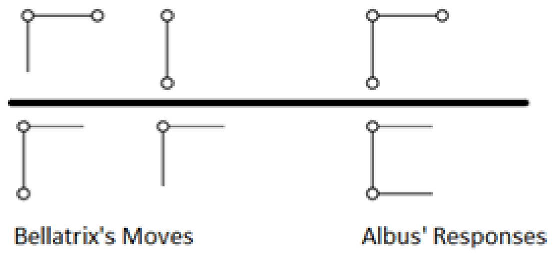


Figure 5.2.1 Top row is set A. Bottom row is set B.

Next, we break Bellatrix's possible moves into three distinct sets. Each set leads to a different position up to symmetry at the end of Albus' third turn, shown in Figures 5.2.2, 5.2.3, and 5.2.4.

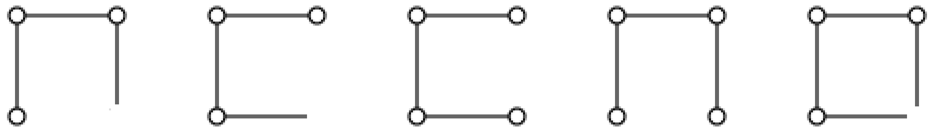


Figure 5.2.2 Set C

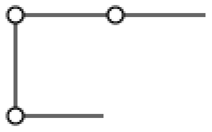


Figure 5.2.3 Set D

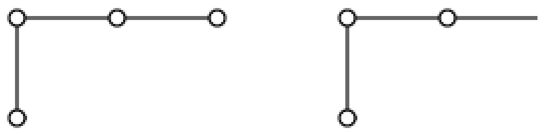


Figure 5.2.4 Set E

We begin by analyzing set C. Albus can place pieces in a manner to complete R_1, with the only pieces played inside R_2 being those common to R_1, and he does so. Since Bellatrix can play at most a horizontal wall of R_2 and its incident tower, then Albus passes Bellatrix the position shown in Figure 5.2.5, after his fourth turn. If Bellatrix encloses R_2, then Albus can enclose R_3 by the ternary lemma and win. Otherwise Albus may enclose R_2 on his next turn and win. This concludes analysis of set C.

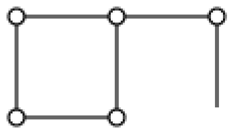


Figure 5.2.5

Next, we look at set D. For Albus' third turn he encloses R_1. On Bellatrix's third turn, if she plays in a manner that Albus can take R_2 on turn four, then he does so and wins. Otherwise there are three cases to examine (Figure 5.2.6). For case 5.2.6.a, Albus just plays a single tower. If Bellatrix does not enclose R_2, then Albus does. If Bellatrix encloses R_2, then Albus can enclose R_3 by the ternary lemma. Case 5.2.6.b follows by analysis of set C, since Albus can pass the position in Figure 5.2.5, if he receives the position in Figure 5.2.6.b. In the case for 5.2.6.c, Albus should play the other vertical wall in R_2. If Bellatrix does not enclose R_2, then Albus does. If Bellatrix encloses R_2, then Albus can enclose R_3 by the ternary lemma.

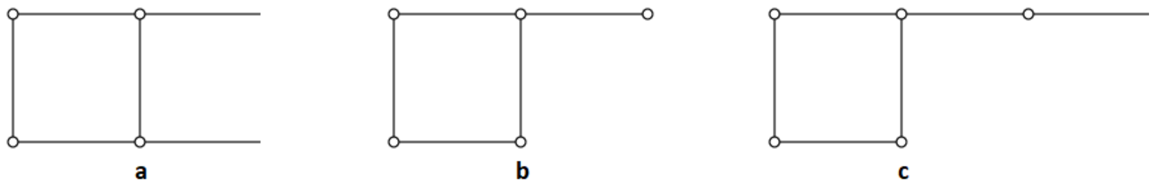


Figure 5.2.6

We finish by examining the case for set E. In this case, Albus uses his third turn to pass Bellatrix the position in Figure 5.2.7.

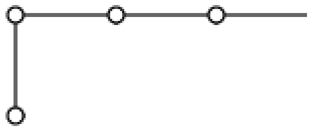


Figure 5.2.7

There are five sets of positions that must be considered that Bellatrix may pass to Albus after her third turn. From the first set (Figure 5.2.8), Albus encloses R_1 on his fourth turn. Next, if Bellatrix plays such that Albus can enclose R_2, then he does and wins. Otherwise, Albus can pass a 2-bucket to Bellatrix after his fifth turn. By the 2-bucket corollary, Albus will be able to enclose at least one unit region on his sixth turn and win.

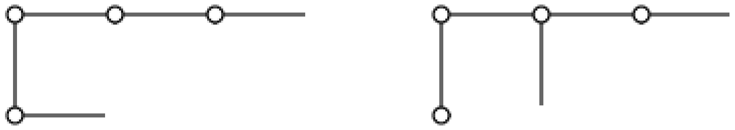


Figure 5.2.8

The next set (Figure 5.2.9) leads to Albus passing the position shown in *Figure 5.2.10*, and has a similar outcome as the set described just above. The only difference is that Bellatrix can pass a 2-bucket to Albus. In this case Albus plays only the vertical wall to divide the 2-bucket. Bellatrix would need to place two walls and one tower to enclose both unit regions, which she can't. Thus, Albus encloses at least one unit region on his next turn and wins.

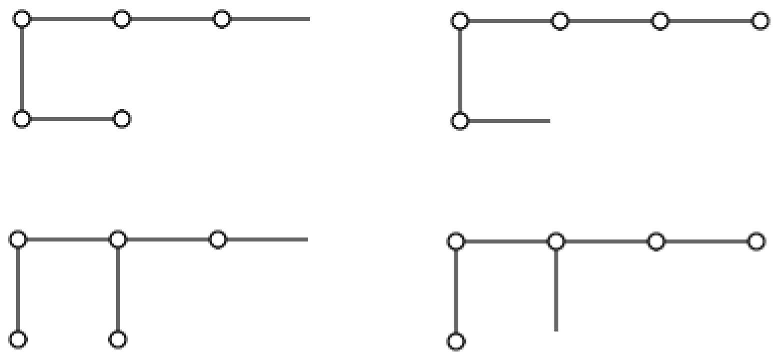


Figure 5.2.9

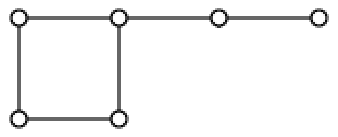


Figure 5.2.10

In the next possible position that Bellatrix could pass to Albus (Figure 5.2.11), Albus plays the last vertical wall of R_1. If Bellatrix plays only a tower on her next turn then so too does Albus. Then, if Bellatrix encloses R_1 or R_2, Albus encloses the other, and he encloses R_3 by the ternary lemma.

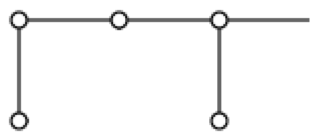


Figure 5.2.11

If Bellatrix plays the last horizontal wall in R_3, then Albus encloses R_1 and plays a tower on R_3. Thus, he will be able to enclose one more unit region on his next turn, as each remaining unit region requires a wall to enclose.

If Bellatrix begins by enclosing R_1 or R_2, then Albus encloses the one she did not, and he places a tower in R_3. In this case, Albus will enclose R_3 by the ternary lemma.

Finally, if Bellatrix places a tower and a wall in R_3, Albus should enclose R_3. Since R_1 and R_2 both require a wall that is not common to both regions, Albus will be able to enclose one unit region on his next turn.

The next possible position Bellatrix may pass to Albus is one where Albus may create the 3-bucket. No matter what Bellatrix plays in the 3-bucket, Albus may enclose a region. Then, only a 2-bucket remains. By the 2-bucket corollary, Albus may enclose at least one region on his next turn, thus winning.

Finally, we examine the following set that Bellatrix may pass to Albus after her third turn (Figure 5.2.12). Figure 5.2.13 shows the position Albus passes to Bellatrix after his fourth turn.

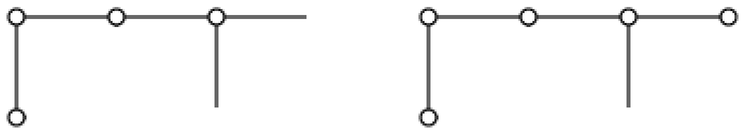


Figure 5.2.12

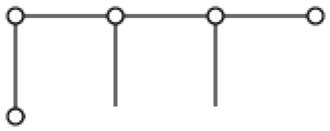


Figure 5.2.13

If Bellatrix encloses R_1, Albus encloses R_2 and then R_3 by the ternary lemma, thus winning. If Bellatrix plays only a tower, then so too does Albus. After which Albus wins by the ternary lemma. If Bellatrix plays a tower of R_2 and the last wall of R_2, A encloses R_1 and R_2, and wins. If Bellatrix plays a tower and a wall in R_3, then Albus encloses R_3. And since two wall locations remain on the board after Albus encloses R_3, then he also encloses R_1 or R_2 on his next turn and wins, as Bellatrix may only claim one of the two remaining regions. If Bellatrix plays the last tower of R_1 along with the last vertical wall of R_3, then Albus takes R_1. Again, since there are two wall locations left, Albus can take at least one more unit region on his next turn and win. The above cases are exhaustive, concluding the case analysis of the $(2,4)$-game. ■

6 Discussion and Future Research

What has been presented has solved some of the fundamental game boards. However, the general rectangular board is not yet solved. We have also examined the $(2,4)$-game board. The case that is missing is the $(2j,2k)$-game board, with $k, j \in \mathbb{Z}$. Using the unit region lemma, it is found that this case must have a winner, as there are an odd number of unit regions. Solving the $(2j,2k)$-game board would finish a solution of the rectangular game board, with players being required to place a tower a wall or one of each on their turn.

6.1 Future Research

What has been presented herein is a starting point for analysis of Castellan. I believe the lemmas presented can be expanded not only to larger game boards but can also act as a basis to more general ideas of how to play the game in its original form. The thought process of the ternary lemma when combined with the cards in a player's hand, along with knowing what cards the opponent has already played, could be used to enclose the largest regions possible.

To continue research, there are two areas to begin with. One, examining what happens when the allowable pieces are expanded; this could include games where skipping one's turn is an option. Two, expanding the size of the game board to include larger rectangular and non-rectangular regions. Once thorough analysis of these instances is complete, it will be possible to delve into games where the cards are introduced in some order, and finally when the cards are randomized.

While the code used to aid in analysis was useful, it is rather limited in its current state. Namely, due to enumerating the entire game tree, the time it takes to run grows rather quickly. Second, it is currently in a state that allows for only short walls and towers. As such, if the code were to be used in further research, it would require a good amount of modification to be able to support more general cases.

Suppose we did want to use the code as is, then it is useful to be able to estimate the limitations of time and storage space, and a manner of calculation to do so follows. If we let b represent the maximum number of moves at any node in our game tree and let h represent the maximum level of our game tree, then we can calculate both the space and time complexity of our current game. Space complexity measures the worst-case scenario of space needed in memory for a calculation. Space complexity is calculated $O(bh)$ and time complexity is $O(b^h)$ [11]. Calculating the maximum number of possible moves is done by summing the number of allowed towers and allowed walls each turn. Finding the maximum level of our game tree is as simple as counting the number of board positions. Letting t represent the number of allowed towers per turn, and w the number of allowed walls per turn, and with p being the total number of board positions at the beginning of the game, then we find: space complexity is $O(bh) = O((t + w) \cdot p)$ and time complexity is $O(b^h) = O((t + w)^p)$.

Since each move consists of a string that has outer parentheses, current score, and moves from each level of the tree, then we can calculate the maximum needed amount of memory by noticing that each character is one byte. We account for bytes for the outer parentheses by the constant 2. Bytes needed for the score will depend on the maximum number of digits in any score, say d_s. We also need to multiply the number of levels l by 3 to account for the outer brackets and comma needed for each move, then we add two to account for the outer bracket of the moves. Finally, we must multiply the maximum number of moves $t + w$ for each turn by 3 plus the number of digits in the maximum row d_r plus the number of digits in the maximum column d_s. We end up with the following equation: $4 + d_s + 3l + [(t + w)(3 + d_r + d_s)]$. Equating this number and converting to megabytes or gigabytes allows an estimation of the amount of random access memory (RAM) needed for computation. In practice, we found that any board greater than 7 unit regions pushed the threshold of a system with 16GB of RAM. Estimating time in seconds is much more involved and includes multiple variables, such as CPU speed, RAM speed, hard disk read and write speed, bus transfer speeds and more.

Works Cited

[1] W. Poundstone, Prisoner's Dilema, New York: Dover Publication, 1992, pp. 6,51.

[2] E. W. Weisstein, "Game -- from Wolfram MathWorld," 2016. [Online]. Available: http://mathworld.wolfram.com/Game.html. [Accessed 11 10 2016].

[3] M. D. Davis, Game Theory A Nontechnical Introduction, New York: Dover Publication INC., 1983, pp. 9,14.

[4] A. Graulund, "GraphJS," 20 May 2011. [Online]. Available: https://github.com/graulund/GraphJS. [Accessed 1 January 2017].

[5] D. B. West, in *Introduction to Graph Theory*, Noida (India), Pearson India Education Services Pvt. Ltd, 2015, pp. 2-36.

[6] "Game Tree - Scratch Wiki," MIT Media Lab, 27 May 2016. [Online]. Available: https://wiki.scratch.mit.edu/wiki/Game_Tree. [Accessed 7 February 2017].

[7] D. Higginbotham, "An Exhaustive Explanation of Minimax, a Staple AI Algorithm," 10 January 2012. [Online]. Available: http://www.flyingmachinestudios.com/programming/minimax/. [Accessed 28 02 2017].

[8] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Branching_factor. [Accessed 02 05 2017].

[9] "Minimax," Wikipedia, 16 February 2017. [Online]. Available: https://en.wikipedia.org/wiki/Minimax. [Accessed 8 March 2017].

[10] M. Fourment, "National Institute of Health," 05 February 2008. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2267699/pdf/1471-2105-9-82.pdf. [Accessed 07 February 2017].

[11] D. Nau, "Game-Tree Search," n.d.. [Online]. Available: https://www.cs.umd.edu/users/nau/game-theory/4%20Game-tree%20search.pdf. [Accessed 9 October 2016].

[12] S. J. G. Incorporated, *Castellan Rules version 1.0,* Austin Texas: Steve Jackson Games Incorporated, 2013.

[13] "chessprogramming - NegaScout," Tangient LLC, 2017. [Online]. Available: https://chessprogramming.wikispaces.com/NegaScout. [Accessed 7 Februrary 2017].

[14] "Alpha-beta pruning - Wikipedia," Wikipedia, 4 February 2017. [Online]. [Accessed 7 February 2017].

[15] A. L. Aradhya, "Minimax Algorithm in Game Theory | Set 1 (Introduction)," Geeks for Geeks, June 2016. [Online]. Available: http://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/. [Accessed 8 March 2017].

[16] C. Matoran, "Game Tree," Massachesetts Institute of Technology, 27 May 2016. [Online]. Available: https://wiki.scratch.mit.edu/wiki/Game_Tree. [Accessed 21 03 2017].

Appendix A Python Code to Model Play

Code is included below for reference. However, if you would like to be able to simply copy and paste the code into a new Python project you can download it at https://sourceforge.net/projects/castellancode/.

```
#Code by Jason Suptic
#Edited and added to by Dr. Thomas Mahoney and Trevor Mahoney
import itertools
import sys
import cProfile

#region GLOBALS
IB = [[0,0,0,0,0,0,0,0,0,0,0],
      [1,0,0,0,0,0,0,0,0,0,0],
      [0,0,0,0,0,0,0,0,0,0,0]]
numRows = len(IB)
numCols = len(IB[0])
allRegionPoints = []
for row in range(numRows):
    for col in range(numCols):
        if row % 2 != 0 and col % 2 != 0 and IB[row][col] == 0:
            allRegionPoints.append((row,col))
allTowerPoints = []
for row in range(numRows):
    for col in range(numCols):
        if row % 2 == 0 and col % 2 == 0:
            allTowerPoints.append((row,col))
allWallPoints = []
for row in range(numRows):
    for col in range(numCols):
        if (row % 2 == 0 and col % 2 != 0) or (row % 2 != 0 and col % 2 == 0):
            allWallPoints.append((row,col))
#endregion GLOBALS

#region SCORING
def fill(M, regionPoints):
    (row,col) = regionPoints[-1]
    if (regionPoints[0] != "Unbounded"
    and ((M[row-1][col] == 0 or M[row-1][col-1] == 0 or M[row-1][col+1] == 0)
and row-2 < 0)
    or ((M[row][col+1] == 0 or M[row-1][col+1] == 0 or M[row+1][col+1] == 0)
and col+2 >= numCols)
    or ((M[row+1][col] == 0 or M[row+1][col-1] == 0 or M[row+1][col+1] == 0)
and row+2 >= numRows)
    or ((M[row][col-1] == 0 or M[row-1][col-1] == 0 or M[row+1][col-1] == 0)
and col-2 < 0)):
        regionPoints.insert(0,"Unbounded",)
    if row-2 >= 0:
        if (M[row-1][col-1] == 0 or M[row-1][col] == 0 or M[row-1][col+1] == 0)
and (row-2,col) not in regionPoints:
            regionPoints.append((row-2,col),)
            regionPoints = fill(M,regionPoints)
    if col+2 < numCols:
        if (M[row-1][col+1] == 0 or M[row][col+1] == 0 or M[row+1][col+1] == 0)
and  (row,col+2) not in regionPoints:
```

```
            regionPoints.append((row,col+2),)
            regionPoints= fill(M,regionPoints)
    if row+2 < numRows:
        if (M[row+1][col-1] == 0 or M[row+1][col] == 0 or M[row+1][col+1] == 0)
and (row+2,col) not in regionPoints:
            regionPoints.append((row+2,col),)
            regionPoints = fill(M,regionPoints)
    if col-2 >= 0:
        if (M[row-1][col-1] == 0 or M[row][col-1] == 0 or M[row+1][col-1] == 0)
and (row,col-2) not in regionPoints:
            regionPoints.append((row,col-2),)
            regionPoints = fill(M,regionPoints)
    return regionPoints

#endregion SCORING


#region MOVEGENERATION
    #region ADJACENT
def adjacent(occupiedTowers, currBoard, TW):
    if TW == "T":
        adjacentTowers = []
        for tower in occupiedTowers:
            if (tower[0]-2 >= 0 and currBoard[tower[0]-2][tower[1]] == 0 and
(tower[0]-2,tower[1]) not in adjacentTowers):
                adjacentTowers.append((tower[0]-2,tower[1]))
            if (tower[1]-2 >= 0 and currBoard[tower[0]][tower[1]-2] == 0 and
(tower[0],tower[1]-2) not in adjacentTowers):
                adjacentTowers.append((tower[0],tower[1]-2))
            if (tower[0]+2 < numRows and currBoard[tower[0]+2][tower[1]] == 0
and (tower[0]+2,tower[1]) not in adjacentTowers):
                adjacentTowers.append((tower[0]+2,tower[1]))
            if (tower[1]+2 < numCols and currBoard[tower[0]][tower[1]+2] == 0
and (tower[0],tower[1]+2) not in adjacentTowers):
                adjacentTowers.append((tower[0],tower[1]+2))
        adjacentTowers.sort()
        return adjacentTowers
    else:
        adjacentWalls = []
        for tower in occupiedTowers:
            if (tower[0]-1 >= 0 and currBoard[tower[0]-1][tower[1]] == 0 and
(tower[0]-1,tower[1]) not in adjacentWalls):
                if(((tower[0]-1) % 2 == 0 and not tower[1] % 2 == 0) or (not
(tower[0]-1) % 2 == 0 and tower[1] % 2 == 0)):
                    adjacentWalls.append((tower[0]-1,tower[1]))
            if (tower[1]-1 >= 0 and currBoard[tower[0]][tower[1]-1] == 0 and
(tower[0],tower[1]-2) not in adjacentWalls):
                if((tower[0] % 2 == 0 and not (tower[1]-1) % 2 == 0) or (not
tower[0] % 2 == 0 and (tower[1]-1) % 2 == 0)):
                    adjacentWalls.append((tower[0],tower[1]-1))
            if (tower[0]+1 < numRows and currBoard[tower[0]+1][tower[1]] == 0
and (tower[0]+1,tower[1]) not in adjacentWalls):
                if(((tower[0]+1) % 2 == 0 and not tower[1] % 2 == 0) or (not
(tower[0]+1) % 2 == 0 and tower[1] % 2 == 0)):
                    adjacentWalls.append((tower[0]+1,tower[1]))
            if (tower[1]+1 < numCols and currBoard[tower[0]][tower[1]+1] == 0
and (tower[0],tower[1]+1) not in adjacentWalls):
                if((tower[0] % 2 == 0 and not (tower[1]+1) % 2 == 0) or (not
tower[0] % 2 == 0 and (tower[1]+1) % 2 == 0)):
                    adjacentWalls.append((tower[0],tower[1]+1))
        adjacentWalls.sort()
        return adjacentWalls
def unclaimed(board,coords,currPlayer):
    (row,col) = coords
    if currPlayer == "L":
        opponent = "R"
    else:
        opponent = "L"
    if (row+col) % 2 == 0: # (row,col) is a tower; check 4 corners
        if row > 0:
            if col > 0 and board[row-1][col-1] != opponent:
                return True
            if col < numCols-1 and board[row-1][col+1] != opponent:
                return True
        if row < numRows-1:
            if col > 0 and board[row+1][col-1] != opponent:
                return True
            if col < numCols-1 and board[row+1][col+1] != opponent:
                return True
    else:
        if row % 2 == 0:
            if row > 0 and board[row-1][col] != opponent:
                return True
            if row < numRows-1 and board[row+1][col] != opponent:
                return True
        else: # vertical edge
            if col > 0 and board[row][col-1] != opponent:
                return True
            if col < numCols-1 and board[row][col+1] != opponent:
                return True
    return False

def legalTowerMoves(currentTowers, currBoard, maxTowers, maxWalls,currPlayer):
    if len(currentTowers) == 0: #Nothing on the board, anything is valid, only
check one quadrant
        firstMoves = []
        for (row,col) in allTowerPoints:
            if row < (numRows+1)/2 and col < (numCols+1)/2:
                move = [(row,col)]
                firstMoves.append(move,)
                yield(move)
    else:
        firstMoves = []
        tempCurrTowers = currentTowers[:]
        for location in adjacent(tempCurrTowers,currBoard,"T"):
            if unclaimed(currBoard,location,currPlayer):
                move = [location]
                firstMoves.append(move,)
                yield(move)
    m = 1
    mMoves = firstMoves
    while m < maxTowers:
        mPlusOneMoves = []
        for towerList in mMoves:
            towersThisTurn = []
            tempBoard = boardcopy(currBoard)
            for tower in towerList:
                if tower not in tempCurrTowers:
                    tempCurrTowers.append(tower)
                    towersThisTurn.append(tower)
                    tempBoard[tower[0]][tower[1]] = 1
            for adjacentTower in adjacent(tempCurrTowers, tempBoard, "T"):
                if unclaimed(tempBoard,adjacentTower,currPlayer):
                    mPlusOneTurn = sorted(towersThisTurn + [adjacentTower])
                    if mPlusOneTurn not in mPlusOneMoves:
                        mPlusOneMoves.append(mPlusOneTurn,)
                        yield(mPlusOneTurn)
                mMoves = mPlusOneMoves
                m+=1
def requiredWallMoves(CB, CT, towerMoveMade):
    if len(CT) == 0:
        yield([])
    else:
        tempTowersToConnect = towerMoveMade[:]
        tempBoard = boardcopy(CB)
        listOfMoves = []
        for count in range(len(tempTowersToConnect)):
            if tempTowersToConnect:
                tempTowersToConnect.append(tempTowersToConnect.pop(0))
            towersToConnect = tempTowersToConnect[:]
            listOfWalls = []
            currTowers = CT[:]
            while towersToConnect:
                tower = towersToConnect[0]
                reqWalls = []
                if (((tower[0]-1 >= 0 and tempBoard[tower[0]-1][tower[1]] == 0)
or (tower[0]-1 < 0))
                and ((tower[1]-1 >= 0 and tempBoard[tower[0]][tower[1]-1] == 0)
or (tower[1]-1 <0))
                and ((tower[0]+1 < numRows and tempBoard[tower[0]+1][tower[1]]
== 0) or (tower[0]+1 >= numRows))
                and ((tower[1]+1 < numCols and tempBoard[tower[0]][tower[1]+1]
== 0) or (tower[1]+1 >= numCols))):
                    if(tower[0] - 2 >= 0 and (tower[0]-2,tower[1]) in
currTowers):
                        reqWalls.append((tower[0]-1,tower[1]))
                        currTowers.append(tower)
                        if tower in towersToConnect:
                            towersToConnect.remove(tower)
                    if(tower[1] - 2 >= 0 and (tower[0],tower[1]-2) in
currTowers):
                        reqWalls.append((tower[0],tower[1]-1))
                        currTowers.append(tower)
                        if tower in towersToConnect:
                            towersToConnect.remove(tower)
                    if(tower[0] + 2 < numRows and (tower[0]+2,tower[1]) in
currTowers):
                        reqWalls.append((tower[0]+1,tower[1]))
                        currTowers.append(tower)
                        if tower in towersToConnect:
                            towersToConnect.remove(tower)
                    if(tower[1] + 2 < numCols and (tower[0],tower[1]+2) in
currTowers):
                        reqWalls.append((tower[0],tower[1]+1))
                        currTowers.append(tower)
                        if tower in towersToConnect:
                            towersToConnect.remove(tower)
                    tempBoard[tower[0]][tower[1]] = 1
                    if tower in towersToConnect:
                        towersToConnect.remove(tower)
                        towersToConnect.append(tower)
                else:
                    tempBoard[tower[0]][tower[1]] = 1
                    currTowers.append(tower)
                    towersToConnect.remove(tower)
                if(reqWalls):
                    listOfWalls.append(reqWalls)
            for wallSet in itertools.product(*listOfWalls):
                    wallSet = list(set(wallSet))
                    wallSet.sort()
                    if wallSet not in listOfMoves:
                        yield(wallSet)
                        listOfMoves.append(wallSet)
def legalWallMoves(currBoard, towerMove, requiredWalls, maxWalls,currPlayer):
    tempCurrTowers = towerMove[:]
    currWalls = requiredWalls[:]
    firstMoves = []
    for (row,col) in allTowerPoints:
        if currBoard[row][col] == 1:
            tempCurrTowers.append((row,col))
    for (row,col) in allWallPoints:
        if currBoard[row][col] == 1:
            currWalls.append((row,col))
    tempBoard = boardcopy(currBoard)
    for wall in currWalls:
        tempBoard[wall[0]][wall[1]] = 1
    for tower in tempCurrTowers:
        tempBoard[tower[0]][tower[1]] = 1
    maxWallMoves = maxWalls - len(requiredWalls)
    if maxWallMoves > 0:
        tempCurrWalls = currWalls[:]
        for location in adjacent(tempCurrTowers, tempBoard, "W"):
            if unclaimed(tempBoard,location,currPlayer):
                move = [location]
                yield(move)
                firstMoves.append(move,)
        m = 1
        mMoves = firstMoves
        while m < maxWallMoves:
            m+=1
            mPlusOneMoves = []
            for wallList in mMoves:
                wallsThisTurn = []
                moveTempBoard = boardcopy(tempBoard)
                for wall in wallList:
                    if wall not in tempCurrWalls:
                        tempCurrWalls.append(wall)
                        wallsThisTurn.append(wall)
                        moveTempBoard[wall[0]][wall[1]] = 1
                for adjacentWall in adjacent(tempCurrTowers, moveTempBoard,
"W"):
                    if unclaimed(tempBoard,adjacentWall,currPlayer):
                        mPlusOneTurn = sorted(wallsThisTurn + [adjacentWall])
                        tempCurrWalls = currWalls[:]
                        if mPlusOneTurn not in mPlusOneMoves:
                            mPlusOneMoves.append(mPlusOneTurn,)
                            yield(mPlusOneTurn)
                mMoves = mPlusOneMoves
def moveGenerator(currBoard, maxTowers, maxWalls,currPlayer):
    currTowers = []
    for (row,col) in allTowerPoints:
        if currBoard[row][col] == 1:
            currTowers.append((row,col))
    for towerMove in
legalTowerMoves(currTowers,currBoard,maxTowers,maxWalls,currPlayer):
        for reqWallMove in requiredWallMoves(currBoard,currTowers, towerMove):
            yield towerMove+reqWallMove
            for wallMoves in legalWallMoves(currBoard,
towerMove,reqWallMove,maxWalls,currPlayer):
                yield towerMove + reqWallMove + wallMoves
    for wallMoves in legalWallMoves(currBoard, [], [], 1,currPlayer):
        yield wallMoves
```

```
def makeMove(preMoveBoard, move,player):
    board = boardcopy(preMoveBoard)
    for (row,col) in move:
        board[row][col] = 1
    regionsToCheck = []
    for (row,col) in move:
        if (row+col) % 2 == 0:
            if row > 0:
                if col > 0 and board[row-1][col-1] == 0:
                    regionsToCheck.append((row-1,col-1),)
                if col < numCols-1 and board[row-1][col+1] == 0:
                    regionsToCheck.append((row-1,col+1),)
            if row < numRows-1:
                if col > 0 and board[row+1][col-1] == 0:
                    regionsToCheck.append((row+1,col-1),)
                if col < numCols-1 and board[row+1][col+1] == 0:
                    regionsToCheck.append((row+1,col+1),)
        else:
            if row % 2 == 0:
                if board[row][col-1] == 1 and board[row][col+1] == 1:
                    if row > 0 and board[row-1][col] == 0:
                        regionsToCheck.append((row-1,col),)
                    if row < numRows-1 and board[row+1][col] == 0:
                        regionsToCheck.append((row+1,col),)
            else:
                if board[row-1][col] == 1 and board[row+1][col] == 1:
                    if col > 0 and board[row][col-1] == 0:
                        regionsToCheck.append((row,col-1),)
                    if col < numCols-1 and board[row][col+1] == 0:
                        regionsToCheck.append((row,col+1),)
    regionsToCheck = list(set(regionsToCheck))
    while regionsToCheck:
        regionPoint = regionsToCheck[0]
        region = fill(board,[regionPoint])
        if region[0] != "Unbounded":
            for (row,col) in region:
                board[row][col] = player
        regionsToCheck = [coords for coords in regionsToCheck if coords not in
region]
    return board

def computeScore(board):
    score = 0
    uncheckedRegionPoints = allRegionPoints[:]
    while uncheckedRegionPoints:
        regionPoint = uncheckedRegionPoints[0]
        region = fill(board,[regionPoint])
        if region[0] != "Unbounded":
            towers = []
            (row,col) = region[0]
            if board[row][col] == "L":
                multiplier = 1
            else:
                multiplier = -1
            for (row,col) in region:
                if board[row-1][col-1] == 1:
                    towers.append((row-1,col-1),)
                if board[row-1][col+1] == 1:
                    towers.append((row-1,col+1),)
                if board[row+1][col-1] == 1:
                    towers.append((row+1,col-1),)
                if board[row+1][col+1] == 1:
                    towers.append((row+1,col+1),)
                score += multiplier * len(set(towers))
        uncheckedRegionPoints = [coords for coords in uncheckedRegionPoints if 
coords not in region]
    return score
#next block was from http://stackoverflow.com/questions/661603/how-do-i-know-
if-a-generator-is-empty-from-the-start
def peek(iterable):
    try:
        first = next(iterable)
    except StopIteration:
        return None
    return first, itertools.chain([first], iterable)

def boardcopy(board):
    newBoard = []
    for row in board:
        newBoard.append(row[:],)
    return newBoard

def boardToTuple(board):
    return tuple([tuple(row) for row in board])

def miniMax(currBoard, currPlayer, maxTowers, maxWalls,depth,seq):
    tupBoard = boardToTuple(currBoard)
    if currPlayer == "L":   #Maximizing player
        if tupBoard in Llookup:
            return Llookup[tupBoard]
        bestScore = -(9999)
        bestSeq = []
        if peek(moveGenerator(currBoard,maxTowers,maxWalls,"L")) is None:
            if peek(moveGenerator(currBoard,maxTowers,maxWalls,"R")) is None:
                score = computeScore(currBoard)
                return (score,seq,currBoard)
            else:
                return miniMax(currBoard, "R", maxTowers, 
maxWalls,depth+1,seq+[[],])
        else:
            for move in moveGenerator(currBoard,maxTowers,maxWalls,currPlayer):
                if depth == 0:
                    print ("Checking first move",move)
                nextBoard = makeMove(currBoard,move,currPlayer)
                val = miniMax(nextBoard, "R", maxTowers, 
maxWalls,depth+1,seq+[move])
                if val[0] > bestScore:
                    bestScore = val[0]
                    bestSeq = move
            Llookup[tupBoard] = (bestScore,bestSeq)
            return (bestScore,bestSeq,currBoard)
    else: #Minimizing player
        if tupBoard in Rlookup:
            return Rlookup[tupBoard]
        bestScore = (9999)
        bestSeq = []
        if peek(moveGenerator(currBoard,maxTowers,maxWalls,"R")) is None:
            if peek(moveGenerator(currBoard,maxTowers,maxWalls,"L")) is None:
                score = computeScore(currBoard)
                return (score,seq,currBoard)
            else:
                return miniMax(currBoard, "L", maxTowers, 
maxWalls,depth+1,seq+[[],])
        else:
            for move in moveGenerator(currBoard,maxTowers,maxWalls,currPlayer):
                if depth == 0:
                    print ("Checking first move",move)
                nextBoard = makeMove(currBoard, move,currPlayer)
                val = miniMax(nextBoard, "L", maxTowers,
maxWalls,depth+1,seq+[move])
                if val[0] < bestScore:
                    bestScore = val[0]
                    bestSeq = move
            Rlookup[tupBoard] = (bestScore,bestSeq)
            return (bestScore,bestSeq,currBoard)

def getSequence(board,currPlayer):
    if (currPlayer == "L"):
        tupleBoard = boardToTuple(board)
        if (tupleBoard in Llookup):
            val = Llookup[tupleBoard]
            nextBoard = makeMove( board, val[1], "L" )
            print(val[1])
            getSequence( nextBoard, "R" )
    else:
        tupleBoard = boardToTuple(board)
        if (tupleBoard in Rlookup):
            val = Rlookup[tupleBoard]
            nextBoard = makeMove( board, val[1], "R" )
            print(val[1])
            getSequence( nextBoard, "L" )


Llookup = dict()
Rlookup = dict()
for row in IB:
    print (row)
val = miniMax(IB,"L",1,1,0,[])[0]
print ("Value:",val)
getSequence(IB,"L")
```

Appendix B Original Rules of *Castellan*

To understand the rules of the simplified version of *Castellan* that we have presented, it is necessary to know the original rules as well. *Castellan* is normally played with the following pieces: 26 Long Walls, 30 Short Walls, 32 Towers, 10 Blue Keeps, 10 Red Keeps, 28 cards (14 for each player). Each player's set of cards are identical except for color. Walls and towers are not color coded as the cards are such that the number of pieces are precisely accounted for.

Each player in turn will play cards to get walls and towers to add to the growing castle. You are trying to completely enclose areas (regions) without helping your rival do the same.

You choose how many cards to play each turn. The more cards you play, the more pieces you get, but if you play too many cards at once, you limit your options for later turns.

When you complete an enclosure, you "claim" it with a keep in your color. At the end of the game, count the towers around each player's courtyards. The one with the most towers wins [12].

Game play must be in a manner so as to ensure the game is connected at all times. A player may choose to subdivide their own enclosed areas but not the other players enclosed regions. Towers and walls connect by the circular nub of a tower fitting into the circular groove of a wall. Walls may not be placed in a way such that a tower cannot be connected to it. If a player makes an earnest mistake it is allowed to be corrected by the other player.

Before play begins each player divides their deck of cards into two piles, one for tower cards and one for wall cards. Both piles are shuffled and each player draws two cards from each pile. After hands are drawn each turn commences in the following manner. The current player will first choose what cards to play, and they must play at least one. Cards are then placed face up on and pieces corresponding to the pictures on the cards are collected. Next, all pieces gathered must be played on the current structure, ensuring the structure is connected after all pieces are played. After the pieces have all been played the player puts a keep in each new courtyard enclosed. Once per game a player may decide to place two keeps in a single courtyard to double the scoring of that courtyard. Once keeps are placed the player draws one card – they may draw additional cards if they played any with a draw symbol on it – and discards their used cards. When one player is out of cards (no more in hand or left to draw), the other player draws all their remaining cards and takes one last turn. The gray pieces should come out even; there are exactly as many pieces in the set as there are symbols on the cards.

Finally, the game is scored by counting the towers of the regions enclosed by each player. Whoever has the higher score is the victor.

I, Jason Robert Suptic, hereby submit this thesis/report to Emporia State University as partial fulfillment of the requirements for an advanced degree. I agree that the Library of the University may make it available to use in accordance with its regulations governing materials of this type. I further agree that quoting, photocopying, digitizing or other reproduction of this document is allowed for private study, scholarship (including teaching) and research purposes of a nonprofit nature. No copying which involves potential financial gain will be allowed without written permission of the author. I also agree to permit the Graduate School at Emporia State University to digitize and place this thesis in the ESU institutional repository.

Signature of Author

Date

Non-Losing Strategies for *Castellan*
Title of Thesis

Signature of Graduate School Staff

Date Received