BLOB: A TWO PLAYER STRATEGY GAME

By

Adam Holers

A SENIOR RESEARCH PAPER PRESENTED TO THE DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE OF STETSON UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF SCIENCE

STETSON UNIVERSITY

2006

ACKNOWLEDGMENTS

I would like to think Dr. Erich Friedman for his guidance on this analysis, my family for their support, and the Stetson faculty for my training.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

BLOB: A TWO PLAYER STRATEGY GAME

By

Adam Holers

May 2007

Advisor: Dr. Erich Friedman

Department: Mathematics and Computer Science

Blob is a two player strategy game played on an infinite triangular, square, or hexagonal grid. A move ratio ***p*** is used to fix a sequence of moves directing the order in which the two players, white and black, move. White is trying to create the largest connected region of shapes (blob) possible while black is trying to border off and contain white's region while keeping its size to a minimum. Multiple versions exist that place restrictions on the locations black and white can move.

Various questions are being addressed. For given versions, what value of ***p*** (referred to as the **critical ratio**) does white require to create an infinite blob? What ***p*** value is required by black to contain a blob to a given size? Are there any patterns relating blob size to ***p***? Geometrically, what is the minimum and maximum number of moves black requires to contain a white blob assuming any given blob shape?

The methods used to analyze these situations include constructing computer generated game trees and direct proof. The critical ratio is bounded from above and below for the different versions.

CHAPTER 1

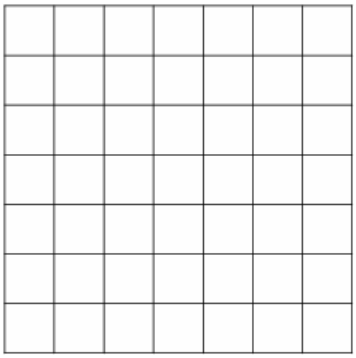
HOW TO PLAY A GAME

1.1. BEGINNING A GAME

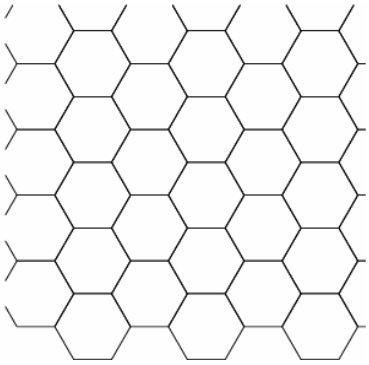
To start a game, the two players agree to play on a grid type (square, triangular, or hexagonal), decide who is playing which color (white and black), agree on a particular version (where white and black are allowed to move), and set a move ratio (that describes the order the two players move). The two players then take turns placing counters on the grid according to the sequence of moves generated from the move ratio. White tries to create the largest region of connected spaces possible but must connect the blob to the original move. Black tries to border off and contain white's region keeping its size to a minimum.

1.1.1 GRIDS

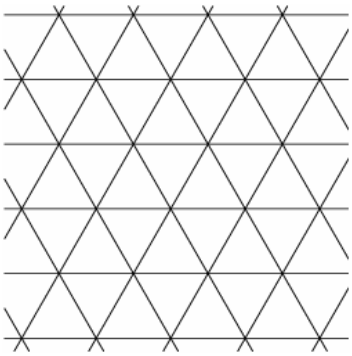
The three grid choices are triangular, hexagonal, and square and are illustrated below. The grids illustrated are finite, but when the game is played the grids are treated as if they were infinite.



(figure 1.1.1.a Square Grid)



(figure 1.1.1.b Hexagonal Grid)



(figure 1.1.1.c Triangular Grid)

1.1.2 COLOR

The two players decide who is playing white and who is playing black. Each player gets an unlimited number of counters of their color to place on the selected grid. White always makes the first move.

1.1.3 VERSIONS

There are up to five versions for the different grids. The thing that separates the versions is the restrictions on white and blacks movement. The terms used to describe movement are connected, adjacent, and attached. Two spaces are **connected** if they share a side, **adjacent** if they share exactly 1 vertex and **attached** if they share at least one vertex. Note that all connected and adja1cent spaces are also attached.

On each grid there are 5 versions with the corresponding rules as listed below.

Version I - Black and white's moves are unrestricted.

Version II - Black's moves are unrestricted. White must move to positions attached to other white positions.

Version III - White must move to positions that result in connections with other white positions. Black's moves are unrestricted.

Version IV - Both black and white must move to positions attached to other white positions.

Version V - White must move to positions connected to other white positions. Black must move to positions attached to white positions.

Note: On the hexagonal grid, all attached hexagons are also connected. Thus, on the hexagonal grid, versions II and III are the same, as are versions IV and V.

1.1.4 THE MOVE RATIO

The move ratio ***p***, where $0 < \boldsymbol{p} < 1$, is used in the following equation to determine who moves on the ***n***th move.

$$\lceil pn \rceil - \lceil p(n-1) \rceil, \text{ where } \lceil\ \rceil \text{ denotes the ceiling function} \quad (1)$$

The above equation evaluates to either 0, which means black moves on the ***n***th turn, or 1, which means white moves on the ***n***th turn. This gives white a proportion of the moves equal to ***p***. For instance a ***p*** value of $\frac{1}{2}$ produces the move sequence w,b,w,b,w,b,etc. A move ratio of $\frac{2}{3}$ produces the move sequence w,w,b,w,w,b,w,w,b,etc. A move ratio of $\frac{2}{5}$ produces the move sequence w,b,w,b,b,w,b,w,b,b,w,etc. Notice that sometimes black gets one move and other times black gets two moves. A move ratio of $\frac{1}{\pi}$ produces a move sequence with no repeating pattern starting w,b,b,w,b,b,b,w,b,etc.

1.2. THE RESULTING GAME

One of two things can happen with any given game. Black either contains white, or black gives up, conceding white can make an infinite blob. **Contained** means every square connected to white's blob is occupied. If white's blob becomes contained, the game is over and the number of spaces in white's blob is counted.

CHAPTER 2

THE FUNCTION BLOB_(***P***)

2.1 STRATEGIES

Consider a game on the square grid Version V. The grids are infinite so white's first move is always at an arbitrary square. Black must then choose the best square in response. Where should black move and where should white move after that? What makes a move good? Intuitively, the best moves for black are the moves that lead to the smallest blob size possible and the best moves for white are the moves that lead to the largest blob size possible.

For a simple example consider $\boldsymbol{p} = \frac{1}{4}$. This produces a sequence of moves w,b,b,b,w,b,b,b,etc. If black moves to three connected squares surrounding white's first move white's second move is forced. If white's second move is not connected to the first white square, black may contain the first move the following turn by moving to the fourth connect square. Black's next turn, by moving to three connected squares again, black will contain white allowing white a blob of size two.

(Figure 2.1.1.a Square Blob of Size Two)

A strategy is called a **best strategy** if it minimizes potential losses against all opposing strategies. The concept of a best strategy is very important to Blob so it will be given a formal treatment.

Considering all the different strategies white or black may employ, let *X* represent the set of all black strategies and *Y* represent the set of all white strategies. Define *M*(*a*, *b*) to be the size of the blob resulting from the execution of two strategies *a* and *b*. Let $y \in Y$ and $x \in X$.

$$\text{Define } v_b(x) = \max_y M(x, y) \text{ and } v_w(y) = \min_x M(x, y).$$

The functions $v_b(x)$ and $v_w(y)$ are called black and white's security levels for strategies *x* and *y* respectively. For $v_b(x)$, if black disclosed his strategy *x* to white, and white was allowed to pick the best strategy *y* to counter black's strategy *x,* the size of the resulting blob would be the value of $v_b(x)$. Likewise, if white picked a particular strategy *y* and allowed black to pick the best strategy *x* to counter white's *y*, the size of the resulting blob would be the value of $v_w(y)$. Another way to think of the two quantities $v_b(x)$ and $v_w(y)$ is as a measure of potential losses for each player when choosing a specific strategy.

If both players seek the best security level, black will choose a strategy x^0 where $v_b(x^0) \le v_b(x)$ for all $x \in X$. Likewise, white will pick a strategy y^0 where $v_w(y^0) \ge v_w(y)$ for all $y \in Y$. Notice the reversal of the inequalities because the two players have competing objectives. These strategies with the best security levels for the two players are known as their **maximin strategies** or **best strategies**. [1]

The **Minimax Theorem** states that every **finite**, **zero-sum**, two-person game has optimal **mixed strategies**. [1] Blob is not necessarily a finite game, but it is zero-sum. Zero-sum means one player's gain is equivalently another player's loss. The growth of the blob by one position is black's loss and white's gain. For the purposes of blob, mixed strategies are just strategies. It will be shown for certain versions and ***p*** values that black can force a finite game. Blob can be treated as a finite game for these versions and values guaranteeing best strategies for white and black.

2.2 BLOB_(***P***)

The functions BlobH(***p***), BlobS(***p***), and BlobT(***p***) are defined as the size of the blob resulting from a sequence of moves from best strategies on the hexagonal, square, and triangular grids. If white can make an unbounded Blob we define Blob_(***p***) = ∞.

2.3 THE CRITICAL MOVE RATIO

When white has enough moves, it seems obvious white should be able to make an infinite blob. This is exactly the case. There are values of ***p*** for which Blob_(***p***) is finite and there are values of ***p*** for which Blob_(***p)*** is infinite. The infimum of all ***p*** values where Blob_(***p***) evaluates to infinity is defined as the **critical move ratio,** denoted p_c.

CHAPTER 3

THE SQUARE GRID

3.1 GEOMETRY OF THE SQUARE GRID

Now the game will be explored on the square grid.

3.1.1 MINIMAL SQUARE ENVELOPE

The following terminology is defined to study black's containment of white's blob.

Definition: A blob is said to be **enveloped** when every position attached to that blob is occupied.

Definition: The smallest number of black moves required to envelope a square blob of size ***n*** is known as the **minimal square envelope**.

Definition: The smallest number of black moves required to contain a square blob of size ***n*** is known as the **minimal square containment**.

Theorem 1: The Minimal Square Envelope Theorem:

The size of the minimal square envelope is 2***n*** + 6.

Proof:

Say a blob has ***n*** squares and white is placing a move connected to that blob. All envelopes occupy every square attached to a blob. So, white is placing a move on a square that would make up a minimal envelope of the blob of size ***n.*** White's placement can be connected to 1, 2, 3, or 4 white squares. The following shows regardless of how many white squares the placement is connected to, the difference between the size of the minimal envelopes for a blob of size ***n*** and a blob of size ***n*** + 1 will be at most 2.

(Figure 3.1.1.a Necessary Envelope Additions)

Figure 1 above illustrates the placement of a white move on the red square which is connected to at least 1 other white square. Here the green and grey squares may be unoccupied, white, or black. Thinking of the red square as part of the minimal envelope of a blob of size ***n***, if white places a move on that red square at most 2 additional black moves are required to envelope the blob of size ***n*** + 1 (3 green squares minus 1 red square). If any of the green squares are occupied by black or white, the number of black squares required to envelope the blob decreases by 1 for each green square occupied. So an increase of 2 occurs only when the placement is connected to 1 other white square and the 3 other connected squares are unoccupied.

It takes 8 moves to envelope a blob of size 1. Adding one white square to this blob can only increase the size of the square envelope by 2. Thus the smallest number of squares required to envelope a blob of size n is $2(\boldsymbol{n}-1) + 8 = 2\boldsymbol{n} + 6$. ■

3.1.2 MINIMAL SQUARE CONTAINMENT

Theorem 2: Minimal Square Containment Theorem:

The size of the minimal containment for a blob of size ***n*** is 2***n*** + 2.

Proof:

Say a blob has ***n*** squares and white is placing a move connected to that blob. All containments occupy every square connected to a blob. So, white is placing a move on a square that would make up a minimal containment of the blob of size ***n.*** White's placement can be connected to 1, 2, 3, or 4 white squares. The following shows regardless of how many white squares the placement is connected to, the difference between the size of the minimal containments for a blob of size ***n*** and a blob of size ***n*** + 1 will be at most 2.

(figure 3.1.2.a Necessary Containment Additions)

Figure 1 above illustrates the placement of a white move on the red square which is connected to at least 1 other white square. Here the green and grey squares may be unoccupied, white, or black. Thinking of the red square as part of the minimal containment of a blob of size ***n***, if white places a move on that red square at most 2 additional black moves are required to contain the blob of size ***n*** + 1 (3 green squares minus 1 red square). If any of the green squares are occupied by black or white, the

number of black squares required to contain the blob decreases by 1 for each green square occupied. So an increase of 2 occurs only when the placement is connected to 1 other white square and the 3 other connected squares are unoccupied.

If either of the 2 grey squares above and below the white square are occupied by white, the size of the containment will not increase by 2. Thus, a blob requiring the largest containment will not bend.

It takes 4 moves to contain a blob of size 1. Adding one white square to this blob can only increase the blob size by 2. Thus the smallest number of squares required to contain a blob of size n is 2(***n***-1) + 4 = 2***n*** + 2. ■

Lemma 1:

If BlobS(***p***) = ***n***, ***p*** $\geq \left(\frac{n}{3n+2} \right)$.

Proof:

Black may move to squares connected to white squares. There are at most $2n+2$ squares connected to a blob of size ***n*** by theorem 2. Thus black may occupy every connected square for a blob of size ***n*** if ***p*** is than or equal to $\left(\frac{n}{3n+2} \right)$. ■

3.2.1 SMALL ***P*** VALUES ON THE SQUARE GRID

Theorem 3:

For all $\boldsymbol{p}$ where $0 < \boldsymbol{p} \leq \frac{1}{5}$, BlobS($\boldsymbol{p}$) = 1.

Proof:

On the square grid with four moves, black can contain any individual white move by moving to the 4 connected squares. ■

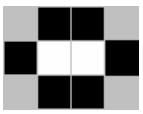
(figure 3.2.1.a Contained Square Blob of Size One)

Theorem 4:

For all $\boldsymbol{p}$ where $\frac{1}{5} < \boldsymbol{p} \leq \frac{1}{4}$, BlobS($\boldsymbol{p}$) = 2.

Proof:

With three moves, black can occupy all but one square connected to any white move. If this move is white's blob, white may play one move connected to that blob and black can then contain the blob with 3 more moves.



(Figure 3.2.1.b Contained Square Blob of Size Two)

For the first example values of $\boldsymbol{p}$ less than $\frac{1}{5}$ could allow black a large number of moves until white moves again, but 4 was all that was required to contain the blob. In the second example, some values of $\boldsymbol{p}$ less than $\frac{1}{4}$ produce the sequence of moves w,b,b,b,w,b,b,b,b,w where black gets 4 moves in the second sequence before white moves again. These were more moves than required to contain the blob. In fact any move ratio less than $\frac{1}{4}$ is more than black requires to contain white. Also, with any ratio larger than $\frac{1}{5}$, black can not contain the blob to size one and with any ratio larger than $\frac{1}{4}$ and black cannot contain the blob to size 2. These values of $\boldsymbol{p}$ where BlobS($\boldsymbol{p}$) changes value are called **steps**. ■

Theorem 5:

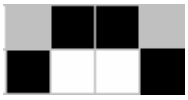
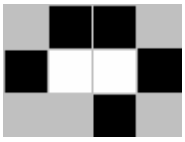
For all $\boldsymbol{p}$ where $\frac{1}{4} < \boldsymbol{p} \leq \frac{3}{10}$, BlobS($\boldsymbol{p}$) = 3.

Proof:

$\boldsymbol{p}$ values in this range produce move sequences starting w,b,b,w,b,b,w,b,b,b, or w,b,b,w,b,b,b,w,b,b,w, or w,b,b,w,b,b,b,w,b,b,b. With any sequence, Black may occupy two squares connected to white's first move as below

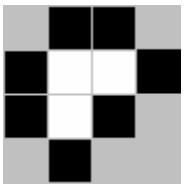
(figure 3.2.1.c Two Connected Black Placements)

For white's next move, if white does not move to one of the two unoccupied squares connected to white's first move, black may contain white's blob. Thus white is forced to play connected to the first move to build a blob larger than one. After one of white's moves, which are the same by symmetry, black may occupy two or three connected squares depending on the move sequence.

 , 

(figure 3.2.1.d Sequential Connected Black Placements)

White may play one connected move by symmetry or black may contain the blob. The following turn, black may contain the blob



(figure 3.2.1.e Contained Square Blob of Size Three)

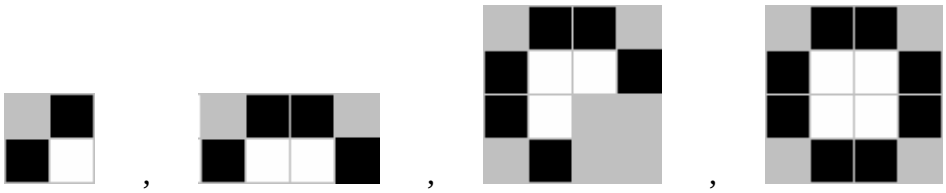
Six black moves are required to contain a blob of size two. All $\boldsymbol{p}$ values greater than $\frac{1}{4}$ do not provide black six moves before white moves a third time. Thus, for all $\boldsymbol{p}$ where $\frac{1}{4} < \boldsymbol{p} \leq \frac{3}{10}$, BlobS($\boldsymbol{p}$) = 3. ■

Theorem 6:

For all $\boldsymbol{p}$ where $\frac{3}{10} < \boldsymbol{p} \leq \frac{1}{3}$, BlobS($\boldsymbol{p}$) = 4.

Proof:

All $\boldsymbol{p}$ values where $\frac{3}{10} < \boldsymbol{p} \leq \frac{1}{3}$ start with the move sequence w,b,b,w,b,b,w,b,b,w. After this a third move may or may not be provided to black before white moves again. The following sequences of moves illustrated below result from symmetry and may be forced



(figure 3.2.1.f Sequential Connected Placements for $\boldsymbol{p} \leq \frac{1}{3}$)

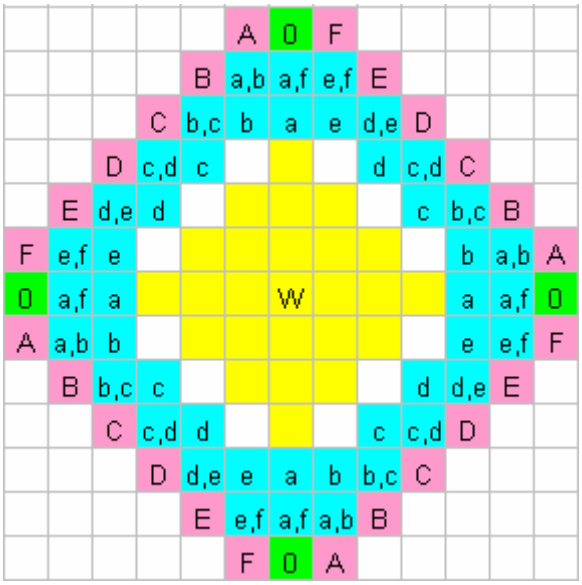
The various arrangements of a blob of size 3 each require at least 7 black moves to contain to blob. The move sequences for $\boldsymbol{p}$, where $\frac{3}{10} < \boldsymbol{p} \leq \frac{1}{3}$, do not provide a seventh move before white moves again. Thus, for all $\boldsymbol{p}$ where $\frac{3}{10} < \boldsymbol{p} \leq \frac{1}{3}$, BlobS($\boldsymbol{p}$) = 4. ■

3.2.2 MIDDLE ***P*** VALUES ON THE SQUARE GRID

Theorem 7: **The Little Diamond Theorem:**

For square version III, $\text{BlobS}\left(\frac{1}{2}\right) \leq 55$, and in particular, $BlobS\left(\frac{1}{2}\right) < \infty$

Proof:



(figure 3.2.2.a Little Diamond)

White's first four moves could not have moved outside the yellow region. Black's first four moves are at the green squares marked 0. White may move onto lettered squares or non-lettered squares. Blue squares direct specific responses by black to the corresponding pink squares marked with the same letter. To save space in the picture, the same letters were used multiple times but only indicate responses nearby. If white does not move onto a lettered square or black has already made a directed response, black

places moves on the pink border until it is completed. The following shows white will never be able to move onto the pink and green border.

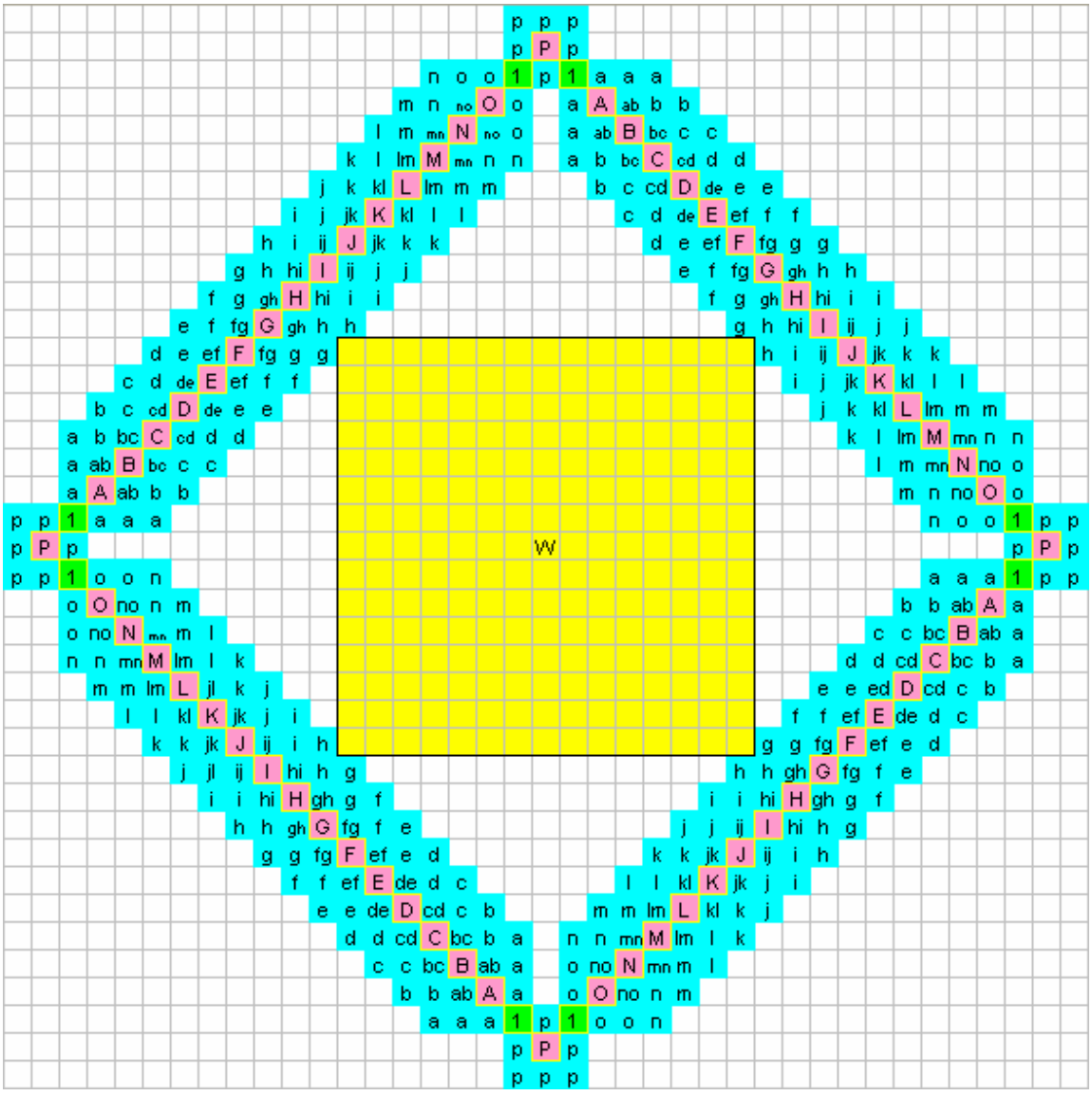
White can move onto pink border squares from double lettered squares and white can move onto double lettered squares from single lettered squares or other double lettered squares. The yellow region is enveloped by single lettered squares so white can not move onto a double lettered square without first moving onto a single lettered square. All single and double lettered squares connected to double lettered squares direct one of the responses directed by the double lettered square. Thus, when moving onto a double lettered square, black will have already made at least one of the moves directed. Any double lettered square directs responses to its connected pink border squares except squares marked a,f which are not connected to pink squares. Thus white can never move onto a pink square or white can not move onto the containment.

There are 28 border squares and 81 squares inside the containment. After the border is complete, white will have 28 squares inside the containment occupied. This leaves 53 squares unoccupied in the containment. White may take 27 and black may take 26 allowing white a blob of 28 + 27 = 55 squares. This is also a bound for all ***p*** less than $\frac{1}{2}$. ■

Theorem 8: **The Big Diamond Theorem:**

For Version II $\text{BlobS}\left(\frac{1}{2}\right) \le 343$, and in particular, $BlobS\left(\frac{1}{2}\right) < \infty$.

Proof:



(figure 3.2.2.b Big Diamond)

White's first move is made at an arbitrary square labeled W. White's next seven moves can not leave the yellow square surrounding W. Black's first eight moves are the

strategic moves placed at the green squares marked 1. The pink and green squares will make up black's containment. Any time white moves onto a blue square, a response for black is directed by a lower case letter. Black makes this response in the pink square with the corresponding capital letter. Blue squares with two letters direct two responses but it will be shown at least one of these responses will have already been made. Black ignores any directions already made. If white's move does not direct black's movement, black moves on the border until it is completed. To save space in the picture, single character labels were used multiple times, but only direct responses nearby.

Considering any pink border square, the eight attached squares are combinations of other border squares, single lettered squares, and double lettered squares. Blue single lettered squares are either adjacent or not attached to pink squares. If white moves onto a single lettered square black responds at the pink border square indicated blocking movement onto any attached pink border square from that single lettered square. Thus, white cannot move from a single lettered square onto a border square.

Considering any double lettered square, again, the eight attached squares are combinations of border squares, single lettered squares, double lettered squares. The double lettered squares are enveloped by single lettered squares as is white's first move. If white moves onto double lettered squares, white must first move onto a single lettered square due to the envelope. Any single lettered squares attached to a double lettered square direct one of the moves directed by that double lettered square. Thus, in moving onto a double lettered square from a single lettered square, black will only be directed to make one of the two moves. These directions are always the border squares connected to

the double lettered square. So, white cannot move onto a border from a double lettered square moved to from a single lettered square.

If white has moved onto a double lettered square from a single lettered square, white can then move from that double lettered square to other adjacent double lettered squares. Any double lettered square attached to another double lettered square each direct at most one different move. So, in moving from a double lettered square to another, at most one new move will be directed. Again, these directions are always the connected border squares. Thus, white cannot move onto a border square from a single lettered square and white cannot move onto a border square from a double lettered square and finally white cannot move onto the border.

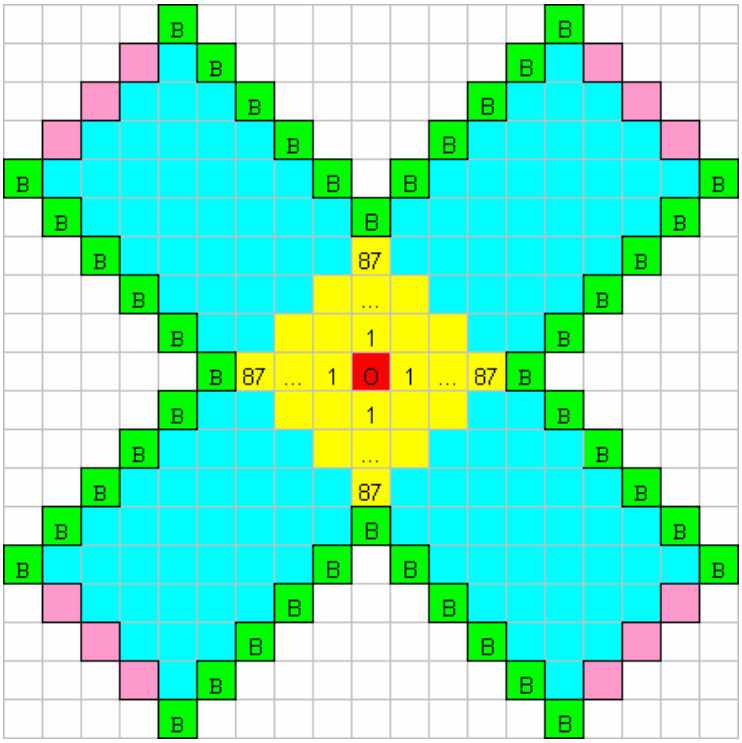
There are 72 squares on the pink border and 615 contained inside the border. By the time the border is completed, white can have 72 squares inside the diamond. This leaves 543 unoccupied squares inside the diamond. White can take 271 of them and black can take 270 since white moves first. This creates a contained blob of at most 271 + 72 = 343 squares. This is also a bound for all $\boldsymbol{p}$ less than $\frac{1}{2}$. ■

Theorem 9: The Giant Diamond Theorem:

When $\boldsymbol{p} = \frac{2}{3}$ for square version III and white's 1st and 2nd moves must be connected $\text{BlobS}\left(\frac{2}{3}\right) \leq$ 20760 or, in particular, $\text{BlobS}\left(\frac{2}{3}\right)$ is finite.

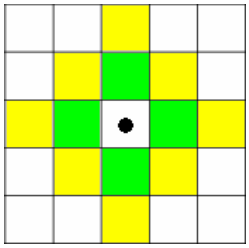
Proof:

White's first move is arbitrary marked in red below. Black's first 44 moves are at the green border squares B as diagrammed below. For purposes of displaying the containment, the distances from white's first move and the containment are compressed. White's first 88 moves could not leave the yellow region allowing black 44 moves to fill in the green border squares. When white moves onto the squares in blue, black responds specifically to corresponding pink border squares as explained below.



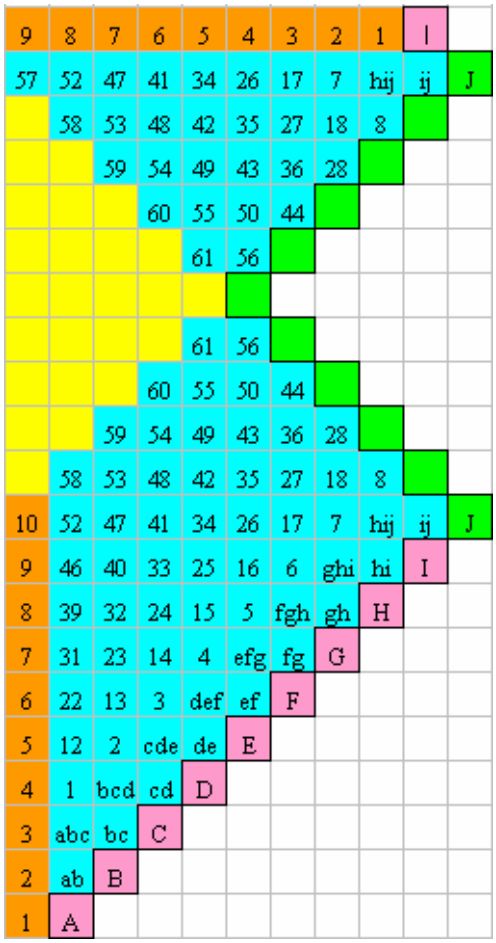
(figure 3.2.2.c Giant Diamond)

There are different ways white may approach the border. When $\boldsymbol{p} = \frac{2}{3}$, white receives two moves before black moves again which are called the **first** and **second moves**. The possible locations white may move to from any location are diagrammed below where the dot represents a white location, green represents white's **first move** and yellow represents white's **second move**.



(figure 3.2.2.d White's possible moves)

The figure below is used with the following charts instructing how Black is to move. Lowercase letters direct responses to corresponding capital lettered pink border squares. The numbers in the blue squares refer to locations in the enumerated tables where lists of lowercase letters can be found. Black uses the lists of lowercase letters in the enumerated tables when picking an appropriate move. Note that orange locations represent diagonal columns or **levels**, pink and green locations represent border squares, yellow locations represent white's distance from the origin, and blue locations represent where black is required to move specifically if white moves on these locations.



(figure 3.2.2.e White Responses)

1	bc	acd	abd
2	cd	bde	bce
3	de	cef	cdf
4	ef	dfg	deg
5	fg	egh	efh
6	gh	fhi	fgi
7	hi	gij	ghj
8	ij	hjk	hik

(Table 3.2.2.a Level 3)

12	bcd	acd	bce
13	cde	bde	cdf
14	def	cef	deg
15	efg	dfg	efh
16	fgh	egh	fgi
17	ghi	fhi	ghj
18	hij	gij	hik

(Table 3.2.2.b Level 4)

22	cd	bd	ce	ade	bcf	adf	acf
23	de	ce	df	bef	cdg	beg	bdg
24	ef	df	eg	cfg	deh	cfh	ceh
25	fg	eg	fh	dgh	efi	dgi	dfi
26	gh	fh	gi	ehi	fgj	ehj	egj
27	hi	gi	hj	fij	ghk	fik	fhk
28	ij	hj	ik	gjk	hil	gjl	gil

(Table 3.2.2.c Level 5)

31	ce	cdf	bde	bef	bcf	cdg	ade
32	df	deg	cef	cfg	cdg	deh	bef
33	eg	efh	dfg	dgh	deh	efi	cfg
34	fh	fgi	egh	ehi	efi	fgj	dgh
35	gi	ghj	fhi	fij	fgj	ghk	ehi
36	hj	hik	gij	gjk	ghk	hil	fij

(Table 3.2.2.d Level 6)

39	d	e	cf	bf	cg	de
40	e	f	dg	cg	dh	ef
41	f	g	eh	dh	ei	fg
42	g	h	fi	ei	fj	gh
43	h	i	gj	fj	gk	hi
44	i	j	hk	gk	hl	ij

(Table 3.2.2.e Level 7)

46	df	de	ef	cf	dg
47	eg	ef	fg	fg	eh
48	fh	fg	gh	eh	fi
49	gi	gh	hi	fi	gj
50	hj	hi	ij	gj	hk
51	ik	ij	jk	hk	il

(Table 3.2.2.f Level 8)

52	e	f
53	f	g
54	g	h
55	h	i
56	i	j

(Table 3.2.2.g Level 9)

58	f
59	g
60	h
61	i

(Table 3.2.2.h Level 10)

Black uses the following algorithm to respond specifically when white places the first or second moves on the blue region. If at any time the location black is directed to move is occupied, black places random moves on the border until it is completed. Once the border is completed, black places moves inside of the border until white is contained. We will prove by induction that after white's moves to squares containing numbers or small letters, black will be able to occupy corresponding border squares with capital letters. This will show that white can not occupy a border square, which shows that white can be contained.

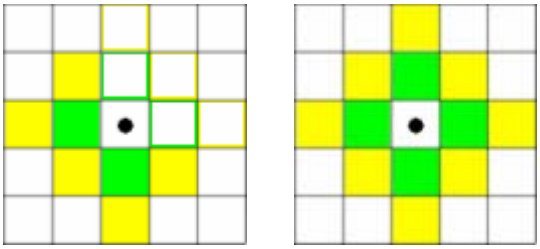
To describe black's algorithm assume that before white makes a first move, whenever white is located on the blue region at say **L** that one of the combinations of corresponding capital letters listed in the tables at **L** are occupied by black. This will be proven shortly. White may make moves onto one or two blue squares with the first and second moves. If white moves onto exactly one blue square with the first and second moves say square numbered **B**, **B** must be in level 10 or level 1. If **B** is in level 10, black responds to the corresponding capital letter indicated in the single lettered column in row **B** of table 10. If exactly one of white's first or second moves are on level 1 white's other move must be on the border, but it will be shown white cannot move onto the border. Thus black is directed to make exactly one move when white makes exclusively either the first or second move on the blue region.

White may also make two moves on the blue region. Black's strategy in response to this is to find combinations of moves in the tables for the two moves that differ by one letter from a combination of moves already occupied on the border. Exactly how this is done is described below.

White's first move say **B** located on number **b** may be connected to multiple white locations. Black picks one of these locations call it **A** located on number **a. A** may also be in the unnumbered yellow region or in the numbered blue region. Call white's second move **C** located on number **c**. Notice we have **A**, **B**, **C** where **A** is a starting location, **B** is the first move and **C** is the second move. Clearly **A** is connected to **B** which is connected to **C**. Let **A** be in level **l, B** in level **m** and **C** in level **n**. Notice A, **B**, **C** corresponds with levels **l**, **m**, **n** respectively. **l** must differ by one from **m** which must differ by one from **n**. Black picks a combination of letters from table **l** at number **a** already occupied by black or if **A** is in the yellow region black picks no letters. For the locations in the first two levels, there is only one combination of letters used for each location listed in the figure above instead of a table. Call this combination of letters chosen by black **the combination**. Black then goes to table **m** in row **b** and table **n** in row **c** and finds one combination of letters in row **b** and one combination of letters in row **c** where there is exactly one letter **z** in these two sets of letters that is not in **combination.**

It will now be shown that regardless where **A** is such a letter **z** can always be found.

An appeal to symmetry will first be made. It is only necessary to consider all combinations of letters from a singular white starting location in every level since the lettering is symmetric across all the levels. Also, it is not necessary to consider all the possible white moves **B** and **C** from any **A** but a subset of them due to the diagonal symmetry of the responses. For instance, if white is located in location 4, considering d, e, g is the same as considering d, f, g. In other words, Only the moves marked below on the left need to be considered for each level. Note **B** is in green and **C** is in yellow.



(figure 3.2.2.f Necessary Considerations And Possible Moves)

So, from every white starting position there are at most 6 different white move combinations that need to be considered. This is done in the tables below using starting locations 59, 53, 48, 41, 34, 25, 16, 5, fgh, fg, and a yellow location connected to 59.

The following charts illustrate possible starting locations, possible 1st and 2nd moves followed by the appropriate response **z**. These charts also show that such a **z** always exists or that black's algorithm directs at most one response. Also these charts imply black's algorithm directs black to occupy at least one of the combinations of letters listed in the first set of tables. Also the tables show that white may never move onto the border since black will be directed to occupy the border squares where white could move onto the border.

Start	1st Move	2nd Move	z
Yellow	Yellow	59	
none	none	g	G

(table 3.2.2.i Yellow Region 1)

Start	1st Move	2nd Move	z
Yellow	59	53	
none	g	g	G

(table 3.2.2.j Yellow Region 2)

Start	1st Move	2nd Move	z
59	53	58	
g	f	fg	F
59	53	47	
g	f	fg	F
59	53	48	
g	f	g	F

(table 3.2.2.k Level 10)

Start	1st Move	2nd Move	z
48	53	59	
fh	f	g	G
fg	f	g	None
gh	f	g	F
eh	g	g	G
fi	f	g	G
48	53	58	
fh	g	f	G
fg	f	f	None
gh	f	f	F
eh	g	g	G
fi	f	g	G
48	53	47	
fh	g	fg	G
fg	f	ef	E
gh	f	fg	F
eh	f	ef	F
fi	f	fg	G
48	41	47	
fh	fg	fg	G
fg	fg	dg	D
gh	fg	fg	F
eh	eh	ef	F
fi	fg	fg	G
48	41	33	
fh	eh	efh	E
fg	fg	cfg	C
gh	dh	dgh	D
eh	eh	efh	F
fi	ei	efi	E
48	41	34	
fh	eh	fh	E
fg	fg	fgi	I
gh	dh	dgh	D
eh	eh	ehi	I
fi	ei	ehi	E

(table 3.2.2.l Level 8)

Start	1st Move	2nd Move	z
53	58	52	
f	f	ef	E
g	f	f	F
53	47	52	
f	ef	e	E
g	fg	f	F
53	47	40	
f	ef	ef	E
g	dg	dg	D
53	47	41	
f	ef	e	E
g	fg	fg	G

(table 3.2.2.m Level 9)

Start	1st Move	2nd Move	z
41	47	53	
eh	eh	f	F
dh	dg	g	G
ei	ef	f	F
fg	dg	g	D
41	47	52	
eh	eh	f	F
dh	eh	e	E
ei	eh	e	H
fg	fg	e	E
41	47	40	
eh	eh	dh	D
dh	eh	dh	E
ei	ef	ef	F
fg	ef	ef	E
41	33	40	
eh	deh	dh	D
dh	dgh	dh	G
ei	efi	ef	F
fg	ef	ef	E
41	33	24	
eh	eg	eg	G
dh	deh	deh	E
ei	efi	ef	F
fg	cfg	cfg	C
41	33	25	
eh	efh	fh	F
dh	dgh	dgh	G
ei	efi	efi	F
fg	dfg	fg	D

(table 3.2.2.n Level 7)

Start	1st Move	2nd Move	z
34	41	47	
fh	fg	fg	G
fgi	fg	dg	D
egh	dh	eh	D
ehi	dh	eh	D
efi	fg	ef	G
fgj	fg	ef	E
dgh	fg	fg	F
34	41	48	
fh	fg	fg	G
fgi	fg	gh	H
egh	eh	ef	F
ehi	eh	ef	F
efi	ei	eh	H
fgj	fg	ef	E
dgh	dh	eg	E
34	41	33	
fh	eh	efh	E
fgi	fg	eg	E
egh	eh	deh	D
ehi	eh	deh	D
efi	ei	ehf	H
fgj	fg	dfg	D
dgh	dh	dfg	F
34	25	33	
fh	fh	efh	E
fgi	fg	dfg	D
egh	eg	dgh	D
ehi	fh	efi	F
efi	fh	efi	H
fgj	eg	eg	E
dgh	dgh	eg	E
34	25	15	
fh	fh	efh	E
fgi	fg	efg	E
egh	fg	efg	F
ehi	fh	efh	F
efi	fh	efh	G
fgj	fg	dfg	D
dgh	fg	dgh	F
34	25	16	
fh	fg	fgh	G
fgi	eg	fgi	E
egh	fg	egh	F
ehi	eg	egh	G
efi	eg	fgi	G
fgj	fg	fgi	I
dgh	fg	fgh	F

(table 3.2.2.o Level 6)

Start	1st Move	2nd Move	z
25	33	40	
fg	eg	ef	G
eg	eg	ef	F
fh	efh	ef	H
dgh	dgh	cg	C
efi	efh	ef	H
dgi	dfg	dg	F
dfi	dfg	dg	G
25	33	41	
fg	eg	fg	E
eg	eg	fg	F
fh	efh	eh	E
dgh	dgh	eh	E
efi	efi	fg	G
dgi	dgh	dh	H
dfi	dfg	fg	G
25	33	24	
fg	cfg	cfg	C
eg	eg	ef	F
fh	efh	ef	E
dgh	dgh	df	F
efi	efh	ef	H
dgi	eg	eg	E
dfi	efi	df	E
25	15	24	
fg	efg	ef	E
eg	efg	ef	F
fh	efh	ef	E
dgh	dfg	df	F
efi	efh	ef	H
dgi	dfg	df	F
dfi	dfg	df	G
25	15	4	
fg	dfg	dfg	D
eg	efg	ef	F
fh	efh	ef	E
dgh	dfg	dfg	F
efi	efg	ef	G
dgi	dfg	dfg	F
dfi	dfg	dfg	G
25	15	5	
fg	efg	fg	E
eg	efg	fg	F
fh	efh	efh	H
dgh	dfg	fg	F
efi	efg	fg	G
dgi	dfg	fg	F
dfi	dfg	fg	G

(table 3.2.2.p Level 5)

Start	1st Move	2nd Move	z
25	33	40	
fg	eg	ef	G
eg	eg	ef	F
fh	efh	ef	H
dgh	dgh	cg	C
efi	efh	ef	H
dgi	dfg	dg	F
dfi	dfg	dg	G
25	33	41	
fg	eg	fg	E
eg	eg	fg	F
fh	efh	eh	E
dgh	dgh	eh	E
efi	efi	fg	G
dgi	dgh	dh	H
dfi	dfg	fg	G
25	33	24	
fg	cfg	cfg	C
eg	eg	ef	F
fh	efh	ef	E
dgh	dgh	df	F
efi	efh	ef	H
dgi	eg	eg	E
dfi	efi	df	E
25	15	24	
fg	efg	ef	E
eg	efg	ef	F
fh	efh	ef	E
dgh	dfg	df	F
efi	efh	ef	H
dgi	dfg	df	F
dfi	dfg	df	G
25	15	4	
fg	dfg	dfg	D
eg	efg	ef	F
fh	efh	ef	E
dgh	dfg	dfg	F
efi	efg	ef	G
dgi	dfg	dfg	F
dfi	dfg	dfg	G
25	15	5	
fg	efg	fg	E
eg	efg	fg	F
fh	efh	efh	H
dgh	dfg	fg	F
efi	efg	fg	G
dgi	dfg	fg	F
dfi	dfg	fg	G

(table 3.2.2.q Level 4)

Start	1st Move	2nd Move	z
5	15	24	
fg	efg	ef	E
egh	efg	ef	F
efh	efg	ef	G
5	15	25	
fg	efg	fg	E
egh	efh	fh	F
efh	efh	fg	G
5	15	4	
fg	efg	ef	E
egh	efg	fg	F
efh	efg	fg	G
5	efg	4	
fg	efg	ef	E
egh	efg	ef	F
efh	efg	ef	G
5	efg	ef	
fg	efg	ef	E
egh	efg	ef	F
efh	efg	ef	G
)5	efg	fg	
fg	efg	fg	E
egh	efg	fg	F
efh	efg	fg	G

(table 3.2.2.r Level 3)

Start	1st Move	2nd Move	z
fgh	5	15	
fgh	fg	efg	E
fgh	5	16	
fgh	fg	egh	E
fgh	5	efg	
fgh	fg	efg	E
fgh	fg	efg	
fgh	fg	efg	E

(table 3.2.2.s Level 2)

Start	1st Move	2nd Move	z
fg	efg	4	
fg	efg	ef	E
fg	efg	4	
fg	efg	fg	E
fg	efg	4	
fg	efg	ef	E

(table 3.2.2.t Level 1)

The tables show by induction that black always has a response that guarantees white will can not move onto the border squares. There are 30741 squares inside the containment. Black's algorithm directs he will occupy the 388 border squares first allowing white to take 776 squares inside the containment. This leaves 29975 squares inside the containment. White may take 19984 and black may take 9991 before all the squares inside the containment are occupied. 19984+776 =20760 so $\text{BlobS}\left(\frac{2}{3}\right) \leq 20760$ or, in particular, $\text{BlobS}\left(\frac{2}{3}\right)$ is finite. ■

3.3 BOUNDS ON THE CRITICAL ***P*** VALUE ON THE SQUARE GRID

3.3.1 BOUNDS ON VERSION I ON THE SQUARE GRID

Theorem 10:

For Version I, $\text{BlobS}\left(\frac{3}{4}+\varepsilon\right)=\infty$.

Proof:

White labels two columns on a 2-strip 1 and 2. White arbitrarily picks a square value, labels it 0, and enumerates the 2 strip with the integers going upward as follows.

7	7
6	6
5	5
4	4
3	3
2	2
1	1

(figure 3.3.1.a Square 2-Strip)

White plays as follows. For the first turn, white makes three connected moves on the 2-strip at (1, 1), (1, 2), and (1, 3) to start a blob. Say black moves on the 2-strip. White's following turn, in response to (1, n) by black, white moves to (2, n-1), (2, n), (2, n+1) or, in response to (2, n) by black, (1, n-1), (1, n), (1, n+1). Some moves may not be possible here due to occupied squares. White's moves not placed or accounted for above are placed as follows.

Find the highest two squares in the 1 and 2 columns that are connected to the designated blob. If they are equal height pick the square in column 1. WLOG say column one is chosen at (1, m). Place a move at (1, m+1) and repeat until there are no additional moves to place.

Possible disconnections by black along the 2-strip will be of the form

(figure 3.3.1.b 2-Strip Disconnections)

However, white's response to any black move on the 2-strip makes all disconnections impossible. The occasional additional move allotted to white by the epsilon term allows white to make an infinite number of moves connected to the blob. ■

To prove that for for Square Version 1 $\text{BlobS}\left(\frac{3}{4}\right) = \infty$, a sequence of terms will be defined along with an algorithm to be used by white. The terms will be shown to be a complete description of all possible black moves. Also a sequence of lemma's about white's algorithm will be produced to show $\text{BlobS}\left(\frac{3}{4}\right) = \infty$. It will be assumed that $\left(\frac{3}{4}\right)$ is the move ratio throughout the proof.

Consider a vertical 4-strip with the columns labeled 1 through 4. Black moves on the 4-strip are categorized as follows. Any black move will be referred to as an **attack on basic essential connections**, **no threat**, an **attack on a future region**, a **single sided attack**, or a **double sided attack**. Each of these are defined as follows.

1. **Attacks on basic essential connections** are moves by black which threaten to disconnect the **knight's move** or **corners**.

Locations are said to be **essentially connected** for a particular move ratio when for any subsequence of the sequence of moves defined by the move ratio, white has a way to connect the locations against any black strategy. This proof will show that all white's placements on the 4-strip are essentially connected and describe how white may connect them. There are two simple types of essential connections used in this proof as shown below. The lower cased lettered squares are defined as **attacks on basic essential connections** because they threaten to disconnect two essentially connected moves and are relatively simple cases.

Lemma 2:

All basic essential connections are essentially connected for the move ratio $\left(\frac{3}{4}\right)$

W	a
c	b
d	W

(figure 3.3.1.c Knight's Move)

W	e
f	W

(figure 3.3.1.d Corner)

White maintains the essential connection between the two moves by moving or connecting the two moves while moving solely on the boxes illustrated. The list of possible black moves followed by the sequence of white responses are shown below for the two types of essential connections.

Black Move	Response
a	c
d	b
b	cd
c	ab

(table 3.3.1.a Knight's Move Responses)

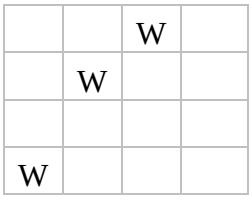
In the case for the knights move white either creates an essentially connected corner to maintain the essential connection as in response to threats a and d or connects the two moves as in response to threats b and c

Black Move	Response
e	f
f	e

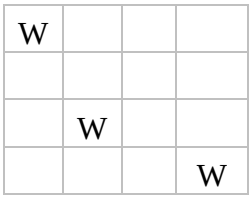
(table 3.3.1.b Corner Responses)

In the case for the corner white simply connects the two moves when the essential connection is threatened. ■

Multiple groups of white locations may form what's called an **essentially connected pathway**. For example, a combination of two knight's moves or a combination of a knight's move and a corner may form an essentially connected pathway as shown below. All white squares in any such pathway are said to be essentially connected.



(figure 3.3.1.e Essentially Connected Pathway 1)



(figure 3.3.1.f Essentially Connected Pathway 2)

2. **No threat** is defined as a black move on the 4-strip where there exists an essentially connected pathway through the row the move is located on (to a

location above and below the move) that does not require a response by white to either connect the pathway or ensure the essential connection. Any black move off the 4-strip is also no threat.

3. **Attacks on future regions** are defined as moves by black that are vertical distance 3 or greater from any white square on the 4-strip.

The attacks on future regions and white's responses are illustrated below.

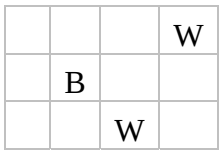
W
B
W
(figure 3.3.1.g Future Region Attack 1)

W
B
W
(figure 3.3.1.h Future Region Attack 2)

W
B
W
(figure 3.3.1.i Future Region Attack 3)

W
B
W
(figure 3.3.1.j Future Region Attack 4)

Projections are white moves that have a vertical distance of 1 away from any black threat. The figure below shows the example of a two projections.



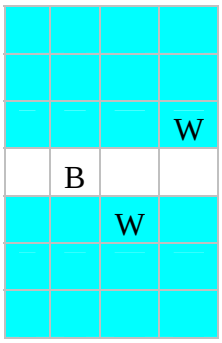
(figure 3.3.1.k Projections)

Lemma 3: White's responses to all single sided attacks create two projections.

Proof:

It is easy to see from the four responses illustrated above that all responses to attacks on future regions create two projections. ■

4. **Single sided attacks** are defined as black moves in the blue region below that have vertical distance 2 or less from projections that are not double sided attacks. Also, no essentially connected pathway may exist through the blue region.



(figure 3.3.1.l Single Sided Attacks)

Lemma 4: In response to any single sided attack, white may create a new projection that is essentially connected to the first projection through a sequence of at most two essentially connected moves.

Proof:

What follows are tables of responses to be used by white to pick moves in response to certain black moves that will secure essential connections and create new projections in response to black threats define as single sided attacks.

Black may threaten projections located in 4 columns on the grid. Two are necessary to consider by symmetry. The two projections and white's responses to single sided attacks are now illustrated in tables.

A	B	C	D
1	2	3	4
5	6	7	8
9	W	10	11

(figure 3.3.1.m Possible Single Sided Attacks 1)

Black Moves	White Responses
3,4	1,A
7,8	1
9,10,11	6
1,2	7,D
5	3
6	10,4

(table 3.3.1.c Responses 1)

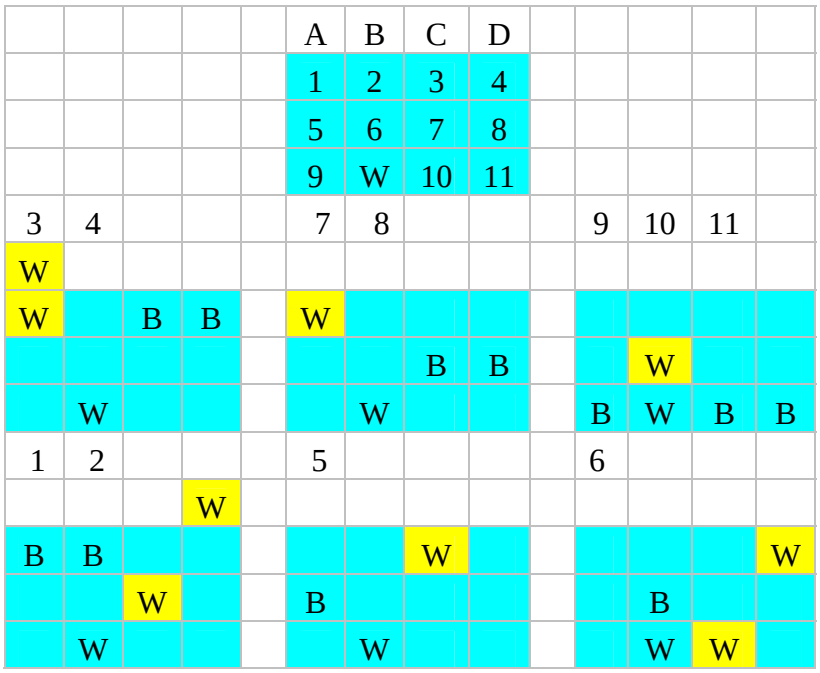
A	B	C	D
1	2	3	4
5	6	7	8
W	9	10	11

(figure 3.3.1.n Possible Single Sided Attacks 2)

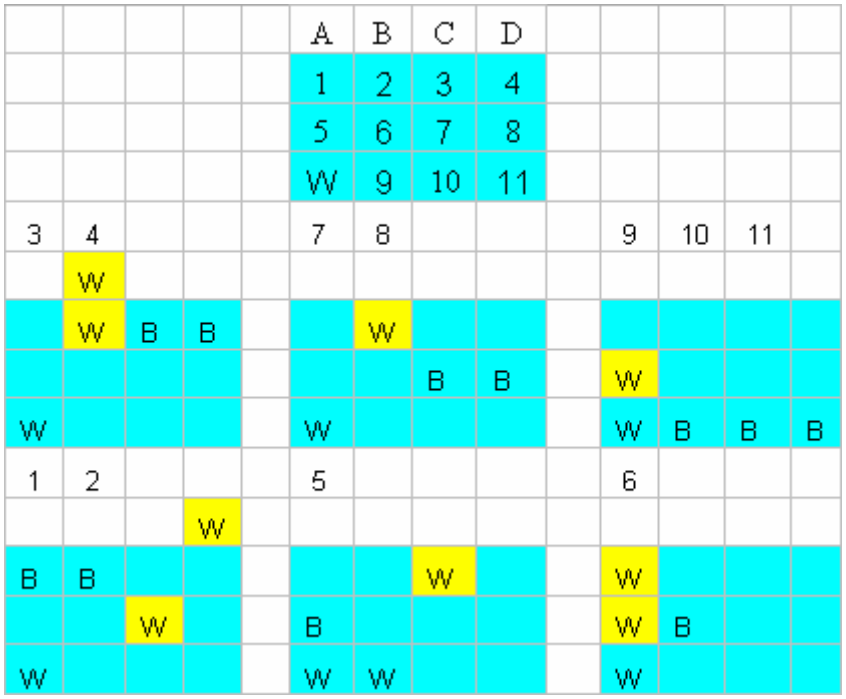
Black Moves	White Responses
3,4	2,B
7,8	5,1
9,10,11	5
1,2	6,7,D
5	9,3
6	5,1

(table 3.3.1.d Responses 2)

The essentially connected pathways are illustrated below where yellow moves represent white's responses to the black threat. Each sequence of moves results in a projection passed the black threat which will have a vertical distance of at least two from any other white location above the projection. This is true because otherwise, black's threat would be a vertical distances of 3 from a white move above the projection and the situation would be a double sided attack not a single sided attack. This will be lucid once a double sided attack is defined.



(figure 3.3.1.o Single Sided Attack Responses 1)



(figure 3.3.1.p Single Sided Attack Responses 2)

Clearly all the above result in a new projection with at most two white moves that are essentially connected pathways. ■

5. **Double sided attacks** are defined as any moves by black that are vertical distance 3 or less in one direction from a projection and vertical distance 3 or less in the other direction from another projection where the total of these two vertical distances is at most 5 and at least 2. Also, to be a double sided attack, there must be no essentially connected pathway connecting the two projections in question. White will respond to this threat establishing an essentially connected pathway through this region essentially connecting the two projections. Below are the possible double sided attacks where the blue squares are locations where black could mount a double sided attack.

A					B					C					D			
W	1	2	3		W	1	2	3		W	1	2	3		W	1	2	3
4	5	6	7		4	5	6	7		4	5	6	7		4	5	6	7
8	9	10	W		8	9	W	10		8	W	9	10		W	8	9	10
E					F					G					H			
W	1	2	3		W	1	2	3		W	1	2	3		W	1	2	3
4	5	6	7		4	5	6	7		4	5	6	7		4	5	6	7
8	9	10	11		8	9	10	11		8	9	10	11		8	9	10	11
12	13	14	W		12	13	W	14		12	W	13	14		W	12	13	14
I					J					K					L			
W	1	2	3		W	1	2	3		W	1	2	3		W	1	2	3
4	5	6	7		4	5	6	7		4	5	6	7		4	5	6	7
8	9	10	11		8	9	10	11		8	9	10	11		8	9	10	11
12	13	14	15		12	13	14	15		12	13	14	15		12	13	14	15
16	17	18	W		16	17	W	18		16	W	17	18		W	16	17	18
M					N					O					P			
W	1	2	3		W	1	2	3		W	1	2	3		W	1	2	3
4	5	6	7		4	5	6	7		4	5	6	7		4	5	6	7
8	9	10	11		8	9	10	11		8	9	10	11		8	9	10	11
12	13	14	15		12	13	14	15		12	13	14	15		12	13	14	15
16	17	18	19		16	17	18	19		16	17	18	19		16	17	18	19
20	21	22	W		20	21	W	22		20	W	21	22		W	20	21	22

(figure 3.3.1.q Top Right White Double Sided Attacks)

A					B					C					D			
1	W	2	3		1	W	2	3		1	W	2	3		1	W	2	3
4	5	6	7		4	5	6	7		4	5	6	7		4	5	6	7
8	9	10	W		8	9	W	10		8	W	9	10		W	8	9	10
E					F					G					H			
1	W	2	3		1	W	2	3		1	W	2	3		1	W	2	3
4	5	6	7		4	5	6	7		4	5	6	7		4	5	6	7
8	9	10	11		8	9	10	11		8	9	10	11		8	9	10	11
12	13	14	W		12	13	W	14		12	W	13	14		W	12	13	14
I					J					K					L			
1	W	2	3		1	W	2	3		1	W	2	3		1	W	2	3
4	5	6	7		4	5	6	7		4	5	6	7		4	5	6	7
8	9	10	11		8	9	10	11		8	9	10	11		8	9	10	11
12	13	14	15		12	13	14	15		12	13	14	15		12	13	14	15
16	17	18	W		16	17	W	18		16	W	17	18		W	16	17	18
M					N					O					P			
1	W	2	3		1	W	2	3		1	W	2	3		1	W	2	3
4	5	6	7		4	5	6	7		4	5	6	7		4	5	6	7
8	9	10	11		8	9	10	11		8	9	10	11		8	9	10	11
12	13	14	15		12	13	14	15		12	13	14	15		12	13	14	15
16	17	18	19		16	17	18	19		16	17	18	19		16	17	18	19
20	21	22	W		20	21	W	22		20	W	21	22		W	20	21	22

(figure 3.3.1.r Top Middle White Double Sided Attacks)

Lemma 5:

White may form an essentially connected pathway in response to double sided attacks between any two projections with three moves

Proof:

What follows are tables along with illustrations showing that essentially connected pathways may be created in response to any double sided attack. The responses by white are shown in yellow and use at most three moves.

A

1	W	2	3
4	5	6	7
8	9	10	W

(figure 3.3.1.s Possible Black Threats A)

Black Moves	Responses
1,4,8	6
5	2
2	5
6	10,9,5
7	10
10	7
9	6
3	5

(table 3.3.1.e Threats and Responses A)

1	4	8				7		
B	W					W		
B		W						B
B			W				W	W
5						10		
	W	W				W		
	B							W
			W				B	W
2						9		
	W	B				W		
	W						W	
			W			B		W
6						3		
	W					W		B
	W	B				W		
	W	W	W					W

(figure 3.3.1.t Response Illustrations A)

E			
1	W	2	3
4	5	6	7
8	9	10	11
12	13	14	W

(figure 3.3.1.u Possible Black Threats E)

Black Moves	Responses
1,4,8,12	6
5,9,13	6,2
2,3	6,5
6	9,5
7	9,5
10	11,7
11	14,5
14	11,2

(table 3.3.1.f Threats And Responses E)

1	4	8	12		7			
B	W					W		
B		W				W		B
B						W		
B			W					W
5	9	13			10			
	W	W				W		
	B	W						W
	B						B	W
	B		W					W
2	3				11			
	W	B	B			W		
	W	W				W		
								B
			W				W	W
6					14			
	W					W	W	
	W	B						
	W							W
			W				B	W

(figure 3.3.1.v Response Illustrations E)

I			
1	W	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	W

(figure 3.3.1.w Possible Black Threats I)

I Black Moves	Responses
4,8,12	10
5,9,13	10,6,2
6,10	13,4
14	15,11,2
15	18,9,5
7,11	13,4

(table 3.3.1.g Threats and Responses I)

4	8	12			14			
	W					W	W	
B								
B		W						W
B								W
			W				B	W
5	9	13			15			
	W	W				W		
	B	W				W		
	B	W				W		
	B							B
			W				W	W
6	10				7	11		
	W					W		
W		B			W			B
		B						B
	W					W		
			W					W

(figure 3.3.1.x Response Illustrations I)

M			
1	W	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	W

(figure 3.3.1.y Possible Black Threats M)

I Black Moves	Responses
4,8,12	10
5,9,13	10,6,2
6,10	13,4
14	15,11,2
15	18,9,5
7,11	13,4

(table 3.3.1.h Threats and Responses M)

8	12				10	14		
	W					W		
	W							
B					W		B	
B		W					B	
						W	W	
			W					W
9	13				11	15		
	W					W		
		W				W		
	B	W				W		B
	B	W						B
							W	
			W					W

(figure 3.3.1.z Response Illustrations M)

B

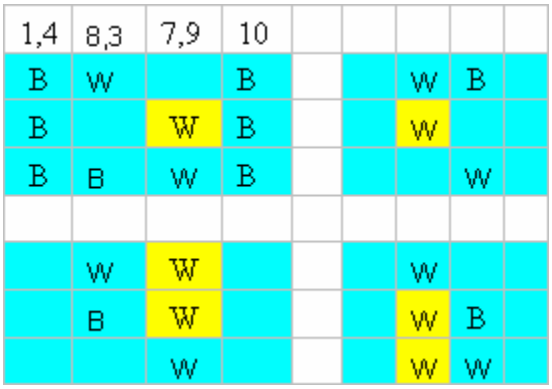
1	W	2	3
4	5	6	7
8	9	W	10

(figure 3.3.1.aa Possible Black Threats B)

B

Black Moves	Responses
1,4,8,3,7,9,10	6
5	6,2
2	5
6	9,5

(table 3.3.1.i Threats and Responses B)



(figure 3.3.1.ab Response Illustration B)

F			
1	W	2	3
4	5	6	7
8	9	10	11
12	13	W	14

(figure 3.3.1.ac Possible Threats F)

F

Black Moves	Responses
1,4,8,12	5
5,9	10,6,2
13	10
2,3,7,11,14	5
6,10	13,9,5

(table 3.3.1.j Threats And Responses)

1	4	8	12		2,3	7	11	14
B	W					W	B	B
B	W					W		B
B								B
B		W					W	B
5	9				6	10		
	W	W				W		
	B	W				W	B	
	B	W				W	B	
		W				W	W	
13								
	W							
		W						
	B	W						

(figure 3.3.1.ad Response Illustration F)

1	W	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	W	18

(figure 3.3.1.ae Possible Threats J)

J

Black Moves	Responses
4,8,12	9,5
7,11,15	9,5
5	2,11,15
9,13	14,10,6
14	17,8,4
6,10	17,8,4

(table 3.3.1.k Threats and Responses J)

4	8	12			9	13		
	W					W		
B	W						W	
B	W					B	W	
B						B	W	
		W					W	
7	11	15			14			
	W					W		
	W		B		W			
	W		B		W			
			B				B	
		W				W	W	
5					6	10		
	W	W				W		
	B				W		B	
			W		W		B	
			W					
		W				W	W	

(figure 3.3.1.af Response Illustrations J)

1	W	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	W	22

(figure 3.3.1.ag Possible Threats N)

N Black Moves	Responses
8,12,11,15	13,9,5
9,13	15,11,2
10,14	17,8,4

(table 3.3.1.l Threats And Responses N)

8	12	11	15		10	14		
	W					W		
	W				W			
B	W		B		W		B	
B	W		B				B	
						W		
		W					W	
9	13							
	W	W						
	B		W					
	B		W					
		W						

(figure 3.3.1.ah Response Illustration N)

	C		
1	W	2	3
4	5	6	7
8	W	9	10

(figure 3.3.1.ai Potential Threats C)

C Black Moves	Responses
1,4,8,2,6	5
3,7,9,10	5
5	9,6,2

(figure 3.3.1.m Threats and Responses C)

1,4	8	2	6		5			
B	W	B				W	W	
B	W	B				B	W	
B	W					W	W	
3	7	9	10					
	W		B					
	W		B					
	W	B	B					

(figure 3.3.1.aj Response Illustrations C)

	G		
1	W	2	3
4	5	6	7
8	9	10	11
12	W	13	14

(figure 3.3.1.ak Potential Threats G)

G Black Moves	Responses
1,4,8,12	5,9
2,6,10,13	5,9
3,7,11,14	5,9
5	2,6,10
9	13,10,6

(figure 3.3.1.n Threats And Responses G)

1	4	8	12		5			
B	W					W	W	
B	W					B	W	
B	W						W	
B	W					W		
2	6	10	13		9			
	W	B				W		
	W	B					W	
	W	B				B	W	
	W	B				W	W	
3	7	11	14					
	W		B					
	W		B					
	W		B					
	W		B					

(figure 3.3.1.al Response Illustrations G)

	K		
1	W	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	W	17	18

(figure 3.3.1.am Possible Threats K)

K Black Moves	Responses
4,8,12	5,9,13
6,10,14	5,9,13
7,11,15	5,9,13
5,9,13	2,11,17

(table 3.3.1.o Threats And Responses K)

4	8	12				7	11	15	
		W					W		
	B	W					W		B
	B	W					W		B
	B	W					W		B
		W					W		
	6	10	14			5	9	13	
		W					W	W	
		W	B				B		
		W	B				B		W
		W	B				B		
		W					W	W	

(figure 3.3.1.an Response Illustrations K)

1	W	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	W	21	22

(figure 3.3.1.ao Possible Threats O)

O Black Moves	Responses
8,12,11,15	14,10,5
10,14	12,8,5
9,13	18,11,6

(table 3.3.1.p Threats And Response O)

8	12	11	15		9	13		
	W					W		
	W						W	
B		W	B			B		W
B		W	B			B		
							W	
	W					W		
10	14							
	W							
	W							
W		B						
W		B						
	W							

(figure 3.3.1.ap Response Illustrations O)

	D		
1	W	2	3
4	5	6	7
W	8	9	10

(figure 3.3.1.aq Potential Threats D)

D Black Moves	Responses
1,4	8,5
2,3,6,7,9,10	8,5
5,8	1,4

(table 3.3.1.q Threats And Responses D)

1	4				5	8		
B	W				W	W		
B	W				W	B		
W	W				W	B		
2,3	6,7	9	10					
	W	B	B					
	W	B	B					
W	W	B	B					

(figure 3.3.1.ar Response Illustrations D)

	H		
1	W	2	3
4	5	6	7
8	9	10	11
W	12	13	14

(figure 3.3.1.as Potential Threats H)

H Black Moves	Responses
2,6,10,13	5
3,7,11,14	5
12	8
1	5
4,8	5,9,12
5,9	1,4,8

(table 3.3.1.r Threats And Responses H)

2	6	10	13		1			
	W	B			B	W		
	W	B				W		
		B						
W		B			W			
3	7	11	14		4	8		
	W		B			W		
	W		B		B	W		
			B		B	W		
W			B		W	W		
12					5	9		
	W				W	W		
					W	B		
W					W	B		
W	B				W			

(figure 3.3.1.at Response Illustrations H)

	L		
1	W	2	3
4	5	6	7
8	9	10	11
12	13	14	15
W	16	17	18

(figure 3.3.1.au Potential Threats L)

L Black Moves	Responses
4,8,12	16,10,6
16,10,14	9,5
7,11,15	9,5
5	14,7,2
9,13	12,8,4

(table 3.3.1.s Threats And Responses L)

4	12	8			5			
	W					W	W	
B		W				B		W
B		W						
B							W	
W	W				W			
16	10	14			9	13		
	W					W		
	W	B			W			
	W	B			W	B		
		B			W	B		
W					W			
7	11	15						
	W							
	W		B					
	W		B					
			B					
W								

(figure 3.3.1.av Response Illustrations L)

	P		
	W	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
W	20	21	22

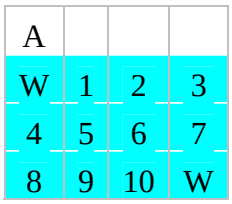
(figure 3.3.1.aw Potential Threats P)

P Black Moves	Responses
8,12,9,13	18,11,2
10,11,14,15	13,9,5

(table 3.3.1.t Threats And Responses P)

	W	W				W		
						W		
B	B		W			W	B	B
B	B					W	B	B
		W						
W					W			

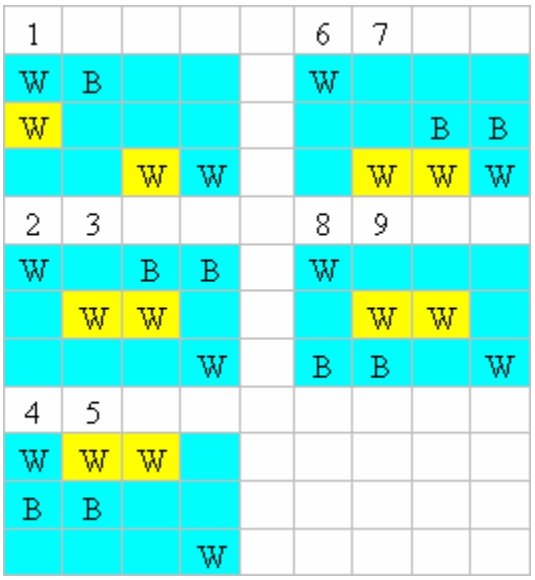
(figure 3.3.1.ax Response Illustrations P)



(figure 3.3.1.ax2 Potential Threats A)

A Black Moves	Responses
1	4,10
2,3	5,6
4,5	1,2
6,7	9,10
8,9	5,6

(table 3.3.1.u Threats And Responses A)



(figure 3.3.1.ay Response Illustrations A)

E			
W	1	2	3
4	5	6	7
8	9	10	11
12	13	14	W

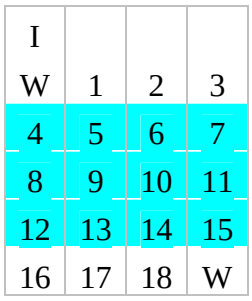
(figure 3.3.1.az Potential Threats E)

E Black Moves	Responses
1,2,3	4,10
12,13,14	11,5
4,8,5,9	1,2,11
6,10,7,11	14,13,4

(table 3.3.1.v Threats And Responses E)

1	2	3			6	10	7	11
W	B	B	B		W			
W					W		B	B
		W					B	B
			W			W	W	W
12	13	14			4	8	5	9
W					W	W	W	
	W				B	B		
			W		B	B		W
B	B	B	W					W

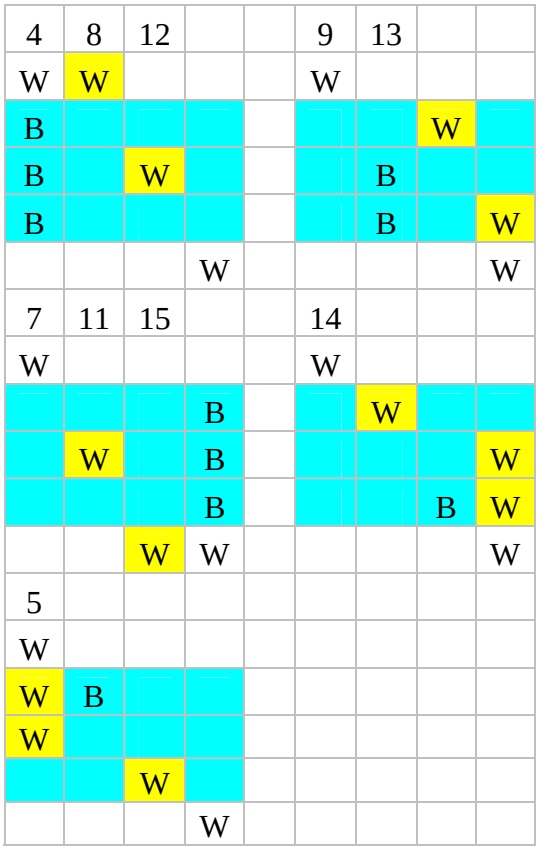
(figure 3.3.1.ba Response Illustrations E)



(figure 3.3.1.bb Potential Threats I)

I Black Moves	Responses
4,8,12	1,10
7,11,15	18,9
5	4,8,14
9,13	6,15
14	15,11,5

(table 3.3.1.w Threats And Responses I)



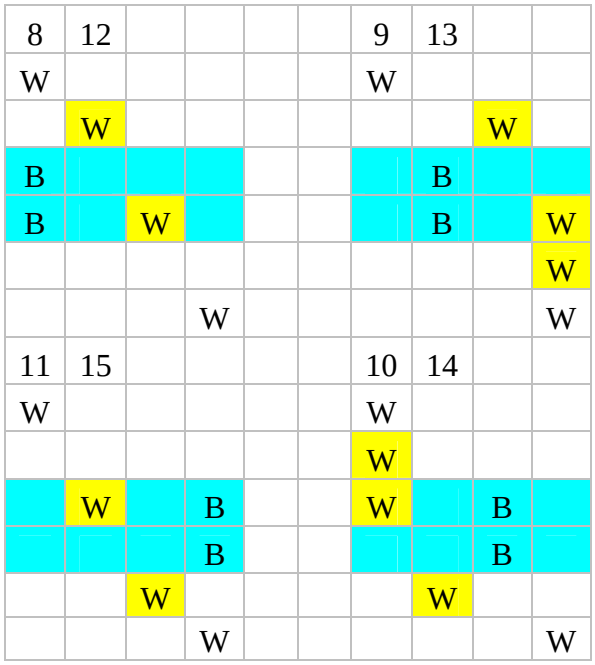
(figure 3.3.1.bc Response Illustrations I)

M W	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	W

(figure 3.3.1.bd Potential Threats M)

M Black Moves	Responses
8,12	14,5
11,15	9,18
9,13	6,15,19
10,14	17,8,4

(table 3.3.1.x Threats And Responses M)



(figure 3.3.1.be Response Illustrations M)

B

W	1	2	3
4	5	6	7
8	9	W	10

(figure 3.3.1.bf Potential Threats B)

B Black Moves	Responses
3,7,10	1
4,8,9	1,6
5	1,2,6
1,2,6	4,9

(table 3.3.1.y Threats And Responses B)

3	7	10			5			
W	W		B		W	W	W	
			B			B	W	
		W	B				W	
4	8	9			1	2	6	
W	W				W	B	B	
B		W			W		B	
B	B	W				W	W	

(figure 3.3.1.bg Response Illustrations B)

F			
W	1	2	3
4	5	6	7
8	9	10	11
12	13	W	14

(figure 3.3.1.bh Potential Threats F)

F Black Moves	Responses
3,7,11,14	5
2,6,10	13,4
4,8,12	1,10
1,5	4,8
13,9	10,6

(table 3.3.1.z Threats And Responses F)

3	7	11	14		4	8	12	
W			B		W	W		
	W		B		B			
			B		B		W	
		W	B		B		W	
2	6	10			1	5		
W		B			W	B		
W		B			W	B		
		B			W			
	W	W					W	
13	9							
W								
		W						
	B	W						
	B	W						

(figure 3.3.1.bi Response Illustrations F)

J			
W	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	W	18

(figure 3.3.1.bj Potential Threats J)

J Black Moves	Responses
4,8,12	1,10,14
7,11,15	9,5
6,10,14	17,8,4
5	4,8,13
9	12,8,4
13	14,10,1

(table 3.3.1.aa Threats And Responses J)

(figure 3.3.1.bk Response Illustrations J)

N			
W	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	W	22

(figure 3.3.1.bl Potential Threats N)

N Black Moves	Responses
8,12	13,9,5
13	15,11,5
9	6,10,19
10,14,11,15	21,12,5

(table 3.3.1.ab Threats And Responses N)

8	12				9			
W					W			
	W						W	
B	W					B	W	
B	W							
								W
		W					W	
13					10	14	11	15
W					W			
	W					W		
			W				B	B
	B		W		W		B	B
		W				W	W	

(figure 3.3.1.bm Response Illustrations N)

	C		
W	1	2	3
4	5	6	7
8	W	9	10

(figure 3.3.1.bn Potential Threats C)

C Black Moves	Responses
2,3,6,7,9,10	
1,5	4,8
4,8	1,5

(table 3.3.1.ac Threats And Responses C)

2,3	6,7	9	10		4	8		
W		B	B		W	W		
		B	B		B	W		
	W	B	B		B	W		
1	5							
W	B							
W	B							
W	W							

(figure 3.3.1.bo Response Illustrations C)

	G		
W	1	2	3
4	5	6	7
8	9	10	11
12	W	13	14

(figure 3.3.1.bp Potential Threats G)

G Black Moves	Responses
2,6,10,13	4
3,7,11,14	4
1,5	4,8
12,8	9,5
4	1,5,9
9	12,8,4

(table 3.3.1.ad Threats And Responses G)

2	6	10	13		12	8		
W		B			W			
W		B				W		
		B			B	W		
	W	B			B	W		
3	7	11	14		4			
W			B		W	W		
W			B		B	W		
			B			W		
	W		B			W		
1	5				9			
W	B				W	W		
W	B					W		
W						W		
	W				B	W		

(figure 3.3.1.bq Response Illustrations G)

	K		
W	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	W	17	18

(figure 3.3.1.br Potential Threats K)

K Black Moves	Responses
6,10,14	8,4
7,11,15	8,4
4	1,10,14
13	17,11,5
8,12	13,9,5
5,9	12,8,4

(table 3.3.1.ae Threats And Responses K)

6	10	14			7	11	15			4			
W					W					W	W		
W		B			W			B		B			
W		B			W			B				W	
		B						B				W	
	W					W					W		

13					8	12				5	9		
W					W					W			
	W					W				W	B		
			W		B	W				W	B		
	B				B	W				W			
	W	W				W					W		

(figure 3.3.1.bs Response Illustrations K)

	O		
W	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	W	21	22

(figure 3.3.1.bt Potential Threats O)

O Black Moves	Responses
10,11,14,15	12,8,4
8,12	18,14,5
9	4,8,12
13	19,10,5

(table 3.3.1.af Threats And Responses O)

10	11	14	15		9			
W					W			
W					W			
W		B	B		W	B		
W			B		W			
	W					W		
8	12				13			
W					W			
	W					W		
B							W	
B		W				B		
		W						W
	W					W		

(figure 3.3.1.bu Response Illustrations O)

	D		
W	1	2	3
4	5	6	7
W	8	9	10

(figure 3.3.1.bv Potential Threats D)

D Black Moves	Responses
1,5,8	4
2,6,9	4
3,7,10	4
4	1,5,8

(table 3.3.1.ag Threats And Responses D)

1	5	8			3	7	10	
W	B				W			B
W	B				W			B
W	B				W			B
2	6	9			4			
W		B			W	W		
W		B			B	W		
W		B			W	W		

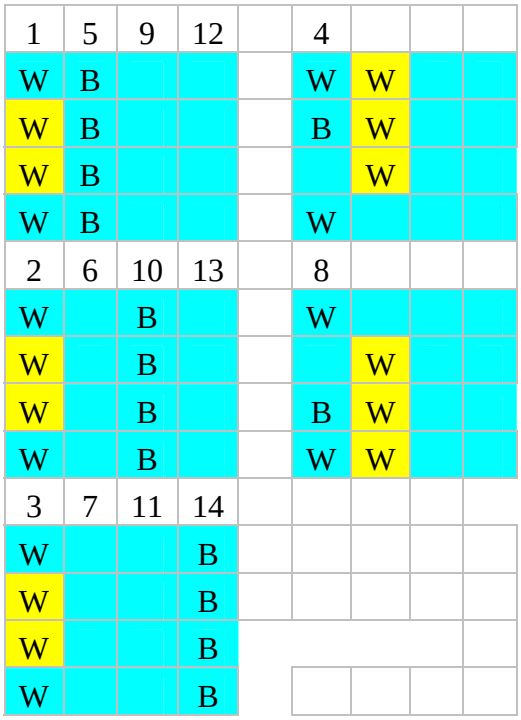
(figure 3.3.1.bw Response Illustrations D)

	H		
W	1	2	3
4	5	6	7
8	9	10	11
W	12	13	14

(figure 3.3.1.x Potential Threats H)

H Black Moves	Responses
1,5,9,12	4,8
2,6,10,13	4,8
3,7,11,14	4,8
4	1,5,9
8	12,9,5

(table 3.3.1.ah Threats And Responses H)



(figure 3.3.1.by Response Illustrations H)

	L		
W	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
W	16	17	18

(figure 3.3.1.bz Potential Threats L)

L Black Moves	Responses
5,9,13	4,8,12
6,10,14	4,8,12
7,11,15	4,8,12
4	1,5,9
8	13,9,5
12	16,13,9

(table 3.3.1.ai Threats And Responses L)

5	9	13			4			
W					W	W		
W	B				B	W		
W	B					W		
W	B							
W					W			
6	10	14			8			
W					W			
W		B				W		
W		B			B	W		
W		B				W		
W					W			
7	11	15			12			
W					W			
W			B					
W			B			W		
W			B		B	W		
W					W	W		

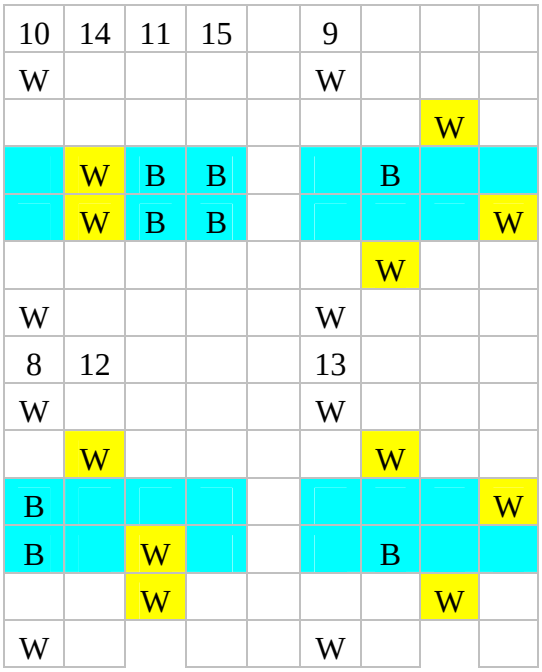
(figure 3.3.1.ca Response Illustrations L)

	P		
W	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
W	20	21	22

(table 3.3.1.cb Potential Threats P)

P Black Moves	Responses
10,14,11,15	13,9
8,12	18,14,5
9	6,15,17
13	18,11,5

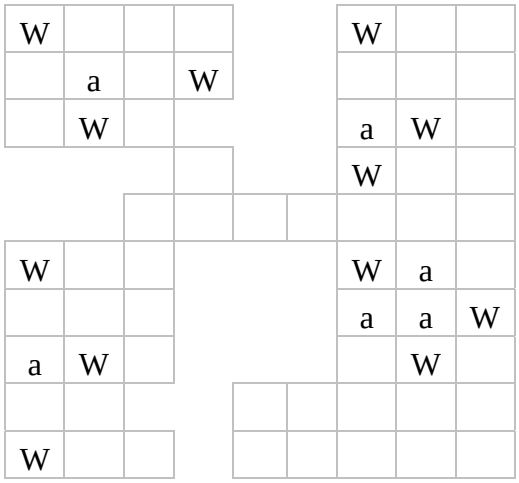
(table 3.3.1.aj Threats And Responses P)



(figure 3.3.1.cc Response Illustrations P)

The preceeding cases are all the types of double sided attacks each of which results in an essentially connected pathway between the projections using at most three white moves. ■

Black may also make threats against essentially connected pathways. **A threat against an essentially connected pathway** is defined as a move by black that is a threat to two or more basic essential connections. For instance, the squares marked a below are threats against essentially connected pathways.



(figure 3.3.1.cd Threat s Against Essentially Connected Pathways)

Lemma 6:

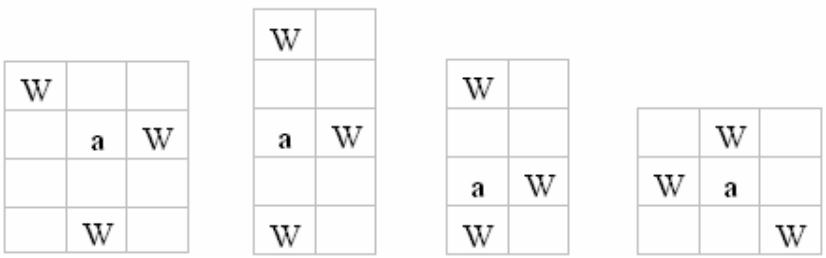
White may connect any essentially connected pathway that results from attacks on future regions, single sided attacks, or double sided attacks.

Proof:

The different locations defined as threats to basic essential connections are the locations where black may make threats against essentially connected pathways. There are different forms of essentially connected pathways generated when white responds to attacks on future regions, single sided attacks, and double sided attacks. The only basic essential connections used by white to create new projections in response to attacks on

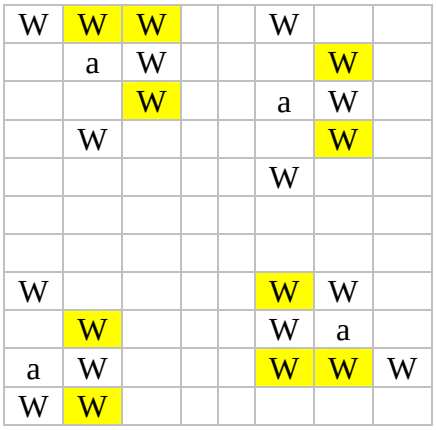
future regions are vertical knight's moves and corners that share no locations used to maintain essential connections. Thus the two projections here are essentially connected.

When responding to single sided and double sided attacks white generates essentially connected pathways of many different forms. Most of which are pathways of basic essential connections or pairs of basic essential connections with no locations in common used to maintain the essential connection. The pathways that do have locations in common used to maintain the basic essential connection have the forms shown below.



(figure 3.3.1.ce Essentially Connected Pathways 1)

To maintain the basic essential connections for the four, white requires at most three moves as shown below.



(figure 3.3.1.cf Essentially Connected Pathways 2)

Thus white may connect any essentially connected pathway that results from attacks on future regions, single sided attacks, and double sided attacks. ■

White's Algorithm:

It has been described how white responds to attacks on future regions, single sided attacks, and double sided attacks. The following describes how white moves in response when black does not move on the 4-strip or black's move is defined as no threat or an attack on a basic essential conneciton.

Firstly, white's first three moves are placed in a 1-strip on the 4-strip so they are all connected. From this point on if black's move is no threat or white has extra moves that are not directed to any specific location, white places the moves at random locations connected to the blob containing white's first move. A special exception occurs when white must move connected to projections that have a vertical distance of two between the projections and there is not an essentially connected pathway between the projections. Here white first plays according to white's response indicated to any double sided attack shown for that distance between the two projections creating an essentially connected pathway between the projections using that one move. This is to prevent a case where white has two projections a vertical distance of one away from each other which is not handled in the case argument.

When white responds to attacks on basic essential connections, white picks the least number of moves possible to maintain the basic essential connection. That is white uses one more in response to threats on corners, one move when black threatens locations

a and d below on the knight's move and two moves when black threatens locations b and c on the knight's move below.

W	a
c	b
d	W

(figure 3.3.1.cg Knight's Move)

Some black moves are attacks on basic essential connections and either single sided attacks or double sided attacks. This occurs when black moves in the same row as a projection that is also a knight's move such as the location marked a in figure 3.3.1.cg. The knight's move here is always vertical, In these two cases, white requires at most one move to secure the essential connection between the knight's move and at most two moves to generate an essentially connected pathway to a new projection for the single sided attack or to generate an essentially connected pathway to an already existing projection for the double sided attack. This may be verified as all single sided attacks and double sided attacks shown above where the white move is in the same row as a projection require at most two white moves to respond.

Lemma 7:

The system used to classify black's moves defines any black move when white plays as above. That is, any black move is defined as no threat, an attack on basic essential connections, an attack on future regions, a single sided attack, or a double sided attack.

Proof:

Black may move anywhere on the grid. Any location outside the grid is defined as no threat.

Consider locations on the 4-strip. Black moves that are on the same row as an essentially connected pathway are either threats if they threaten basic essential connections or no threat. It's clear when a black move is a threat on a basic essential connection, but attacks on basic essential connections may also be single sided attacks or double sided attacks. In the cases where the black move is an attack on a basic essential connection and either a single or double sided attack the black move is on the same row as a projection.

Consider black moves on the 4-strip that are not on the same row as an essentially connected pathway. Any black move on the 4-strip with a vertical distance greater than or equal to three from all white moves is defined as an attack on a future region. Any move by black within a vertical distance of two or less from a single white projection in the blue region as shown earlier is called a single sided attack if is not a double sided attack. Black moves within a vertical distance of three from each of two white projections in opposite directions (one above the white projection above and another below the black move) where the sum of the distances between them is at most five and at least two is defined as a double sided attack if no essentially connected pathway between the moves exists.

White's algorithm does not create a situation where there are two projections with a vertical distance of one away from each other. This is easy to see in how white creates projections. White creates projections when responding to single sided attacks or attacks

on future regions. White's projections created in response to attacks on future regions have a vertical distance of one from the black attack. Since the black attack must be a vertical distance of three or more to all previous white moves to be an attack on a future region, the two projections must be a vertical distance of at least two away from all other white moves. Also, when white is responding to single sided attacks, the projection created is a distance of one away from the black move. If this new projection were a distance of one away from another white projection, black's move would not have been defined as a single sided attack but as a double sided attack since the two vertical distances between the projections and blacks move would be at most two for each projection. Thus, a vertical distance of one between two projections will never occur because white will never create them. From this same argument, a vertical distance of zero between two projections will never occur as white never create such projections.

This defines all black moves on the four strip in relation to any white position. ■

Theorem 11:

For Square Version 1 $\text{BlobS}\left(\frac{3}{4}\right) = \infty$

Proof:

Lemmas 2-7 verify white will always make an essentially connected pathway up the 4-strip which cannot be disconnected by any black move.

Black may create infinite attacks on future regions which require two moves by white to respond. Black may also create infinite single sided attacks which require at

most two moves from white to respond according to the algorithm. These attacks allow white one extra move to build the blob.

Black may also make double sided attacks and attacks on basic essential connections which require three white moves in response. However, the number of double sided attacks black may make is limited by the number of attacks on future regions black has made. Exactly one less double sided attack can be made.

Also, Black may make attacks on essentially connected pathways which require at most three white moves in response to secure the basic essential connections. These instances occur after white secures an essentially connected pathway against double sided attacks. However, the number of such attacks on essentially connected pathways is less than or equal too the sum of three times the number of double sided attacks and the number of single sided attacks. This So, the number of instances where white is required to respond with three moves is less than or equal to the sum of the number of single sided attacks and three times the number of double sided attacks. The number of double sided attacks is bounded by the number of attacks on future. Thus, the only instances where white is required to respond with three moves is less than three times the number of extra moves white receives. This is because white receives an extra move for each single sided attack (only two responses required) and each double sided attack (bounded by the number of attacks on future regions which require two responses).

This shows black has no strategy against white to prevent white from using extra moves to build an infinite blob up the 4-strip when white plays according to the algorithm as all black strategies allow white extra moves.■

3.3.2 BOUNDS ON VERSIONS IV AND V ON THE SQUARE GRID

Theorem 12:

For Versions IV and V, $\text{BlobS}\left(\frac{3}{5}\right) = \infty$

Proof:

When p = $\left(\frac{3}{5}\right)$, the move equation produces the following sequence of moves,

w,b,w,w,b,w,b,w,w,b,w,etc. White makes a first move arbitrarily on the grid and then the second move above the first. Black may move to any of the green squares below where L is white's last move.

(figure 3.3.2.a Possible Black Placements)

If black moves to a non-green square, white responds by moving above the last white move. If black moves to one of the green squares, white responds specifically as diagramed in figure 2 where B is black's move, L is white's last move, and N is white's response.

(figure 3.3.2.b One Move Responses)

(figure 3.3.2.c Next Turns Possibilities)

If white moves again, before black moves, white moves above the last white move made (N above in figure 2) and we have the situation in figure 1. If white does not get to move again, black may move to the orange square illustrated in figure 3. White then receives two moves and moves to two Ns in figure 3. If black does not move to the orange square white responds by moving to the two squares above L. In any case, white's last move results in the situation in figure 1. These moves are all connected and, by induction, white may build an infinite blob.

3.4 SUMMARY OF FINDINGS ON THE SQUARE GRID

$\boldsymbol{P}$ values	$0 < \boldsymbol{p} \leq \frac{1}{5}$	$\frac{1}{5} < \boldsymbol{p} \leq \frac{1}{4}$	$\frac{1}{4} < \boldsymbol{p} \leq \frac{3}{10}$	$\frac{3}{10} < \boldsymbol{p} \leq \frac{1}{3}$
Blob sizes	1	2	3	4

(Table 3.4.a: Square Blob Sizes)

For Version 2: $\text{BlobS}\left(\frac{1}{2}\right) \leq 343$. For Version 3: $\text{BlobS}\left(\frac{1}{2}\right) \leq 55$

For Version 3: $\text{BlobS}\left(\frac{2}{3}\right) \leq 20760$ when white's first and second moves must be connected

Version	1	2	3	4	5
Bounds	$\frac{1}{3} \leq p_c \leq \frac{3}{4}$	$\frac{1}{2} \leq p_c \leq \infty$	$\frac{1}{2} \leq p_c \leq \infty$	$\frac{1}{3} \leq p_c \leq \frac{3}{5}$	$\frac{1}{3} \leq p_c \leq \frac{3}{5}$

(Table 3.4.b: Square Critical Move Ratio Bounds)

CHAPTER 4: THE TRIANGULAR GRID

4.1 GEOMETRY ON THE TRIANGULAR GRID

Now the triangular grid will be explored.

4.1.1 MINIMAL TRIANGULAR ENVELOPE

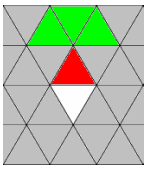
Definition: The smallest number of black moves required to envelope a triangular blob of size ***n*** is known as the **minimal triangular envelope**.

Theorem 13: The Minimal Triangular Envelope Theorem:

The size of the minimal triangular envelope for a blob of size ***n*** is $2n + 10$.

Proof:

Say a blob has ***n*** triangles and white is placing a move connected to that blob. All envelopes occupy every triangle attached to a blob. So, white is placing a move on a triangle that would make up a minimal envelope of the blob of size ***n.*** White's placement can be connected to 1, 2, or 3, triangles. The following shows regardless of how many white triangles the placement is connected to, the difference between the size of the minimal envelopes for a blob of size ***n*** and a blob of size $n + 1$ will be at most 2.



(figure 4.1.1.a Triangular Envelope Extensions)

Figure 4 above illustrates the placement of a white move on the red triangle which is connected to at least 1 other white triangle. Here the green and grey triangles may be unoccupied, white, or black. Thinking of the red triangle as part of the minimal envelope of a blob of size ***n***, if white places a move on that red triangle at most 2 additional black moves are required to envelope the blob of size ***n*** + 1 (3 green triangles minus 1 red triangle). If any of the green triangles are occupied by black or white, the number of black triangles required to envelope the blob decreases by 1 for each green triangle occupied. So an increase of 2 occurs only when the placement is connected to 1 other white triangle and the 3 other connected triangle are unoccupied.

It takes 12 moves to envelope a blob of size 1. Adding one white triangle to this blob can only increase the blob size by 2. Thus the smallest number of triangles required to contain a blob of size n is $2(\boldsymbol{n}-1) + 8 = 2\boldsymbol{n} + 12$. ■

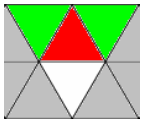
4.1.2 MINIMAL TRIANGULAR CONTAINMENT

Theorem 14: Minimal Triangular Containment Theorem:

The size of a minimal containment on a triangular grid for a blob of size ***n*** is ***n*** + 2.

Proof:

Say a blob has ***n*** triangles and white is placing a move connected to that blob. All containments occupy every triangle connected to a blob. So, white is placing a move on a triangle that would make up a minimal containment of the blob of size ***n.*** White's placement can be connected to 1, 2, or 3 white triangles. The following shows regardless of how many white triangles the placement is connected to, the difference between the size of the minimal containments for a blob of size ***n*** and a blob of size ***n*** + 1 will be at most 1.



(figure 4.1.2.a Triangular Containment Additions)

Figure 1 above illustrates the placement of a white move on the red triangles which is connected to at least 1 other white triangles. Here the green and grey triangles may be unoccupied, white, or black. Thinking of the red triangle as part of the minimal containment of a blob of size ***n***, if white places a move on that red triangle at most 1 additional black moves are required to contain the blob of size ***n*** + 1 (2 green triangle

minus 1 red triangle). If any of the green triangles are occupied by black or white, the number of black triangles required to contain the blob decreases by 1 for each green triangle occupied. So an increase of 1 occurs only when the placement is connected to 1 other white triangle and the 2 other connected triangles are unoccupied.

It takes 3 moves to contain a blob of size 1. Adding one white triangle to this blob can only increase the border size by 1. Thus the smallest number of triangles required to contain a blob of size n is $\boldsymbol{n} + 2$. ■

Lemma 8:

If BlobT($\boldsymbol{p}$) = $\boldsymbol{n}$, $\boldsymbol{p} \geq \left(\frac{n}{2n+2} \right)$.

Proof:

Black may move to triangles connected to white triangles. There are at most $\boldsymbol{n} + 2$ triangles connected to a blob of size $\boldsymbol{n}$ by theorem 12. Thus $\boldsymbol{p}$ must be great than or equal to $\left(\frac{n}{2n+2} \right)$. ■

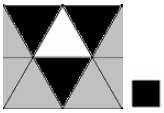
4.2 SMALL TO MIDDLE ***P*** VALUES ON THE TRIANGULAR GRID

Theorem 15:

For all $\boldsymbol{p}$ where $0 < \boldsymbol{p} \leq \frac{1}{4}$, BlobT($\boldsymbol{p}$) = 1.

Proof:

If $\boldsymbol{p} \leq \frac{1}{4}$, black may contain the blob by moving to the three connected triangles.



(figure 4.2.1.a Triangular Blob of Size One Contained)

Theorem 16:

For all $\boldsymbol{p}$ where $\frac{1}{4} < \boldsymbol{p} \leq \frac{1}{3}$, BlobT($\boldsymbol{p}$) = 2.

Proof:

$\boldsymbol{p}$ values where $\frac{1}{4} < \boldsymbol{p} \leq \frac{1}{3}$ produce a move sequence that starts w,b,b,w,b,b. By moving to triangles connected to white, black will contain any blob of size two and can not contain the blob to size one by the MTCT. ■

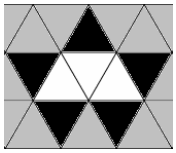
(figure 4.2.1.b Triangular Blob of Size Two Contained)

Theorem 17:

For all $\boldsymbol{p}$ where $\frac{1}{3} < \boldsymbol{p} \leq \frac{3}{8}$, BlobT($\boldsymbol{p}$) = 3.

Proof:

The $\boldsymbol{p}$ values where $\frac{1}{3} < \boldsymbol{p} \leq \frac{3}{8}$ produce a move sequence that starts w,b,w,b,b,w,b,b. By symmetry, there is only one way to connect three triangles. If black plays to triangles connected to white he will contain white to a blob of size three by the MTCT (Minimal Triangular Containment Theorem). Also by MTCT, black could not contain white to a blob of size two since white moves a third time before black moves a fourth. ■



(figure 4.2.1.c Triangular Blob of Size Three Contained)

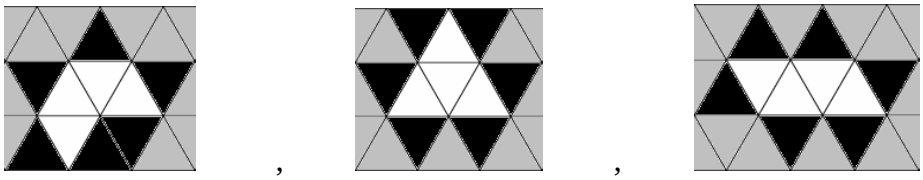
Theorem 18:

For all $\boldsymbol{p}$ where $\frac{3}{8} < \boldsymbol{p} \leq \frac{2}{5}$, BlobT($\boldsymbol{p}$) = 4

Proof:

T he $\boldsymbol{p}$ values where $\frac{3}{8} < \boldsymbol{p} \leq \frac{2}{5}$ produce a move sequence that starts w,b,w,b,b,w,b,w,b,b. By symmetry, there are three ways to connect four triangles. Each arrangement requires six black triangles to contain its structure. Thus, if white and black

play to triangles connected to white's blob, black will contain white to a blob of size four. Black cannot contain the blob to size three since white receives a fourth move before black gets a fifth by MTCT and there being only one arrangement for three connected white triangles by symmetry. ■



(figure 4.2.1.d Triangular Blob Contained Arrangements of Size Four)

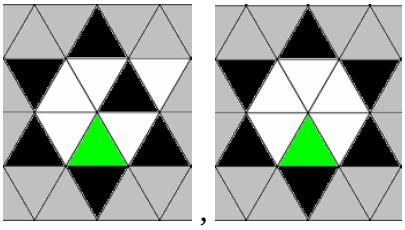
Theorem 19:

For all $\boldsymbol{p}$ where $\frac{2}{5} < \boldsymbol{p} \leq \frac{5}{11}$, BlobT($\boldsymbol{p}$) = 5.

Proof:

The $\boldsymbol{p}$ values where $\frac{2}{5} < \boldsymbol{p} \leq \frac{5}{11}$ produce a move sequences that start w,b,w,b,w,b,b,w,b,w,b, w,b,w,b,w,b,w,b,b,w,b, or w,b,w,b,w,b,w,b,w,b,b. White's first move and a connected response by black define a hexagon as illustrated below. If white moves inside the hexagon, black responds to the triangle on the black border connected to whites move. If white moves on the border, black responds inside the hexagon to the triangle connected to whites move. If white moves elsewhere or black moves twice, black moves inside the hexagon. The hexagon is connected to the six triangles occupied by black's border. This algorithm dictates black will occupy one of these triangles. Black will make a second move once before white responds with any sequence. This move is

inside the hexagon and is represented in green. Following this algorithm black will contain the blob to five triangles if white moves inside the hexagon. Black cannot contain the blob to size four with either sequence because white always receives a fifth move before black receives his sixth and Theorem 16. ■



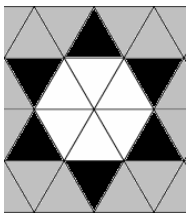
(figure 4.2.1.e Triangular Blob of Size Five)

Theorem 20:

For all $\boldsymbol{p}$ where $\frac{5}{11} < \boldsymbol{p} \leq \frac{1}{2}$, BlobT($\boldsymbol{p}$) = 6.$\left(\frac{1}{2}\right)$

Proof:

The $\boldsymbol{p}$ values where $\frac{5}{11} < \boldsymbol{p} \leq \frac{1}{2}$ define a sequence of moves w,b,w,b,w,b,w,b,w,b,w,b. White's first move is arbitrary. Black responds to a connected triangle defining a hexagon as below.



(figure 4.2.1.f The Hexagon)

Again, each triangle in the hexagon is connected to one triangle outside the hexagon. Following the same algorithm in the previous theorem, black will occupy one of these triangles meaning white will never be able to connect the blob outside of the hexagon. Black cannot contain the blob to size five by the move sequence produced and the previous theorem.■

Theorem 21:

For all version, BlobT(***p***) is not onto the Natural Numbers.

Proof:

From the theorem above its clear that $\text{BlobT}\left(\frac{1}{2}\right) = 6$ for all versions on the triangular grid.

Geometrically, there is no triangular blob size 7 that may be contained with 6 black moves. Any move ratio greater than $\left(\frac{1}{2}\right)$ does not allow black a 7^{th} move before white moves again. Thus, black may not contain white to a blob of size 7 or BlobT(***p***) is not onto the Natural Numbers.

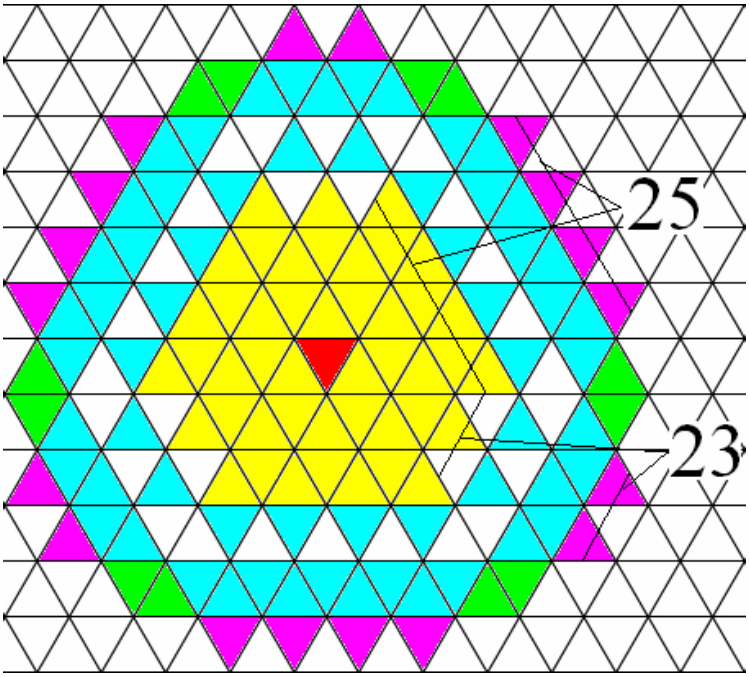
Theorem 22: Giant Triangle Theorem:

When **p** = $\frac{2}{3}$ and whites successive two moves must be connected $\text{BlobT}\left(\frac{2}{3}\right) \leq$ 933 for version III and, in particular, $\text{BlobT}\left(\frac{2}{3}\right)$ is finite.

Proof:

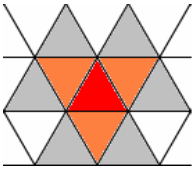
White's first move is arbitrary marked in red below. Black's first 12 moves are at the green border squares B as diagrammed below. For purposes of displaying the containment, the distances from white's first move and the containment are compressed.

The yellow region is larger with 23 triangles on the jagged yellow edges and 25 triangles on the smooth edges. The corresponding sides made up of pink border squares match the lengths of the sides of the yellow region. When white moves onto the squares in blue, black responds specifically to corresponding pink border squares as explained below.

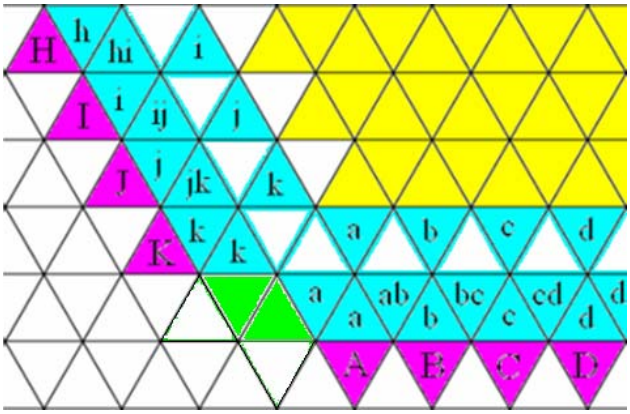


(figure 4.2.1.g Triangular Containment)

Any two moves by white before black moves again are called the first and second moves. These two moves must be connected. From any position white occupies marked in red, the possible white moves are illustrated below. Red represents a starting triangle, orange represents possible first moves, and grey represents possible second moves.

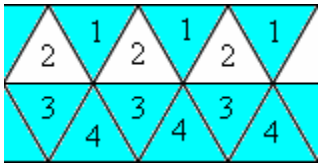

(Figure 4.2.1.h Possible Moves)

The figure below is used by black to pick specific responses on the border when white's first or second move is in the blue region. When white makes a move on the blue region black responds to a corresponding capital pink border triangle. If that border triangle is already occupied, black places moves on the border until it is complete and then places moves randomly inside the containment until white is contained.



(figure 4.2.1.i Triangle Corner)

The following system will be used to refer to triangles in different locations. The triangles are in levels 1 through 4.



(figure 4.2.1.j Triangle Locations)

If white moves into the blue region there must be a first move onto level one in this region. When white is moving from the yellow region into the blue region white may place moves into level one and then level two. At most one single lettered triangle may be moved to in this case and Black responds here.

If white has entered the blue region and has a location on level one white may make moves to levels two and one. The new white location in level one directs a response by black and black responds to this letter.

If white has entered the blue region and has a location on level one white may make moves to levels two and three. Each letter in level three that white may move to differs from the location in level one by exactly one letter. Black responds to this letter.

If white has moved through level one onto level two at least one letter must be occupied in the connected triangle in level three. If white moves to level three and then to four, at most one letter differs between the triangles in level three and level four and at least one of the letters in level 3 is occupied by black. Black moves to the other letter which may or may not be the letter in level 4. In either case, the letter in level four is occupied by black.

If white has moved through level one onto level two white may move onto level one and two. One response is directed by the new triangle in level one and black moves here.

If white has moved onto level three from level one or two the above demonstrates both attached border triangles must be occupied. White may here move to level two then one and no new moves are directed. White may also move to level four then three and at

most one new response is directed by both triangles. The responses here are always the connected and attached pink border triangles.

If white has moved onto level four white may move to level three then two and at most one new response is directed. Black moves here. White may also move to level three then four and again at most one new response is directed. The responses here are always the connected and attached pink border squares

The above considers all possible first and second moves from and onto levels one, two, three, and four. White may only move onto the border from levels three and four however blacks algorithm dictates black will occupy these border triangles. Thus white may not move onto the border and black's algorithm contains white.

There are 1243 triangles inside the containment. White may take 312 of these while black completes the border. White may then take 621 triangles inside the containment and black may take 310 before white is contained. This gives white a blob size of 933. ■

4.3 BOUNDS ON THE CRITICAL ***P*** VALUE ON THE TRIANGULAR GRID

4.3.1 BOUNDS ON VERSION I

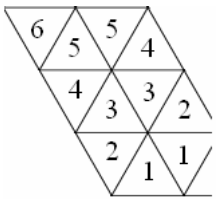
Theorem 23:

For Version 1, $\text{BlobT}\left(\frac{5}{6}+\varepsilon\right)=\infty..$

Proof:

White labels two columns on a 2-strip 1 and 2. White arbitrarily picks a triangle, labels it 1, and envisions an enumeration up the 2 strip with the integers as follows.

Columns: 1 2

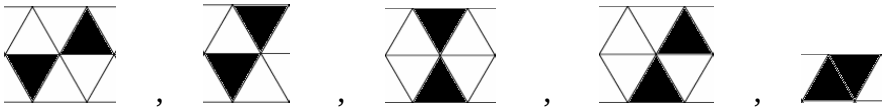


(figure 4.3.1.a Triangular 2-Strip)

For the first turn, white makes 6 connected moves on the 2-strip at (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), to start a blob. Say black moves on the 2-strip. White's following turn, in response to (1, n) by black, white moves to (2, n-2), (2, n-1), (2, n), (2, n+1), and (2, n+2). In response to (2, n) by black, white moves to (1, n-2), (1, n-1), (1, n), (1, n+1), (1, n+2). Some moves may not be possible here due to occupied triangles. The moves not placed or accounted for above are placed as follows.

Find the highest two triangles in the 1 and 2 columns that are connected to the blob. If they are equal height pick the square in column 1. Without loss of generality say column one is chosen at (1, m). Place a move at (1, m+1) and repeat until there are no additional moves to place.

Possible disconnections by black along the 2-strip will be of the form



(figure 4.3.1.b Possible Disconnections on a Triangular 2-Strip)

However, white's response to any black move on the 2-strip makes all disconnections impossible. The occasional additional move allotted to white by the ε term allows white to make an infinite number of moves connected to the blob started with the first move. ■

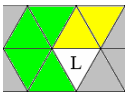
4.3.2 BOUNDS ON VERSIONS IV AND V

Theorem 24:

For Version IV & V, $\text{BlobT}\left(\frac{3}{4}\right) = \infty$

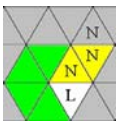
Proof:

White places a first move at an arbitrary triangle labeled L for white's last move. Black may respond to the yellow or green regions below.



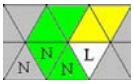
(figure 4.3.2.a Possible Black Moves)

If black does not move to a yellow or green colored region or if black moves to a triangle in the green region, white's moves to the three triangles labeled N below.



(figure 4.3.2.b White's Response to Green Moves)

If black moves to a triangle in the yellow region, white's moves to the three triangles labeled N below.

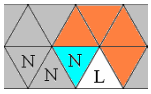


(figure 4.3.2.c White's Response to Yellow Moves)

The previous two directs result in the situation below where the blue and orange regions are unoccupied below.

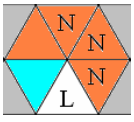
(figure 4.3.2.d Possible Black Moves)

If black does not move to a blue or orange colored region or if black moves to a triangle in the orange region, white's moves to the three triangles labeled N below.



(figure 4.3.2.e White's Response to Orange Moves)

If black moves to the blue triangle, white moves to the three triangles labeled N below.



(figure 4.3.2.f White's Response to Blue Moves)

The previous two instructions result in the situation below where the yellow and green regions are unoccupied in the first figure. All these moves are connected. Thus, by induction, white may build an infinite blob. ■

4.4 SUMMARY OF FINDINGS ON THE TRIANGULAR GRID

P Values	$0 < \boldsymbol{p} \leq \frac{1}{4}$	$\frac{1}{4} < \boldsymbol{p} \leq \frac{1}{3}$	$\frac{1}{3} < \boldsymbol{p} \leq \frac{3}{8}$	$\frac{3}{8} < \boldsymbol{p} \leq \frac{2}{5}$	$\frac{2}{5} < \boldsymbol{p} \leq \frac{5}{11}$	$\frac{5}{11} < \boldsymbol{p} \leq \frac{1}{2}$
Blob Size	1	2	3	4	5	6

(Table 4.4.a: Blob Sizes)

For Version III, $\text{BlobT}\left(\frac{2}{3}\right) \leq$ 933 when white's first and second moves must be connected.

Version	1	2	3	4	5
Bounds	$\frac{1}{2} \leq p_c \leq \frac{5}{6} + e$	$\frac{1}{2} \leq p_c \leq \infty$	$\frac{1}{2} \leq p_c \leq \infty$	$\frac{1}{2} \leq p_c \leq \frac{3}{4}$	$\frac{1}{2} \leq p_c \leq \frac{3}{4}$

(Table 4.4.b: Critical Move Ratio Bounds)

CHAPTER 5 THE HEXAGONAL GRID

5.1 GEOMETRY ON THE HEXAGONAL GRID

Now the Hexagonal grid will be explored.

5.1.1 MINIMAL HEXAGONAL CONTAINMENT

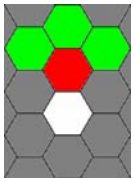
Definition: The smallest number of black moves required to contain a hexagonal blob of size ***n*** is known as the **minimal hexagonal containment**.

Theorem 24: The Minimal Hexagonal Containment Theorem:

The size of the minimal hexagonal containment for a blob of size ***n*** is $2n + 4$.

Proof:

Say a blob has ***n*** hexagons and white is placing a move connected to that blob. All containments occupy every hexagon connected to a blob. So, white is placing a move on a square that would make up a minimal containment of the blob of size ***n.*** White's placement can be connected to 1, 2, 3, 4, 5, or 6 white hexagons. The following shows regardless of how many white hexagons the placement is connected to, the difference between the size of the minimal containments for a blob of size ***n*** and a blob of size $n + 1$ will be at most 2.



(figure 5.1.1.a Hexagonal Containment Expansion)

The figure above illustrates the placement of a white move on the red square which is connected to at least 1 other white hexagon. Here the green and grey squares may be unoccupied, white, or black. Thinking of the red hexagon as part of the minimal containment of a blob of size ***n***, if white places a move on that red hexagon at most 2 additional black moves are required to contain the blob of size ***n*** + 1 (3 green hexagons minus 1 red hexagon). If any of the green hexagons are occupied by black or white, the number of black hexagons required to contain the blob decreases by 1 for each green hexagon occupied. So an increase of 2 occurs only when the placement is connected to 1 other white hexagon and the 3 other connected hexagons are unoccupied.

If either of the 2 grey hexagons above and below the white hexagon are occupied by white, the size of the containment will not increase by 2.

It takes 6 moves to contain a blob of size 1. Adding one white hexagon to this blob can only increase the blob size by 2. Thus the smallest number of hexagons required to contain a blob of size ***n*** is 2(***n***-1) + 4 = 2***n*** + 4. ■

Lemma 9:

If BlobH(***p***) = ***n***, $\boldsymbol{p} \geq \left(\frac{n}{3n+4}\right)$.

Proof:

Black may move to hexagons connected to white hexagons. There are at most $2n+4$ hexagons attached to a blob of size ***n*** by theorem 21. Thus ***p*** must be great than or equal to $\left(\frac{n}{2n+4}\right)$. ■

5.2 ***P*** VALUES ON THE HEXAGONAL GRID

5.2.1 SMALL ***P*** VALUES ON THE HEXAGONAL GRID

Theorem 26:

For all $\boldsymbol{p}$ where $0 < \boldsymbol{p} \leq \frac{1}{7}$, BlobH($\boldsymbol{p}$) = 1.

Proof:

On the hexagonal grid with six moves, black can contain any individual white move by moving to the six connected squares. ■

(figure 5.2.1.a Hexagonal Blob of Size One Contained)

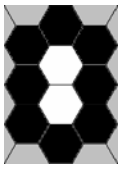
Theorem 27:

For all $\boldsymbol{p}$ where $\frac{1}{7} < \boldsymbol{p} \leq \frac{1}{5}$, BlobH($\boldsymbol{p}$) = 2.

Proof:

There is one way to connect two hexagons on the grid. Black cannot contain white when $\boldsymbol{p}$ is greater than $\frac{1}{7}$ because black does not receive six moves before white moves again. Also, any move ration less than or equal to $\frac{1}{5}$ will give black at least eight

moves before white moves a third time. The conclusion follows from the minimal hexagonal containment theorem.



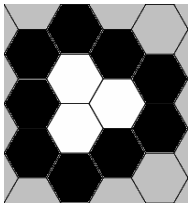
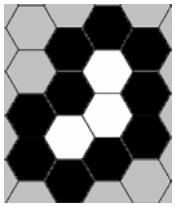
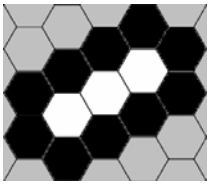
(figure 5.2.1.b Hexagonal Blob of Size Two Contained)

Theorem 28:

When $\frac{1}{5} < \boldsymbol{p} \leq \frac{3}{13}$, BlobH($\boldsymbol{p}$) = 3

Proof:

There are three ways to connect three hexagons requiring nine to ten squares to contain them. Without an eight move before white moves again, black cannot contain white's blob to two by the previous theorem. $\boldsymbol{p} = \frac{3}{13}$ produces the move sequence w,b,b,b,w,b,b,b,w,b,b,b,w. Anything less than $\frac{3}{13}$ produces the move sequence w,b,b,b,w,b,b,b,w,b,b,b,b allowing black to contain any blob of size three ■

 ,  , 

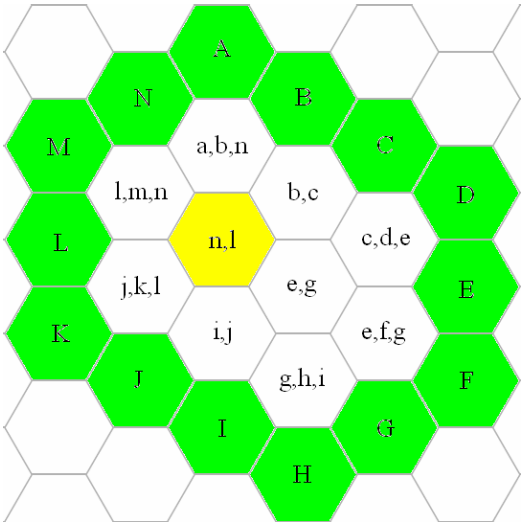
(figure 5.2.1.c Hexagonal Blobs of Size Three Contained)

5.2.2 MIDDLE ***P*** VALUES ON THE HEXAGONAL GRID

Theorem 29: Little Hexagon:

For Version III, $\text{BlobH}\left(\frac{1}{3}\right) \leq 8$.

Proof:



(figure 5.2.2.a BlobH of $\frac{1}{3}$ Contained)

White's first move is at an arbitrary square marked in yellow. Black then responds to the hexagons labeled N and L. The lower case letters direct black's movement at the corresponding capital letter in the green hexagons. Black responds to all the lower case letters that have not already been responded to. Black places moves on the border until it is complete and then places moves inside the border until white is contained. The following will show white cannot move onto the green border.

White may move onto a border hexagon from any hexagon connected to that border hexagon. Any white hexagon connected to two border hexagons direct black to

respond to the two connected border hexagons. So, white can not move onto a border hexagon from a hexagon with two letters.

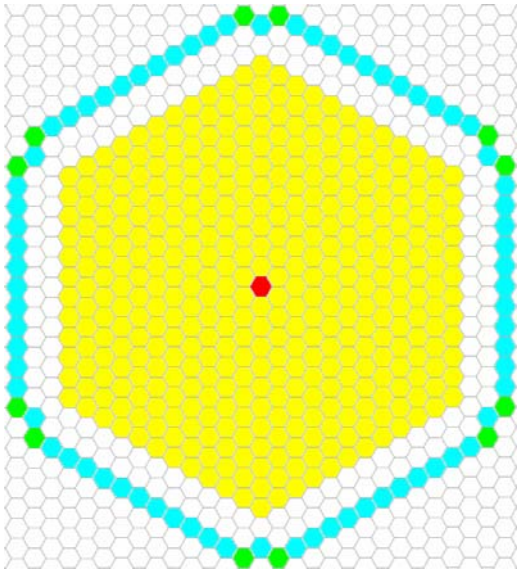
White may also move onto a border hexagon from a white hexagon connected to three border hexagons. However, white must first move onto one of these three lettered hexagons before white may move onto the border. Each hexagon connected to a white three lettered hexagon is a border hexagon or a white hexagon. Connected white hexagons direct one of the moves directed by the three lettered white hexagon. Thus, black will never be directed to respond to more than two moves. These moves are the two connected border hexagons. The other border hexagon will already be occupied

There are ten hexagons inside the green border. White may take seven of them and then black's completes the border. White takes an additional hexagon and black takes the remaining two. Thus, $\text{BlobH}\left(\frac{1}{3}\right) \leq 8$ ■

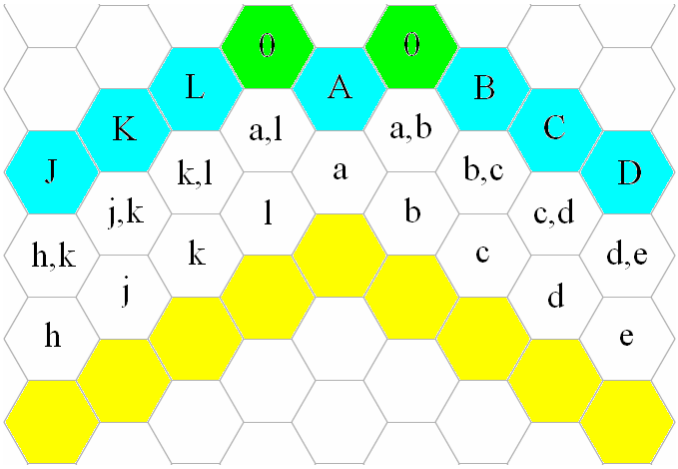
Theorem 30: Big Hexagon Theorem:

For Version III $\text{BlobH}\left(\frac{1}{2}\right) \leq 294$, or in particular $\text{BlobH}\left(\frac{1}{2}\right) < \infty$.

Proof:



(figure 5.2.2.b Big Hexagon)



(figure 5.2.2.c A Section of Big Hexagon)

Whites first move is at an arbitrary hexagon marked in red. Whites first 12 moves can not leave the yellow hexagon. Blacks first 12 moves are at the green border hexagons. The above labeling applies to any side and corner. The following will show white cannot move onto the blue and green border.

The lower case letters direct responses by white at the corresponding capital letters. Black ignores any directions already followed. White may move onto the border from a single lettered hexagon or a double lettered hexagon. All single lettered hexagons are marked “a” and direct a response to the only attached border hexagon “A”. Thus white cannot move onto a border hexagon from a single lettered hexagon.

The double lettered hexagons are enveloped by single lettered hexagons as is white’s first move. If white moves onto double lettered hexagons, white must first move onto a single lettered hexagon due to the envelope. Any single lettered hexagons connected to a double lettered hexagon direct one of the moves directed by that double lettered hexagon. Thus, in moving onto a double lettered hexagon from a single lettered hexagon, black will only be directed to make one of the two moves. These two directions are always the two border hexagons connected to the double lettered hexagon. So, white cannot move onto a border from a double lettered hexagon moved to from a single lettered hexagon.

If white has moved onto a double lettered hexagon from a single lettered hexagon, white can then move from that double lettered hexagon to other connected double lettered hexagons. Any double lettered hexagon attached to another double lettered hexagon each direct at most one different move. So, in moving from a double lettered hexagon to another, at most one new move will be directed. Again, these two directions are always

the two connected border hexagons. Thus, white cannot move onto a border hexagon from a single lettered hexagon and white cannot move onto a border hexagon from a double lettered hexagon and finally white cannot move onto the border.

There are 84 hexagons on the border and 503 inside of it. White may take 84 hexagons inside the border white black completes it leaving 419. White may take 210 of these and black may take 209 since white moves first allowing white a blob of 294 squares. Thus, $\text{BlobH}\left(\frac{1}{2}\right) \leq 294$. ■

5.3 BOUNDS FOR THE CRITICAL ***P*** VALUE ON THE HEXAGONAL GRID

5.3.1 BOUNDS ON VERSION I ON THE HEXAGONAL GRID

Theorem 31:

For Version I, where $0 < \varepsilon < \frac{1}{3}$, $\text{BlobH}\left(\frac{2}{3}+\varepsilon\right) = \infty$.

Proof:

White labels two columns on a 2-strip 1 and 2. White arbitrarily picks a square value, labels it 0, and envisions an enumeration up the 2 strip with the integers as follows.

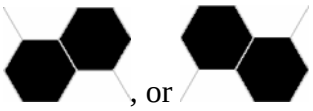
(figure 5.3.1.a Hexagonal 2-Strip)

For the first turn, white makes two connected moves on the 2-strip at (1, 0) (1, 1) to start a blob. Say black moves on the 2-strip. White's following turn, in response to (1, n) by black, white moves to (2, n-1), (2, n) or, in response to (2, n) by black, white moves to (1,

n), (1, n+1). Some moves may not be possible here due to occupied hexagons. The moves not placed or accounted for above are placed as follows.

> Find the highest two hexagons in the 1 and 2 columns that are connected to the designated blob. If they are equal height pick the square in column 1. WLOG say column one is chosen at (1, m). Place a move at (1, m+1) and repeat until there are no additional moves to place.

Possible disconnections by black along the 2-strip will be of the form



(figure 5.3.1.b Hexagonal 2-Strip Disconnections)

However, white's response to any black move on the 2-strip makes all disconnections impossible. The occasional additional move allotted to white by the *e* term allows white to make an infinite number of moves connected to the blob. ■

5.3.2 BOUNDS ON VERSION V ON THE HEXAGONAL GRID

Theorem 32:

For Version V, $\text{BlobH}\left(\frac{1}{2}\right) = \infty$

Proof:

White places a move at an arbitrary hexagon on the grid and black may respond to the two hexagons in green below.

(figure 5.3.2.a Possible Black Placements)

For any turn, if black responds to one of green hexagons where W is white's last move, white moves to the other the following turn. If black does not respond to one of the green hexagons, white places a move at the hexagon above white's last move. Either case results in the situation above where black may move to either of the green squares. White creates an infinite blob with this strategy. ■

5.4 SUMMARY OF FINDINGS ON THE HEXAGONAL GRID

Recall Versions II and III are the same as are IV and V on the hexagonal grid.

P Values	$0 < \boldsymbol{p} \leq \frac{1}{7}$	$\frac{1}{7} < \boldsymbol{p} \leq \frac{1}{5}$	$\frac{1}{5} < \boldsymbol{p} \leq \frac{3}{13}$
Blob Sizes	1	2	3

(Table 5.4.a: Blob Sizes)

For Version 2 and 3 $\text{BlobH}\left(\frac{1}{3}\right) \leq 8$ and $\text{BlobH}\left(\frac{1}{2}\right) \leq 294$

Version	1	2	3	4	5
Bounds	$\frac{3}{13} \leq p_c \leq \frac{2}{3} + \varepsilon$	$\frac{1}{2} \leq p_c \leq \infty$	$\frac{1}{2} \leq p_c \leq \infty$	$\frac{3}{13} \leq p_c \leq \frac{1}{2}$	$\frac{3}{13} \leq p_c \leq \frac{1}{2}$

(Table 5.4.b: Critical Move Ratio Bounds)

CHAPTER 6

SQUARE GRID AI

6.1 INTRODUCTION TO GAME TREES

One way to keep track of the moves of a game is with a **game tree**. A game tree is a **directed graph** with **vertices** (or **nodes**) and **edges**. The nodes (or vertices), are points at which players can take actions, connected by edges, which represent the actions that may be taken at that node. An initial (or root) node represents the first decision to be made. The **depth** of a particular game tree corresponds directly with the number of moves that have been made. The **complete game tree** for a game is the game tree starting at the initial node where each node contains all possible moves from that position. A complete game tree describes all the possible strategies the two players may employ as the nodes represent the choices.

6.2 COMPUTER GENERATED GAME TREES

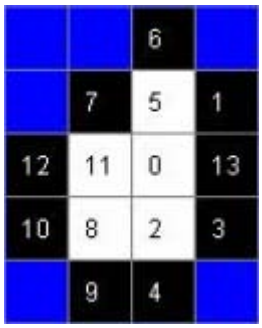
Computer generated game trees are perhaps the most useful game trees. This is due to the computational power of the computer. This gives us a tool to do advanced case arguments. A program was made using Java to produce computer generated game trees of the game blob. The program creates game trees on versions I, IV, and V on the square grid. However, a bounded box must be used when analyzing version I. At its current status, the program can eliminate symmetrical positions and prune itself for versions IV and V in search of an "**optimal path"** or a sequence of moves that reflects the best strategies for white and black. The program generates the trees depth first, and prunes the trees while they're being created to avoid running out of memory. It uses the idea of an expected blob size to prune a large portion of the nodes that do not result in containments after a given number of moves. Also an AI that picks moves for Square Version V has been created and was used in conjunction with the pruner to find the best white strategy or the best black strategy against the AI for a specific move ratio.

6.3 AI RESULTS

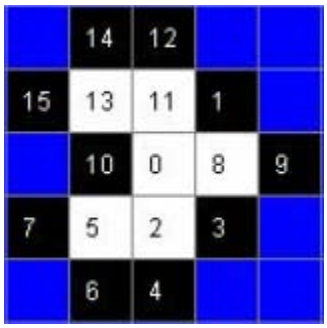
The AI for black may be run picking black moves in a tree that considers all possible white moves. Also, the AI for white may be run in a tree that considers all possible black moves. The pruner used returns the best strategy for white against the black AI or the best strategy for black against the white AI depending on the settings of the AI.

The AI for black when run in conjuction with all possible white moves returns an upper bound on the maximum blob size white may obtain against blacks best strategy. Likewize, the AI for white when run in conjunction with all possible black moves returns an upper bound on the minimum blob size white may obtain against blacks best strategy. However, the number of possible black moves far outweights the number of possible white moves and the only trees with substantial depth produced signifigant results. These came when black's AI when run against all white strategies.

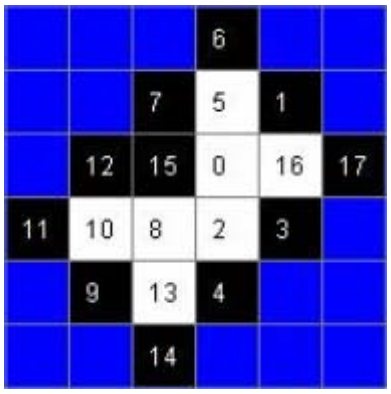
The AI was designed as a function of the current position and the move sequence. The results or upper bounds on the blob sizes for different move ratios are shown below.



(figure 6.3.a $\frac{1}{3} < \boldsymbol{p} \leq \frac{5}{14}$)

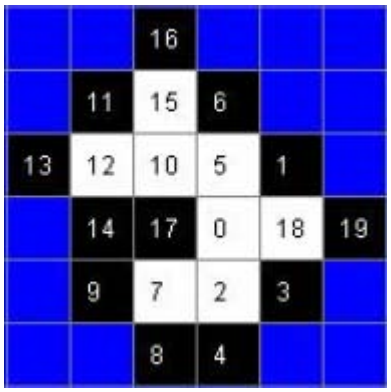


(figure 6.3.b $\frac{5}{14} < \boldsymbol{p} \leq \frac{4}{11}$)



(figure 6.3. $\frac{4}{11} < \boldsymbol{p} \leq \frac{3}{8}$)

(figure 6.3.d $\frac{3}{8} < \boldsymbol{p} \leq \frac{5}{13}$)



(figure 6.3.e $\frac{5}{13} < \boldsymbol{p} \leq \frac{7}{18}$)

(figure 6.3.f $\frac{7}{18} < \boldsymbol{p} \leq \frac{2}{5}$)

(figure 6.3.g $\frac{2}{5} < \boldsymbol{p} \leq \frac{9}{22}$)

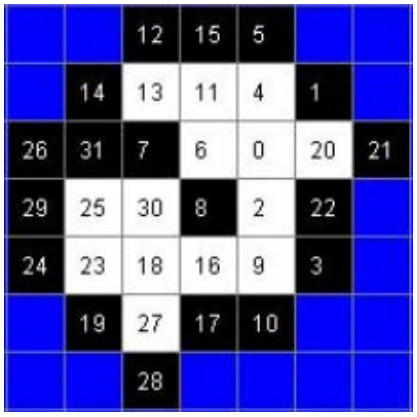
(figure 6.3.h $\frac{9}{22} < \boldsymbol{p} \leq \frac{7}{17}$)

(figure 6.3.i $\frac{7}{17} < \boldsymbol{p} \leq \frac{5}{12}$)

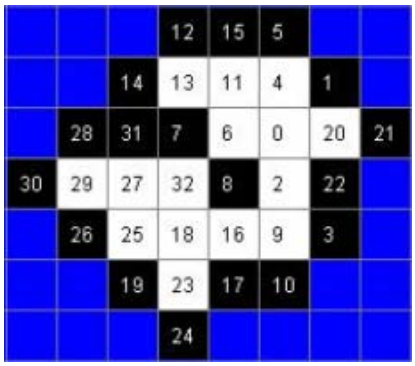
(figure 6.3.j $\frac{5}{12} < \boldsymbol{p} \leq \frac{8}{19}$)

(figure 6.3.k $\frac{8}{19} < \boldsymbol{p} \leq \frac{11}{26}$)

(figure 6.3.l $\frac{11}{26} < \boldsymbol{p} \leq \frac{3}{7}$)

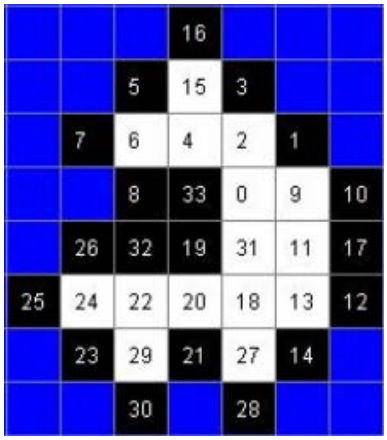


(figure 6.3.m $\frac{3}{7} < \boldsymbol{p} \leq \frac{13}{30}$)

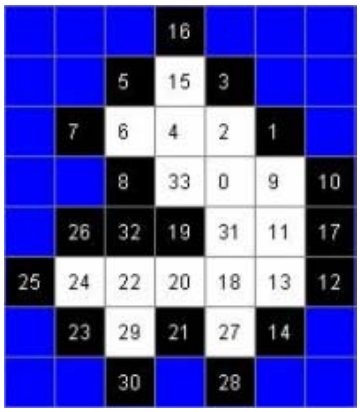


(figure 6.3.n $\frac{13}{30} < \boldsymbol{p} \leq \frac{10}{23}$)

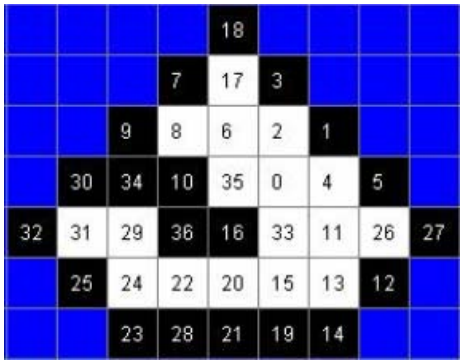
(figure 6.3.o $\frac{7}{16} < \boldsymbol{p} \leq \frac{11}{25}$)



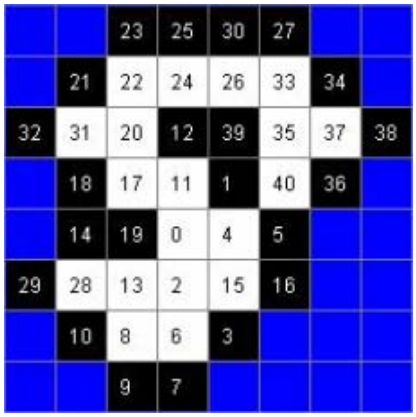
(figure 6.3.p $\frac{11}{25} < \boldsymbol{p} \leq \frac{15}{34}$)



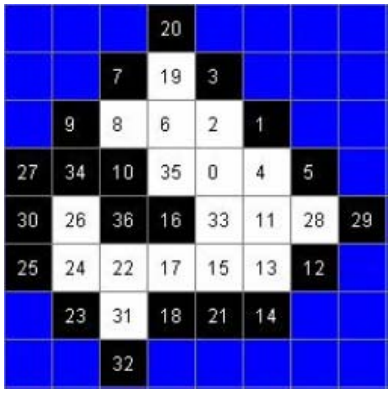
(figure 6.3.q $\frac{15}{34} < \boldsymbol{p} \leq \frac{4}{9}$)



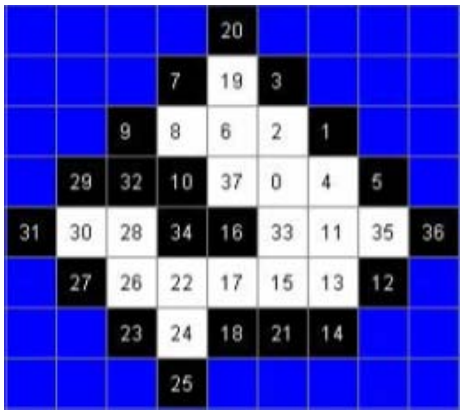
(figure 6.3.r $\frac{4}{9} < \boldsymbol{p} \leq \frac{13}{29}$)



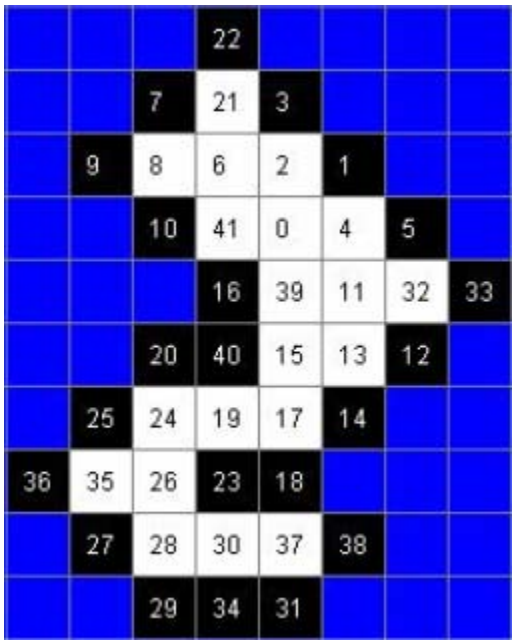
(figure 6.3.s $\frac{13}{29} < \boldsymbol{p} \leq \frac{9}{20}$)



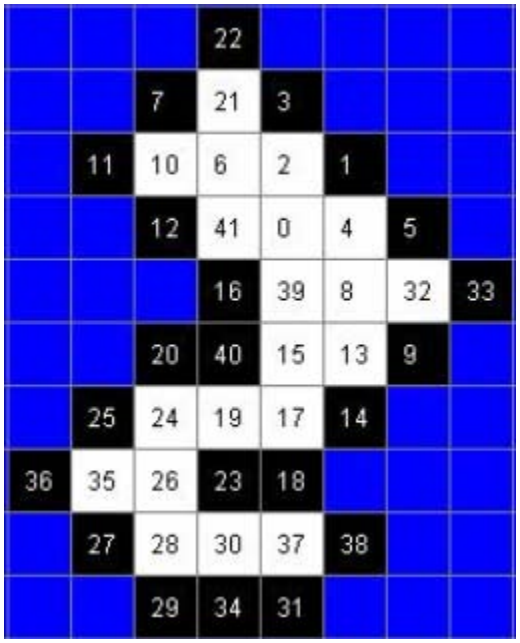
(figure 6.3.t $\frac{9}{20} < \boldsymbol{p} \leq \frac{14}{31}$)



(figure 6.3.u $\frac{14}{31} < \boldsymbol{p} \leq \frac{5}{11}$)

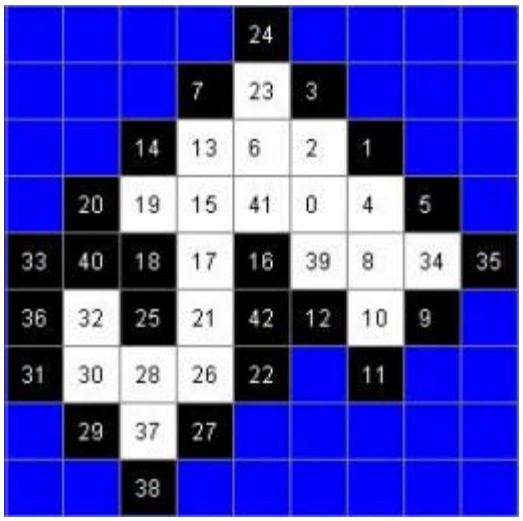


(figure 6.3.v $\frac{5}{11} < \boldsymbol{p} \leq \frac{16}{35}$)

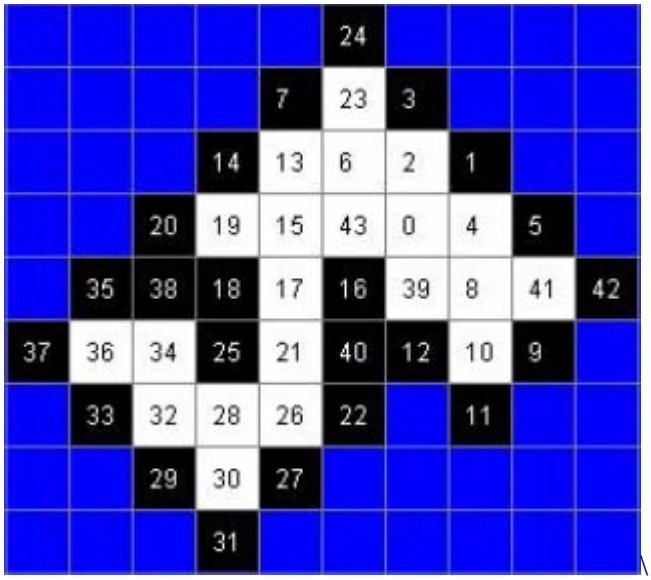


(figure 6.3.w $\frac{5}{11} < \boldsymbol{p} \leq \frac{16}{35}$)

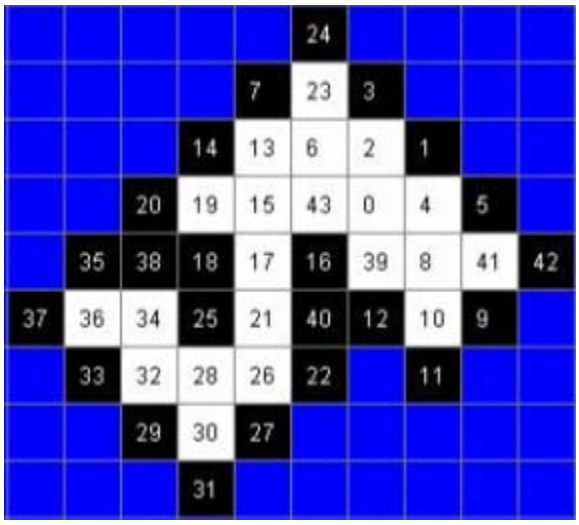
(figure 6.3.x $\frac{5}{11} < \boldsymbol{p} \leq \frac{16}{35}$)



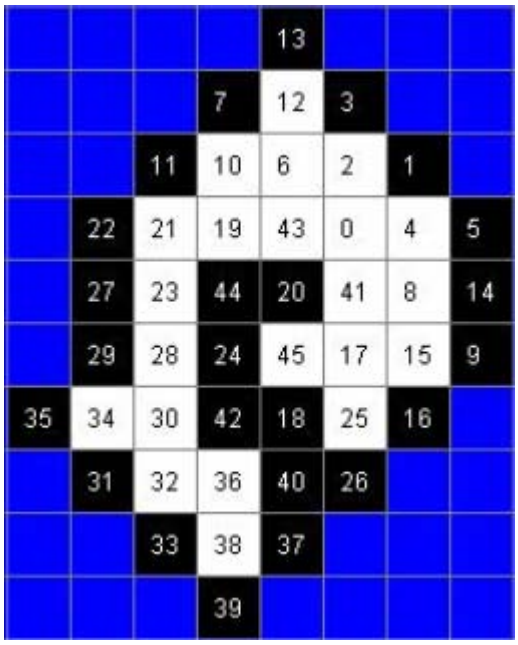
(figure 6.3.y $\frac{11}{24} < \boldsymbol{p} \leq \frac{17}{37}$)



(figure 6.3.z $\frac{17}{37} < \boldsymbol{p} \leq \frac{6}{13}$)

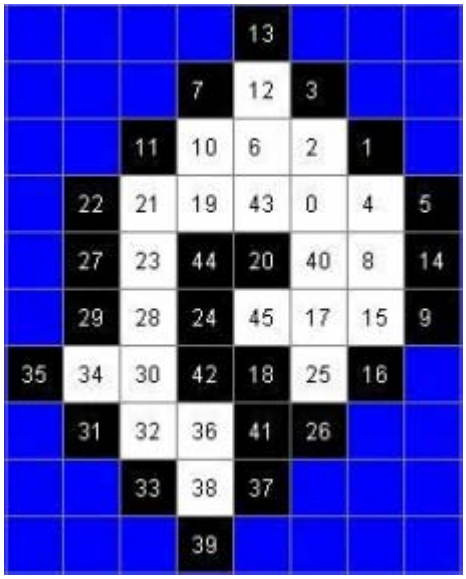


(figure 6.3.aa $\frac{6}{13} < \boldsymbol{p} \leq \frac{19}{41}$)



(figure 6.3.ab $\frac{6}{13} < \boldsymbol{p} \leq \frac{19}{41}$)

(figure 6.3.ac $\frac{6}{13} < \boldsymbol{p} \leq \frac{19}{41}$)

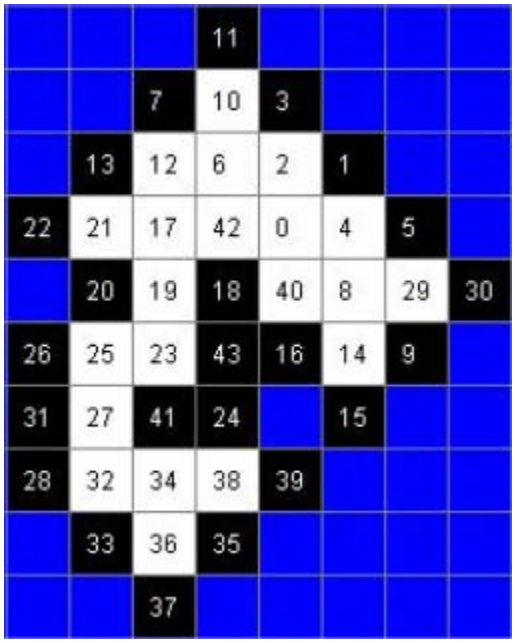


(figure 6.3.ad $\frac{6}{13} < \boldsymbol{p} \leq \frac{19}{41}$)

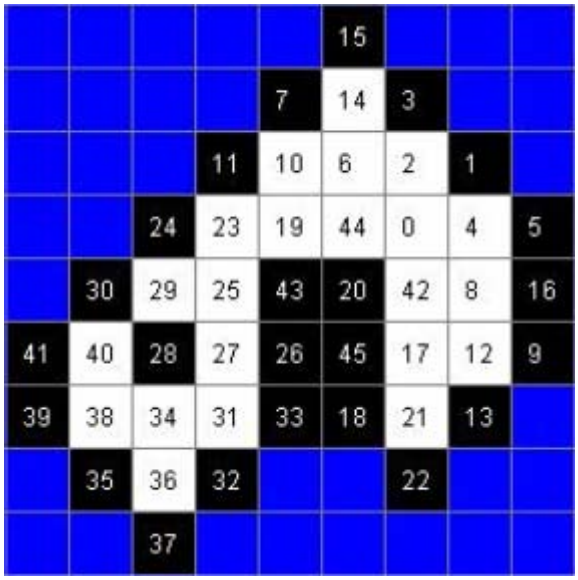
(figure 6.3.ae $\frac{7}{15} < \boldsymbol{p} \leq \frac{20}{43}$)

(figure 6.3.af $\boldsymbol{p} = \frac{7}{15} < \boldsymbol{p} \leq \frac{20}{43}$)

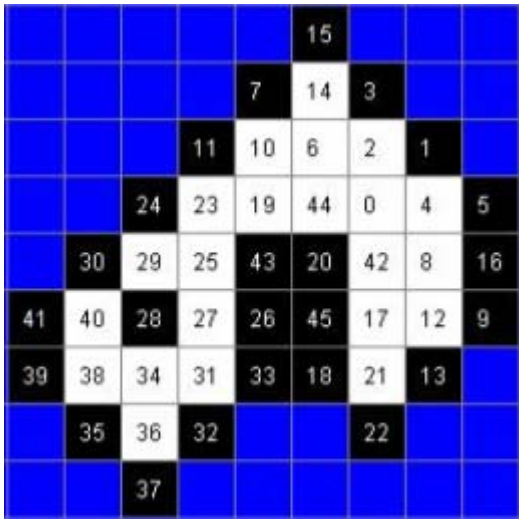
(figure 6.3.ag $\frac{7}{15} < \boldsymbol{p} \leq \frac{15}{32}$)



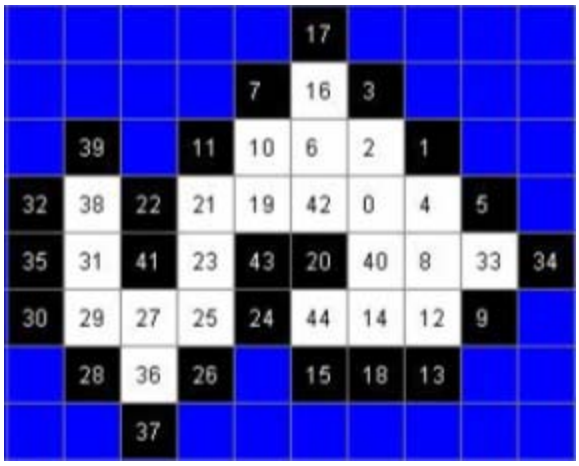
(figure 6.3.ah $\frac{7}{15} < \boldsymbol{p} \leq \frac{15}{32}$)



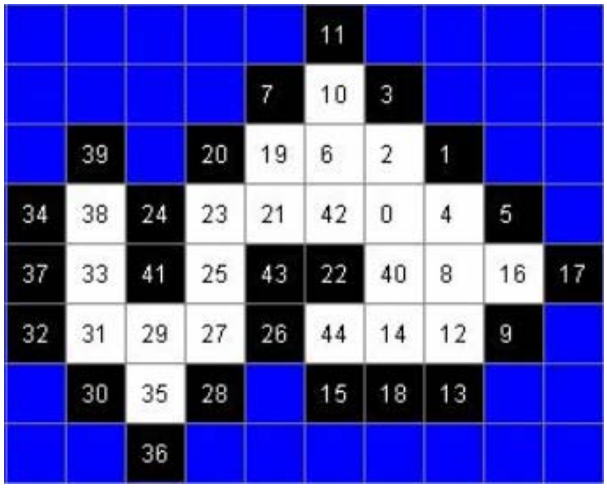
(figure 6.3.ai $\frac{15}{32} < \boldsymbol{p} \leq \frac{8}{7}$)



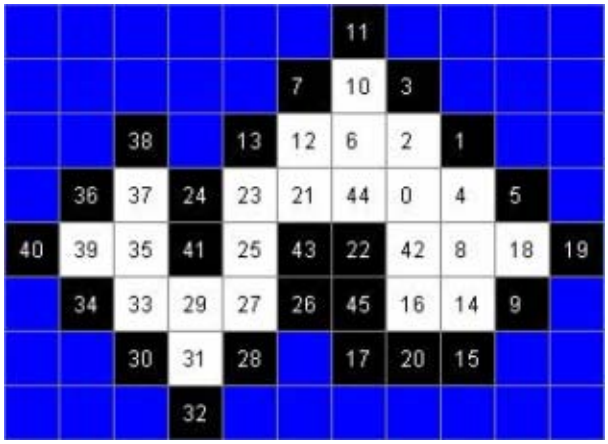
(figure 6.3.aj $\frac{15}{32} < \boldsymbol{p} \leq \frac{8}{17}$)



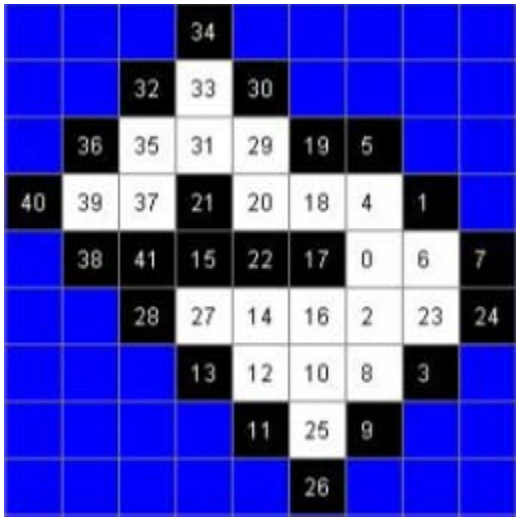
(figure 6.3.ak $\frac{8}{17} < \boldsymbol{p} \leq \frac{17}{36}$)



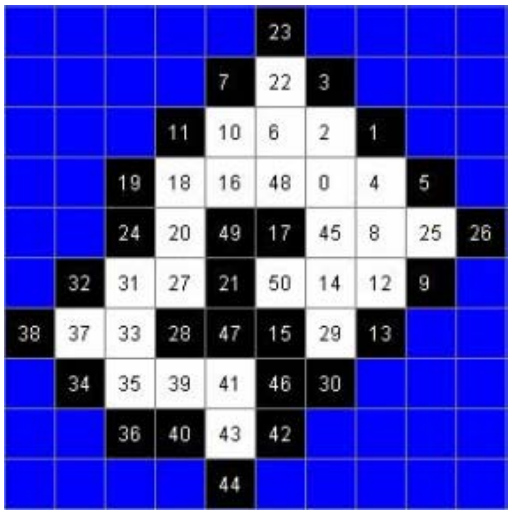
(figure 6.3.al $\frac{17}{36} < \boldsymbol{p} \leq \frac{19}{40}$)



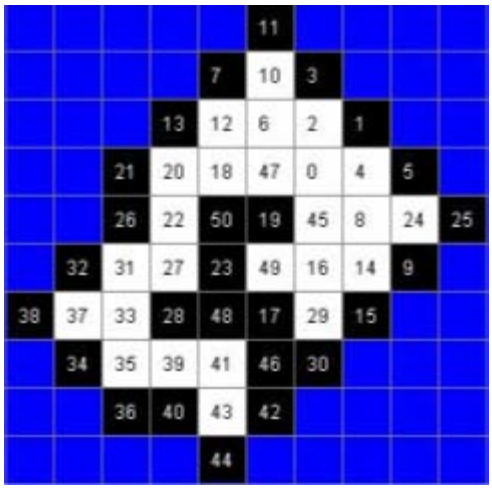
(figure 6.3.am $\frac{9}{19} < \boldsymbol{p} \leq \frac{19}{40}$)



(figure 6.3.an $\frac{19}{40} < \boldsymbol{p} \leq \frac{10}{21}$)



(figure 6.3.ao $\frac{11}{23} < \boldsymbol{p} \leq \frac{23}{48}$)



(figure 6.3.ap $\frac{12}{25} < \boldsymbol{p} \leq \frac{13}{27}$)

Move Ratios				Blob Size Bound	
	1		5		5
	3		14		
	5		4		6
	14		11		
	4		3		7
	11		8		
	3		5		8
	8		13		
	5		7		9
	13		18		
	7		2		9
	18		5		
	2		9		10
	5		22		
	9		7		11
	22		17		
	7		5		12
	17		12		

(table: 6.3.a BlobSizes 1)

Move Ratios				Blob Size Bound	
	5		11		13
	12		26		
	11		13		14
	26		30		
	13		10		15
	30		23		
	7		11		14
	16		25		
	11		15		15
	25		34		
	15		4		16
	34		9		
	4		13		17
	9		29		
	13		9		19
	29		20		
	9		14		17
	20		31		

(table: 6.3.a BlobSizes 2)

Move Ratios				Blob Size Bound	
	14		5		18
	31		11		
	5		17		20
	11		37		
	17		19		21
	37		41		
	19		7		22
	41		15		
	7		15		21
	15		32		
	15		19		22
	32		40		
	19		10		20
	40		21		
	11		23		24
	23		48		
	12		13		24
	25		27		

(table: 6.3.a BlobSizes 2)

CHAPTER 7

FUTURE WORK

7.1 EXPANDED FIGURES

For Versions II and III, arguments similar to Little Diamond, Big Diamond, Big Hexagon and Giant Diamond offer potential for better bounds on the critical move ratio. Expanding the figures may allow the placement of moves to create a **dense border**. A dense border is a border with moves evenly placed a set number of spaces apart. For example, a border that is $\frac{1}{2}$ dense has a move placed every other location. A border that is $\frac{2}{21}$ dense has two moves placed after 19 empty locations. It seems obvious that certain border densities may be created. The problem here is producing an algorithm that prevents white from moving onto a border with a given density.

7.2 BETTER UPPER BOUNDS

The computer generated game tree program offers potential for case arguments that could provide better upper bounds on the critical move ratios for the various versions. The concept of **essentially connected** moves for the versions may be used to get better upper bounds. Moves are essentially connected when white may connect the two moves the following turn wherever black moves. This may allow white to build a chain of loosely connected islands that could be connected when the connections are threatened.

7.3 THE CRITICAL MOVE RATIO FOR HEXAGONAL VERSION V

The critical move ration for Hexagonal Version V is believed to be $\frac{1}{2}$. It has been shown that white can escape at $\boldsymbol{p} = \frac{1}{2}$ but it remains to be shown that black can contain white when $\boldsymbol{p}$ is less than $\frac{1}{2}$.

Black may create what are called Tentacles. These tentacles limit the direction white may expand the blob by damming the tentacles on one side. The extra move allotted when the move ratio is less than $\frac{1}{2}$ to black allows black to **turn** tentacles preventing them from growing in a specific direction. Eventually it is believed that black has a strategy to turn all these tentacles back towards eachother which will eventually contain the blob.

7.4 THE CRITICAL MOVE RATIO FOR SQUARE VERSION V

For Square Version V if all black moves must be connected to white, it is believe that the critical move ratio is $\boldsymbol{p} = \frac{1}{2}$. It has been shown white can escape at $\boldsymbol{p} = \frac{1}{2}$ but it remains to be shown that black has a strategy that guaranteed to contain any white strategy when $\boldsymbol{p}$ is less than $\frac{1}{2}$

7.5 BETTER RESULTS FOR SMALL P

Through case argument better results for small ***p*** values are possible. But as the move ratio gets bigger the number of cases grows quickly making this approach cumbersome. The game tree program has help here providing upper bounds on the maximum blob size white may obtain.

Heuristics or rules governing the movement of black may have the potential to reduce the complexity of the game tree program allowing better results. However, programming a better artificial intelligence to pick more appropriate move for black for any given move ratio and any given position appears difficult.

7.6 UPPER BOUNDS FOR VERSION II AND III

Providing upper bounds for Versions II and III has proven remarkably hard to do. It appears obvious that, with enough moves, white may completely envelope any potential containment black may try to create. However creating an algorithm that describes exactly how white needs to move to envelope black and avoid containment has not yet been achieved.

7.7 BLOB_(***P***)

Blob_(***p***) is believed to be a strictly non-decreasing, left continuous function with an asymptote at x = p_c. None of these have been shown.

REFERENCES

[1] Luce, Duncan, and Howard Raiffa. GAMES AND DECISIONS. New York: John Wiley & Sons, Inc.:1958.

[2] Berlekamp, Elwyn, John Conway, and Richard Guy. Winning Ways for your mathematical plays. New York: Academic Press Inc.: 1982.

BIOGRAPHICAL SKETCH

Adam Jay Holers was born on August 3rd 1984 in Defiance Ohio to his proud parents Bobbie and Jeff Holers. He attended Deland High School and is currently an undergraduate at Stetson pursuing his Pre-Med and Mathematics interests. An avid soccer player, Adam won defensive player of the year at Deland High School and led his intramural soccer team to the championship in 2005 with an astounding goal total of zero. He has always had trouble shooting.

Adam eats chocolate ice cream and fried rice but not together. His favorite things include starcraft, chess, apple juice, orange juice, grape juice, cranberry juice, and various fruity juices.

Adam volunteers at the Deland Hospital, tutors mathematics, and does habitat for humanity construction. One day he hopes to leave Stetson to pursue graduate studies.

Appendix A:

Code for Gaming Tree Analyzer

```
/*
 * Created on Sep 4, 2006
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */

/**
 * @author Adam
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */

public class TreeAnalyzer
{
   public static int    BOARDSIZE = 40;
   private int          UNOCCUPIED = 0, WHITE = 1, BLACK = 2;
   private double       myWhiteMoves, myBlackMoves, myMoveRatio;
   private GamingTree   myGamingTree;
   private int[]        myMoveArray, myAdjacentPossibleMoves;
   private int[][]      myBoardArray;
   private int[][]      my1Moves, my2Moves;
   private int[]        myFutureMovesArray;
   private int[]        myMoveOrderArray;
               private int          mySizeNumber;
               private SquareNode[] mySquareNodes;
               private SquareSymmetryChecker mySquareSymmetryChecker;

/* 8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
 * Constructors
  8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888*/

   public TreeAnalyzer()
   {
     myWhiteMoves       = 1;
     myBlackMoves       = 1;
     myMoveRatio        = .5;
     myGamingTree       = null;
                 myMoveArray           = new int[0];
                 myBoardArray          = new int[BOARDSIZE][BOARDSIZE];
                 myFutureMovesArray    = new int[100];
                 myMoveOrderArray      = new int[100];
                 myAdjacentPossibleMoves = new int[16];
                 mySizeNumber          = 0;
                 mySquareNodes         = null;
                 mySquareSymmetryChecker = new SquareSymmetryChecker();
                 my1Moves = null;
                 my2Moves = null;
                 resetBoard();
                 resetFutureMoves();
                 myMoveRatio = 0;
   }

   public TreeAnalyzer(GamingTree tree)
   {
                           myWhiteMoves       = 1;
                           myBlackMoves       = 1;
                           myMoveRatio        = .5;
               myGamingTree       = tree;
                           myMoveArray              = new int[0];
```

```
                        myAdjacentPossibleMoves = new int[16];
                        myBoardArray              = new int[BOARDSIZE][BOARDSIZE];
                        myFutureMovesArray  = new int[100];
                        myMoveOrderArray    = new int[100];
                        mySizeNumber        = 0;
                        mySquareNodes       = null;
                        mySquareSymmetryChecker = new SquareSymmetryChecker();
                my1Moves = null;
                my2Moves = null;
                        resetBoard();
                        resetFutureMoves();
                        myMoveRatio = 0;
    }

/*8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
 * Accessor Methods
 8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888*/

    public GamingTree getGamingTree()
    {
                return myGamingTree;
    }

    public void setGamingTree(GamingTree tree)
    {
                myGamingTree = tree;
    }

    public double getWhiteMoves()
    {
                return myWhiteMoves;

    }

                public double getBlackMoves()
                {
                        return myWhiteMoves;
                }

                public void setWhiteMoves(double w)
                {
                        myWhiteMoves = w;
                }

                public void setBlackMoves(double b)
                {
                        myBlackMoves = b;
                }

                public int[] getMoveArray()
                {
                        return myMoveArray;
                }

                public void setMoveArray(int[] m)
                {
                        myMoveArray = m;
                }

                public int[][] getBoard()
                {
                        return myBoardArray;
                }

                public void setBoard(int[][] b)
                {
                        myBoardArray = b;
                }

                public int getBoardMove(int x, int y)
```

```
		{
			return myBoardArray[x][y];
		}

		public void setmyMoveRatio(double d)
		{
			myMoveRatio = d;
		}

		public double getmyMoveRatio()
		{
			return myMoveRatio;
		}

		public void setSquareSymmetryChecker(SquareSymmetryChecker s)
		{
			mySquareSymmetryChecker = s;
		}

		public SquareSymmetryChecker getmySquareSymmetryChecker()
		{
			return mySquareSymmetryChecker;
		}

		public void setBoardMove(int x, int y, int m)
		{
			myBoardArray[x][y] = m;
		}

	public int[][] getmy1Moves()
	{
		return my1Moves;
	}

	public int[][] getmy2Moves()
	{
		return my2Moves;
	}

		public int[] getFutureMoves()
		{
			return myFutureMovesArray;
		}


		public void setFutureMoves(int[] f)
		{
			myFutureMovesArray = f;
		}

		public int[] getMoveOrder()
		{
			return myMoveOrderArray;
		}

		public void setMoveOrder(int[] m)
		{
			myMoveOrderArray = m;
		}


/* 88888888888888888888888888888888888888888888888888888888888888888888888888888888
 * Class Algorithms
  88888888888888888888888888888888888888888888888888888888888888888888888888888888*/

	public void resetBoard()
	{
			for(int x = 0; x < BOARDSIZE; x++)
			{
			  for(int y = 0; y < BOARDSIZE; y++)
```

```
                        {
                                myBoardArray[x][y] = UNOCCUPIED;
                        }
                    }
    }

    public void setAdjacentPossibleMoves(int[] a)
    {
            myAdjacentPossibleMoves = a;
    }

    public int[] getAdjacentPossibleMoves()
    {
            return myAdjacentPossibleMoves;
    }

    public void resetFutureMoves()
    {
                    myFutureMovesArray = new int[100];
                    for(int i = 0; i < 100; i++)
                    {
                            myFutureMovesArray[i] = 100;
                    }
    }

    public boolean isBlacksMove(int i)
    {
            return(myMoveOrderArray[i] == 2);
    }

    public boolean isWhitesMove(int i)
    {
            return(myMoveOrderArray[i] == 1);
    }

/*
 * Generates an array of the previous moves and stores them under (MyMoveArray)
 * in such a way that the X coordinate and the Y Coordinate Alternate for the various
 * levels. xyxyxy = 012345 There are thus twice as many components of the array
 * as there are moves.* Note: the moves are stored in reverse order,
 * newest move >> first move.
 */
            public void generateMoveArray(GamingTree tree)
            {
                    int level = tree.getLevel();
                    int index = 0;
                    myMoveArray = new int[2*level+2];
                    GamingTree temptree;
                    temptree = tree;
                    while(temptree != null)
                    {
                            //System.out.println(myMoveArray.length + " myMoveArrayLength");
                            //System.out.println(index + " index");
                            myMoveArray[index]   = temptree.getX();
                            myMoveArray[index+1] = temptree.getY();
                            index = index+2;
                            temptree = temptree.getParent();
                    }
            }

/*
 * This Populates the Board with moves from myMoveArray, The moves are
 * stored in Order xyxyxy, note: the color of the move is not accounted
 * for in mymovearray. To account for the color of the move, it is neccessary
 * to use myMoveIndexArray which stores the move order.

 */

            public void populateBoardArray()
            {
```

```
                    resetBoard();
                    int Order = 0;
                    /* This partition populates the board in the order the moves were made
                     *
                     */
                    for(int i = myMoveArray.length-1; i > -1; i = i-2)

                    {
        myBoardArray[myMoveArray[i-1]][myMoveArray[i]] = myMoveOrderArray[Order];
        Order++;
                    }
            }

// needs checking

            public void populateBoardArray(int[] a)
            {
                    resetBoard();
                    int Order = 0;
                    /* This partition populates the board in the order the moves were made
                     *
                     */
                    for(int i = a.length-1; i > -1; i = i-2)
                    {
        myBoardArray[a[i-1]][a[i]] = myMoveOrderArray[Order];
        Order++;
                    }
            }

            public void determineMoveOrderArray()
            {
               double blackFractionTotal = 0;
               double whiteFractionTotal = 0;
               int blackMoveTotal = 0;
               int whiteMoveTotal = 0;
               int index = 0;
               int currentBlackMoves = 0;
               int currentWhiteMoves = 0;
               int[] myMoveOrderA = new int[100];
               myMoveOrderArray = myMoveOrderA;
               while(index < 100)
               {
                  /*
                   * increments the total move fraction
                   */
                  blackFractionTotal = blackFractionTotal + myBlackMoves;
                  whiteFractionTotal = whiteFractionTotal + myWhiteMoves;
        /*
         * determines the number of moves each player recieves each turn
         */
                  currentBlackMoves = (int) (blackFractionTotal - (double)blackMoveTotal);
                  currentWhiteMoves = (int) (whiteFractionTotal - (double)whiteMoveTotal);
                  /*
                   * assigns the moves to myMoveOrderArray;
                   */
                  for(int i = 0; i < currentWhiteMoves; i++)
                  {
                     if(index < 100)
                     {
                        myMoveOrderArray[index] = WHITE;
                     }
                     index++;
                  }

                  for(int i = 0; i < currentBlackMoves; i++)
                  {
                     if(index < 100)
                     {
                        myMoveOrderArray[index] = BLACK;
                     }
```

```
            index++;
          }
          /*
           * increments blackMoveTotal and whiteMoveTotal
           */
          blackMoveTotal = blackMoveTotal + currentBlackMoves;
          whiteMoveTotal = whiteMoveTotal + currentWhiteMoves;
        }
    }

    public int[] determineAndGetMoveOrderArray()
    {
        double blackFractionTotal = 0;

        double whiteFractionTotal = 0;
        int blackMoveTotal = 0;
        int whiteMoveTotal = 0;
        int index = 0;
        int currentBlackMoves = 0;
        int currentWhiteMoves = 0;
        int[] myMoveOrderA = new int[100];
        int[] a = new int[100];
        myMoveOrderArray = myMoveOrderA;
        while(index < 100)
        {
          /*
           * increments the total move fraction
           */
          blackFractionTotal = blackFractionTotal + myBlackMoves;
          whiteFractionTotal = whiteFractionTotal + myWhiteMoves;
/*
 * determines the number of moves each player recieves each turn
 */
          currentBlackMoves = (int) (blackFractionTotal - (double)blackMoveTotal);
          currentWhiteMoves = (int) (whiteFractionTotal - (double)whiteMoveTotal);
          /*
           * assigns the moves to myMoveOrderArray;
           */
          for(int i = 0; i < currentWhiteMoves; i++)
          {
            if(index < 100)
            {
              myMoveOrderArray[index] = WHITE;
              a[index] = WHITE;
            }
            index++;
          }

          for(int i = 0; i < currentBlackMoves; i++)
          {
            if(index < 100)
            {
              myMoveOrderArray[index] = BLACK;
              a[index] = BLACK;
            }
            index++;
          }
          /*
           * increments blackMoveTotal and whiteMoveTotal
           */
          blackMoveTotal = blackMoveTotal + currentBlackMoves;
          whiteMoveTotal = whiteMoveTotal + currentWhiteMoves;
        }
        return a;
    }

    public void determineMoveOrderArrayWithRatio()
    {
                double r = 0;
                r = myMoveRatio;
```

```
		int[] moveOrderArray;
		moveOrderArray = new int[100];
		int data = 0;
		moveOrderArray[0] = 1;
		for(int i = 1; i < 101; i++)
		{
			data = (int) Math.ceil(r*((double)i)) - (int) Math.ceil(r*((double)(i-1)));
			moveOrderArray[i-1] = (2-data);
		}
		myMoveOrderArray = moveOrderArray;
	}

	public boolean futureMoveArrayContains(int x, int y)
	{
		boolean b = false;
		int index = 0;
		while(myFutureMovesArray[index] != 100)
		{
			if(myFutureMovesArray[index] == x & myFutureMovesArray[index+1] == y)
			{
				b = true;
			}
			index = index+2;
		}
		return b;
	}


/* This algorithm generates a future move array. The move array just consists
 * of moves around the squares for either white or black depending on whose
 * move it is. myMoveOrderArray is used to determine whose turn it is. and
 * thus myFutureMoveArray consists of either white or black moves. What
 * is created is an array of moves xyxyxyxyxy, stored in no signifigant order
 * Just a set of the possible moves. This is for the variation where white can only
 * move to adjacent squares. (side to side, up & down), Black can do the same but
 * also move to adjacent corners.
 *
 *   NOTE this algorthm duplicates some moves. When children are created, it is necessary
 *   to check for duplicates.
 */

	public boolean isWhiteContainedV1(GamingTree tree)
	{
		boolean contained = true;
	   generateMoveArray(tree);
	   determineMoveOrderArray();
	   populateBoardArray();
		int moveOrderIndex = 0;
		for(int i = myMoveArray.length-1; i > -1; i = i-2)
		{
		  int x = myMoveArray[i-1];
			int y = myMoveArray[i];
			if (myMoveOrderArray[moveOrderIndex] == WHITE && contained)

			{
	if(myBoardArray[x][y+1] == 0)
	{
	    contained = false;
	}
	if(myBoardArray[x][y-1] == 0)
	{
	    contained = false;
	}
	if(myBoardArray[x-1][y] == 0)
	{
	    contained = false;
	}
				if(myBoardArray[x+1][y] == 0)
				{
			contained = false;
```

```
                                        }
                                }
                                moveOrderIndex++;
                }
        return contained;
        }

////////////////////////////////////////////////////////////////////
        ////////////////  Geometric Blob stuff//////////////////////////
        ////////////////////////////////////////////////////////////////

        public int getSizeNumber()
        {
                return mySizeNumber;
        }

        public void setSizeNumber(int s)
        {
                mySizeNumber = s;
                mySquareNodes = new SquareNode[s];
                for(int i = 0; i < mySquareNodes.length; i++)
                {
                        mySquareNodes[i] = new SquareNode(i+1,1);
                }
        }

        public SquareNode[] getSquareNodes()
        {
                return mySquareNodes;
        }

////////////////////////////////////////////////////////////////////
        ////////////////  Geometric Blob stuff//////////////////////////
        ////////////////////////////////////////////////////////////////
        public int getXDimmension(int[] m1)
        {
        int maxX, minX;
        maxX = m1[0];
        minX = m1[0];

        for(int i = 0; i < m1.length; i=i+2)
        {
                if(maxX < m1[i])
                {
                        maxX = m1[i];
                }
                if(minX > m1[i])
                {
                        minX = m1[i];
                }
        }
        return(maxX-minX+1);
        }

        public int getYDimmension(int[] m1)
        {
        int maxY, minY;
        maxY = m1[1];
        minY = m1[1];

        for(int i = 0; i < m1.length; i=i+2)
        {
                if(maxY < m1[i+1])
                {
                        maxY = m1[i+1];
                }
                if(minY > m1[i+1])
                {
                        minY = m1[i+1];
                }
```

```
		}
		return(maxY-minY+1);
		}

		public boolean dimensionExists(int[] a)
		{
			boolean dimension = false;
			int x, y;
			SquareNode SN;
			x = getXDimmension(a);
			y = getYDimmension(a);
			if(y > x)
			{
				int temp;
				temp = x;
				x = y;
				y = temp;
			}
			SN = mySquareNodes[a.length/2-1];
			while(SN.getChild() != null)
			{
				if(SN.getX() == x && SN.getY() == y)
				{
					dimension = true;
				}
				SN = SN.getChild();
			}
			if(SN.getX() == x && SN.getY() == y)
			{
				dimension = true;
			}
			return dimension;
		}

		public boolean symmetricPositionExists(int n, int[] a)
		{
			boolean symmetric = false;
			int x, y;
			x = getXDimmension(a);
			y = getYDimmension(a);
			if(y > x)
			{
				int temp;
				temp = x;
				x = y;
				y = temp;
			}


			return symmetric;
		}

////////////////////////////////////////////////////////////////////////
		//////////////////  Geometric Blob stuff//////////////////////////
		////////////////////////////////////////////////////////////////////

	public boolean is12Symmetric(int[] m1, int[] m2)
	{
		boolean symmetric = true;
		generateM1Moves(m1);
		generateM2Moves(m2);
		int x1, x2, y1, y2;
		x1 = my1Moves.length;
		x2 = my2Moves.length;
		y1 = my1Moves[0].length;
		y2 = my2Moves[0].length;
		String myString = new String("");
		String tempString = new String("");
		// Checks to see if the dimensions match.
```

```
            if(x1 == x2)
            {
                        if(y1 != y2)
                        {
                           symmetric = false;
                        }
            }
            else
            {
                        if(x1 == y2)
                        {
                                    if(x2 != y1)
                                    {
                                       symmetric = false;
                                    }
                        }
                        else
                        {
                                    symmetric = false;
                        }
            }

            if(symmetric)
            {
                        symmetric = false;
                        myString = generateSymmetries(my1Moves);
                        for(int i = 0; i < my2Moves.length; i++)
                        {
                                    for(int j = 0; j < my2Moves[0].length; j++)
                                    tempString = tempString + my2Moves[i][j];
                        }
                        for(int i = 0; i < 8; i++)
                        {
                                    if(tempString.equalsIgnoreCase(myString.substring(3+
                                      i*my1Moves.length*my1Moves[0].length, 3+
                                      (i+1)*my1Moves.length*my1Moves[0].length)))
                                    {
                                                symmetric = true;
                                    }
                        }
            }
            return symmetric;
   }

public int getSymmetryStringLength(int[][] a)
{
            int    counter = 0;
            counter = 8*(a.length)*(a[0].length) + 3;
            return counter;
}

public void resetAdjacentPossibleMoves()
{
            myAdjacentPossibleMoves = new int[16];
            for(int i = 0; i < 16; i++)
            {
                        myAdjacentPossibleMoves[i] = 0;
            }
}

public int getNumberOfMovesInArray(int[][] a)
{
            int    counter = 0;
            for(int i = 0; i < a.length; i++)
            {
                        for(int j = 0; j < a[0].length; j++)
                        {
                                    if(a[i][j] == 1 || a[i][j] == 2)
                                    {
                                                counter++;
```

```
                                        }
                                }
                        }
                        return counter;
}

    /*
     * Pass in a clipped 2D-Array, returns a string with the first 3 characters
     * representing X length, Y length, # of moves.
     */

    public String generateSymmetries(int[][] a)
    {
                String tempString;
                int    counter = getNumberOfMovesInArray(a);

                //creates String (x,y,Move #)

                tempString = new String("");
                tempString = tempString+a.length+a[0].length+counter;

                //+x,+y

                for(int i = 0; i < a.length; i++)
                {
                        for(int j = 0; j < a[0].length; j++)
                        {
                                tempString = tempString+a[i][j];
                        }
                }

                //+x,-y

                for(int i = 0; i < a.length; i++)
                {
                        for(int j = a[0].length-1; j > -1; j--)
                        {
                                tempString = tempString+a[i][j];
                        }
                }

                //-x,+y

                for(int i = a.length-1; i > -1; i--)
                {
                        for(int j = 0; j < a[0].length; j++)
                        {
                                tempString = tempString+a[i][j];
                        }
                }

                //-x,-y

                for(int i = a.length-1; i > -1; i--)
                {
                        for(int j = a[0].length-1; j > -1; j--)
                        {
                                tempString = tempString+a[i][j];
                        }
                }

                //+y,+x

                for(int j = 0; j < a[0].length; j++)
                {
                        for(int i = 0; i < a.length; i++)
                        {
                                tempString = tempString+a[i][j];
                        }
                }
```

```
		//+y,-x

		for(int j = 0; j < a[0].length; j++)
		{
			for(int i = a.length-1; i > -1; i--)
			{
				tempString = tempString+a[i][j];
			}
		}

		//-y,+x

		for(int j = a[0].length-1; j > -1; j--)
		{
			for(int i = 0; i < a.length; i++)
			{
				tempString = tempString+a[i][j];
			}
		}

		//-y,-x

		for(int j = a[0].length-1; j > -1; j--)
		{
			for(int i = a.length-1; i > -1; i--)
			{
				tempString = tempString+a[i][j];
			}
		}

		return tempString;
	}

	/* pass in an array of moves, x,y,x,y,... generates
	 * a minimal dimensional array
	 */

	public void generateM1Moves(int[] m1)
	{
		determineMoveOrderArray();
		int maxX, minX, maxY, minY;
		maxX = m1[0];
		minX = m1[0];
		maxY = m1[1];
		minY = m1[1];
/*
 * This determines the maximum and minimum positions of the
 * move Array so we can trim it.
 */
		for(int i = 0; i < m1.length; i=i+2)
		{
			if(maxX < m1[i])
			{
				maxX = m1[i];
			}
			if(minX > m1[i])
			{
				minX = m1[i];
			}
			if(maxY < m1[i+1])
			{
				maxY = m1[i+1];
			}
			if(minY > m1[i+1])
			{
				minY = m1[i+1];
			}
		}
		//System.out.println("max X "+maxX+" min X "+minX);
```

```
		//System.out.println("max Y "+maxY+" min Y "+minY);
		my1Moves = new int[maxX-minX+1][maxY-minY+1];

		/*
		 * Initializes terms of my1Moves
		 */

		for(int i = 0; i < maxX-minX+1; i++)

		{
			for(int j = 0; j < maxY-minY+1; j++)
			{
				my1Moves[i][j] = 0;
			}
		}

		/*
		 * Assigns appropriate terms of my the moveArray to 1 or 2
		 */

		for(int i = 0; i < m1.length; i = i+2)
		{
			my1Moves[m1[i]-minX][m1[i+1]-minY] = myMoveOrderArray[i/2];
		}
	}

	public void generateM2Moves(int[] m2)
	{
		determineMoveOrderArray();
		int maxX, minX, maxY, minY;
		maxX = m2[0];
		minX = m2[0];
		maxY = m2[1];
		minY = m2[1];
/*
 * This determines the maximum and minimum positions of the
 * move Array so we can trim it.
 */
		for(int i = 0; i < m2.length; i=i+2)
		{
			if(maxX < m2[i])
			{
				maxX = m2[i];
			}
			if(minX > m2[i])
			{
				minX = m2[i];
			}
			if(maxY < m2[i+1])
			{
				maxY = m2[i+1];
			}
			if(minY > m2[i+1])
			{
				minY = m2[i+1];
			}
		}
		my2Moves = new int[maxX-minX+1][maxY-minY+1];

		/*
		 * Initializes all terms of my2Moves to 0
		 */

		for(int i = 0; i < maxX-minX+1; i++)
		{
			for(int j = 0; j < maxY-minY+1; j++)
			{
				my2Moves[i][j] = 0;
			}
		}
```

```
	/*
	 * Assigns appropriate terms of my the moveArray to 1 or 2
	 */

	for(int i = 0; i < m2.length; i = i+2)
	{
		my2Moves[m2[i]-minX][m2[i+1]-minY] = myMoveOrderArray[i/2];
	}
}

	public void geometricBlobFutureMoveGenerator(GamingTree tree)
	{
		myWhiteMoves = 200;
		myBlackMoves = 0;
		generateMoveArray(tree);
		determineMoveOrderArray();
		populateBoardArray();
		resetFutureMoves();

		  /* note Level ordering is 0,1,2,3,4... */

		int level = tree.getLevel();

		  /*this section generate the future moves if it is whites turn*/

		if(myMoveOrderArray[level+1] == WHITE)
		{
			int total = 0;
			int moveOrderIndex = 0;
			for(int i = myMoveArray.length-1; i > -1; i = i-2)
			{
			    int x = myMoveArray[i-1];
				int y = myMoveArray[i];
				if (myMoveOrderArray[moveOrderIndex] == WHITE)
				{
		if(myBoardArray[x][y+1] == 0 & !futureMoveArrayContains(x, y+1))
		{
		  myFutureMovesArray[total] = x;
		  myFutureMovesArray[total+1] = (y+1);
		  total = total+2;
		}
		if(myBoardArray[x][y-1] == 0 & !futureMoveArrayContains(x, y-1))
		{
		  myFutureMovesArray[total] = x;
				myFutureMovesArray[total+1] = (y-1);
				total = total+2;
		}
		if(myBoardArray[x-1][y] == 0& !futureMoveArrayContains(x-1, y))
		{

				myFutureMovesArray[total] = (x-1);
				myFutureMovesArray[total+1] = y;
				total = total+2;
		}
			if(myBoardArray[x+1][y] == 0& !futureMoveArrayContains(x+1, y))
				{
				myFutureMovesArray[total] = (x+1);
				myFutureMovesArray[total+1] = y;
				total = total+2;
				}
			}
			moveOrderIndex++;
		    }
		  }
		int i = 0;
		while(myFutureMovesArray[i] != 100)
		{
			i++;
		}
```

```
                int[] b;
                b = new int[i];
                for(int a = 0; a < i; a++)
                {
                        b[a] = myFutureMovesArray[a];
                }
                myFutureMovesArray = b;
}

public void generateFutureMoveArrayVersion1(GamingTree tree)
{
  generateMoveArray(tree);
  determineMoveOrderArray();
  populateBoardArray();
  resetFutureMoves();

  /* note Level ordering is 0,1,2,3,4... */

  int level = tree.getLevel();

  /*this section generate the future moves if it is whites turn*/

  if(myMoveOrderArray[level+1] == WHITE)
  {
                int total = 0;
                int moveOrderIndex = 0;
                for(int i = myMoveArray.length-1; i > -1; i = i-2)
                {
                  int x = myMoveArray[i-1];
                        int y = myMoveArray[i];
                        if (myMoveOrderArray[moveOrderIndex] == WHITE)
                        {
        if(myBoardArray[x][y+1] == 0 & !futureMoveArrayContains(x, y+1))
        {
          myFutureMovesArray[total] = x;
          myFutureMovesArray[total+1] = (y+1);
          total = total+2;
        }
        if(myBoardArray[x][y-1] == 0 & !futureMoveArrayContains(x, y-1))
        {
          myFutureMovesArray[total] = x;
                                        myFutureMovesArray[total+1] = (y-1);
                                        total = total+2;
        }
        if(myBoardArray[x-1][y] == 0& !futureMoveArrayContains(x-1, y))
        {
                                        myFutureMovesArray[total] = (x-1);
                                        myFutureMovesArray[total+1] = y;
                                        total = total+2;
        }
                                  if(myBoardArray[x+1][y] == 0& !futureMoveArrayContains(x+1, y))
                                        {
                                          myFutureMovesArray[total] = (x+1);
                                          myFutureMovesArray[total+1] = y;
                                          total = total+2;
                                        }
                        }
                        moveOrderIndex++;
                }
  }

  /* Future Moves for Black Works*/

  else
  {
  int total = 0;
  int moveOrderIndex = 0;
  for(int i = myMoveArray.length-1; i > -1; i = i-2)
  {
                int x = myMoveArray[i-1];
```

```
                int y = myMoveArray[i];
                if (myMoveOrderArray[moveOrderIndex] == WHITE)
                {
                  if(myBoardArray[x][y+1] == 0 & !futureMoveArrayContains(x, y+1))
                        {
                          myFutureMovesArray[total] = x;
                                myFutureMovesArray[total+1] = (y+1);
                                total = total+2;
                        }
                        if(myBoardArray[x][y-1] == 0 & !futureMoveArrayContains(x, y-1))
                        {
                          myFutureMovesArray[total] = x;
                          myFutureMovesArray[total+1] = (y-1);
                                total = total+2;
                        }
                        if(myBoardArray[x-1][y] == 0 & !futureMoveArrayContains(x-1, y))
                        {
                          myFutureMovesArray[total] = (x-1);
                          myFutureMovesArray[total+1] = y;
                          total = total+2;
                        }
                        if(myBoardArray[x+1][y] == 0& !futureMoveArrayContains(x+1, y))
                        {
                          myFutureMovesArray[total] = (x+1);
                          myFutureMovesArray[total+1] = y;
                                total = total+2;
                        }
                        if(myBoardArray[x+1][y+1] == 0& !futureMoveArrayContains(x+1, y+1))
                        {
                          myFutureMovesArray[total] = (x+1);
                          myFutureMovesArray[total+1] = (y+1);
                          total = total+2;
                        }
                        if(myBoardArray[x+1][y-1] == 0& !futureMoveArrayContains(x+1, y-1))
                        {
                          myFutureMovesArray[total] = (x+1);
                          myFutureMovesArray[total+1] = (y-1);
                          total = total+2;
                        }
                        if(myBoardArray[x-1][y+1] == 0 & !futureMoveArrayContains(x-1, y+1))
                        {
                          myFutureMovesArray[total] = (x-1);
                          myFutureMovesArray[total+1] = (y+1);
                          total = total+2;
                        }
                        if(myBoardArray[x-1][y-1] == 0& !futureMoveArrayContains(x-1, y-1))
                        {
                          myFutureMovesArray[total] = (x-1);
                          myFutureMovesArray[total+1] = (y-1);
                          total = total+2;
                        }
                }
                moveOrderIndex++;
          }
        }
        int i = 0;
        while(myFutureMovesArray[i] != 100)
        {
                i++;
        }
        int[] b;
        b = new int[i];
        for(int a = 0; a < i; a++)
        {
                b[a] = myFutureMovesArray[a];
        }
        myFutureMovesArray = b;
}

//////////////////////////////////////
```

```
// GFMAWSV1
////////////////////////////////////

        public void generateFutureMoveArrayVersion1WS(GamingTree tree)
        {
          generateMoveArray(tree);
          determineMoveOrderArray();
          populateBoardArray();
          resetFutureMoves();

          /* note Level ordering is 0,1,2,3,4... */

          int level = tree.getLevel();

          /*this section generate the future moves if it is whites turn*/

          if(myMoveOrderArray[level+1] == WHITE)
          {
                    int total = 0;
                    int moveOrderIndex = 0;
                    for(int i = myMoveArray.length-1; i > -1; i = i-2)
                    {
                      int x = myMoveArray[i-1];
                              int y = myMoveArray[i];
                              if (myMoveOrderArray[moveOrderIndex] == WHITE)
                              {
                  if(myBoardArray[x][y+1] == 0 & !futureMoveArrayContains(x, y+1))
                  {
                    myFutureMovesArray[total] = x;
                    myFutureMovesArray[total+1] = (y+1);
                    total = total+2;
                  }
                  if(myBoardArray[x][y-1] == 0 & !futureMoveArrayContains(x, y-1))
                  {
                    myFutureMovesArray[total] = x;
                                              myFutureMovesArray[total+1] = (y-1);
                                              total = total+2;
                  }
                  if(myBoardArray[x-1][y] == 0& !futureMoveArrayContains(x-1, y))
                  {
                                              myFutureMovesArray[total] = (x-1);
                                              myFutureMovesArray[total+1] = y;
                                              total = total+2;
                  }
                                    if(myBoardArray[x+1][y] == 0& !futureMoveArrayContains(x+1, y))
                                              {
                                                myFutureMovesArray[total] = (x+1);
                                                myFutureMovesArray[total+1] = y;
                                                total = total+2;
                                              }
                              }
                              moveOrderIndex++;
                    }
          }

          /* Future Moves for Black Works*/

          else
          {
          int total = 0;
          int moveOrderIndex = 0;
          for(int i = myMoveArray.length-1; i > -1; i = i-2)
          {
                    int x = myMoveArray[i-1];
                    int y = myMoveArray[i];
                    if (myMoveOrderArray[moveOrderIndex] == WHITE)
                    {
                      if(myBoardArray[x][y+1] == 0 & !futureMoveArrayContains(x, y+1))
                              {
                                myFutureMovesArray[total] = x;
```

```
                                        myFutureMovesArray[total+1] = (y+1);
                                        total = total+2;
                                }
                                if(myBoardArray[x][y-1] == 0 & !futureMoveArrayContains(x, y-1))
                                {
                                  myFutureMovesArray[total] = x;
                                  myFutureMovesArray[total+1] = (y-1);
                                        total = total+2;
                                }
                                if(myBoardArray[x-1][y] == 0 & !futureMoveArrayContains(x-1, y))
                                {
                                  myFutureMovesArray[total] = (x-1);
                                  myFutureMovesArray[total+1] = y;
                                  total = total+2;
                                }
                                if(myBoardArray[x+1][y] == 0& !futureMoveArrayContains(x+1, y))
                                {
                                  myFutureMovesArray[total] = (x+1);
                                  myFutureMovesArray[total+1] = y;
                                        total = total+2;
                                }
                                if(myBoardArray[x+1][y+1] == 0& !futureMoveArrayContains(x+1, y+1))
                                {
                                  myFutureMovesArray[total] = (x+1);
                                  myFutureMovesArray[total+1] = (y+1);
                                  total = total+2;
                                }
                                if(myBoardArray[x+1][y-1] == 0& !futureMoveArrayContains(x+1, y-1))
                                {
                                  myFutureMovesArray[total] = (x+1);
                                  myFutureMovesArray[total+1] = (y-1);
                                  total = total+2;
                                }
                                if(myBoardArray[x-1][y+1] == 0 & !futureMoveArrayContains(x-1, y+1))
                                {
                                  myFutureMovesArray[total] = (x-1);
                                  myFutureMovesArray[total+1] = (y+1);
                                  total = total+2;
                                }
                                if(myBoardArray[x-1][y-1] == 0& !futureMoveArrayContains(x-1, y-1))
                                {
                                  myFutureMovesArray[total] = (x-1);
                                  myFutureMovesArray[total+1] = (y-1);
                                  total = total+2;
                                }
                        }
                        moveOrderIndex++;
                  }
                }
                int i = 0;
                while(myFutureMovesArray[i] != 100)
                {
                        i++;
                }
                int[] b;
                b = new int[i];
                for(int a = 0; a < i; a++)
                {
                        b[a] = myFutureMovesArray[a];
                }
                myFutureMovesArray = b;
                myFutureMovesArray = eliminateSymmetricPositions();
}


// eliminates symmetric positions from future move array
// presumes future move array has already been generated*

public int[] eliminateSymmetricPositions()
{
```

```
		int[] MO = myMoveArray;
		int[] FM = myFutureMovesArray;
		int[] NFM = new int[FM.length+3];
		NFM[0] = FM[0];
		NFM[1] = FM[1];
		for(int i = 2; i < NFM.length; i++)
		{
			NFM[i] = 100;
		}
		int[] tempArray1 = new int[MO.length+2];
		int[] tempArray2 = new int[MO.length+2];
		for(int i = 0; i < MO.length; i ++)
		{
			tempArray1[i] = MO[i];
			tempArray2[i] = MO[i];
		}
		tempArray1[MO.length]   = 100;
		tempArray2[MO.length]   = 100;
		tempArray1[MO.length+1] = 100;
		tempArray2[MO.length+1] = 100;
		for(int i = 2; i < FM.length; i=i+2)
		{
			int j = 0;
			boolean notFound = true;
			tempArray1[MO.length]   = FM[i];
			tempArray1[MO.length+1] = FM[i+1];
			while(NFM[j] != 100 && j < FM.length && notFound)
			{
				tempArray2[MO.length]   = NFM[j];
				tempArray2[MO.length+1] = NFM[j+1];
				boolean b = is12Symmetric(tempArray1, tempArray2);
				if(b)
				{
					notFound = false;
				}
				j = j+2;
			}
			if(notFound && j < FM.length)
			{
				int k = 2;
				while(NFM[k] != 100)
				{
					k = k+2;
				}
				NFM[k]  = FM[i];
				NFM[k+1]= FM[i+1];
			}
		}
		int k = 2;
		while(NFM[k] != 100)
		{
			k = k+2;
		}
		int a[] = new int[k];
   for(int i = 0; i < k; i=i+2)
   {
		a[i] = NFM[i];
		a[i+1] = NFM[i+1];
   }
   return a;
}

/* This algorithm generates a future move array. The move array just consists
 * of positions around the squares for either white or black depending on whose
 * move it is. myMoveOrderArray is used to determine whose turn it is. and
 * thus myFutureMoveArray consists of either white or black moves not both. What
 * is created is an array of moves xyxyxyxyxy, stored in with some order
 * The order is the possible moves around move#1, the possible moves around move #2...
 * Just a set of the possible moves. This is for the variation where white can only
 * move to adjacent squares and corners, Black has the same limitations
```

```
 *
 * NOTE this algorthm duplicates some moves. When children are created, it is necessary
 * to check for duplicates.
 */

public void generateFutureMoveArrayVersion2(GamingTree tree)
{
	generateMoveArray(tree);
	populateBoardArray();
	resetFutureMoves();

	/* note Level ordering is 0,1,2,3,4... */

	int level = tree.getLevel();

	/*this section generate the future moves if it is whites turn*/

	if(myMoveOrderArray[level+1] == WHITE)
	{
		int total = 0;
		int moveOrderIndex = 0;
		for(int i = myMoveArray.length-1; i > -1; i = i-2)
		{
		  int x = myMoveArray[i-1];
		  int y = myMoveArray[i];
		  if (myMoveOrderArray[moveOrderIndex] == WHITE)
		  {
			if(myBoardArray[x][y+1] == 0)
			{
				myFutureMovesArray[total] = x;
				myFutureMovesArray[total+1] = (y+1);
				total = total+2;
			}
			if(myBoardArray[x][y-1] == 0)
			{
				myFutureMovesArray[total] = x;
				myFutureMovesArray[total+1] = (y-1);
				total = total+2;
			}
			if(myBoardArray[x-1][y] == 0)
			{
				myFutureMovesArray[total] = (x-1);
				myFutureMovesArray[total+1] = y;
				total = total+2;
			}
			if(myBoardArray[x+1][y] == 0)
			{
				myFutureMovesArray[total] = (x+1);
				myFutureMovesArray[total+1] = y;
				total = total+2;
			}
		    if(myBoardArray[x+1][y+1] == 0)
			{
			  myFutureMovesArray[total] = (x+1);
			  myFutureMovesArray[total+1] = (y+1);
			  total = total+2;
			}
			if(myBoardArray[x+1][y-1] == 0)
			{
			  myFutureMovesArray[total] = (x+1);
			  myFutureMovesArray[total+1] = (y-1);
			  total = total+2;
			}
			if(myBoardArray[x-1][y+1] == 0)
			{
			  myFutureMovesArray[total] = (x-1);
			  myFutureMovesArray[total+1] = (y+1);
			  total = total+2;
			}
			if(myBoardArray[x-1][y-1] == 0)
```

```
                        {
                          myFutureMovesArray[total] = (x-1);
                          myFutureMovesArray[total+1] = (y-1);
                          total = total+2;
                        }
                }
                moveOrderIndex++;
        }
}

/* Future Moves for Black */

else
{
int total = 0;
int moveOrderIndex = 0;
for(int i = myMoveArray.length-1; i > -1; i = i-2)
{
        int x = myMoveArray[i-1];
        int y = myMoveArray[i];
        if (myMoveOrderArray[moveOrderIndex] == WHITE)
        {
          if(myBoardArray[x][y+1] == 0)
          {
                myFutureMovesArray[total] = x;
                myFutureMovesArray[total+1] = (y+1);
                total = total+2;
          }
          if(myBoardArray[x][y-1] == 0)
          {
                myFutureMovesArray[total] = x;
                myFutureMovesArray[total+1] = (y-1);
                total = total+2;
          }
          if(myBoardArray[x-1][y] == 0)
          {
                myFutureMovesArray[total] = (x-1);
                myFutureMovesArray[total+1] = y;
                total = total+2;
          }
          if(myBoardArray[x+1][y] == 0)
          {
                myFutureMovesArray[total] = (x+1);
                myFutureMovesArray[total+1] = y;
                total = total+2;
          }
          if(myBoardArray[x+1][y+1] == 0)
          {
                myFutureMovesArray[total] = (x+1);
                myFutureMovesArray[total+1] = (y+1);
                total = total+2;
          }
          if(myBoardArray[x+1][y-1] == 0)
          {
                myFutureMovesArray[total] = (x+1);
                myFutureMovesArray[total+1] = (y-1);
                total = total+2;
          }
          if(myBoardArray[x-1][y+1] == 0)
          {
                myFutureMovesArray[total] = (x-1);
                myFutureMovesArray[total+1] = (y+1);
                total = total+2;
          }
          if(myBoardArray[x-1][y-1] == 0)
          {
                myFutureMovesArray[total] = (x-1);
                myFutureMovesArray[total+1] = (y-1);
                total = total+2;
          }
```

```
                        }
                        moveOrderIndex++;
                }
        }
        int i = 0;
        while(myFutureMovesArray[i] != 100)
        {
                i++;
        }
        int[] b;
        b = new int[i];
        for(int a = 0; a < i; a++)
        {
                b[a] = myFutureMovesArray[a];
        }
        myFutureMovesArray = b;
}

////////////////////////////////////////////////////////////////////
///       Game AI Algorithms Used to Analyse situation/pick moves     //
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// These Algorithms get the Max and Min positions from myMoveArray      //
////////////////////////////////////////////////////////////////////


        public void generateFutureMoveArrayVersion1WSWP(GamingTree tree)
        {
          generateMoveArray(tree);
          determineMoveOrderArray();
          populateBoardArray();
          resetFutureMoves();

          /* note Level ordering is 0,1,2,3,4... */

          int level = tree.getLevel();

          /*this section generate the future moves if it is whites turn*/

          if(myMoveOrderArray[level+1] == WHITE)
          {
                int total = 0;
                int moveOrderIndex = 0;
                for(int i = myMoveArray.length-1; i > -1; i = i-2)
                {
                  int x = myMoveArray[i-1];
                        int y = myMoveArray[i];
                        if (myMoveOrderArray[moveOrderIndex] == WHITE)
                        {
              if(myBoardArray[x][y+1] == 0 & !futureMoveArrayContains(x, y+1))
              {
                myFutureMovesArray[total] = x;
                myFutureMovesArray[total+1] = (y+1);
                total = total+2;
              }
              if(myBoardArray[x][y-1] == 0 & !futureMoveArrayContains(x, y-1))
              {
                myFutureMovesArray[total] = x;
                                        myFutureMovesArray[total+1] = (y-1);
                                        total = total+2;
              }
              if(myBoardArray[x-1][y] == 0& !futureMoveArrayContains(x-1, y))
              {
                                        myFutureMovesArray[total] = (x-1);
                                        myFutureMovesArray[total+1] = y;
                                        total = total+2;
              }
                                if(myBoardArray[x+1][y] == 0& !futureMoveArrayContains(x+1, y))
                                        {
```

```
                                        myFutureMovesArray[total] = (x+1);
                                        myFutureMovesArray[total+1] = y;
                                        total = total+2;
                                    }
                        }
                        moveOrderIndex++;
            }
}

/* Future Moves for Black Works*/

else
{
int total = 0;
int moveOrderIndex = 0;
for(int i = myMoveArray.length-1; i > -1; i = i-2)

{
            int x = myMoveArray[i-1];
            int y = myMoveArray[i];
            if (myMoveOrderArray[moveOrderIndex] == WHITE)
            {
              if(myBoardArray[x][y+1] == 0 & !futureMoveArrayContains(x, y+1))
                        {
                          myFutureMovesArray[total] = x;
                                    myFutureMovesArray[total+1] = (y+1);
                                    total = total+2;
                        }
                        if(myBoardArray[x][y-1] == 0 & !futureMoveArrayContains(x, y-1))
                        {
                          myFutureMovesArray[total] = x;
                          myFutureMovesArray[total+1] = (y-1);
                                    total = total+2;
                        }
                        if(myBoardArray[x-1][y] == 0 & !futureMoveArrayContains(x-1, y))
                        {
                          myFutureMovesArray[total] = (x-1);
                          myFutureMovesArray[total+1] = y;
                          total = total+2;
                        }
                        if(myBoardArray[x+1][y] == 0& !futureMoveArrayContains(x+1, y))

                        {
                          myFutureMovesArray[total] = (x+1);
                          myFutureMovesArray[total+1] = y;
                                    total = total+2;
                        }
                        if(myBoardArray[x+1][y+1] == 0& !futureMoveArrayContains(x+1, y+1))
                        {
                          myFutureMovesArray[total] = (x+1);
                          myFutureMovesArray[total+1] = (y+1);
                          total = total+2;
                        }
                        if(myBoardArray[x+1][y-1] == 0& !futureMoveArrayContains(x+1, y-1))
                        {
                          myFutureMovesArray[total] = (x+1);
                          myFutureMovesArray[total+1] = (y-1);
                          total = total+2;
                        }
                        if(myBoardArray[x-1][y+1] == 0 & !futureMoveArrayContains(x-1, y+1))
                        {
                          myFutureMovesArray[total] = (x-1);
                          myFutureMovesArray[total+1] = (y+1);
                          total = total+2;
                        }
                        if(myBoardArray[x-1][y-1] == 0& !futureMoveArrayContains(x-1, y-1))
                        {
                          myFutureMovesArray[total] = (x-1);
                          myFutureMovesArray[total+1] = (y-1);
                          total = total+2;
```

```
                                }
                        }
                        moveOrderIndex++;
                }
        }
        if(myMoveOrderArray[level+1] == WHITE)
        {
                int a[];
                a = new int[2];
                a[0] = myFutureMovesArray[0];
                a[1] = myFutureMovesArray[1];
                resetFutureMoves();
                myFutureMovesArray[0] = a[0];
                myFutureMovesArray[1] = a[1];
        }
        int i = 0;
        while(myFutureMovesArray[i] != 100)
        {
                i++;
        }
        int[] b;
        b = new int[i];
        for(int a = 0; a < i; a++)
        {
                b[a] = myFutureMovesArray[a];
        }
        myFutureMovesArray = b;
        myFutureMovesArray = eliminateSymmetricPositions();
}


public void generateFutureMoveArrayVersion1WSBP(GamingTree tree)
        {
          generateMoveArray(tree);
          populateBoardArray();
          resetFutureMoves();

          /* note Level ordering is 0,1,2,3,4... */

          int level = tree.getLevel();

          /*this section generate the future moves if it is whites turn*/

          if(myMoveOrderArray[level+1] == WHITE)
          {
                int total = 0;
                int moveOrderIndex = 0;
                for(int i = myMoveArray.length-1; i > -1; i = i-2)
                {
                  int x = myMoveArray[i-1];
                        int y = myMoveArray[i];
                        if (myMoveOrderArray[moveOrderIndex] == WHITE)
                        {
              if(myBoardArray[x][y+1] == 0 & !futureMoveArrayContains(x, y+1))
              {
                myFutureMovesArray[total] = x;
                myFutureMovesArray[total+1] = (y+1);
                total = total+2;
              }
              if(myBoardArray[x][y-1] == 0 & !futureMoveArrayContains(x, y-1))
              {
                myFutureMovesArray[total] = x;
                                        myFutureMovesArray[total+1] = (y-1);
                                        total = total+2;
              }
              if(myBoardArray[x-1][y] == 0& !futureMoveArrayContains(x-1, y))
              {
                                        myFutureMovesArray[total] = (x-1);
                                        myFutureMovesArray[total+1] = y;
```

```
                                        total = total+2;
                }
                                if(myBoardArray[x+1][y] == 0& !futureMoveArrayContains(x+1, y))
                                        {
                                          myFutureMovesArray[total] = (x+1);
                                          myFutureMovesArray[total+1] = y;
                                          total = total+2;
                                        }
                                }
                                moveOrderIndex++;
                        }
                }

//////////////////////////////////////////////////////
///////   Please Work you beautiful code you         //
//////////////////////////////////////////////////////

                  else
                  {
                          int[] c = determineBlackMove(tree);
                          myFutureMovesArray[0] = c[0];
                          myFutureMovesArray[1] = c[1];
                  }
                int i = 0;
                while(myFutureMovesArray[i] != 100)
                {
                        i++;
                }
                int[] b;
                b = new int[i];
                for(int a = 0; a < i; a++)
                {
                        b[a] = myFutureMovesArray[a];
                }
                myFutureMovesArray = b;
                myFutureMovesArray = eliminateSymmetricPositions();
}


  public int getNumberOfMovesInMyMoveArray()
  {
                int i = 0;
                i = (myMoveArray.length/2);
                return i;
  }

  public int getNumberOfMovesInMyAdjacentPossibleMoves()
  {
                int i = 0;
                i = (myAdjacentPossibleMoves.length/2);
                return i;
  }

  public int getNumberOfMovesInMyFutureMoveArray()
  {
                int i = 0;
                i = (myFutureMovesArray.length/2);
                return i;
  }

  public int getMaxXWhitePosition(GamingTree tree)
  {
                generateMoveArray(tree);
                int x = 0;
                int moves = getNumberOfMovesInMyMoveArray();
    for(int i=0; i < moves; i++)
    {

                if(myMoveOrderArray[i] == WHITE)
                {
```

```
                        if(myMoveArray[moves*2-2*i-2] > x)
                        {
                                x = myMoveArray[moves*2-2*i-2];
                        }
                }
  }
                return x;
}

public int getMinXWhitePosition(GamingTree tree)
{
                generateMoveArray(tree);
                int x = 100;
                        int moves = getNumberOfMovesInMyMoveArray();
                        for(int i=0; i < moves; i++)
                  {

                        if(myMoveOrderArray[i] == WHITE)
                        {
                                if(myMoveArray[2*moves-2*i-2] < x)
                                {
                                        x = myMoveArray[2*moves-2*i-2];
                                }
                        }
                  }
                        return x;
}

public int getMaxXBlackPosition(GamingTree tree)
{
                generateMoveArray(tree);
                int x = 0;
                        int moves = getNumberOfMovesInMyMoveArray();
                  for(int i=0; i < moves; i++)
                  {

                        if(myMoveOrderArray[i] == BLACK)
                        {
                                if(myMoveArray[2*moves-2*i-2] > x)
                                {
                                        x = myMoveArray[2*moves-2*i-2];
                                }
                        }
                  }
                        return x;
}

public int getMinXBlackPosition(GamingTree tree)
{
                generateMoveArray(tree);
                int x = 100;
                        int moves = getNumberOfMovesInMyMoveArray();
                  for(int i=0; i < moves; i++)
                  {

                        if(myMoveOrderArray[i] == BLACK)
                        {
                                if(myMoveArray[2*moves-2*i-2] < x)
                                {
                                        x = myMoveArray[2*moves-2*i-2];
                                }
                        }
                  }
                        return x;
}

public int getMaxYWhitePosition(GamingTree tree)
{
                generateMoveArray(tree);
                int y = 0;
```

```
                int moves = getNumberOfMovesInMyMoveArray();
            for(int i=0; i < moves; i++)
            {

                if(myMoveOrderArray[i] == WHITE)
                {
                    if(myMoveArray[2*moves-2*i-1] > y)
                    {
                        y = myMoveArray[2*moves-2*i-1];
                    }
                }
            }
                return y;
}

public int getMinYWhitePosition(GamingTree tree)
{
        generateMoveArray(tree);
        int y = 100;
                int moves = getNumberOfMovesInMyMoveArray();
            for(int i=0; i < moves; i++)
            {

                if(myMoveOrderArray[i] == WHITE)
                {
                    if(myMoveArray[2*moves-2*i-1] < y)
                    {
                        y = myMoveArray[2*moves-2*i-1];
                    }
                }
            }
                return y;

}

public int getMaxYBlackPosition(GamingTree tree)
{
        generateMoveArray(tree);
        int y = 0;
                int moves = getNumberOfMovesInMyMoveArray();
            for(int i=0; i < moves; i++)
            {

                if(myMoveOrderArray[i] == BLACK)
                {
                    if(myMoveArray[2*moves-2*i-1] > y)
                    {
                        y = myMoveArray[2*moves-2*i-1];
                    }
                }
            }
                return y;
}


public int getMinYBlackPosition(GamingTree tree)
{
        generateMoveArray(tree);
        int y = 100;
                int moves = getNumberOfMovesInMyMoveArray();
            for(int i=0; i < moves; i++)
            {

                if(myMoveOrderArray[i] == BLACK)
                {
                    if(myMoveArray[2*moves-2*i-1] < y)
                    {
                        y = myMoveArray[2*moves-2*i-1];
                    }
                }
```

```
                }
                        return y;
 }

// Must pass in the current move number n ARRAY LAND!!! so n-1

 public int getNumberOfSequentialBlackMoves(int i)
 {
        int totalup = i;
        int totaldown = i;
        while(myMoveOrderArray[totalup] == 2)
        {
                totalup = totalup +1;
        }
        while(myMoveOrderArray[totaldown] == 2)
        {
                                        totaldown = totaldown - 1;
        }
        return(totalup - totaldown - 1);
 }

//must pass in the current move number n starts at 0 Array rules!

 public int getNumberOfBlackMovesUntilWhiteMovesAgain(int i)
 {
        int totalup = i;
        while(myMoveOrderArray[totalup] == 2)
        {
                totalup = totalup +1;
        }
        return(totalup - i);
 }

// must pass in the current move number n starts at 0

 public int determineWhitesLastMoveNumber(int i)
 {
         int totaldown = i;
         while(myMoveOrderArray[totaldown] == BLACK)
    {
            totaldown--;
         }
         return totaldown;
 }

 public int[] getArrayOfBlackMoves(GamingTree tree)
 {
         generateMoveArray(tree);
         int[] a;
         int movesNumber = myMoveArray.length;
         int counter = 0;
         for(int i = 0; i < movesNumber/2; i++)
         {
                 if(myMoveOrderArray[i] == BLACK)
                 {
                         counter++;
                 }
         }
         a = new int[2*counter];
         counter = 0;
         for(int i = 0; i < movesNumber/2; i++)
         {
                 if(myMoveOrderArray[i] == BLACK)
                 {
                         a[2*counter] = myMoveArray[movesNumber-2*i-2];
                         a[2*counter+1] = myMoveArray[movesNumber-2*i-1];
                         counter++;
                 }
         }
         return a;
```

```
}

public int[] getArrayOfWhiteMoves(GamingTree tree)
{
	generateMoveArray(tree);
	int[] a;
	int movesNumber = myMoveArray.length;
	int counter = 0;
	for(int i = 0; i < movesNumber/2; i++)
	{
		if(myMoveOrderArray[i] == WHITE)
		{
			counter++;
		}
	}
	a = new int[2*counter];
	counter = 0;
	for(int i = 0; i < movesNumber/2; i++)
	{
		if(myMoveOrderArray[i] == WHITE)
		{
			a[2*counter] = myMoveArray[movesNumber-2*i-2];
			a[2*counter+1] = myMoveArray[movesNumber-2*i-1];
			counter++;
		}
	}
	return a;
}

////// if this works the first run... //////

public int[] getOpenEdgeClosestToOtherBlackPositions(int x, int y, GamingTree tree)
{
	int[] a = getArrayOfOpenEdgesAroundPosition(x,y,tree);
	int[] b = new int[2];
	double[] minDistances;
	minDistances = new double[getNumberOfUnoccupiedConnectedPositionsTo(x,y,tree)];
	for(int i = 0; i < minDistances.length; i++)
	{
		minDistances[i] = 100;
	}
	double counter = 0;
	int length = myMoveArray.length;
//	calculates the mininum distance for each location
	for(int i = 0; i < minDistances.length; i++)
	{

		for(int j = 0; j < length/2; j++)
		{
			if(myMoveOrderArray[j] == BLACK)
			{
				double distance;
				distance = getDistanceBetweenPositions(myMoveArray[length-2*j-2],
myMoveArray[length-2*j-1], a[2*i], a[2*i+1]);
				if(distance < minDistances[i])
				{
					minDistances[i] = distance;
				}
			}
		}

	}
	counter = 100;
	int min = 100;
	for(int i = 0; i < minDistances.length;i++)
	{
		if(minDistances[i] < counter)
		{
			counter = minDistances[i];
			min = i;
```

```
                        }
                }
                b[0] = a[2*min];
                b[1] = a[2*min+1];
                return b;
        }

        public int[] getOpenEdgeClosestToBlacksLastMove(int x, int y, GamingTree tree)
        {
                int[] a = getArrayOfOpenEdgesAroundPosition(x,y,tree);
                int[] b = new int[2];
                int[] c = getBlacksLastMove(tree);
                double[] minDistances;
                minDistances = new double[getNumberOfUnoccupiedConnectedPositionsTo(x,y,tree)];
                for(int i = 0; i < minDistances.length; i++)
                {
                        minDistances[i] = 100;
                }
                double counter = 0;
//              calculates the mininum distance for each location
                for(int i = 0; i < minDistances.length; i++)
                {
                        double distance;
                        distance = getDistanceBetweenPositions(c[0], c[1], a[2*i], a[2*i+1]);
                        if(distance < minDistances[i])
                        {
                                minDistances[i] = distance;
                        }
                }
                counter = 100;
                int min = 100;
                for(int i = 0; i < minDistances.length;i++)
                {
                        if(minDistances[i] < counter)
                        {
                                counter = minDistances[i];
                                min = i;
                        }
                }
                b[0] = a[2*min];
                b[1] = a[2*min+1];
                return b;
        }

        public int[] getOpenBoxEdgeClosestToOtherBlackPositions(int x, int y, GamingTree tree)
        {
                int[] a = getUnnoccupiedSquaresOnTheBoxConnectedTo(x,y,tree);
                int[] b = new int[2];
                double[] minDistances;
                minDistances = new double[getNumberOfUnoccupiedConnectedBoxPositionsTo(x,y,tree)];
//              System.out.println("minDistances.length "+minDistances.length);
                for(int i = 0; i < minDistances.length; i++)
                {
                        minDistances[i] = 100;
                }
                double counter = 0;
                int length = myMoveArray.length;
//              calculates the mininum distance for each location
                for(int i = 0; i < minDistances.length; i++)
                {
                        for(int j = 0; j < length/2; j++)
                        {
                                if(myMoveOrderArray[j] == BLACK)
                                {
                                        double distance;
                                        distance = getDistanceBetweenPositions(myMoveArray[length-2*j-2],
myMoveArray[length-2*j-1], a[2*i], a[2*i+1]);
                                        if(distance < minDistances[i])
                                        {
                                                minDistances[i] = distance;
```

```
                                        }
                                }
                        }

                }
                counter = 100;
                int min = 100;
                for(int i = 0; i < minDistances.length;i++)
                {
                        if(minDistances[i] < counter)
                        {
                                counter = minDistances[i];
                                min = i;
                        }
                }
                b[0] = a[2*min];
                b[1] = a[2*min+1];
                return b;
        }

        public int[] getOpenEdgeClosestToOtherBlackConnectedPositions(int x, int y, GamingTree tree)
        {
                int[] a = getArrayOfOpenEdgesAroundPosition(x,y,tree);
                int[] b = new int[2];
                double[] minDistances;
                minDistances = new double[getNumberOfUnoccupiedConnectedPositionsTo(x,y,tree)];
                for(int i = 0; i < minDistances.length; i++)
                {
                        minDistances[i] = 100;
                }
                double counter = 0;
                int[] d = getArrayOfBlackEdgesConnectedToPosition(x,y,tree);
                int length = d.length;
        //      calculates the mininum distance for each location
                for(int i = 0; i < minDistances.length; i++)
                {
                        for(int j = 0; j < length/2; j++)
                        {
                                double distance;
                                distance = getDistanceBetweenPositions(d[length-2*j-2], d[length-2*j-1], a[2*i], a[2*i+1]);
                                if(distance < minDistances[i])
                                {
                                        minDistances[i] = distance;
                                }
                        }
                }
                counter = 100;
                int min = 100;
                for(int i = 0; i < minDistances.length;i++)
                {
                        if(minDistances[i] < counter)
                        {
                                counter = minDistances[i];
                                min = i;
                        }
                }
                b[0] = a[2*min];
                b[1] = a[2*min+1];
                return b;
        }

        public int[] getOpenCornerClosestToOtherBlackPositions(int x, int y, GamingTree tree)
        {
                int[] a = getArrayOfOpenCornersAroundPosition(x,y,tree);
                int[] b = new int[2];
                double[] minDistances;
                minDistances = new double[getNumberOfUnoccupiedCornersTo(x,y,tree)];
                for(int i = 0; i < minDistances.length; i++)
                {
                        minDistances[i] = 100;
```

```
		}
		double counter = 0;
		int length = myMoveArray.length;
//		calculates the mininum distance for each location
		for(int i = 0; i < minDistances.length; i++)
		{
			for(int j = 0; j < length/2; j++)
			{
				if(myMoveOrderArray[j] == BLACK)
				{
					double distance;
					distance = getDistanceBetweenPositions(myMoveArray[length-2*j-2],
myMoveArray[length-2*j-1], a[2*i], a[2*i+1]);
					if(distance < minDistances[i])
					{
						minDistances[i] = distance;

					}
				}
			}

		}
		counter = 100;
		int min = 100;
		for(int i = 0; i < minDistances.length;i++)
		{
			if(minDistances[i] < counter)
			{
				counter = minDistances[i];
				min = i;
			}
		}
		b[0] = a[2*min];
		b[1] = a[2*min+1];
		return b;
	}

	public boolean isPositionConnectedToOtherMoves(int x, int y, GamingTree tree)
	{
		boolean b = false;
		generateMoveArray(tree);
		for(int i=0; i < myMoveArray.length;i = i+2)
		{
			if(x == myMoveArray[i])
			{
				if(y+1 == myMoveArray[i+1] || y-1 == myMoveArray[i+1])
				{
					b = true;
				}
			}
			if(y == myMoveArray[i+1])
			{
				if(x+1 == myMoveArray[i] || x-1 == myMoveArray[i])
				{
					b = true;
				}
			}
		}
		return b;
	}

	public int[] getArrayOfOpenCornersAroundPositionNotConnectedToOtherPositions(int x, int y, GamingTree tree)
	{
		int[] a = getArrayOfOpenCornersAroundPosition(x,y,tree);
		int counter = 0;
		for(int i = 0; i < a.length; i = i+2)
		{
			if(!isPositionConnectedToOtherMoves(a[i],a[i+1],tree))
			{
				counter++;
```

```
                        }
                }
                int[] b;
                b = new int[2*counter];
                counter = 0;
                for(int i = 0; i < a.length; i = i+2)
                {
                        if(!isPositionConnectedToOtherMoves(a[i],a[i+1],tree))
                        {
                                b[2*counter] = a[i];
                                b[2*counter+1] = a[i+1];
                                counter++;
                        }
                }
                return b;
        }

        public int[] getOpenCornerClosestToOtherBlackPositionsNotConnectedToOtherPositions(int x, int y, GamingTree tree)

        {
                int[] a = getArrayOfOpenCornersAroundPositionNotConnectedToOtherPositions(x,y,tree);
                int[] b = new int[2];
                double[] minDistances;
                minDistances = new double[a.length/2];
                for(int i = 0; i < minDistances.length; i++)
                {
                        minDistances[i] = 100;
                }
                double counter = 0;
                int length = myMoveArray.length;
//              calculates the mininum distance for each location
                for(int i = 0; i < minDistances.length; i++)
                {
                        for(int j = 0; j < length/2; j++)
                        {
                                if(myMoveOrderArray[j] == BLACK)
                                {
                                        double distance;
                                        distance = getDistanceBetweenPositions(myMoveArray[length-2*j-2],
myMoveArray[length-2*j-1], a[2*i], a[2*i+1]);
                                        if(distance < minDistances[i])
                                        {
                                                minDistances[i] = distance;
                                        }
                                }
                        }

                }
                counter = 100;
                int min = 100;
                for(int i = 0; i < minDistances.length;i++)
                {
                        if(minDistances[i] < counter)
                        {
                                counter = minDistances[i];
                                min = i;
                        }
                }
                b[0] = a[2*min];
                b[1] = a[2*min+1];
                return b;
        }

        public double getDistanceBetweenPositions(int x1, int y1, int x2, int y2)
        {
                double distance = 0;
                double x11, x22, y11, y22;
                x11 = (double) x1;
                x22 = (double) x2;
                y11 = (double) y1;
```

```
		y22 = (double) y2;
		distance = Math.sqrt((y22-y11)*(y22-y11) + (x22-x11)*(x22-x11));
		return distance;
	}

// determines an array of open corners of myMoveArray from position x,y

	public int[] getArrayOfOpenCornersAroundPosition(int x, int y, GamingTree tree)
	{
		generateMoveArray(tree);
		populateBoardArray(myMoveArray);
		/*int counter = 0;
		int[] tempArray;
		tempArray = new int[8];
		for(int i = 0; i < 8; i++)
		{
			tempArray[i] = 0;
		}
		for(int i = 0; i < myMoveArray.length; i = i+2)
		{
			if(myMoveArray[i] == x+1 && myMoveArray[i+1] == y+1)
			{
				tempArray[2*counter] = x+1;
				tempArray[2*counter+1] = y+1;
				counter++;
			}
			if(myMoveArray[i] == x-1 && myMoveArray[i+1] == y+1)
			{
				tempArray[2*counter] = x-1;
				tempArray[2*counter+1] = y+1;
				counter++;
			}
			if(myMoveArray[i] == x+1 && myMoveArray[i+1] == y-1)
			{
				tempArray[2*counter] = x+1;
				tempArray[2*counter+1] = y-1;
				counter++;
			}
			if(myMoveArray[i] == x-1 && myMoveArray[i+1] == y-1)
			{
				tempArray[2*counter] = x-1;
				tempArray[2*counter+1] = y-1;
				counter++;
			}
		}
		int[] a = new int[counter*2];
		for(int i = 0; i < a.length; i++)
		{
			a[i] = tempArray[i];
		}
		counter = 0;
		tempArray[0] = x+1;
		tempArray[1] = y+1;
		tempArray[2] = x-1;
		tempArray[3] = y+1;
		tempArray[4] = x+1;
		tempArray[5] = y-1;
		tempArray[6] = x-1;
		tempArray[7] = y-1;
		for(int i = 0; i < a.length; i=i+2)
		{
			for(int j = 0; j < 8; j=j+2)
			{
				if(a[i] == tempArray[j] && a[i+1] == tempArray[j+1])
				{
					tempArray[j] = 0;
					tempArray[j+1] = 0;
					counter++;
				}
			}
```

```
		}
		a = new int[8-2*counter];
		counter = 0;
		for(int i = 0; i < 8; i=i+2)
		{
			if(tempArray[i] != 0)
			{
				a[2*counter] = tempArray[i];
				a[2*counter+1] = tempArray[i+1];
				counter++;
			}
		}
		return a; */
		int counter2 = 0;
		if(myBoardArray[x+1][y+1] == UNOCCUPIED)
		{
			counter2++;
		}
		if(myBoardArray[x+1][y-1] == UNOCCUPIED)
		{
			counter2++;
		}
		if(myBoardArray[x-1][y+1] == UNOCCUPIED)
		{
			counter2++;
		}
		if(myBoardArray[x-1][y-1] == UNOCCUPIED)
		{
			counter2++;
		}
		int[] c = new int[2*counter2];
		counter2 = 0;
		if(myBoardArray[x+1][y+1] == UNOCCUPIED)
		{
			c[2*counter2]   = x+1;
			c[2*counter2+1] = y+1;
			counter2++;
		}
		if(myBoardArray[x+1][y-1] == UNOCCUPIED)
		{
			c[2*counter2]   = x+1;
			c[2*counter2+1] = y-1;
			counter2++;
		}
		if(myBoardArray[x-1][y+1] == UNOCCUPIED)
		{
			c[2*counter2]   = x-1;
			c[2*counter2+1] = y+1;
			counter2++;
		}
		if(myBoardArray[x-1][y-1] == UNOCCUPIED)
		{
			c[2*counter2]   = x-1;
			c[2*counter2+1] = y-1;
			counter2++;
		}
		return c;
}

public int[] getArrayOfOpenEdgesAroundPosition(int x, int y, GamingTree tree)
{
		generateMoveArray(tree);
		populateBoardArray(myMoveArray);
		int counter = 0;
		if(myBoardArray[x+1][y] == UNOCCUPIED)
		{
			counter++;
		}
		if(myBoardArray[x-1][y] == UNOCCUPIED)
		{
```

```
                counter++;
        }
        if(myBoardArray[x][y+1] == UNOCCUPIED)
        {
                counter++;
        }
        if(myBoardArray[x][y-1] == UNOCCUPIED)
        {
                counter++;
        }
        int[] a;
        a = new int[2*counter];
        counter = 0;
        if(myBoardArray[x+1][y] == UNOCCUPIED)
        {
                a[2*counter] = x+1;
                a[2*counter+1] = y;
                counter++;
        }
        if(myBoardArray[x-1][y] == UNOCCUPIED)
        {
                a[2*counter] = x-1;
                a[2*counter+1] = y;
                counter++;
        }
        if(myBoardArray[x][y+1] == UNOCCUPIED)
        {
                a[2*counter] = x;
                a[2*counter+1] = y+1;
                counter++;
        }
        if(myBoardArray[x][y-1] == UNOCCUPIED)
        {
                a[2*counter] = x;
                a[2*counter+1] = y-1;
                counter++;
        }
        return a;
}


public int[] getArrayOfBlackEdgesConnectedToPosition(int x, int y, GamingTree tree)
{
        generateMoveArray(tree);
        populateBoardArray(myMoveArray);
        int counter = 0;
        if(myBoardArray[x+1][y] == BLACK)
        {
                counter++;
        }
        if(myBoardArray[x-1][y] == BLACK)
        {
                counter++;
        }
        if(myBoardArray[x][y+1] == BLACK)
        {
                counter++;
        }
        if(myBoardArray[x][y-1] == BLACK)
        {
                counter++;
        }
        int[] a;
        a = new int[2*counter];
        counter = 0;
        if(myBoardArray[x+1][y] == BLACK)
        {
                a[2*counter] = x+1;
                a[2*counter+1] = y;
```

```
                counter++;
        }
        if(myBoardArray[x-1][y] == BLACK)
        {
                a[2*counter] = x-1;
                a[2*counter+1] = y;
                counter++;
        }
        if(myBoardArray[x][y+1] == BLACK)
        {
                a[2*counter] = x;
                a[2*counter+1] = y+1;
                counter++;
        }
        if(myBoardArray[x][y-1] == BLACK)
        {
                a[2*counter] = x;
                a[2*counter+1] = y-1;
                counter++;
        }
        return a;
}

// determines a array called myAdjacentPossible moves around whites last move

 public void determineMovesAroundWhitesLastMove(GamingTree tree)
 {
        generateMoveArray(tree);
        determineMoveOrderArray();
   populateBoardArray();
        resetAdjacentPossibleMoves();

        int x = myMoveArray[0];
        int y = myMoveArray[1];
        int total = 0;

        if(myBoardArray[x][y+1] == 0)
        {
                myAdjacentPossibleMoves[total] = x;
                myAdjacentPossibleMoves[total+1] = (y+1);
                total = total+2;
        }
        if(myBoardArray[x][y-1] == 0)
        {
                 myAdjacentPossibleMoves[total] = x;
                 myAdjacentPossibleMoves[total+1] = (y-1);
                 total = total+2;
        }
        if(myBoardArray[x-1][y] == 0)

        {
                 myAdjacentPossibleMoves[total] = (x-1);
                 myAdjacentPossibleMoves[total+1] = y;
                 total = total+2;
        }
        if(myBoardArray[x+1][y] == 0)
        {
                 myAdjacentPossibleMoves[total] = (x+1);
                 myAdjacentPossibleMoves[total+1] = y;
                 total = total+2;
        }
   if(myBoardArray[x+1][y+1] == 0)
        {
        myAdjacentPossibleMoves[total] = (x+1);
        myAdjacentPossibleMoves[total+1] = (y+1);
           total = total+2;
        }
        if(myBoardArray[x+1][y-1] == 0)
        {
                 myAdjacentPossibleMoves[total] = (x+1);
```

```
                        myAdjacentPossibleMoves[total+1] = (y-1);
                    total = total+2;
                }
                if(myBoardArray[x-1][y+1] == 0)
                {
                        myAdjacentPossibleMoves[total] = (x-1);
                        myAdjacentPossibleMoves[total+1] = (y+1);
                    total = total+2;
                }
                if(myBoardArray[x-1][y-1] == 0)
                {
                        myAdjacentPossibleMoves[total] = (x-1);
                        myAdjacentPossibleMoves[total+1] = (y-1);
                    total = total+2;
                }
                int i = 0;
                while(myAdjacentPossibleMoves[i] != 0)
                {
                  i++;
                }
                int[] b;
                b = new int[i];
                for(int a = 0; a < i; a++)
                {
                        b[a] = myAdjacentPossibleMoves[a];
                }
                myAdjacentPossibleMoves = b;
}

public int getNumberOfUnoccupiedConnectedPositionsToWhitesLastMove(GamingTree tree)

{
                this.generateMoveArray(tree);
                int i = 4;
                int x,y;
                int length = myMoveArray.length/2;
                int lastmove = this.determineWhitesLastMoveNumber(length-1);
                int difference = length-lastmove-1;
                x = myMoveArray[2*(difference)];
                y = myMoveArray[2*difference+1];
                this.populateBoardArray();
                if(myBoardArray[x+1][y] != UNOCCUPIED)
                {
                        i--;
                }
                if(myBoardArray[x][y+1] != UNOCCUPIED)
                {
                        i--;
                }
                if(myBoardArray[x-1][y] != UNOCCUPIED)
                {
                        i--;
                }
                if(myBoardArray[x][y-1] != UNOCCUPIED)
                {
                        i--;
                }
                return i;
}

public int getNumberOfUnoccupiedConnectedPositionsTo(int x, int y, GamingTree tree)
{
                this.generateMoveArray(tree);
                int i = 4;
                this.populateBoardArray();
                if(myBoardArray[x+1][y] != UNOCCUPIED)
                {
                        i--;
                }
                if(myBoardArray[x][y+1] != UNOCCUPIED)
```

```
		{
			i--;
		}
		if(myBoardArray[x-1][y] != UNOCCUPIED)
		{
			i--;
		}
		if(myBoardArray[x][y-1] != UNOCCUPIED)
		{
			i--;
		}
		return i;
}

public int getNumberOfUnoccupiedConnectedBoxPositionsTo(int x, int y, GamingTree tree)
{
		this.generateMoveArray(tree);
		int i = 0;
		this.populateBoardArray();

		int MaxX = getMaxXWhitePosition(tree);
		int MinX = getMinXWhitePosition(tree);
		int MaxY = getMaxYWhitePosition(tree);
		int MinY = getMinYWhitePosition(tree);

		if(myBoardArray[x+1][y] == UNOCCUPIED && MaxX == x)
		{
			i++;
		}
		if(myBoardArray[x][y+1] == UNOCCUPIED && MaxY == y)
		{
			i++;
		}
		if(myBoardArray[x-1][y] == UNOCCUPIED && MinX == x)
		{
			i++;
		}
		if(myBoardArray[x][y-1] == UNOCCUPIED && MinY == y)
		{
			i++;
		}
		return i;
}

public int getNumberOfUnoccupiedCornersTo(int x, int y, GamingTree tree)
{
		this.generateMoveArray(tree);
		int i = 4;
		this.populateBoardArray();
		if(myBoardArray[x+1][y+1] != UNOCCUPIED)
		{
			i--;
		}
		if(myBoardArray[x-1][y+1] != UNOCCUPIED)
		{
			i--;
		}
		if(myBoardArray[x+1][y-1] != UNOCCUPIED)
		{
			i--;
		}
		if(myBoardArray[x-1][y-1] != UNOCCUPIED)
		{
			i--;
		}
		return i;
}

public boolean areThereOpenConnectionsFromLocation(int x, int y, GamingTree tree)
{
```

```
		boolean b = false;
		generateMoveArray(tree);
		populateBoardArray();
		if(myBoardArray[x+1][y] == UNOCCUPIED)
		{
			b = true;
		}
		if(myBoardArray[x-1][y] == UNOCCUPIED)
		{
			b = true;
		}
		if(myBoardArray[x][y+1] == UNOCCUPIED)
		{
			b = true;
		}
		if(myBoardArray[x][y-1] == UNOCCUPIED)
		{
			b = true;
		}
		return b;
	}

	public boolean isPositionXYOccupied(int x, int y, GamingTree tree)
	{
		boolean b = false;
		this.generateMoveArray(tree);
		for(int i = 0; i < myMoveArray.length; i=i+2)
		{
			if(myMoveArray[i] == x && myMoveArray[i+1] == y)
			{
				b = true;
			}
		}
		return b;
	}

	// Needs Checking

	public int[] getLastWhiteMoveWithOpenConnections(GamingTree tree)
	{
		int[] a;
		a = new int[2];
		a[0] = 0;
		a[1] = 0;
		generateMoveArray(tree);
		int length = myMoveArray.length;
/*		for(int i = myMoveArray.length; i > 0; i=i-2)
		{
			if(myMoveOrderArray[i/2-1] == WHITE)
			{
				if(areThereOpenConnectionsFromLocation(myMoveArray[myMoveArray.length-i-
2],myMoveArray[myMoveArray.length-i-1],tree))
				{
					a[0] = myMoveArray[myMoveArray.length-i-2];
					a[1] = myMoveArray[myMoveArray.length-i-1];
		}
			}
		} */
		for(int i = 0; i < myMoveArray.length; i = i+2)
		{
			if(myMoveOrderArray[i/2] == WHITE)
			{
				if(areThereOpenConnectionsFromLocation(myMoveArray[length-i-2],myMoveArray[length-i-
1],tree))
				{
					a[0] = myMoveArray[length-i-2];
					a[1] = myMoveArray[length-i-1];
				}
			}
		}
```

```
		return a;
}

public boolean myMoveArrayContains(int x, int y)
{
		boolean b = false;
		for(int i = 0; i < myMoveArray.length; i=i+2)
		{
			if(myMoveArray[i] == x && myMoveArray[i+1] == y)
			{
				b = true;
			}
		}
		return b;
}

public int[] getWhitesLastMove(GamingTree tree)
{
		int a[];
		a = new int[2];
		generateMoveArray(tree);
		int length = myMoveArray.length;
		for(int i = 0; i < length/2; i++)

		{
			if(myMoveOrderArray[i] == WHITE)
			{
				a[0] = myMoveArray[(length/2-i-1)*2];
				a[1] = myMoveArray[(length/2-i-1)*2+1];
			}
		}
		return a;
}

public int[] getBlacksLastMove(GamingTree tree)
{
		int a[];
		a = new int[2];
		generateMoveArray(tree);
		int length = myMoveArray.length;
		for(int i = 0; i < length/2; i++)
		{
			if(myMoveOrderArray[i] == BLACK)
			{
				a[0] = myMoveArray[(length/2-i-1)*2];
				a[1] = myMoveArray[(length/2-i-1)*2+1];
			}
		}
		return a;
}

public boolean areThereUnnocupiedSquaresOnTheBoxConnectedToWhitesLastMove(GamingTree tree)
{
		int a[] = getWhitesLastMove(tree);
		boolean b = false;
		int x = a[0];
		int y = a[1];
		//  System.out.println( a[0]+" "+a[1]);
		generateMoveArray(tree);
		populateBoardArray();
		int MaxX, MinX, MaxY, MinY;
		MaxX = getMaxXWhitePosition(tree);
		//  System.out.println("MaxX "+MaxX);
		MinX = getMinXWhitePosition(tree);
		//  System.out.println("MinX "+MinX);
		MaxY = getMaxYWhitePosition(tree);
		//  System.out.println("MaxY "+MaxY);
		MinY = getMinYWhitePosition(tree);
		//  System.out.println("MinY "+MinY);
```

```
		if(myBoardArray[x+1][y] == UNOCCUPIED && MaxX == x)
		{
			b = true;
		}
		if(myBoardArray[x-1][y] == UNOCCUPIED && MinX == x)
		{
			b = true;
		}
		if(myBoardArray[x][y+1] == UNOCCUPIED && MaxY == y)
		{
			b = true;
		}
		if(myBoardArray[x][y-1] == UNOCCUPIED && MinY == y)
		{
			b = true;
		}
		return b;
	}

	public int[] getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove(GamingTree tree)
	{
		generateMoveArray(tree);
		populateBoardArray();
		int a[] = getWhitesLastMove(tree);
		int x = a[0];
		int y = a[1];
		int[] b;
		b = new int[10];
		int MaxX, MinX, MaxY, MinY;
		MaxX = getMaxXWhitePosition(tree);
		MinX = getMinXWhitePosition(tree);
		MaxY = getMaxYWhitePosition(tree);
		MinY = getMinYWhitePosition(tree);
		for(int i = 0; i < 10; i++)
		{
			b[i] = 0;
		}
		int counter = 0;
		if(myBoardArray[x+1][y] == UNOCCUPIED && MaxX == x)
		{
			b[counter*2] = x+1;
			b[counter*2+1] = y;
//			System.out.println(" b[counter*2] = x+1 "+x+1);
//			System.out.println(" b[counter*2+1] = y "+ y);
//			System.out.println(" counter "+counter);
			counter++;
//			System.out.println(" counter+2 "+counter);
		}
		if(myBoardArray[x-1][y] == UNOCCUPIED && MinX == x)
		{
			b[counter*2] = x-1;
		   b[counter*2+1] = y;
//		   System.out.println(" b[counter*2] = x-1 "+(x-1));
//			System.out.println(" b[counter*2+1] = y "+ y);
//			System.out.println(" counter "+counter);
			counter++;
//			System.out.println(" counter+2 "+counter);
		}
		if(myBoardArray[x][y+1] == UNOCCUPIED && MaxY == y)
		{
			b[counter*2] = x;
			b[counter*2+1] = y+1;
//			System.out.println(" b[counter*2] = x "+x);
//			System.out.println(" b[counter*2+1] = y+1 "+ (y+1));
//			System.out.println(" counter "+counter);
			counter++;
//			System.out.println(" counter+2 "+counter);
		}
		if(myBoardArray[x][y-1] == UNOCCUPIED && MinY == y)
		{
```

```
                        b[counter*2] = x;
                        b[counter*2+1] = y-1;
//                      System.out.println(" b[counter*2] = x "+x);
//                      System.out.println(" b[counter*2+1] = y-1 "+ (y-1));
//                      System.out.println(" counter "+counter);
                        counter++;
//                      System.out.println(" counter+2 "+counter);
                }
                int counter2 = 0;
/*              System.out.println(" b.length "+b.length);
                for(int i = 0; i < b.length; i ++)
                {
                        System.out.println(" b[i] "+b[i]);
                }*/
                while(counter2 < 8 && b[counter2] != 0)
                {
               //       System.out.println("counter2 "+counter2);
               //       System.out.println("b[counter2] " +b[counter2]);
               //       System.out.println("b[counter2+2] " +b[counter2+2]);
                        counter2 = counter2 +2;
                }
                a = new int[counter2];
                for(int i = 0; i < counter2; i++)
                {
                        a[i] = b[i];
                }
                return a;
 }

 public int[] getUnnoccupiedSquaresOnTheBoxConnectedTo(int x, int y, GamingTree tree)
 {
                generateMoveArray(tree);
                populateBoardArray();
                int[] a;
                int[] b;
                b = new int[10];
                int MaxX, MinX, MaxY, MinY;
                MaxX = getMaxXWhitePosition(tree);
                MinX = getMinXWhitePosition(tree);
                MaxY = getMaxYWhitePosition(tree);
                MinY = getMinYWhitePosition(tree);
                for(int i = 0; i < 10; i++)
                {
                        b[i] = 0;
                }
                int counter = 0;
                if(myBoardArray[x+1][y] == UNOCCUPIED && MaxX == x)
                {
                        b[counter*2] = x+1;
                        b[counter*2+1] = y;
//                      System.out.println(" b[counter*2] = x+1 "+x+1);
//                      System.out.println(" b[counter*2+1] = y "+ y);
//                      System.out.println(" counter "+counter);
                        counter++;
//                      System.out.println(" counter+2 "+counter);
                }
                if(myBoardArray[x-1][y] == UNOCCUPIED && MinX == x)
                {
                        b[counter*2] = x-1;
                   b[counter*2+1] = y;
//                 System.out.println(" b[counter*2] = x-1 "+(x-1));
//                      System.out.println(" b[counter*2+1] = y "+ y);
//                      System.out.println(" counter "+counter);
                        counter++;
//                      System.out.println(" counter+2 "+counter);
                }
                if(myBoardArray[x][y+1] == UNOCCUPIED && MaxY == y)
                {
                        b[counter*2] = x;
                        b[counter*2+1] = y+1;
```

```
//                      System.out.println(" b[counter*2] = x "+x);
//                      System.out.println(" b[counter*2+1] = y+1 "+ (y+1));
//                      System.out.println(" counter "+counter);
                        counter++;
//                      System.out.println(" counter+2 "+counter);
                }
                if(myBoardArray[x][y-1] == UNOCCUPIED && MinY == y)
                {
                        b[counter*2] = x;
                        b[counter*2+1] = y-1;
//                      System.out.println(" b[counter*2] = x "+x);
//                      System.out.println(" b[counter*2+1] = y-1 "+ (y-1));
//                      System.out.println(" counter "+counter);
                        counter++;
//                      System.out.println(" counter+2 "+counter);
                }
                int counter2 = 0;
/*              System.out.println(" b.length "+b.length);
                for(int i = 0; i < b.length; i ++)
                {
                        System.out.println(" b[i] "+b[i]);
                }*/
                while(counter2 < 8 && b[counter2] != 0)
                {
               //               System.out.println("counter2 "+counter2);
               //               System.out.println("b[counter2] " +b[counter2]);
               //               System.out.println("b[counter2+2] " +b[counter2+2]);
                        counter2 = counter2 +2;
                }
                a = new int[counter2];
                for(int i = 0; i < counter2; i++)
                {
                        a[i] = b[i];
                }
                return a;
  }

/* DOES NOT!!!! WORK FIX IF NEEDED coppied from
EareThereUnnocupiedSquaresOnTheBoxCorneredToWhitesLastMove(GamingTree tree)
  public boolean areThereUnnocupiedSquaresOnTheBoxCorneredToWhitesLastMove(GamingTree tree)
  {
                int a[] = getWhitesLastMove(tree);
                boolean b = false;
                int x = a[0];
                int y = a[1];
               //  System.out.println( a[0]+" "+a[1]);
                generateMoveArray(tree);
                populateBoardArray();
                int MaxX, MinX, MaxY, MinY;
                MaxX = getMaxXWhitePosition(tree);
               //  System.out.println("MaxX "+MaxX);
                MinX = getMinXWhitePosition(tree);
               //  System.out.println("MinX "+MinX);
                MaxY = getMaxYWhitePosition(tree);
               //  System.out.println("MaxY "+MaxY);
                MinY = getMinYWhitePosition(tree);
               //  System.out.println("MinY "+MinY);
                if((myBoardArray[x+1][y] == UNOCCUPIED && ((MaxX == x || MinX == x) || (MaxY == y || MinY == y))))
                {
                        b = true;
                }
                if(myBoardArray[x-1][y] == UNOCCUPIED && ((MaxX == x || MinX == x) || (MaxY == y || MinY == y)))
                {
                        b = true;
                }
                if(myBoardArray[x][y+1] == UNOCCUPIED && ((MaxX == x || MinX == x) || (MaxY == y || MinY == y)))
                {
                        b = true;
                }
                if(myBoardArray[x-1][y] == UNOCCUPIED && ((MaxX == x || MinX == x) || (MaxY == y || MinY == y)))
```

```
		{
			b = true;
		}
		return b;
	}
*/

	public int[] getArrayOfBoxWhiteCorners(GamingTree tree)
	{
		int[] a;
		a = new int[8];
		a[0] = getMaxXWhitePosition(tree);
		a[1] = getMaxYWhitePosition(tree);
		a[2] = getMaxXWhitePosition(tree);
		a[3] = getMinYWhitePosition(tree);
		a[4] = getMinXWhitePosition(tree);
		a[5] = getMaxYWhitePosition(tree);
		a[6] = getMinXWhitePosition(tree);
		a[7] = getMinYWhitePosition(tree);
		return a;
	}

	public boolean isPositionAKeyCorner(int x, int y, GamingTree tree)
	{
		boolean b = false;
		int[] corners = getArrayOfBoxWhiteCorners(tree);
		for(int i = 0; i < 8; i=i+2)
		{
			if(x == corners[i] && y == corners[i+1])
			{
				b = true;
			}
		}
		return b;
	}

	public boolean isWhitesLastMoveAKeyCorner(GamingTree tree)

	{
		int x = getWhitesLastMove(tree)[0];
		int y = getWhitesLastMove(tree)[1];
		return(isPositionAKeyCorner(x,y,tree));
	}

	public int[] getKeyCornersFromPosition(int x, int y, GamingTree tree)
	{
		int[] a;
		int MaxX = getMaxXWhitePosition(tree);
		int MinX = getMinXWhitePosition(tree);
		int MaxY = getMaxYWhitePosition(tree);
		int MinY = getMinYWhitePosition(tree);
		int counter = 0;
		if(x == MaxX)
		{
			if(y == MaxY)
			{
				counter++;
			}
			if(y == MinY)
			{
				counter++;
			}
		}
		if(x == MinX)
		{
			if(y == MaxY)
			{
				counter++;
			}
			if(y == MinY)
```

```
                {
                        counter++;
                }
        }
        a = new int[2*counter];
        counter = 0;
        if(x == MaxX)
        {
                if(y == MaxY)
                {
                        a[2*counter]   = x+1;
                        a[2*counter+1] = y+1;
                        counter++;
                }
                if(y == MinY)
                {
                        a[2*counter]   = x+1;
                        a[2*counter+1] = y-1;
                        counter++;
                }
        }
        if(x == MinX)
        {
                if(y == MaxY)
                {
                        a[2*counter]   = x-1;
                        a[2*counter+1] = y+1;
                        counter++;
                }
                if(y == MinY)
                {
                        a[2*counter]   = x-1;
                        a[2*counter+1] = y-1;
                        counter++;
                }
        }
        return a;
 }

 public int[] determineKeyCornerClosestToOtherBlackPositions(int x, int y, GamingTree tree)
 {
        int[] a;
        a = getKeyCornersFromPosition(x,y,tree);
        int[] b = new int[2];
        double[] minDistances;
        minDistances = new double[a.length/2];
        for(int i = 0; i < minDistances.length; i++)

        {
                minDistances[i] = 100;
        }
        double counter = 0;
        int length = myMoveArray.length;
//      calculates the mininum distance for each location
        for(int i = 0; i < minDistances.length; i++)
        {
                for(int j = 0; j < length/2; j++)
                {
                        if(myMoveOrderArray[j] == BLACK)
                        {
                                double distance;
                                distance = getDistanceBetweenPositions(myMoveArray[length-2*j-2],
myMoveArray[length-2*j-1], a[2*i], a[2*i+1]);
                                if(distance < minDistances[i])
                                {
                                        minDistances[i] = distance;
                                }
                        }
                }
```

```
		}
		counter = 100;
		int min = 100;
		for(int i = 0; i < minDistances.length;i++)
		{
			if(minDistances[i] < counter)
			{
				counter = minDistances[i];
				min = i;
			}
		}
		b[0] = a[2*min];
		b[1] = a[2*min+1];
		return b;
	}

// READ THIS ABOUT THIS METHOD, ONLY WORKS WITH KEY CORNERS!

	public boolean isPositionConnectedToBlackBoxSquares(int x, int y, GamingTree tree)
	{
		boolean b = false;
		generateMoveArray(tree);
		populateBoardArray(myMoveArray);
		int MaxX = getMaxXWhitePosition(tree);
		int MinX = getMinXWhitePosition(tree);
		int MaxY = getMaxYWhitePosition(tree);
		int MinY = getMinYWhitePosition(tree);

		if(x == MaxX)
		{
			if(y == MaxY)
			{
				if(myBoardArray[MaxX+1][MaxY] != UNOCCUPIED || myBoardArray[MaxX][MaxY+1] !=
UNOCCUPIED)
				{
					b = true;
				}
			}
			if(y == MinY)
			{
				if(myBoardArray[MaxX+1][MinY] != UNOCCUPIED || myBoardArray[MaxX][MinY-1] !=
UNOCCUPIED)
				{
					b = true;
				}
			}
		}
		if(x == MinX)
		{
			if(y == MaxY)
			{
				if(myBoardArray[MinX-1][MaxY] != UNOCCUPIED || myBoardArray[MinX][MaxY+1] !=
UNOCCUPIED)
				{
					b = true;
				}
			}
			if(y == MinY)
			{
				if(myBoardArray[MinX-1][MinY] != UNOCCUPIED || myBoardArray[MinX][MinY-1] !=
UNOCCUPIED)
				{
					b = true;
				}
			}
		}
		return b;
	}

	public int getNumberOfOccupiedCornersAroundPosition(int x, int y, GamingTree tree)
```

```
{
	generateMoveArray(tree);
	populateBoardArray(myMoveArray);
	int counter = 0;
	if(myBoardArray[x+1][y] != UNOCCUPIED)
	{
		counter++;
	}
	if(myBoardArray[x-1][y] != UNOCCUPIED)
	{
		counter++;
	}
	if(myBoardArray[x][y+1] != UNOCCUPIED)
	{
		counter++;
	}
	if(myBoardArray[x][y-1] != UNOCCUPIED)
	{
		counter++;
	}
	return counter;
}

public boolean isPositionConnectedToBlackSquares(int x, int y, GamingTree tree)
{
	boolean b = false;
	generateMoveArray(tree);
	for(int i=0; i < myMoveArray.length;i = i+2)
	{
		if(myMoveOrderArray[myMoveArray.length/2-i/2-1] == BLACK)
		{
			if(x == myMoveArray[i])
			{
				if(y+1 == myMoveArray[i+1] || y-1 == myMoveArray[i+1])
				{
					b = true;
				}
			}
			if(y == myMoveArray[i+1])
			{
				if(x+1 == myMoveArray[i] || x-1 == myMoveArray[i])
				{
					b = true;
				}
			}
		}
	}
	return b;
}

public boolean isPositionATrueKeyCorner(int x, int y, GamingTree tree)
{
	boolean b = false;
	generateMoveArray(tree);
	int MaxX = getMaxXWhitePosition(tree);
	int MinX = getMinXWhitePosition(tree);
	int MaxY = getMaxYWhitePosition(tree);
	int MinY = getMinYWhitePosition(tree);
	int counter = 0;
	if(MaxX == x)
	{
		if(MaxY == y)
		{
			counter++;
		}
		if(MinY == y)
		{
			counter++;
		}
	}
```

```
		if(MinX == x)
		{
			if(MaxY == y)
			{
				counter++;
			}
			if(MinY == y)
			{
				counter++;
			}
		}
		if(counter == 1)
		{
			b = true;
		}
		return b;
}

public int[] getTrueKeyCornerFromPosition(int x, int y, GamingTree tree)
{
		int[] b;
		b = new int[2];
		generateMoveArray(tree);
		int MaxX = getMaxXWhitePosition(tree);
		int MinX = getMinXWhitePosition(tree);
		int MaxY = getMaxYWhitePosition(tree);
		int MinY = getMinYWhitePosition(tree);
		if(MaxX == x)
		{
			if(MaxY == y)
			{
				b[0] = x+1;
				b[1] = y+1;
			}
			if(MinY == y)
			{
				b[0] = x+1;
				b[1] = y-1;
			}
		}
		if(MinX == x)
		{
			if(MaxY == y)
			{
				b[0] = x-1;
				b[1] = y+1;
			}
			if(MinY == y)
			{
				b[0] = x-1;
				b[1] = y-1;
			}
		}
		return b;
}

public int[] getArrayOfOpenConnectedLocations(GamingTree tree)
{
		generateMoveArray(tree);
		int a[];
		a = new int[4*myMoveArray.length];
		for(int i = 0; i < a.length; i++)
		{
			a[i] = 100;
		}

		int counter = 0;
		for(int i = 0; i < myMoveArray.length; i = i+2)
		{
			if(myMoveOrderArray[myMoveArray.length/2-i/2-1] == WHITE)
```

```
                {
                        int[] b;
                        b = this.getArrayOfOpenEdgesAroundPosition(myMoveArray[i], myMoveArray[i+1], tree);
                        for(int j = 0; j < b.length;j++)
                        {
                                a[counter] = b[j];
                                counter++;
                        }
                }
        }
        int[]b = new int[counter];
        for(int i = 0; i < counter; i++)
        {
                b[i] = a[i];
        }
        return b;
}

public int[] getFurthestConnectedLocationAwayFromTheOrigin(GamingTree tree)
{
        int[] a = getArrayOfOpenConnectedLocations(tree);
        double[] distances = new double[a.length/2];
        for(int i = 0; i < distances.length; i++)
        {
                distances[i] = getDistanceBetweenPositions(a[2*i], a[2*i+1],20,20);
        }
        double max = 0;
        int x,y;
        x = 0;
        y = 0;
        for(int i = 0; i < distances.length; i++)
        {
                if(distances[i] > max)
                {
                        max = distances[i];
                        x = a[2*i];
                        y = a[2*i+1];
                }
        }
        int[] b = new int[2];
        b[0] = x;
        b[1] = y;
        return b;
}

public int[] getConnectedEdgeFurthestAwayFromTheOrigin(int x, int y, GamingTree tree)
{
        int[] a = this.getArrayOfOpenEdgesAroundPosition(x, y, tree);
        double[] distances = new double[a.length/2];
        for(int i = 0; i < distances.length; i++)
        {
                distances[i] = getDistanceBetweenPositions(a[2*i], a[2*i+1],20,20);
        }
        double max = 0;
        int c,d;
        c = 0;
        d = 0;
        for(int i = 0; i < distances.length; i++)
        {
                if(distances[i] > max)
                {
                        max = distances[i];
                        c = a[2*i];
                        d = a[2*i+1];
                }
        }
        int[] b = new int[2];
        b[0] = c;
        b[1] = d;
        return b;
```

```
 }

 public int[] determineBlackMove(GamingTree tree)
 {
            int[] theMove;
            theMove = new int[2];
            generateMoveArray(tree);
            int length = myMoveArray.length/2;
            determineMoveOrderArray();
   populateBoardArray();
            int maxWX, minWX, maxWY, minWY, maxBX,
            minBX, maxBY, minBY;
            maxWX = getMaxXWhitePosition(tree);
            minWX = getMinXWhitePosition(tree);
            maxWY = getMaxYWhitePosition(tree);
            minWY = getMaxYWhitePosition(tree);
            maxBX = getMaxXBlackPosition(tree);
            minBX = getMinXBlackPosition(tree);
            maxBY = getMaxYBlackPosition(tree);
            minBY = getMaxYBlackPosition(tree);
            int blackNumberOfMovesInARow = getNumberOfSequentialBlackMoves(length);
            int numberOfUnoccupiedConnectionsToWhitesLastMove =
getNumberOfUnoccupiedConnectedPositionsToWhitesLastMove(tree);
          //  System.out.println("blackNumberOfMovesUntilWhiteMovesAgain
"+getNumberOfBlackMovesUntilWhiteMovesAgain(length));
          //  System.out.println("blackNumberOfMovesInARow "+blackNumberOfMovesInARow);

/////// One Move In A Row     /////////////////

            if(blackNumberOfMovesInARow == 1)
            {
                  if(numberOfUnoccupiedConnectionsToWhitesLastMove == 0);
                  {
                        //Find New White Location Move Randomly
                        theMove = getFurthestConnectedLocationAwayFromTheOrigin(tree);
                  }
                  if(numberOfUnoccupiedConnectionsToWhitesLastMove == 1);
                  {
                        //Move Connected filling in remaining location
                        int[] a;
                        a = new int[0];
                        a = getWhitesLastMove(tree);
                        if(!myMoveArrayContains(a[0]+1,a[1]))
                        {
                              theMove[0] = a[0]+1;
                              theMove[1] = a[1];
                        }
                        if(!myMoveArrayContains(a[0]-1,a[1]))
                        {
                              theMove[0] = a[0]-1;
                              theMove[1] = a[1];
                        }
                        if(!myMoveArrayContains(a[0],a[1]+1))
                        {
                              theMove[0] = a[0];
                              theMove[1] = a[1]+1;
                        }
                        if(!myMoveArrayContains(a[0],a[1]-1))
                        {
                              theMove[0] = a[0];
                              theMove[1] = a[1]-1;
                        }
                  }
                  if(numberOfUnoccupiedConnectionsToWhitesLastMove == 2)
                  {
                        //Pick correct Edge
                        /* there are some cases when a corner must be picked
                         * How do we find this corner? If three corners are occupied,
                         * occupy the 4th corner. This may help... but not completely
                         * What else would help? any other hueristics?
```

```
                                * DAMN THE RIGHT CORNER! I CANT ALWAYS PICK IT

                                */
                                int x = getWhitesLastMove(tree)[0];
                                int y = getWhitesLastMove(tree)[1];
                                if(isPositionAKeyCorner(x,y,tree) && !isPositionConnectedToBlackBoxSquares(x,y,tree))
                                {
                                        theMove = determineKeyCornerClosestToOtherBlackPositions(x,y,tree);
                                        if(isPositionConnectedToBlackSquares(x,y,tree))
                                        {
                                                theMove = getOpenEdgeClosestToOtherBlackConnectedPositions(x, y, tree);
                                        }
                                        if(2 == this.getArrayOfOpenCornersAroundPosition(x, y, tree).length/2)
                                        {
                                                theMove = determineKeyCornerClosestToOtherBlackPositions(x,y,tree);
                                        }
                                }
                                else
                                {
                                        if(areThereUnnocupiedSquaresOnTheBoxConnectedToWhitesLastMove(tree))
                                   {
                                      theMove = getOpenBoxEdgeClosestToOtherBlackPositions(x,y,tree);
                                        }
                                   else
                                        {
                                      //theMove = getOpenEdgeClosestToOtherBlackPositions(x,y,tree);
                                        theMove = getConnectedEdgeFurthestAwayFromTheOrigin(x,y,tree);
                                        }
                                }
                                //
                                if(1 == getArrayOfOpenCornersAroundPosition(x,y,tree).length/2)
                                {
                                        theMove = getArrayOfOpenCornersAroundPosition(x, y, tree);
                                }
                        }
                        if(numberOfUnoccupiedConnectionsToWhitesLastMove == 3)
                        {
                                //Use Bounding Box to pick a corner

////////////////////////////////////////////////////////////////////
/*Picking the right corner in all situations seems impossible to code up
Too many cases to consider, Sometimes an Edge should be picked when NOUCTWLM 3 occurs
Too difficult to specify when the edge should be picked, not even sure of all the cases
when the edge should be picked. Picking a corner on the Box will have to suffice
Well see if something better is needed. Containment results mostly, */
////////////////////////////////////////////////////////////////////

                                int x = getWhitesLastMove(tree)[0];
                                int y = getWhitesLastMove(tree)[1];
                                if(isPositionAKeyCorner(x,y,tree))
                                {
                                        theMove = determineKeyCornerClosestToOtherBlackPositions(x,y,tree);
                                }
                                else
                                {

if(getArrayOfOpenCornersAroundPositionNotConnectedToOtherPositions(x,y,tree).length != 0)
                                        {
                                                theMove =
getOpenCornerClosestToOtherBlackPositionsNotConnectedToOtherPositions(x, y, tree);
                                        }
                                        else
                                        {
                                                //theMove = getOpenEdgeClosestToOtherBlackPositions(x,y,tree);
                                                theMove = getConnectedEdgeFurthestAwayFromTheOrigin(x,y,tree);
                                        }

                                }
                        }
                        if(numberOfUnoccupiedConnectionsToWhitesLastMove == 4)
```

```
                {
                        theMove[0] = 21;
                        theMove[1] = 19;
                }
        }

/////// Two Moves In A Row     ///////////////

        if(blackNumberOfMovesInARow == 2)
        {
                if(numberOfUnoccupiedConnectionsToWhitesLastMove == 0);
                {
                        //Find a new Location Connected to White's last Move Fill it in
                        theMove = getFurthestConnectedLocationAwayFromTheOrigin(tree);
                }
                if(numberOfUnoccupiedConnectionsToWhitesLastMove == 1);
                {
        // Move Connected filling in remaining location
                        int[] a;
                        a = new int[0];
                        a = getWhitesLastMove(tree);
                        if(!myMoveArrayContains(a[0]+1,a[1]))
                        {
                                theMove[0] = a[0]+1;
                                theMove[1] = a[1];
                        }
                        if(!myMoveArrayContains(a[0]-1,a[1]))
                        {
                                theMove[0] = a[0]-1;
                                theMove[1] = a[1];
                        }
                        if(!myMoveArrayContains(a[0],a[1]+1))
                        {
                                theMove[0] = a[0];
                                theMove[1] = a[1]+1;
                        }
                        if(!myMoveArrayContains(a[0],a[1]-1))
                        {
                                theMove[0] = a[0];
                                theMove[1] = a[1]-1;
                        }
                }
                if(numberOfUnoccupiedConnectionsToWhitesLastMove == 2)
                {
        // Move Connected filling in remaining location
                        int x = getWhitesLastMove(tree)[0];
                        int y = getWhitesLastMove(tree)[1];
                        if(areThereUnnocupiedSquaresOnTheBoxConnectedToWhitesLastMove(tree))
                        {
                                theMove = getOpenBoxEdgeClosestToOtherBlackPositions(x,y,tree);
                                if(getNumberOfBlackMovesUntilWhiteMovesAgain(length) == 1)
                                {
                                        theMove = getOpenEdgeClosestToBlacksLastMove(x, y, tree);
                                }
                        }
                        else
                        {
                                theMove = getOpenEdgeClosestToOtherBlackPositions(x,y,tree);
                                if(getNumberOfBlackMovesUntilWhiteMovesAgain(length) == 1)
                                {
                                        theMove = getOpenEdgeClosestToBlacksLastMove(x, y, tree);
                                }
                        }
                }
                if(numberOfUnoccupiedConnectionsToWhitesLastMove == 3)
                {
                        int x = getWhitesLastMove(tree)[0];
                        int y = getWhitesLastMove(tree)[1];
                        if(areThereUnnocupiedSquaresOnTheBoxConnectedToWhitesLastMove(tree))
                        {
```

```
                                        theMove = getOpenBoxEdgeClosestToOtherBlackPositions(x,y,tree);
                                }
                                else
                                {
                                        theMove = getOpenEdgeClosestToOtherBlackPositions(x,y,tree);
                                }
                        }
                        if(numberOfUnoccupiedConnectionsToWhitesLastMove == 4)
                        {
                                theMove[0] = 21;
                                theMove[1] = 20;
                        }
                }

/////// Three Moves Until White Moves Again     //////////////


                if(blackNumberOfMovesInARow == 3)
                {
                        if(numberOfUnoccupiedConnectionsToWhitesLastMove == 0);
                        {
//                                      Find a new Location Connected to White Move Fill it in
                                        theMove = getFurthestConnectedLocationAwayFromTheOrigin(tree);
                        }
                        if(numberOfUnoccupiedConnectionsToWhitesLastMove > 0);
                        {
//                                     Move Connected filling in remaining location
                                        int[] a;
                                        a = new int[0];
                                        a = getWhitesLastMove(tree);
                                        if(!myMoveArrayContains(a[0]+1,a[1]))
                                        {
                                                theMove[0] = a[0]+1;
                                                theMove[1] = a[1];
                                        }
                                        if(!myMoveArrayContains(a[0]-1,a[1]))
                                        {
                                                theMove[0] = a[0]-1;
                                                theMove[1] = a[1];
                                        }
                                        if(!myMoveArrayContains(a[0],a[1]+1))
                                        {
                                                theMove[0] = a[0];
                                                theMove[1] = a[1]+1;
                                        }
                                        if(!myMoveArrayContains(a[0],a[1]-1))
                                        {
                                                theMove[0] = a[0];
                                                theMove[1] = a[1]-1;
                                        }
                        }
                }

////////////////Four or More Black Moves ////////////////////

                if(blackNumberOfMovesInARow >= 4)
                {
                        if(numberOfUnoccupiedConnectionsToWhitesLastMove == 0)
                        {
                                theMove = getFurthestConnectedLocationAwayFromTheOrigin(tree);
                        }
                        else
                        {
//                                      Move Connected filling in remaining location
                                        int[] a;
                                        a = new int[0];
                                        a = getWhitesLastMove(tree);
                                        if(!myMoveArrayContains(a[0]+1,a[1]))
                                        {
                                                theMove[0] = a[0]+1;
```

```
                                        theMove[1] = a[1];
                                }
                                if(!myMoveArrayContains(a[0]-1,a[1]))
                                {
                                        theMove[0] = a[0]-1;
                                        theMove[1] = a[1];
                                }
                                if(!myMoveArrayContains(a[0],a[1]+1))
                                {
                                        theMove[0] = a[0];
                                        theMove[1] = a[1]+1;
                                }
                                if(!myMoveArrayContains(a[0],a[1]-1))
                                {
                                        theMove[0] = a[0];
                                        theMove[1] = a[1]-1;
                                }
                        }
                }
                return theMove;
        }

        public static void main(String[] args)
        {
          // Containment Tester Works*

                GamingTree tree = new GamingTree(20,20,0,false);
                GamingTree[] child1,child2,child3,child4;
          GamingTree child0 = new GamingTree();
          child0.determineLevel();
        //int[] moveOrderArray;
                child1 = new GamingTree[1];
                child2 = new GamingTree[1];
                child3 = new GamingTree[1];
                child4 = new GamingTree[1];
                child1[0] = new GamingTree(20,19,1,true,tree);
                tree.setChildren(child1);
                child2[0] = new GamingTree(20,21,2,true,child1[0]);
                child1[0].setChildren(child2);
                child3[0] = new GamingTree(19,20,3,true,child2[0]);
                child2[0].setChildren(child3);
                child4[0] = new GamingTree(21,20,4,true,child3[0]);
                child3[0].setChildren(child4);
                TreeAnalyzer treeAnaly = new TreeAnalyzer(tree);
                treeAnaly.setWhiteMoves(1);
                treeAnaly.setBlackMoves(1);
                treeAnaly.setmyMoveRatio(.5);

        /*    for(int i = 0; i < moveOrderArray.length; i++)
            {
                System.out.println("i= "+i+" MOA " + moveOrderArray[i]);
            } */

        /*    for(int i = 0; i < moveOrderArray.length; i++)
            {
                System.out.println("i= "+i+" MOA " + moveOrderArray[i]);
            } */


        /*    int j = 0;
            while(treeAnaly.getMoveArray()[j] != 0)
            {
                System.out.println(" i "+ treeAnaly.getMoveArray()[j]);
                j++;
            } */

                /* my move Order Array Tester */

                int[] array = treeAnaly.getMoveOrder();
```

```
		array = treeAnaly.getMoveOrder();

		/* my move Order Array Tester Done, Works perfectly*/

		/* determineMoveOrder Tester */

		int[] a;
	tree.setBlack(false);
		System.out.println("level " + tree.getLevel() + "X " + tree.getX() + "Y "
		+ tree.getY() + "Children " + tree.getChildNumber());
		treeAnaly.generateMoveArray(tree);
		a = treeAnaly.getMoveArray();
		tree = new GamingTree(20,20,0,false);
		treeAnaly.setWhiteMoves(1);
		treeAnaly.setBlackMoves(1.5);
		treeAnaly.determineMoveOrderArray();
		array = treeAnaly.getMoveOrder();
		for(int i = 0; i < 25; i++)
		{
			System.out.println(" order " + array[i]);
		}
		GamingTree Child1 = new GamingTree(21,21,1, true);
		GamingTree Child2 = new GamingTree(20,21,2, false);
		GamingTree Child3 = new GamingTree(19,21,3, true);
		GamingTree Child4 = new GamingTree(21,22,4, true);
		GamingTree Child5 = new GamingTree(20,22,5, false);
		GamingTree Child6 = new GamingTree(21,21,6, true);
		GamingTree Child7 = new GamingTree(18,20,7, false);
		GamingTree Child8 = new GamingTree(18,21,8, true);
	tree.setChildNumber(1);
	tree.setChild(0, Child1);
	Child1.setChildNumber(1);
	Child2.setChildNumber(1);
	Child3.setChildNumber(1);
	Child4.setChildNumber(1);
	Child5.setChildNumber(1);
	Child6.setChildNumber(1);
	Child7.setChildNumber(1);
	System.out.println("child number " + Child1.getChildNumber());
	Child1.setChild(0, Child2);
	Child2.setChild(0, Child3);

	Child3.setChild(0, Child4);
	Child4.setChild(0, Child5);
	Child5.setChild(0, Child6);
	Child6.setChild(0, Child7);
	Child7.setChild(0, Child8);
	treeAnaly.generateMoveArray(Child2);
		a = treeAnaly.getMoveArray();
		treeAnaly.determineMoveOrderArrayWithRatio();
	treeAnaly.determineMovesAroundWhitesLastMove(Child3);
	treeAnaly.determineMoveOrderArrayWithRatio();
	System.out.println("White's Last Move Number(Array) input 3 " +treeAnaly.determineWhitesLastMoveNumber(3));
	System.out.println("Number of Sequential Black Moves "+treeAnaly.getNumberOfSequentialBlackMoves(1));
	System.out.println("Number of Unoccupied Positions to White's last move
"+treeAnaly.getNumberOfUnoccupiedConnectedPositionsToWhitesLastMove(Child4));

	System.out.println("Are there open connections from location x, y "+ treeAnaly.areThereOpenConnectionsFromLocation(20, 20,
Child4));
	System.out.println("getNumberOfPositionsConnectedToWhitesLastMove
"+treeAnaly.getNumberOfUnoccupiedConnectedPositionsToWhitesLastMove(Child2));
	System.out.println("getLastWhiteMoveWithOpenConnections "+treeAnaly.getLastWhiteMoveWithOpenConnections(Child4)[0]+"
"+treeAnaly.getLastWhiteMoveWithOpenConnections(Child3)[1] );
	System.out.println("isLocationOccupied " + treeAnaly.isPositionXYOccupied(20, 21, child3[0]));
	System.out.println("getWhite'sLastMove "+ treeAnaly.getWhitesLastMove(Child3)[0]+ "
"+treeAnaly.getWhitesLastMove(Child3)[1]);
	System.out.println("   Array of Open Corners displaying ");

	for(int i = 0; i < treeAnaly.getArrayOfOpenCornersAroundPosition(21, 21, Child4).length; i = i+2)
	{
```

```
            System.out.println(treeAnaly.getArrayOfOpenCornersAroundPosition(21,21,Child4)[i]);
            System.out.println(treeAnaly.getArrayOfOpenCornersAroundPosition(21,21,Child4)[i+1]);
    }
    System.out.println("   Array of Open Corners displaying ");
    for(int i = 0; i < treeAnaly.getArrayOfOpenCornersAroundPosition(21, 20, Child4).length; i = i+2)
    {
            System.out.println(treeAnaly.getArrayOfOpenCornersAroundPosition(21,20,Child4)[i]);
            System.out.println(treeAnaly.getArrayOfOpenCornersAroundPosition(21,20,Child4)[i+1]);
    }
    System.out.println("   Array of Open Edges displaying ");
    for(int i = 0; i < treeAnaly.getArrayOfOpenEdgesAroundPosition(20, 20, Child4).length; i = i+2)
    {
            System.out.println(treeAnaly.getArrayOfOpenEdgesAroundPosition(20,20,Child4)[i]);
            System.out.println(treeAnaly.getArrayOfOpenEdgesAroundPosition(20,20,Child4)[i+1]);
    }
    System.out.println("   Array of Open Edges displaying ");
    for(int i = 0; i < treeAnaly.getArrayOfOpenEdgesAroundPosition(21, 20, Child4).length; i = i+2)
    {
            System.out.println(treeAnaly.getArrayOfOpenEdgesAroundPosition(21,20,Child4)[i]);
            System.out.println(treeAnaly.getArrayOfOpenEdgesAroundPosition(21,20,Child4)[i+1]);
    }
    System.out.println(" Distance between 3,3 and 6,7 ");
    System.out.println(treeAnaly.getDistanceBetweenPositions(3,3, 6, 7));
    System.out.println(" Array of Black Moves ");
    for(int i = 0; i < treeAnaly.getArrayOfBlackMoves(Child4).length; i=i+2)
    {
            System.out.println(treeAnaly.getArrayOfBlackMoves(Child4)[i]);
            System.out.println(treeAnaly.getArrayOfBlackMoves(Child4)[i+1]);
    }

    System.out.println(" Array of White Moves ");
    for(int i = 0; i < treeAnaly.getArrayOfWhiteMoves(Child4).length; i=i+2)
    {
            System.out.println(treeAnaly.getArrayOfWhiteMoves(Child4)[i]);
            System.out.println(treeAnaly.getArrayOfWhiteMoves(Child4)[i+1]);
    }
    System.out.println(" Closest Open Edge to 22,23 ");
    System.out.println(treeAnaly.getOpenEdgeClosestToOtherBlackPositions(22,23,Child4)[0]);
    System.out.println(treeAnaly.getOpenEdgeClosestToOtherBlackPositions(22,23,Child4)[1]);
    System.out.println(" Closest Open Corner to 19,22 ");
    System.out.println(treeAnaly.getOpenCornerClosestToOtherBlackPositions(19,22,Child4)[0]);
    System.out.println(treeAnaly.getOpenCornerClosestToOtherBlackPositions(19,22,Child4)[1]);

    System.out.println(treeAnaly.getNumberOfBlackMovesUntilWhiteMovesAgain(1));
    treeAnaly.generateFutureMoveArrayVersion1WSBP(Child4);
    System.out.println(" GenerateFutureMovesArrayWSBPlength "+treeAnaly.getFutureMoves().length);
    System.out.println(" areThereUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove
"+treeAnaly.areThereUnnocupiedSquaresOnTheBoxConnectedToWhitesLastMove(Child4));
    System.out.println(" White'sLastMove x "+treeAnaly.getWhitesLastMove(Child4)[0]);
    System.out.println(" WHite'sLastMove y "+treeAnaly.getWhitesLastMove(Child4)[1]);
    System.out.println(" Length of Array " +
treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove(Child4).length);
    for(int i = 0; i < treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove(Child4).length; i=i+2)
    {
            System.out.println(treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove(Child4)[i]);
            System.out.println(treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove(Child4)[i+1]);
    }
    System.out.println(treeAnaly.getNumberOfUnoccupiedConnectedBoxPositionsTo(19, 20, Child4));
    System.out.println(" getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove.length
"+treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove(Child4).length);
    for(int i = 0; i < treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove(Child4).length; i++)
    {
            System.out.println(treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedToWhitesLastMove(Child4)[i]);
    }
    System.out.println(" getUnnoccupiedSquaresOnTheBoxConnectedTo.length
"+treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedTo(21,20,Child4).length);
    for(int i = 0; i < treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedTo(21,20,Child4).length; i++)
    {
            System.out.println(treeAnaly.getUnnoccupiedSquaresOnTheBoxConnectedTo(21,20,Child4)[i]);
    }
```

```
  System.out.println(" get#OfUnnoccupiedConnectedBoxPositions ");
  System.out.println(treeAnaly.getNumberOfUnoccupiedConnectedBoxPositionsTo(21, 20, Child3));
  System.out.println(" treeAnaly.getOpenBoxEdgeClosestToOtherBlackPositions(20, 21, Child2) ");
  System.out.println( treeAnaly.getOpenBoxEdgeClosestToOtherBlackPositions(20, 21, Child3)[0]);
  System.out.println( treeAnaly.getOpenBoxEdgeClosestToOtherBlackPositions(20, 21, Child3)[1]);
  System.out.println("treeAnaly.getOpenCornerClosestToOtherBlackPositions(19, 20, Child5)[0] ");
  System.out.println( treeAnaly.getOpenCornerClosestToOtherBlackPositions(19, 20, Child5)[0]);
  System.out.println( treeAnaly.getOpenCornerClosestToOtherBlackPositions(19, 20, Child5)[1]);



  int[] d = treeAnaly.getArrayOfBoxWhiteCorners(Child5);
  System.out.println("Box Corners");
/*
  for(int i = 0; i < d.length;i++)
  {
        System.out.println("d[i] "+ d[i]);
  }
  System.out.println("is20,20akeyCorner ");
  System.out.println(treeAnaly.isPositionAKeyCorner(20, 20, Child5));
  System.out.println("is19,20akeyCorner ");
  System.out.println(treeAnaly.isPositionAKeyCorner(19, 20, Child5));
  System.out.println("is21,20akeyCorner ");
  System.out.println(treeAnaly.isPositionAKeyCorner(21, 20, Child5));
  System.out.println("getKeyCornersFromPosition 21,20");
  d = treeAnaly.getKeyCornersFromPosition(21,20,Child5);
  for(int i = 0; i < d.length; i++)
  {
        System.out.println("i, d[i] "+i+" "+d[i]);
  }
  d = treeAnaly.getKeyCornersFromPosition(19,20,Child5);
  System.out.println("getKeyCornersFromPosition 19,20");
  for(int i = 0; i < d.length; i++)
  {
        System.out.println("i, d[i] "+i+" "+d[i]);
  }
  System.out.println("determinekeyCornerClosestToOtherBlackPositions(20,20,tree");
  d = treeAnaly.determineKeyCornerClosestToOtherBlackPositions(20, 20, Child5);
  for(int i = 0; i < d.length; i++)
  {
        System.out.println("i, d[i] "+i+" "+d[i]);
  }
  System.out.println("IsPositionConnectedToOtherMoves(x,y,tree");

  System.out.println(treeAnaly.isPositionConnectedToOtherMoves(19, 20, Child5));

  System.out.println("getOpenCornerClosestToOtherBlackPositionsNotCOnnectedToOtherPositions(x,y,tree");

  d = treeAnaly.getOpenCornerClosestToOtherBlackPositionsNotConnectedToOtherPositions(22, 19, Child5);

  for(int i = 0; i < d.length; i++)
  {
        System.out.println("i, d[i] "+i+" "+d[i]);
  }
  */

 System.out.println(" hihi ");
 d = treeAnaly.determineBlackMove(Child8);
 for(int i = 0; i < d.length;i++)
 {
        System.out.println("i, d[i] "+i+", "+d[i]);
 }
 System.out.println("isPositionConnectedToBlackBoxSquares ");
 System.out.println(treeAnaly.isPositionConnectedToBlackBoxSquares(18, 19, Child1));

 System.out.println(treeAnaly.isPositionConnectedToBlackBoxSquares(20, 22, Child5));

 System.out.println("isPositionConnectedToOtherBlackSquares ");
 System.out.println(treeAnaly.isPositionConnectedToBlackSquares(20, 18, Child5));
 //hihi
```

```
   /*    for(int i = 0; i < treeAnaly.getAdjacentPossibleMoves().length; i++)
   {
            System.out.println(treeAnaly.getAdjacentPossibleMoves()[i]);
   }
  //  System.out.println("White's last moveNumber "+treeAnaly.determineWhitesLastMoveNumber(8));
            /* Generate Move Array traces the trees well. Producing the reverse ordered Array of moves*/

/*

            treeAnaly.generateFutureMoveArrayVersion1(tree);
            int[] moved  = treeAnaly.getMoveArray();
            int[] future = treeAnaly.getFutureMoves();
   for(int i = 0; i < moved.length; i = i+2)
            {
                     System.out.println("x " + moved[i]);
                     System.out.println("y " + moved[i+1]);
            }
            System.out.println("Future Length "+future.length);
            for(int i = 0; i < future.length; i++)
            {
                     System.out.println("/n " + future[i]);
            }

            tree.generateChildren(treeAnaly.getFutureMoves(),
            treeAnaly.isBlacksMove(tree.getLevel()+1));
            System.out.println("tree'sMove; " + treeAnaly.isBlacksMove(tree.getLevel()+1));


            for(int i = 0; i < tree.getChildNumber(); i++)
            {
                     System.out.println("/n x " + tree.getChild(i).getX());
                     System.out.println("/n y " + tree.getChild(i).getY());
                     System.out.println("/n L " + tree.getChild(i).getLevel());
                     System.out.println("/n L " + tree.getChild(i).isBlack());
            }

            for(int b = 0; b < tree.getChildNumber(); b++)
            {
                     child0 = tree.getChild(b);
                     treeAnaly.generateFutureMoveArrayVersion1(child0);
                     child0.generateChildren(treeAnaly.getFutureMoves(),
                     treeAnaly.isBlacksMove(child0.getLevel()+1));
                     System.out.println("Child "+b+"'s Children");
                     for(int i = 0; i < child0.getChildNumber(); i++)
                     {
                              System.out.println("/n x " + child0.getChild(i).getX());
                              System.out.println("/n y " + child0.getChild(i).getY());
                              System.out.println("/n L " + child0.getChild(i).getLevel());
                              System.out.println("/n L " + child0.getChild(i).isBlack());
                     }
            }

            child0 = tree.getChild(0).getChild(0);
            treeAnaly.generateFutureMoveArrayVersion1(child0);
            child0.generateChildren(treeAnaly.getFutureMoves(),
            treeAnaly.isBlacksMove(child0.getLevel()+1));
            System.out.println("Child 0's Children");
            for(int i = 0; i < child0.getChildNumber(); i++)
            {
                     System.out.println("/n x " + child0.getChild(i).getX());
                     System.out.println("/n y " + child0.getChild(i).getY());
                     System.out.println("/n L " + child0.getChild(i).getLevel());
                     System.out.println("/n L " + child0.getChild(i).isBlack());
            }

            child0 = tree.getChild(0).getChild(0).getChild(0);
            treeAnaly.generateFutureMoveArrayVersion1(child0);
            child0.generateChildren(treeAnaly.getFutureMoves(),
            treeAnaly.isBlacksMove(child0.getLevel()+1));
```

```
            System.out.println("Child 0's Children");
            for(int i = 0; i < child0.getChildNumber(); i++)
            {
                        System.out.println("/n x " + child0.getChild(i).getX());
                        System.out.println("/n y " + child0.getChild(i).getY());
                        System.out.println("/n L " + child0.getChild(i).getLevel());
                        System.out.println("/n L " + child0.getChild(i).isBlack());
            }
            child0 = tree.getChild(0).getChild(0).getChild(0).getChild(0);
            treeAnaly.generateFutureMoveArrayVersion1(child0);
            child0.generateChildren(treeAnaly.getFutureMoves(),
            treeAnaly.isBlacksMove(child0.getLevel()+1));
            System.out.println("Child 0's Children");
            for(int i = 0; i < child0.getChildNumber(); i++)
            {
                        System.out.println("/n x " + child0.getChild(i).getX());
                        System.out.println("/n y " + child0.getChild(i).getY());
                        System.out.println("/n L " + child0.getChild(i).getLevel());
                        System.out.println("/n L " + child0.getChild(i).isBlack());
            }

  */
 /*treeAnaly.setWhiteMoves((double)1);
  treeAnaly.setBlackMoves((double)2);
  treeAnaly.determineMoveOrderArray();
  for(int i =0; i <treeAnaly.getMoveOrder().length;i++)
  {
              System.out.println(treeAnaly.getMoveOrder()[i]);
  }
  */
                        treeAnaly.setWhiteMoves(1);
                        treeAnaly.setBlackMoves(1.75);
                        int[] f = treeAnaly.determineAndGetMoveOrderArray();
                        int white = 0;
                        int black = 0;
                        int whitenumber = 0;
                        int blacknumber = 0;



                        for(int i = 0; i < 16; i++)
                        {
                           System.out.println("i, f[i] "+i+", "+f[i]);
                           if(treeAnaly.getMoveOrder()[i] == 1)
                           {
                                    white++;
                                    whitenumber = i+1;
                           }
                           else
                           {
                                    black++;
                                    blacknumber = i+1;
                           }
                        }
                        System.out.println((white-1)+" white");
                        System.out.println(black+" black");
                        System.out.println(whitenumber+" white#");
                        System.out.println(blacknumber+" blacknumber");
  }
}
```

```
/*
 * Created on Sep 3, 2006
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */

/**
 * @author Adam
 *
 *
 */

public class GamingTree
{
  ////////////////////////////////////////////
  //          Properties                    //
  ////////////////////////////////////////////
  private GamingTree[] myChildrenArray;
  private GamingTree myParent;
  private int myX, myY, myLevel;
  private boolean isBlack;

  ////////////////////////////////////////////
  //          Methods                       //
  ////////////////////////////////////////////
  public GamingTree()
  {
              myParent = null;
              myX = 0;
              myY = 0;
              myLevel = 0;
              myChildrenArray = null;
              isBlack = false;
```

```
}

public GamingTree(int x, int y)
{
		myParent = null;
		myX = x;
		myY = y;
		myLevel = 0;
		myChildrenArray = null;
		isBlack = false;
}

public GamingTree(int x, int y, int L)
{
		myParent = null;
		myX = x;
		myY = y;
		myLevel = L;
		myChildrenArray = null;
		isBlack = false;
}

public GamingTree(int x, int y, int L, boolean b)
{
		myParent = null;
		myX = x;
		myY = y;

		myLevel = L;
		isBlack = b;
		myChildrenArray = null;
}

public GamingTree(int x, int y, int L, boolean b, GamingTree rent)
{
		myParent = rent;
		myX = x;
		myY = y;
		myLevel = L;
		isBlack = b;
		myChildrenArray = null;
}

public GamingTree(int x, int y, int L, GamingTree rent)
{
		myParent = rent;
  myX = x;
  myY = y;
		myLevel = L;
  myChildrenArray = new GamingTree[0];
		isBlack = false;
}

public GamingTree(int x, int y, int L, int children, GamingTree rent)
{
		myParent = rent;
		myX = x;
		myY = y;
		myLevel = L;
		myChildrenArray = new GamingTree[children];
		isBlack = false;
}

public void setX(int x)
{
  myX = x;
}

public void setY(int y)
{
```

```
		myY = y;
}

public void setLevel(int l)
{
		myLevel = l;
}

public void setXY(int x, int y)
{
  myY = y;
  myX = x;
}

public void setXYL(int x, int y, int l)
{
		myY = y;
		myX = x;
		myLevel = l;
}

public int getX()
{
  return(myX);
}

public int getY()
{
		return(myY);
}

public int getLevel()
{
		return(myLevel);
}

public void setParent(GamingTree rent)
{
		myParent = rent;
}

public GamingTree getParent()
{
		return myParent;
}

public int getChildNumber()
{
		 if(myChildrenArray != null)
		 {
		   return myChildrenArray.length;
		 }
		 else
		 {
				return 0;
		 }
}

public void setChildNumber(int a)
{
		 myChildrenArray = new GamingTree[a];
}

public GamingTree getChild(int c)
{
		return myChildrenArray[c];
}

public void setChild(int c, GamingTree tree)
{
```

```
                myChildrenArray[c] = tree;

                //changed
                myChildrenArray[c].setParent(this);
  }

  public GamingTree[] getChildren()
  {
                return myChildrenArray;
  }

  public void setChildren(GamingTree[] g)
  {
                myChildrenArray = g;
                for(int i = 0; i < g.length; i++)
                {
                  g[i].setParent(this);
                }
  }

  public boolean isBlack()
  {
                 return isBlack;
  }


  public void setBlack(boolean b)
  {
                 isBlack = b;
  }

/* must pass in the color of the next move, the Level is determined from
 * the previous level of the parent.
 */

  public void generateChildren(int[] c, boolean b)
  {
                 myChildrenArray = new GamingTree[c.length/2];
                 int index = 0;
                 for(int i = 0; i < c.length/2; i++)
                 {
                                myChildrenArray[i] = new GamingTree(c[index], c[index+1], this.getLevel()
                    +1, b, this);
                                index = index+2;
                 }
  }

  public void disownChildren()
  {
     for(int i=0; i < myChildrenArray.length; i++)
     {
        myChildrenArray[i].setParent(null);
     }
     myChildrenArray = null;
  }

  public void determineLevel()
  {
    if(this.getParent() != null)
    {
      myLevel = this.getParent().getLevel()+1;
    }
    else
    {
              myLevel = 0;
    }
  }


/*
```

```
 * Note: Input here must be a valid Future Move Array see Tree Analyzer Comments
 * for additional information about structure of Future Move Array
 */

  public void createChildren(int[] c, int L, boolean b)
  {
                myChildrenArray = new GamingTree[c.length/2];
                for(int i = 0; i < c.length/2; i = i++)
                {
                  myChildrenArray[i] = new GamingTree(c[2*i], c[2*i+1], L, b);
                  myChildrenArray[i].setParent(this);
                }
  }


}
```

```
/*
 * Created on Oct 1, 2006
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */

public class SquareSymmetryChecker
{
   private int[][]  my1Moves, my2Moves;
   private int[]    myMoveOrderArray;
   private double   myWhiteMoves, myBlackMoves;

   public SquareSymmetryChecker()
   {
                my1Moves = null;
                my2Moves = null;
                myMoveOrderArray = null;
                myWhiteMoves = 0;
                myBlackMoves = 0;
   }

   public SquareSymmetryChecker(double w, double b)
   {
                my1Moves = null;
                my2Moves = null;
                myMoveOrderArray = null;
                myWhiteMoves = w;
                myBlackMoves = b;
                determineMoveOrderArray();
```

```
}

public void setWhiteMoves(double w)
{
	myWhiteMoves = w;
}

public void setBlackMoves(double b)
{
	myBlackMoves = b;
}

public double getWhiteMoves()
{
	return myWhiteMoves;
}

public double getBlackMoves()
{
	return myBlackMoves;
}

public int[] getMoveOrderArray()
{
	return myMoveOrderArray;
}

public int[][] getmy1Moves()
{
	return my1Moves;
}

public int[][] getmy2Moves()
{
	return my2Moves;
}

public boolean is12Symmetric(int[] m1, int[] m2)
{
	boolean symmetric = true;
	generateM1Moves(m1);
	generateM2Moves(m2);
	int x1, x2, y1, y2;
	x1 = my1Moves.length;
	x2 = my2Moves.length;
	y1 = my1Moves[0].length;
	y2 = my2Moves[0].length;
	String myString = new String("");
	String tempString = new String("");
	// Checks to see if the dimensions match.

	if(x1 == x2)
	{
		if(y1 != y2)
		{
		   symmetric = false;
		}
	}
	else
	{
		if(x1 == y2)
		{
			if(x2 != y1)
			{
			   symmetric = false;
			}
		}
		else
		{
			symmetric = false;
```

```
                }
        }

        if(symmetric)
        {
                symmetric = false;
                myString = generateSymmetries(my1Moves);
                for(int i = 0; i < my2Moves.length; i++)
                {
                        for(int j = 0; j < my2Moves[0].length; j++)
                        tempString = tempString + my2Moves[i][j];
                }
                for(int i = 0; i < 8; i++)
                {
                        if(tempString.equalsIgnoreCase(myString.substring(3+
                          i*my1Moves.length*my1Moves[0].length, 3+
                          (i+1)*my1Moves.length*my1Moves[0].length)))
                        {
                                symmetric = true;
                        }
                }
        }
        return symmetric;
   }

public int getSymmetryStringLength(int[][] a)
{
        int    counter = 0;
        counter = 8*(a.length)*(a[0].length) + 3;
        return counter;
}

public int getNumberOfMovesInArray(int[][] a)
{
        int    counter = 0;
        for(int i = 0; i < a.length; i++)
        {
                for(int j = 0; j < a[0].length; j++)
                {
                        if(a[i][j] == 1 || a[i][j] == 2)
                        {
                                counter++;
                        }
                }
        }
        return counter;
}

   /*

    * Pass in a clipped 2D-Array, returns a string with the first 3 characters
    * representing X length, Y length, # of moves.
    */

   public String generateSymmetries(int[][] a)
   {
        String tempString;
        int    counter = getNumberOfMovesInArray(a);

        //creates String (x,y,Move #)

        tempString = new String("");
        tempString = tempString+a.length+a[0].length+counter;

        //+x,+y

        for(int i = 0; i < a.length; i++)
        {
                for(int j = 0; j < a[0].length; j++)
                {
```

```
                        tempString = tempString+a[i][j];
                }
}

//+x,-y

for(int i = 0; i < a.length; i++)
{
                for(int j = a[0].length-1; j > -1; j--)
                {
                        tempString = tempString+a[i][j];
                }
}

//-x,+y

for(int i = a.length-1; i > -1; i--)
{
                for(int j = 0; j < a[0].length; j++)
                {
                        tempString = tempString+a[i][j];
                }
}

//-x,-y

for(int i = a.length-1; i > -1; i--)
{
                for(int j = a[0].length-1; j > -1; j--)
                {
                        tempString = tempString+a[i][j];
                }
}

//+y,+x

for(int j = 0; j < a[0].length; j++)
{
                for(int i = 0; i < a.length; i++)
                {
                        tempString = tempString+a[i][j];
                }
}

//+y,-x

for(int j = 0; j < a[0].length; j++)
{
                for(int i = a.length-1; i > -1; i--)
                {
                        tempString = tempString+a[i][j];
                }
}

//-y,+x

for(int j = a[0].length-1; j > -1; j--)
{
                for(int i = 0; i < a.length; i++)
                {
                        tempString = tempString+a[i][j];
                }
}

//-y,-x

for(int j = a[0].length-1; j > -1; j--)
{
                for(int i = a.length-1; i > -1; i--)
                {
```

```
                                        tempString = tempString+a[i][j];
                                }
                        }

                        return tempString;
                }

                /* pass in an array of moves, x,y,x,y,... generates
                 * a minimal dimensional array
                 */

                public void generateM1Moves(int[] m1)
                {
                        determineMoveOrderArray();
                        int maxX, minX, maxY, minY;
                        maxX = m1[0];
                        minX = m1[0];
                        maxY = m1[1];
                        minY = m1[1];
        /*
         * This determines the maximum and minimum positions of the
         * move Array so we can trim it.
         */
                        for(int i = 0; i < m1.length; i=i+2)
                        {
                                if(maxX < m1[i])
                                {
                                        maxX = m1[i];
                                }
                                if(minX > m1[i])
                                {
                                        minX = m1[i];
                                }
                                if(maxY < m1[i+1])
                                {
                                        maxY = m1[i+1];
                                }
                                if(minY > m1[i+1])
                                {
                                        minY = m1[i+1];
                                }
                        }
                        //System.out.println("max X "+maxX+" min X "+minX);
                        //System.out.println("max Y "+maxY+" min Y "+minY);
                        my1Moves = new int[maxX-minX+1][maxY-minY+1];

                        /*
                         * Initializes terms of my1Moves
                         */

                        for(int i = 0; i < maxX-minX+1; i++)
                        {
                                for(int j = 0; j < maxY-minY+1; j++)
                                {
                                        my1Moves[i][j] = 0;
                                }
                        }

                        /*
                         * Assigns appropriate terms of my the moveArray to 1 or 2
                         */

                        for(int i = 0; i < m1.length; i = i+2)
                        {
                                my1Moves[m1[i]-minX][m1[i+1]-minY] = myMoveOrderArray[i/2];
                        }
                }

                public void generateM2Moves(int[] m2)
                {
```

```
		determineMoveOrderArray();
		int maxX, minX, maxY, minY;
		maxX = m2[0];
		minX = m2[0];
		maxY = m2[1];
		minY = m2[1];
/*
 * This determines the maximum and minimum positions of the
 * move Array so we can trim it.
 */
		for(int i = 0; i < m2.length; i=i+2)
		{
			if(maxX < m2[i])
			{
				maxX = m2[i];
			}
			if(minX > m2[i])
			{
				minX = m2[i];
			}
			if(maxY < m2[i+1])
			{
				maxY = m2[i+1];
			}
			if(minY > m2[i+1])
			{
				minY = m2[i+1];
			}
		}
		my2Moves = new int[maxX-minX+1][maxY-minY+1];

		/*
		 * Initializes all terms of my2Moves to 0
		 */

		for(int i = 0; i < maxX-minX+1; i++)
		{
			for(int j = 0; j < maxY-minY+1; j++)
			{
				my2Moves[i][j] = 0;
			}
		}

		/*
		 * Assigns appropriate terms of my the moveArray to 1 or 2
		 */

		for(int i = 0; i < m2.length; i = i+2)
		{
			my2Moves[m2[i]-minX][m2[i+1]-minY] = myMoveOrderArray[i/2];
		}
	}



		public void determineMoveOrderArray()
		{
		   double blackFractionTotal = 0;
		   double whiteFractionTotal = 0;
		   int blackMoveTotal = 0;
		   int whiteMoveTotal = 0;
		   int index = 0;
		   int currentBlackMoves = 0;
		   int currentWhiteMoves = 0;
		   int[] myMoveOrderA = new int[100];
		   myMoveOrderArray = myMoveOrderA;
		   while(index < 100)
		   {
		      /*
```

```
                * increments the total move fraction
                */
               blackFractionTotal = blackFractionTotal + myBlackMoves;
               whiteFractionTotal = whiteFractionTotal + myWhiteMoves;
          /*
           * determines the number of moves each player recieves each turn
           */
               currentBlackMoves = (int) (blackFractionTotal - (double)blackMoveTotal);
               currentWhiteMoves = (int) (whiteFractionTotal - (double)whiteMoveTotal);
               /*
                * assigns the moves to myMoveOrderArray;
                */
               for(int i = 0; i < currentWhiteMoves; i++)
               {
                  if(index < 100)
                  {
                    myMoveOrderArray[index] = 1;
                  }
                  index++;
               }

               for(int i = 0; i < currentBlackMoves; i++)
               {
                  if(index < 100)
                  {
                    myMoveOrderArray[index] = 2;
                  }
                  index++;
               }
               /*
                * increments blackMoveTotal and whiteMoveTotal
                */
               blackMoveTotal = blackMoveTotal + currentBlackMoves;
               whiteMoveTotal = whiteMoveTotal + currentWhiteMoves;
            }
         }



         public static void main(String args[])
         {
//checks nXm array Maker given a move Array. Works
//also checks logic of Symmetric checker

                  int[] moveOrder, moveOrder2, moveOrder3, moveOrder4;
                  moveOrder = new int[6];
                  moveOrder2 = new int[6];
                  moveOrder3 = new int[6];
                  moveOrder4 = new int[6];
                  int[][] mMoves;
                  moveOrder[0] = 2;
                  moveOrder[1] = 1;
                  moveOrder[2] = 3;
                  moveOrder[3] = 2;
                  moveOrder[4] = 2;
                  moveOrder[5] = 3;
                  moveOrder2[0] = 1;
                  moveOrder2[1] = 0;
                  moveOrder2[2] = 0;
                  moveOrder2[3] = 1;
                  moveOrder2[4] = 1;
                  moveOrder2[5] = 2;
                  moveOrder3[0] = 1;
                  moveOrder3[1] = 5;
                  moveOrder3[2] = 2;
                  moveOrder3[3] = 4;
                  moveOrder3[4] = 3;
                  moveOrder3[5] = 5;
                  moveOrder4[0] = 5;
                  moveOrder4[1] = 8;
```

```
                moveOrder4[2] = 6;
                moveOrder4[3] = 9;
                moveOrder4[4] = 7;
                moveOrder4[5] = 9;
                SquareSymmetryChecker SSC = new SquareSymmetryChecker(1,1);
                SSC.generateM2Moves(moveOrder);
                mMoves = SSC.getmy2Moves();
                System.out.println(SSC.generateSymmetries(mMoves));
                System.out.println(SSC.generateSymmetries(mMoves).length());
                System.out.println(SSC.getNumberOfMovesInArray(mMoves));
                System.out.println(SSC.getSymmetryStringLength(mMoves));
                String string = new String("abcdefg");
                System.out.println(string.substring(3,5));
                System.out.println(SSC.is12Symmetric(moveOrder, moveOrder2));
                System.out.println(SSC.is12Symmetric(moveOrder, moveOrder3));
                System.out.println(SSC.is12Symmetric(moveOrder, moveOrder4));
                System.out.println(SSC.is12Symmetric(moveOrder2, moveOrder3));
                System.out.println(SSC.is12Symmetric(moveOrder2, moveOrder4));
                System.out.println(SSC.is12Symmetric(moveOrder3, moveOrder4));

        }
}
```

```
/*
 * Created on Sep 12, 2006
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */

public class TreeMaker
{
  private GamingTree        myGamingTree;
  private TreeAnalyzer      myTreeAnalyzer;

  public TreeMaker()
  {
        myGamingTree = new GamingTree(20,20,0,false);
        myTreeAnalyzer = new TreeAnalyzer(myGamingTree);
  }

  public TreeMaker(GamingTree tree)
  {
        myGamingTree = tree;
```

```
		myTreeAnalyzer = new TreeAnalyzer(tree);
	}

	public TreeMaker(double w, double b)
	{
		myGamingTree = new GamingTree(20,20,0,false);
		myTreeAnalyzer = new TreeAnalyzer(myGamingTree);
		myTreeAnalyzer.setBlackMoves(b);
		myTreeAnalyzer.setWhiteMoves(w);
		myTreeAnalyzer.determineMoveOrderArray();
	}

	public GamingTree getGamingTree()
	{
		return myGamingTree;
	}

	public void setGamingTree(GamingTree tree)
	{
		myGamingTree = tree;
	}


	public TreeAnalyzer getTreeAnalyzer()
	{
		return myTreeAnalyzer;
	}

	public void setMoveNumbers(double w, double b)
	{
		myTreeAnalyzer.setBlackMoves(b);
		myTreeAnalyzer.setWhiteMoves(w);
		myTreeAnalyzer.determineMoveOrderArray();
	}

	public void setMoveRatio(double d)
	{
		myTreeAnalyzer.setmyMoveRatio(d);
		myTreeAnalyzer.determineMoveOrderArrayWithRatio();
	}

	public void createTree(int[] a)
	{
		int index = 2;
		myGamingTree = new GamingTree(a[0],a[1],0,myTreeAnalyzer.isBlacksMove(0));
		GamingTree temp = myGamingTree;
		while(index < a.length)
		{
			int[] c;
			c = new int[2];
			c[0]=a[index];
			c[1]=a[index+1];
			temp.generateChildren(c, myTreeAnalyzer.isBlacksMove(index/2));
			temp = temp.getChild(0);
			index = index+2;
		}
	}

public void generateTree(int level, GamingTree tree)
	{
	  if(tree.getLevel() == level)
	  {

	  }
	  else
	  {
	    GamingTree tempTree = tree;
	    int[] futureMoveArray;
	    int[] moveOrder;
	    myTreeAnalyzer.setGamingTree(tempTree);
```

```
        moveOrder = myTreeAnalyzer.getMoveOrder();
        myTreeAnalyzer.generateFutureMoveArrayVersion1WS(tree);
        futureMoveArray = myTreeAnalyzer.getFutureMoves();
        tempTree.generateChildren(futureMoveArray, (moveOrder[tree.getLevel()+1] == 2));
                    for(int i = 0; i < tempTree.getChildNumber(); i++)
                    {
                      System.out.println("-----------------");
                      System.out.println("i "+i);
                      System.out.println("Child# "+tempTree.getChildNumber());
                      System.out.println("x " +tempTree.getChild(i).getX());
                      System.out.println("y " +tempTree.getChild(i).getY());
                      System.out.println("L " +tempTree.getChild(i).getLevel());
                      System.out.println("isB "+tempTree.getChild(i).isBlack());
          generateTree(level, tempTree.getChild(i));
                    }
      }
    }

public void generateBlackPickingTree(int level, GamingTree tree)
{
    if(tree.getLevel() == level && !myTreeAnalyzer.isWhiteContainedV1(tree))
    {

    }
    else
    {
      GamingTree tempTree = tree;
      int[] futureMoveArray;
      int[] moveOrder;
      myTreeAnalyzer.setGamingTree(tempTree);
      moveOrder = myTreeAnalyzer.getMoveOrder();
      System.out.println("myTreeAnalyzer.isWhiteContainedV1(tempTree) "+myTreeAnalyzer.isWhiteContainedV1(tempTree));
      if(!myTreeAnalyzer.isWhiteContainedV1(tempTree))
      {
                myTreeAnalyzer.generateFutureMoveArrayVersion1WSBP(tree);
                futureMoveArray = myTreeAnalyzer.getFutureMoves();
                tempTree.generateChildren(futureMoveArray, (moveOrder[tree.getLevel()+1] == 2));
                            for(int i = 0; i < tempTree.getChildNumber(); i++)
                            {
                             /*System.out.println("-----------------");
                            System.out.println("i "+i);
                            System.out.println("Child# "+tempTree.getChildNumber());
                            System.out.println("x " +tempTree.getChild(i).getX());
                            System.out.println("y " +tempTree.getChild(i).getY());
                            System.out.println("L " +tempTree.getChild(i).getLevel());
                            System.out.println("isB "+tempTree.getChild(i).isBlack());*/
                            generateBlackPickingTree(level, tempTree.getChild(i));
                             }
      }
    }
}

public int[] generateBlackPickingTreePruned(int level, GamingTree tree)
{
                myTreeAnalyzer.setGamingTree(tree);
                myTreeAnalyzer.generateMoveArray(tree);
                int[] TheMoves = myTreeAnalyzer.getMoveArray();
    if(tree.getLevel() == level && !myTreeAnalyzer.isWhiteContainedV1(tree))
    {

    }
    else
    {
       GamingTree tempTree = tree;
       int[] futureMoveArray;
       int[] moveOrder;
       myTreeAnalyzer.setGamingTree(tempTree);
       moveOrder = myTreeAnalyzer.getMoveOrder();
      // System.out.println("myTreeAnalyzer.isWhiteContainedV1(tempTree) "+myTreeAnalyzer.isWhiteContainedV1(tempTree));
       if(!myTreeAnalyzer.isWhiteContainedV1(tempTree))
```

```
    {
            myTreeAnalyzer.generateFutureMoveArrayVersion1WSBP(tree);
            futureMoveArray = myTreeAnalyzer.getFutureMoves();
            tempTree.generateChildren(futureMoveArray, (moveOrder[tree.getLevel()+1] == 2));
                    for(int i = 0; i < tempTree.getChildNumber(); i++)
                    {
                            /* System.out.println("-----------------");
                    System.out.println("i "+i);
                    System.out.println("Child# "+tempTree.getChildNumber());
                    System.out.println("x " +tempTree.getChild(i).getX());
                    System.out.println("y " +tempTree.getChild(i).getY());
                    System.out.println("L " +tempTree.getChild(i).getLevel());
                    System.out.println("isB "+tempTree.getChild(i).isBlack()); */
                    int[] a = generateBlackPickingTreePruned(level, tempTree.getChild(i));
                    if(TheMoves.length < a.length)
                    {
                            TheMoves = a;
                    }
                   }
                   tempTree.disownChildren();
    }
  }
  return TheMoves;
}


public void generateWhitePickingTree(int level, GamingTree tree)
{
  if(tree.getLevel() == level)
  {

  }
  else
  {
    GamingTree tempTree = tree;
    int[] futureMoveArray;
    int[] moveOrder;
    myTreeAnalyzer.setGamingTree(tempTree);
    moveOrder = myTreeAnalyzer.getMoveOrder();
    myTreeAnalyzer.generateFutureMoveArrayVersion1WSWP(tree);
    futureMoveArray = myTreeAnalyzer.getFutureMoves();
    tempTree.generateChildren(futureMoveArray, (moveOrder[tree.getLevel()+1] == 2));
            for(int i = 0; i < tempTree.getChildNumber(); i++)
            {
              System.out.println("-----------------");
              System.out.println("i "+i);
              System.out.println("Child# "+tempTree.getChildNumber());
              System.out.println("x " +tempTree.getChild(i).getX());
              System.out.println("y " +tempTree.getChild(i).getY());
              System.out.println("L " +tempTree.getChild(i).getLevel());
              System.out.println("isB "+tempTree.getChild(i).isBlack());
              generateWhitePickingTree(level, tempTree.getChild(i));
            }
  }
}

public int[] generateWhitePickingTreePruned(int level, GamingTree tree)
{
            myTreeAnalyzer.setGamingTree(tree);
            myTreeAnalyzer.generateMoveArray(tree);
            int[] TheMoves = myTreeAnalyzer.getMoveArray();
  if(tree.getLevel() == level && !myTreeAnalyzer.isWhiteContainedV1(tree))
  {

  }
  else
  {
    GamingTree tempTree = tree;
    int[] futureMoveArray;
    int[] moveOrder;
```

```
      myTreeAnalyzer.setGamingTree(tempTree);
      moveOrder = myTreeAnalyzer.getMoveOrder();
      System.out.println("myTreeAnalyzer.isWhiteContainedV1(tempTree) "+myTreeAnalyzer.isWhiteContainedV1(tempTree));
      if(!myTreeAnalyzer.isWhiteContainedV1(tempTree))

      {
            myTreeAnalyzer.generateFutureMoveArrayVersion1WSWP(tree);
            futureMoveArray = myTreeAnalyzer.getFutureMoves();
            tempTree.generateChildren(futureMoveArray, (moveOrder[tree.getLevel()+1] == 2));
                  for(int i = 0; i < tempTree.getChildNumber(); i++)
                  {
                        /*  System.out.println("-----------------");
                  System.out.println("i "+i);
                  System.out.println("Child# "+tempTree.getChildNumber());
                  System.out.println("x " +tempTree.getChild(i).getX());
                  System.out.println("y " +tempTree.getChild(i).getY());
                  System.out.println("L " +tempTree.getChild(i).getLevel());
                  System.out.println("isB "+tempTree.getChild(i).isBlack()); */
                  int[] a = generateWhitePickingTreePruned(level, tempTree.getChild(i));
                  if(TheMoves.length > a.length)
                  {
                        TheMoves = a;
                  }
                 }
      }
    }
    return TheMoves;
}

public void generateTreePrune(int level, GamingTree tree)
{
  if(tree.getLevel() == level)
  {

  }
  else
  {
    GamingTree tempTree = tree;
    int[] futureMoveArray;
    int[] moveOrder;
    myTreeAnalyzer.setGamingTree(tempTree);
    moveOrder = myTreeAnalyzer.getMoveOrder();
    myTreeAnalyzer.generateFutureMoveArrayVersion1WS(tree);
    futureMoveArray = myTreeAnalyzer.getFutureMoves();
    tempTree.generateChildren(futureMoveArray, (moveOrder[tree.getLevel()+1] == 2));
          for(int i = 0; i < tempTree.getChildNumber(); i++)
          {
            System.out.println("-----------------");
            System.out.println("i "+i);
            System.out.println("Child# "+tempTree.getChildNumber());
            System.out.println("x " +tempTree.getChild(i).getX());
            System.out.println("y " +tempTree.getChild(i).getY());
            System.out.println("L " +tempTree.getChild(i).getLevel());
            generateTree(level, tempTree.getChild(i));
          }
  }
}

  public static void main(String args[])
  {
     TreeMaker myTreeMaker;
     GamingTree myGamingTree;
     myGamingTree = new GamingTree(20,20,0,false);
     myTreeMaker = new TreeMaker(1,2);
     int[] b = myTreeMaker.getTreeAnalyzer().getMoveOrder();
  //  myTreeMaker.setMoveNumbers((double)1, (double)2);
    //myTreeMaker.setMoveRatio(.4);
     myTreeMaker.setGamingTree(myGamingTree);
     int a[] = myTreeMaker.generateBlackPickingTreePruned(12,myGamingTree);
     for(int i = 0; i < a.length; i=i+2)
```

```
    {
                System.out.println("isBlacksMove "+(b[a.length/2 - i/2-1] == 2));
                System.out.println("Level# "+(a.length/2 - i/2-1)+" x "+a[i]+" y "+a[i+1]);
    }
  }


}
```

```
/*
 * Created on Sep 26, 2006
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */

import java.applet.Applet;
import java.awt.Button;
import java.awt.Label;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TreeApplet extends Applet implements ActionListener
{
        public final static int Y_SIZE = 542,
```

```
                    X_SIZE = 450;
private int         UNOCCUPIED = 0, WHITE = 1, BLACK = 2;
            private Button      myGoButton;
private double      myBlackMoves,
                                                        myWhiteMoves;
private TextField   myWTextField,
                                                        myBTextField,
                                                        myNTextField;
private Label       myWLabel,
                                                        myBLabel,
                                                        myNLabel;
private int         myDepth;
private TreeMaker   myTreeMaker;
private GamingTree  myGamingTree;
private int[]       myArrayOfLife;

public void init()
{
                        initializeVars();
                        this.setSize(X_SIZE, Y_SIZE);
                        this.setBackground(Color.white);
                        this.add(myWLabel);
                        this.add(myWTextField);
                        this.add(myBLabel);
                        this.add(myBTextField);
                        this.add(myNLabel);
                        this.add(myNTextField);
                        this.add(myGoButton);
}

public void initializeVars()
{
                        myNTextField = new TextField();
            myWTextField = new TextField();
                        myBTextField = new TextField();
                        myNLabel     = new Label("Depth");
                        myWLabel     = new Label("White Moves");
                        myBLabel     = new Label("Black Moves");
                        myNLabel.setBackground(Color.yellow);
                        myWLabel.setBackground(Color.yellow);
                        myBLabel.setBackground(Color.yellow);
                        myGoButton = new Button("Input Data");
                        myNTextField.setBounds(1,1,18,3);
                        myGoButton.setBounds(20,0,20,3);
                        myGoButton.addActionListener(this);
                        myBlackMoves = 0;
                        myWhiteMoves = 0;
                        myDepth = 0;
                        myGamingTree = new GamingTree(20,20,0,false);
                        myTreeMaker = new TreeMaker(myGamingTree);
                        myArrayOfLife = new int[0];
}

            public void actionPerformed(ActionEvent e)
            {
               if(e.getSource() == myGoButton)
               {
      try
      {
         myDepth = Integer.parseInt(myNTextField.getText());
          myWhiteMoves = Double.parseDouble(myWTextField.getText());
          myBlackMoves = Double.parseDouble(myBTextField.getText());
          myTreeMaker.setMoveNumbers(myWhiteMoves, myBlackMoves);
          myArrayOfLife = myTreeMaker.generateBlackPickingTreePruned(myDepth, myGamingTree);
      }
      catch(NumberFormatException f)
      {
      }
               }
               else
```

```
                {

                }
                this.repaint();
            }

    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        this.setBackground(Color.blue);
        int l = myArrayOfLife.length;
        int a[] = myArrayOfLife;
        for(int i = 0; i < myArrayOfLife.length; i = i+2)
        {

                if(myTreeMaker.getTreeAnalyzer().getMoveOrder()[i/2] == WHITE)
                {
                        g.setColor(Color.WHITE);
                        g.fillRect(30*a[l-i-2]-375, 30*a[l-i-1]-350, 30, 30);
                        g.setColor(Color.black);
                        g.drawString(i/2+"", 30*a[l-i-2]-367,  30*a[l-i-1]-330);
                }
                else
                {
                        g.setColor(Color.BLACK);
                        g.fillRect(30*a[l-i-2]-375, 30*a[l-i-1]-350, 30, 30);
                        g.setColor(Color.WHITE);
                        g.drawString(i/2+"", 30*a[l-i-2]-367,  30*a[l-i-1]-330);

                }
        }
        g.setColor(Color.gray);
        for(int i = 0; i < 50; i++)
        {
                g.drawLine(i*30+15, 40, i*30+15, 600);
                g.drawLine(0, i*30+40, 500, i*30+40);
        }
    }

}
```