Rectangular polyomino set (1,2)-achievement games

by Edgar Lee Fisher

A Thesis
Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in Mathematics

Northern Arizona University
August 2005

Approved:

Nándor Sieben, Ph.D., Chair

Michael Falk, Ph.D.

Stephen E. Wilson, Ph.D.

Abstract

Rectangular polyomino set (1,2)-achievement games

Edgar Lee Fisher

We determine sets of rectangular polyominoes as winning or losing in the (1,2) weak achievement game. The (1,2) weak achievement game on the rectangular board is a game in which two players alternately mark unmarked squares on a rectangular board. The first player has one mark and the second player has two marks. A relationship between sets is established to simplify the process and narrow the classifications to a few important sets. All sets of size 4 or less are completely determined as winning or losing. Some infinite sets are also determined as winning or losing as they arise naturally in the theory.

Acknowledgements

I am grateful to Dr. Nándor Sieben for all of his ideas, help and support throughout this thesis. The topic was interesting enough for him to bring it to me and I enjoyed the discoveries that were made. His knowledge in the area made it possible for me to go to him whenever I had a question and get a good answer or a thought provoking question.

I thank my family for all of their support in my Graduate career through their thoughts and prayers. Just as important are the friends who cared enough for me to talk to me, even though I was too busy to talk to them.

Thanks to Dr. Judy Clarke and Dr. Dennis Nemzer for encouraging me to continue my schooling and providing moral and written support.

Contents

List of Tables

List of Figures

To Julie,
For all of your patience and experience which made it easier for me to do this. I love you.

Chapter 1

Introduction

Tic-Tac-Toe is a widely played game in which players alternate placing a mark of their own color in a previously unmarked square of a 3×3 rectangular board. The first player to get three marks in a line is the winner. Hypergraph games are generalizations of this game.

A (finite) *hypergraph* is a pair $(X, \mathcal{F})$ where X is a finite set, called the set of vertices, and $\mathcal{F}$ is an arbitrary family of subsets of X. The elements of $\mathcal{F}$ are called *hyperedges*. In a *hypergraph game*, two players alternate marking previously unmarked vertices of the hypergraph with their respective color. The first player to mark all the vertices of some element of $\mathcal{F}$ is the winner. Hypergraph games are also called strong positional games.

According to the *strategy stealing argument* the first player is guaranteed to either win or draw. To see this let us assume first that the second player has a winning strategy. To begin the game, the first player should place a random mark on the board. Ignoring his first mark, the first player uses, "steals", the second player's strategy. At this point, the first player has a winning strategy. At some point, the strategy may require the first player to mark in a cell that is already marked by himself. This could happen if the first player's previous mark is in the cell that the strategy requires. Then the first player should just make another random mark on the board. Therefore the first player always has some extra mark on the board which can only help him.

Our focus of attention is on games between *perfect* players. That is, each player knows and plays the best possible moves to win or draw in a game. If we consider less than perfect players, then a win is more likely for some players. Thus we are in essence considering the worst case scenario for the movements between players. If the first player has a winning strategy, regardless of the other player's moves, the first player will win. However, if the strategy just forces a draw, a bad play from the second player can allow the first player to win.

Since the first player can always win or draw, the second player could instead focus on keeping the first from winning. In this case, the first player is called the *maker* and the second the *breaker*. These games are called *weak hypergraph games* and will be the focus of this thesis. The maker wins in the standard sense while the breaker wins if she keeps the maker from winning. In this case, there is no draw game as either the maker achieved the goal or the breaker kept him from doing so.

Some connections between the strong and weak games are clear. If the first player has a winning strategy in the strong game, then that same strategy will guarantee a win in the weak game. However, a win for the maker in the weak game can become a draw in the strong game. Similarly, if the breaker has a winning strategy in the weak game, then she has a drawing strategy in the strong game. However, a draw in the strong game, for the second player, could be a loss in the weak game for the breaker, as in Tic-Tac-Toe.

Since both players are defensive and offensive in the strong game, there needs to be a more complicated strategy to play the games. In the weak game, the maker can focus on trying to achieve his goal and the breaker can focus on trying to stop the maker. This simplifies the ideas for the different strategies. Note, however, that this does not reduce the problem to a trivial question. Instead it focuses on different types of strategies.

For certain weak games, Erdős and Selfridge found a sufficient condition for the second player's win. Given a hypergraph $(X, \mathcal{F})$, if

$$\sum_{P \in \mathcal{F}} 2^{-|P|} < \frac{1}{2}$$

then the second player has a winning strategy, [14]. The argument is based on weight functions that use a potential function to measure the likely outcome of the game.

The weak game in which the players alternate placing a single mark on the board is called the (1,1) weak game. This is where the Erdős, Selfridge result holds. A (p, q) achievement game is similar to a (1,1) game, except that in each turn the first player places p marks and the second player places q marks. In [3], Beck extended the Erdős, Selfridge result to the (p, q) weak game as: If

$$\sum_{P \in \mathcal{F}} (1+q)^{\frac{-|P|}{p}} < \frac{1}{1+q}$$

then the second player has a winning strategy. Let d be the number of vertices in $\mathcal{F}$ and e the maximum number of edges containing two vertices of $\mathcal{F}$. Beck result [3] says that if

$$\sum_{P \in \mathcal{F}} \left(1+\frac{q}{p}\right)^{-|P|} > \frac{p^2 q^2}{(p+q)^3} de$$

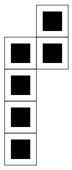


Figure 1.1: The polyomino known as Snaky.

then the first player has a winning strategy. Note that this result can give us useful results and bounds on certain items, but only when the board is finite. The results fail on the infinite board.

In this thesis, we use an infinite rectangular board. The description of the board and polyominoes (shapes made from cells on the board) corresponding to this board are discussed further in Chapter 2. By extending the size of the board, the game in essence becomes a (1,q) game for calculations. That is, we need to use Beck's result on (1,q) game as opposed to the (1,1) result for even rough estimates if we want to use the weight function. In this fashion, it is determined that Snaky (see Figure 1.1) is a 41-dimensional winner [38] although Snaky is in fact a 3-dimensional winner [39]. This invites the study of *biased* games to help understand the infinite board. That is, games where one player has more moves than the other.

Achievement games are special hypergraph games when there is a concrete set or object that is trying to be achieved. The hyperedges are then defined as the goal objects and the set of vertices, known as the *game board*, is some superset of the union of the cells of the goal objects.

The importance of weak achievement games is due in part to Ramsey Theory. In essence, Ramsey's Theorem states "For all $a, b \in \mathbb{N}$ there exists an $R(a, b)$, called the Ramsey Number, such that for all $n \geq R(a, b)$ any simple graph G on n vertices contains either a clique on a vertices or an independent set of b vertices" [18]. Then if the game board were the edges of a complete graph with n vertices, Ramsey Theory might be used to help determine results.

Consider the game where players are marking the edges of the complete graph K_n and are trying to achieve K_a for some $a < n$. The vertices of the corresponding hypergraph $(X, \mathcal{F})$ are the edges of K_n. The hyperedges are all the a element subsets of X.

If $n \geq R(a, a)$ then Ramsey's Theorem guarantees that the game is not a draw. To see this, let the players mark edges until all the edges are marked. Then by Ramsey's Theorem, there is a subgraph isomorphic to K_a marked by a single color. This means one of the players achieved K_a. So one of the players has a winning strategy and by the strategy stealing argument, this player must be the first player. In the game where $a = 3$ the first player wins if $n \geq 6 = R(3,3)$.

Other achievement games could be on a complete graph to achieve a spanning tree or a on bipartite graphs [16]. If the set of vertices were instead the elements of a group, then a goal could be to select the generator(s) of the group or subgroup [1].

Tic-Tac-Toe and other polyomino games are also achievement games. Achievement games for polyominoes were introduced by Frank Harary [22, 20, 19, 25]. The cells in the polyominoes are the vertices in a graph and the edges for the hypergraph game are the group of vertices that make the shape of the polyomino. Since all isomorphic polyominoes are also winners, only one representative edge (polyomino) needs to be given. There can be many different versions of polyomino games. The board can be different shapes: Platonic solids [6], a torus [21], hyperbolic plane or multidimensional [39]. Even if the board is on the plane, it can have different tilings of the plane, which also changes the shape of the polyominoes. The tiling could be by triangles [9, 26], rectangles [26, 36], mosaics [8, 5], tessellations [10] or hexagons [7, 37].

A consideration for the infinite board is taken into account for this thesis. Since the board is infinite the play continues until either the maker has actually achieved the goal or the breaker has proven conclusively to the maker that regardless of his moves, she can keep him from winning. Larger board size gives an advantage to the maker as the breaker must now have a strategy that does not just stall the maker but in fact stops him. Thus the win of the breaker is a matter of proof since the maker can play forever.

For this reason, the unbiased single polyomino game is difficult on the infinite board. When sets of polyominoes are considered, it becomes even more complex. To balance the game, we add bias in favor of the breaker. That is, we consider the game where the breaker gets two marks after every one mark of the maker's. In this fashion, the number of singleton winning sets is limited.

In Chapter 3 the terminology for a single polyomino to be winning or losing for the (1,1) and (1,2) achievement games is discussed. Then Chapter 4 extends these ideas to sets of polyominoes and establishes a relationship between sets to simplify the classification of each set as winning or losing. Some basic facts about sets are also discussed. Chapter 5 uses the information from Chapters 3 and 4 to classify all sets up to size 4 as winning or losing. Following this, Chapter 6 gives some basic results for size 5 sets and establishes some limitations on sets with specific attributes that might be larger. The infinite sized polyominoes and sets are discussed in Chapter 7. Finally, Chapter 8 explores the programs and algorithms that were used to establish some results and generate the graphics throughout the thesis.

Chapter 2

Preliminaries

2.1 Game Board

In this thesis we focus on a single board, the rectangular board. Other boards that could have been used are triangular, hexagonal or cubic (3-dimensional rectangular) boards [7, 8, 9, 28, 33, 37, 39].

Definition 2.1 The *rectangular game board* is $\mathbb{Z} \times \mathbb{Z}$. The *geometric representation* of the game board is the set $\{[x - \frac{1}{2}, x + \frac{1}{2}] \times [y - \frac{1}{2}, y + \frac{1}{2}] \mid (x, y) \in \mathbb{Z} \times \mathbb{Z}\}$. The elements of the board are called *cells*.

The rectangular board is based on a Euclidean tiling of the plane. We can think of it as an infinite chessboard. See Figure 2.1 for a visual representation of the game board with coordinates imposed.

Definition 2.2 Let $c_1 = (x_1, y_1)$ and $c_2 = (x_2, y_2)$ be cells of the game board. We say $c_1 \leq c_2$ if one of the following two conditions holds:

(a) $x_1 < x_2$;

(b) $x_1 = x_2$ and $y_1 \leq y_2$.

This gives the usual lexicographic ordering of the cells of the game board.

2.2 Polyominoes

In [17], Golomb defines a polyomino as "shapes made by connecting certain numbers of equal-sized squares, each joined together with at least one other square along an edge." However in the games with polyominoes, as described in [36, 37, 33, 27],

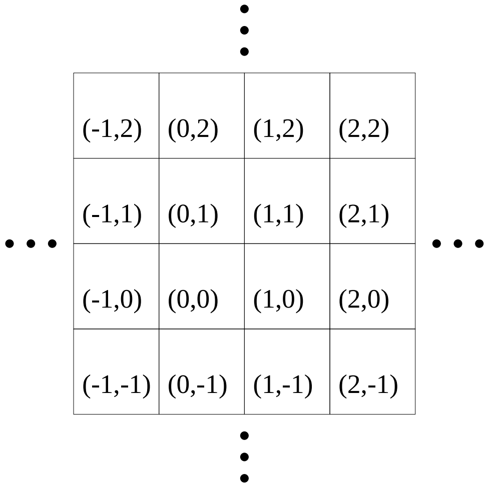


Figure 2.1: Game board with coordinates imposed.

there are some added constraints to the definition. For clarity, we will define the polyominoes in the algebraic setting.

Definition 2.3 Two cells (x_1, y_1) and (x_2, y_2) are *adjacent* if $|x_1 - x_2| + |y_1 - y_2| = 1$.

Note that this means one of the coordinates of the cells are the same and the other coordinates differ by one. An equivalent geometric description is that the cells share an edge.

Definition 2.4 A *path of cells* is a finite sequence $(c_1, c_2, \ldots, c_n)$ of cells whose consecutive cells are adjacent. We say that the path *connects* c_1 to c_n.

Definition 2.5 A subset P of the game board is *connected* if for any two cells $c, d \in P$ there is a path of cells in P that connects c to d.

Definition 2.6 An *animal* is a finite connected set of cells whose complement is also connected. A *polyomino* is the geometric representation of an animal.

This definition eliminates polyominoes that are connected only through a corner or have a hole in them. This is the standard definition [36, 39, 9, 34] of a polyomino for achievement games. In percolation theory, the term animal is also used to represent adjacent cells to simplify the physics. Thus the idea of an animal is not new to the scientific community, but in this case is being applied directly to shapes. With this restriction of the polyominoes, the set of polyominoes to consider is reduced to a manageable size. Therefore the questions that arise are not completely out of reach.

Some of the concepts in this paper were presented originally in a geometric setting. However, the algebraic representation can be more convenient. Therefore we will use both representations throughout. The polyomino is the geometric representation and an animal is the algebraic representation. Since polyominoes and animals are in a bijective correspondence, they can almost always be used interchangeably. Hence, if a statement is made about a polyomino or animal then it has a version for the other unless there is a specific difference noted.

Definition 2.7 The *size* of an animal P is the number of cells within the animal. This we denote by $|P|$.

Definition 2.8 Two animals P and Q are *equivalent* if their polyomino representations are congruent. We denote this by $P \sim Q$. Note that this is an equivalence relation.

To classify each equivalence class, we need to pick a representative in some normal position from each class. This requires a few definitions.

Definition 2.9 Let $c = (c_1, c_2, \ldots, c_m)$ and $d = (d_1, d_2, \ldots, d_n)$ be finite sequences of cells. We say $c < d$ in the *lexicographic order* if one of the following holds:

(a) $c_i = d_i$ for all $1 \leq i \leq m$ and $m < n$;

(b) There exists a $k \leq m$ such that $c_i = d_i$ for all $1 \leq i < k$ and $c_k < d_k$.

We say $c \leq d$ if $c < d$ or $c = d$.

Definition 2.10 Given two animals P and Q, with lexicographically ordered sequences of cells $(p_1, \ldots, p_m)$ and $(q_1, \ldots, q_n)$ respectively. We say that $P \leq Q$ if $(p_1, \ldots, p_m) \leq (q_1, \ldots, q_n)$ in the lexicographic order.

For the following definition we use the notation $\mathbb{W} = \{0, 1, 2, \ldots\}$ for the set of whole numbers to distinguish from the set of natural numbers $\mathbb{N} = \{1, 2, \ldots\}$.

Definition 2.11 Let P be an animal. The set $\mathcal{A} = \{Q \subseteq \mathbb{W} \times \mathbb{W} \mid Q \sim P\}$ is well ordered by $\leq$. The minimum element of $\mathcal{A}$ is the *normal position* of P. It is also called the *normalization* of P.

To determine the normal position of an animal, all the rotations, reflections and combinations of rotations and reflections of the polyomino are determined. These are then placed so that all the coordinates are non-negative. This is effectively pushing the polyominoes as "close to the axes" as possible. The lexicographic order of these placements is then established. The normal position is the placement with the

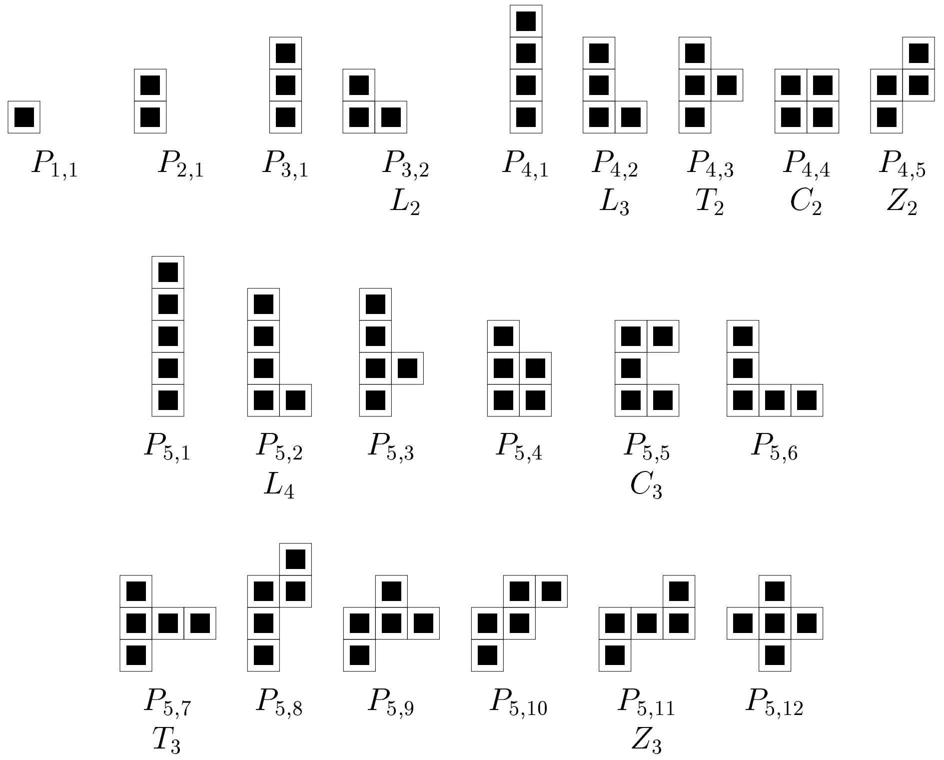


Figure 2.2: Polyominoes up to size 5 in normal position, ordered by size and then by lexicographic order.

smallest lexicographic order. Note that since the polyomino is always within the first quadrant, all of the cells will have non-negative coordinates. The algorithm for this procedure is discussed in Chapter 8.1.

In Figure 2.2 we have a representative in normal position of all the polyomino equivalence classes up to size five.

Definition 2.12 We denote the set of animals of size n in normal position by $\mathcal{P}_n = \{P_{n,i} \mid i = 1, 2, \ldots, k_n\}$, where k_n is the number of animals of size n. The indices are chosen such that $P_{n,i} < P_{n,j}$ whenever $i < j$.

Definition 2.13 We call the collection of animals with n linearly adjacent cells *skinny polyominoes*.

Note that an animal P is skinny if and only if $P \sim P_{n,1}$ for some n. Also note that for $n = 1$ or 2, $\mathcal{P}_n = \{P_{n,1}\}$.

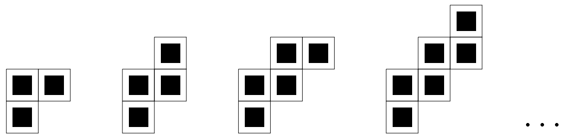


Figure 2.3: Squiggle up to size 6.

n	k_n	n	k_n
1	1	9	1248
2	1	10	4460
3	2	11	16094
4	5	12	58937
5	12	13	217117
6	35	14	805475
7	107	15	3001211
8	363		

Table 2.1: The number of non-equivalent animals up to size 15.

Definition 2.14 We call the collection of animals equivalent to the polyominoes depicted in Figure 2.3 *squiggle animals.*

The total number of non-equivalent polyominoes of a given size has no known closed formula. From [17] we get the table in Figure 2.1 which gives k_n for $n \leq 15$. Using the algorithm described in Chapter 8.1, we verified the numbers for $n \leq 7$.

Definition 2.15 An animal P is an *ancestor* of the animal Q, if there is an animal R such that $R \sim P$ and $R \subseteq Q$. This is denoted by $P \sqsubseteq Q$.

Proposition 2.16 *The ancestor relation is a partial order of the set of normalized animals.*

Proof: It is clear that $P \sqsubseteq P$ for any animal P. Transitivity is also clear.

To verify antisymmetry, let P and Q be normalized animals such that $P \sqsubseteq Q$ and $Q \sqsubseteq P$. Then there exists an R such that $R \sim P$ and $R \subseteq Q$. Also there exists an $S \sim Q$ such that $S \subseteq P$. Therefore we have that $P \sim R \subseteq Q \sim S \subseteq P$ which implies that $P \sim Q$.

Since both P and Q are normalized, this means that $P = Q$. Therefore the relation is reflexive, transitive and antisymmetric and as such is a partial order of the normalized animals. $\square$

Chapter 3

Single Animal Achievement Games

3.1 Winning Animals

To classify an animal as a winner, a strategy needs to be determined for the maker to follow. This strategy must enable the maker to achieve the animal regardless of the breaker's moves, even if the breaker knows the maker's strategy. One way to describe a winning strategy is to consider situations, defined below. This section focuses on the (1,k)-achievement game. That is, the breaker marks k cells after every mark of the maker.

Definition 3.1 A *situation* s is a pair (C_s, N_s) where the *core* C_s and the *neighborhood* N_s are sets of cells such that $C_s \cap N_s = \emptyset$.

A situation captures the essence of the game board after the maker's move. The core contains the maker's marks while the neighborhood is some set of unmarked cells. This neighborhood contains all the future moves of the maker, thus, the cells are crucial to the strategy of the maker. See Figure 3.2 for an illustration of some situations. The core is indicated by dark cells, while the neighborhood consists of all cells with letters in them.

Capital letters denote cells that could be the next mark for the maker. Lowercase versions of a capital letter identify cells that must be vacant in order for the maker's mark on that capital letter to be strategic. Thus the choice of moves for the maker is limited by the breaker's moves within the neighborhood.

Once the maker places a mark, he has established a new situation. The core of this new situation is some subset of the previous core along with the most recent mark. The neighborhood of this new situation is the set of cells containing the lowercase version of the capital letter marked by the maker.

The cells outside of the neighborhood are not displayed in a picture of a situation as they do not affect the playability of that situation. Each mark the breaker places outside the neighborhood, gives the maker more freedom for his next move.

Definition 3.2 Let (C, N) be a situation. If $c \in C$, then $(C \backslash \{c\}, N \cup \{c\})$ is called a *deletion* of the situation.

Notice that $(C \backslash \{c\}) \cap (N \cup \{c\}) = \emptyset$ so a deletion of a situation is itself a situation.

Definition 3.3 Let S be a set of situations. If $C = \bigcup_{s \in S} C_s$ and $N = \bigcup_{s \in S} N_s$ are disjoint then the situation (C, N) is called the *join* of S. A set K of cells is a *k-killer set* for S if $|K| \leq k$ and for all $s \in S$, $K \cap N_s \neq \emptyset$. If there is no k-killer set for S, then we say the join of S is *k-good.*

In essence, a k-killer set is a set of cells the breaker intends to mark to prevent the maker from attaining any future situations.

Definition 3.4 A *winning position sequence* for an animal P in the (1,k)-achievement game is a finite sequence $(s_n, s_{n-1}, \ldots, s_1, s_0)$ of situations with the following criteria:

(a) C_{s_0} is the goal animal, P;

(b) For all i, the situation s_i is a k-good join of situations that are equivalent to deletions of some situations from $\{s_0, s_1, \ldots, s_{i-1}\}$;

(c) $|C_{s_n}| = 1$.

The deletions and joins are created from situations that represent a future state of the game board. We start with the animals we want to achieve and create deletions and joins until we have k-good joins. Eventually we want a situation with a singleton core. This singleton core is the first mark of the maker.

See Figure 3.2 for an example of a winning position sequence. Figure 3.1 shows the details to create this winning position sequence. It shows deletions, the situations from which these deletions originated and the joins for which they will be used. The joins of these deletions are derived by overlapping their cores. Since we are playing the (1,1) game, each join is a 1-good join.

Consider situation s_1 in Figure 3.2. This is a 1-good join of two deletions of s_0 because there is no singleton set of cells that intersects the neighborhoods of both deletions. This means that there is no single cell the breaker can mark to ruin both of the desired moves of the maker. We could easily create up to a 4-good join with four deletions of s_0. However, this is unnecessary since the breaker only has one mark in the (1,1) game. Using a k-good join for $k > 1$ creates a situation with a larger

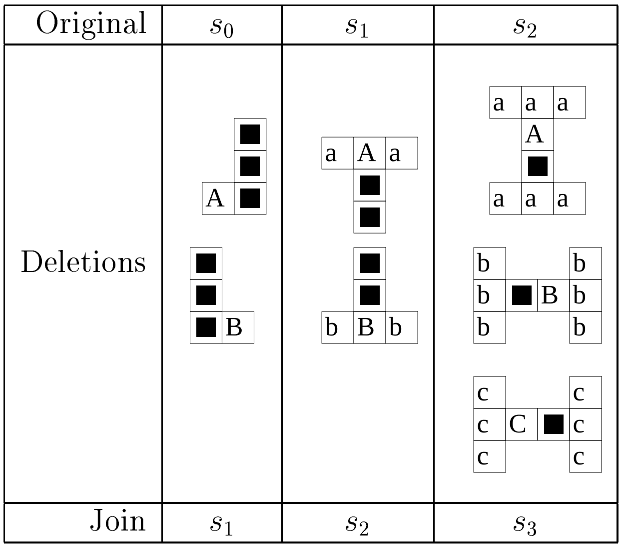


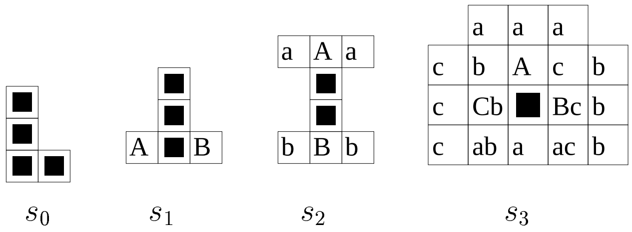
Figure 3.1: Deletions of situations which have been used to create 1-good joins in Figure 3.2.

neighborhood that is harder to achieve. Future 1-good joins based on deletions of these more complicated situations would be harder or impossible to create.

Now consider situation s_3. We can see cells that have multiple letters in them. This is the first situation in this winning position sequence in which this occurs. Suppose we only join the first two of the deletions with letters A and B in their neighborhoods. This join is not 1-good since the cell containing "ab" is in the intersection of the neighborhoods of both deletions. That is, the breaker can mark this cell and prevent the maker from achieving s_2. We cannot create a 1-good join from any other two deletions since there is at least one cell that ruins any two of the three deletions. Therefore a third deletion is needed to create a 1-good join. There is no need to add further deletions to the situation s_3 since all we need is a 1-good join in the (1,1) game.

In this thesis, the winning position sequences are represented with geometric illustrations of situations. Along with the situations are a table and a flowchart. The table shows possible future situations based on the maker's mark. Each row in the table lists different situations achievable from a situation s. If all of a particular letter are free from a breaker's mark, then the maker can mark the cell with the upper case letter and the situation with that letter in the table has been achieved. The flowchart shows the possible paths of the game as it is played.

Let us consider a game played using this strategy for the (1,1) game. We shall play the maker, starting with situation s_3, after a single mark. Assume that the breaker



s	A	B	C
s_1	s_0	s_0	
s_2	s_1	s_1	
s_3	s_2	s_2	s_2

$$s_3 \longrightarrow s_2 \longrightarrow s_1 \longrightarrow s_0$$

Figure 3.2: A winning position sequence for the maker in the (1,1)-achievement game. Notice that any situation s_i has a row in the table. For example, in s_2 we can achieve s_1 by marking the capital letters A or B.

marks in the cell with Cb. Then the situations corresponding to B and C are not attainable. Therefore we mark in the cell containing A. In the table from Figure 3.2, this corresponds to s_2, seen in Figure 3.1 in the upper right corner. So we have now achieved s_2 and it is the breaker's move again. Assume the breaker marks in the cell containing A. Then the situation corresponding to A is not attainable so we mark in the cell containing B. From the table in Figure 3.2 we have achieved s_1. For the breaker's final move, assume she marks the cell containing B. Then we mark the cell containing A and have achieved $s_0 = P_{4,2}$, the goal animal.

Proposition 3.5 *An ancestor of a winning animal is a winner.*

Proof: Let P and Q be animals such that $P \sqsubseteq Q$ and Q is a winning animal. Then there exists a winning strategy for Q. Using this same strategy the maker can achieve P at the same time or before he achieves Q. $\square$

3.2 Losing Animals

For an animal to be a loser, the breaker must have a strategy to keep the maker from achieving the target animal. The most frequently used tool to define this strategy is a paving.

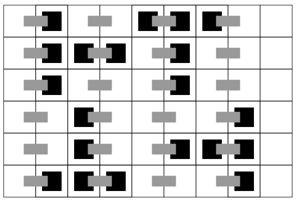


Figure 3.3: $P_{5,7}$ defeated by $\mathsf{SP_A}$

Definition 3.6 A k-paving is a symmetric relation on $\mathbb{Z} \times \mathbb{Z}$ in which no cell in the board is related to itself and each cell is related to at most k other cells. Two cells that are related are called a *pair*.

Remark 3.7 A *domino* is a pair of adjacent cells. A paving is called a *domino paving* if all of its pairs are dominoes. We use the term *single paving* for a 1-paving and *double paving* for a 2-paving. See Figures 3.5 and 3.7 for illustrations.

Definition 3.8 A *fundamental region* F of a k-paving is a pair (V, S) where $V = \{u, v\}$ is a set of two integer vectors called the *fundamental vectors* and S is a set of pairs, the *fundamental set*. If p is a pair of the paving, then p is in the orbit of an element of S through a translation by integer linear combinations of the vectors in V. The group generated by V acts on S and propagates the paving over $\mathbb{Z} \times \mathbb{Z}$.

The fundamental region captures the idea of the paving in two vectors and a set of pairs. These pairs are copied across the plane to create the paving for an infinite board. Assume that $\hat{p}$ is a pair in the paving, then there exists a $p \in S$ such that $\hat{p} = mv_1 + nv_2 + p$ where $m, n \in \mathbb{Z}$.

In the illustrations for a k-paving, see Figures 3.5, 3.7, V is the two vectors as shown by the arrows and S is the set of dark pairs. The designation SP_α is a single paving while DP_α is a double paving, where α is some letter. Each illustration represents four copies of the fundamental region. The dark pairs are the identity copy and the light pairs are the copies translated by u, v and $u + v$. For clarity and simplicity, we have tried to find the smallest fundamental region to represent each paving.

Definition 3.9 A k-paving *kills* an animal P if for every animal R such that $R \sim P$ there is a pair from the paving in R. An animal P is said to be *immune* to a k-paving if it is not killed by the k-paving. In Figure 3.3 we see single paving $\mathsf{SP_A}$ killing $P_{5,7}$.

Definition 3.10 The *strategy based on a paving*, which is a strategy for the breaker, is to mark in all cells that are paired with the cell that the maker marked. If fewer

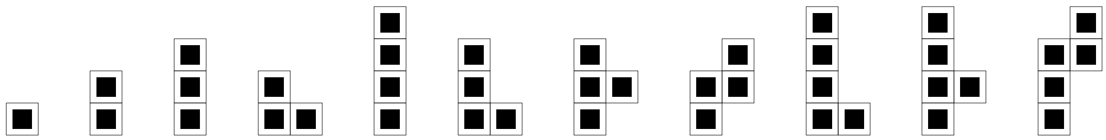


Figure 3.4: Winners for the (1,1) (weak) achievement game.

than k unmarked cells are paired with the maker's move, the breaker should mark all of these and place her remaining marks in any unmarked cell. Note this extra mark will always be in favor of the breaker.

Theorem 3.11 *If a k-paving A kills an animal P, then the strategy based on A will keep the maker from achieving P.*

Proof: Suppose the maker achieved P. Since A kills P, there is a pair within P, call them c_i and c_j. Without loss of generality, let us assume that c_i was marked before c_j. Then when the maker marked c_i, the breaker marked c_j. Thus the maker could not have achieved P. □

Proposition 3.12 *A descendant of a loser is a loser.*

Proof: This is the contrapositive of Proposition 3.5. □

3.2.1 (1,1)-Achievement Game

For the (1,1)-achievement game, all but one question has been answered about whether a given animal is a winner or a loser [27]. The known winners are in Figure 3.4. The rest of the polyominoes are losers except possibly for Snaky, $P_{6,11}$, see Figure 1.1. It is not known if Snaky is a winner or a loser, see [25, 32, 31] for further results.

Figure 3.5 has some examples of single pavings. Some strategies for the breaker for the (1,1) game are defined from these pavings. Note that each of these pavings is a domino paving.

3.2.2 (1,2)-Achievement Game

For the (1,2)-achievement game, the breaker gets two marks and the paving should reflect this.

A useful way to create a double paving is to combine two single pavings. This is discernible in $\mathsf{DP_A}$. See Figure 3.6 for an example of a double paving defeating a polyomino.

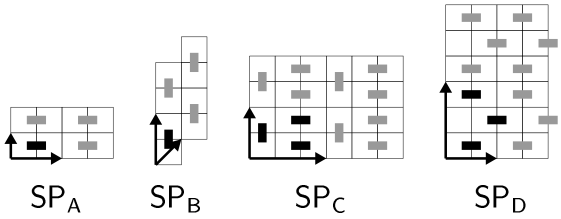


Figure 3.5: Pavings for the (1,1)-achievement game. The dark pairs form S while the lighter pairs are copies by translation through u, v and $u + v$. The arrows are the vectors in V.

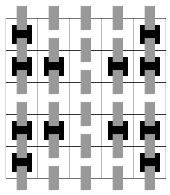


Figure 3.6: $P_{3,2}$ defeated by $\mathsf{DP_A}$. Note that each cell here in $\mathsf{DP_A}$ is related to two other cells while those in $\mathsf{SP_A}$ are related to only one.

Proposition 3.13 *All animals $P_{n,i}$, for $n \geq 3$, are losers in the $(1,2)$-achievement game.*

Proof: $P_{3,1}$ and $P_{3,2}$ are losers [36]. Every animal $P_{n,i}$ for $n > 3$ is a descendant of either $P_{3,1}$ or $P_{3,2}$. Therefore they are all losers by Proposition 3.12. $\square$

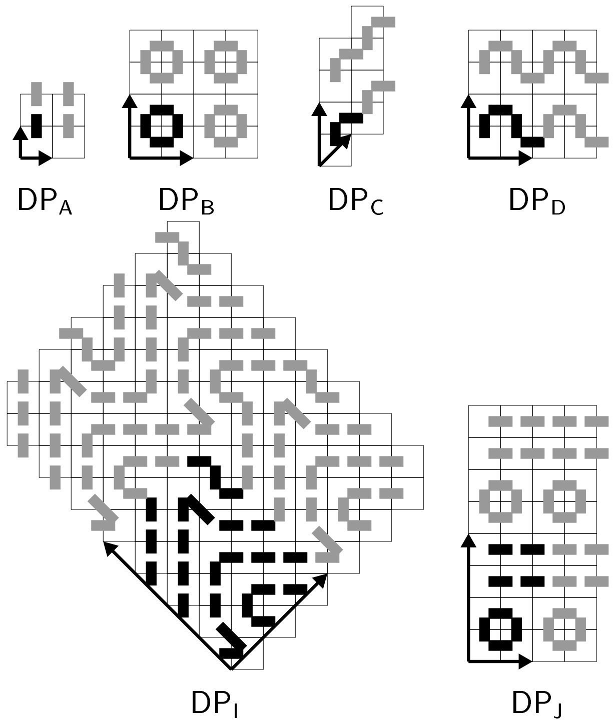


Figure 3.7: Some examples of double pavings. The arrows are the elements in V while the dark pairs form S. The light pairs are the copies of S through u, v and $u + v$.

Chapter 4

Set Polyomino Games

4.1 Set Games

Set games are polyomino achievement games in which a set of polyominoes becomes the goal of the game.

Definition 4.1 A set of animals is a *winning set* if the maker can always achieve at least one of the animals in the set. A set is a *losing set* if it is not a winning set.

The following are reformulations of definitions for single animals.

Definition 4.2 A *winning strategy* for a set M of animals, with $|M| = j$ in the (1,k)-achievement game is a finite sequence $(s_n, s_{n-1}, \ldots, s_1, s_0, s_{-1}, \ldots, s_{1-j})$ of situations with the following criteria:

(a) C_{s_i} is a goal animal for $i = 0, -1, \ldots, 1-j$;

(b) For all $i > 0$, the situation s_i is a k-good join of situations that are equivalent to deletions of situations $s_{1-j}, s_{2-j}, \ldots, s_{i-1}$;

(c) $|C_{s_n}| = 1$.

Definition 4.3 A k-paving *kills* a set if every animal in the set is killed by the k-paving.

If a set is a winning set then regardless of the breaker's moves, the maker is guaranteed to be able to mark cells until one of the animals in the set has been achieved.

If a set is a losing set then the breaker can keep the maker from achieving any of the animals in the set. The most common way to determine that a set is losing is to find a k-paving that defeats all the animals in the set.

We refer to a set as classified if it is determined as winning or losing.

4.2 Partial Order

Now we establish a relationship between sets of animals. This relationship enables us to simplify the process of finding all winning sets. To make a set easier to achieve, we can replace a member animal by an ancestor or add more members to the set. This motivates the following definition which has been adapted from [9] where it was called *at least* and *at most*.

Definition 4.4 If $\mathcal{F} = \{P_1, P_2, \ldots, P_m\}$ and $\mathcal{G} = \{Q_1, Q_2, \ldots, Q_n\}$ are sets of polyominoes, then $\mathcal{F}$ is *simpler* than $\mathcal{G}$ if for all $Q \in \mathcal{G}$ there exists a $P \in \mathcal{F}$ such that $P \sqsubseteq Q$. We use the notation $\mathcal{F} \preceq \mathcal{G}$.

The following proposition is the main reason for Definition 4.4.

Proposition 4.5 *Let $\mathcal{F} \preceq \mathcal{G}$. If $\mathcal{G}$ is a winner, then so is $\mathcal{F}$. If $\mathcal{F}$ is a loser then so is $\mathcal{G}$.*

Proof: Let $\mathcal{F} \preceq \mathcal{G}$ be families of polyominoes and suppose that $\mathcal{G}$ is a winner. Then the maker is able to mark one of the animals $Q \in \mathcal{G}$ after finitely many moves. By definition, there exists a polyomino $P \in \mathcal{F}$ such that $P \sqsubseteq Q$. Thus P is marked at the same time or earlier than Q and therefore $\mathcal{F}$ is a winning family.

The second part of the proposition is the contrapositive of the first part. □

Although the second part of the proposition is merely the contrapositive of the first it is actually the most frequently used portion of the proposition. It is easier to prove something is a loser than a winner.

Definition 4.6 A *family* of animals is a non-empty set of animals such that no member is an ancestor of any other member.

Definition 4.7 Let $\mathcal{M}$ be a set of animals. A set $\mathcal{L}(\mathcal{M})$ is the *legalization* of $\mathcal{M}$ if $\mathcal{L}(\mathcal{M})$ consists of the minimal animals of $\mathcal{M}$ in the ordering $\sqsubseteq$.

Proposition 4.8 *The legalization $\mathcal{L}(\mathcal{M})$ of a set $\mathcal{M}$ is a family.*

Proof: Suppose $\mathcal{L}(\mathcal{M})$ is not a family. Then there exist distinct animals P and Q in $\mathcal{L}(\mathcal{M})$ such that $P \sqsubseteq Q$. However, this means that Q is not minimal, which is a contradiction. □

The notion of legalization relies heavily on the finiteness of polyominoes. If a polyomino were infinite, then there could be many problems, one of which is that another infinite polyomino is an ancestor and a descendant, see Figure 4.1 for an example.

Figure 4.1: A set of infinite polyominoes that does not have a legalization.

Remark 4.9 If $\mathcal{M}$ and $\mathcal{N}$ are sets of polyominoes such that $\mathcal{N} \subseteq \mathcal{M}$, then $\mathcal{M} \preceq \mathcal{N}$. That is, a set is simpler than any of its subsets.

Proposition 4.10 *Let $\mathcal{L}$ be the legalization of a set of polyominoes $\mathcal{M}$. Then $\mathcal{L}$ is a winner if and only if $\mathcal{M}$ is a winner.*

Proof: First assume that $\mathcal{L}$ is a winner. Since $\mathcal{L} \subseteq \mathcal{M}$ then $\mathcal{M} \preceq \mathcal{L}$, by Remark 4.9, and therefore $\mathcal{M}$ is a winner by Proposition 4.5.

Next assume that $\mathcal{M}$ is a winner. By the definition of $\mathcal{L}$, for all $Q \in \mathcal{M}$ there exists a $P \in \mathcal{L}$ such that $P \sqsubseteq Q$. Therefore $\mathcal{L} \preceq \mathcal{M}$ and so $\mathcal{L}$ is a winner by Proposition 4.5. $\square$

Proposition 4.11 *The relation $\preceq$ is a partial ordering of families of animals in standard position.*

Proof: It is clear that $\preceq$ is reflexive and transitive.

To verify antisymmetry, let $\mathcal{F}$ and $\mathcal{G}$ be families such that $\mathcal{F} \preceq \mathcal{G}$ and $\mathcal{G} \preceq \mathcal{F}$. Now if $P \in \mathcal{F}$ then there exists a $Q \in \mathcal{G}$ such that $Q \sqsubseteq P$ since $\mathcal{G} \preceq \mathcal{F}$. Since $\mathcal{F} \preceq \mathcal{G}$ then there exists a $\tilde{P} \in \mathcal{F}$ such that $\tilde{P} \sqsubseteq Q$. Therefore we have that $\tilde{P} \sqsubseteq Q \sqsubseteq P$. This means that $\tilde{P} \sqsubseteq P$ and since $\mathcal{F}$ is a family it follows that $\tilde{P} = P$. Thus $P = Q \in \mathcal{G}$ so $\mathcal{F} \subseteq \mathcal{G}$. Similarly, $\mathcal{G} \subseteq \mathcal{F}$ and so $\mathcal{F} = \mathcal{G}$. This means the relation is reflexive, transitive and antisymmetric and is therefore a partial order. $\square$

Remark 4.12 Notice that $\preceq$ is not a partial ordering of sets, even if the animals are in standard position. A counterexample exists in the proof of 4.10 where it is shown that $\mathcal{M} \preceq \mathcal{L}$ and $\mathcal{L} \preceq \mathcal{M}$. However, if $\mathcal{M}$ has a polyomino that is a descendant of another member polyomino then $\mathcal{L} \neq \mathcal{M}$.

4.3 General Results

Proposition 4.13 *If a family $\mathcal{F}$ is a winner, then $P_{n,1} \in \mathcal{F}$, for some n.*

Proof: If $P_{n,1} \notin \mathcal{F}$ for any $n \in \mathbb{N}$ then $\{P_{3,2}\} \preceq \mathcal{F}$. $P_{3,2}$ is a loser by the strategy based on $\mathsf{DP_A}$, see Figure 3.7 for the visualization. Thus $\mathcal{F}$ is a loser by Proposition 4.5. $\square$

Proposition 4.14 *A family of size four or greater does not have any polyominoes of size* 3 *or less.*

Proof: If $P_{1,1}$ or $P_{2,1}$ were in $\mathcal{F}$, then $\mathcal{F}$ would consist of only that animal.

Now, there are two polyominoes of size three. If both of these polyominoes were in a family $\mathcal{F}$, then the size of the family would be two. Let us therefore assume that there is only one polyomino of size three in $\mathcal{F}$.

Let us first assume that $P_{3,1} \in \mathcal{F}$. Since $\mathcal{F}$ is a family, then no ancestors of $P_{3,1}$ are in $\mathcal{F}$. The only polyominoes that are not ancestors of $P_{3,1}$ are n-Squiggle and $P_{4,4}$. If n-Squiggle $\in \mathcal{F}$ for some n, then no other n-Squiggle is in $\mathcal{F}$. Therefore if $P_{3,1} \in \mathcal{F}$, then the size of $\mathcal{F}$ is at most three.

Now assume that $P_{3,2} \in \mathcal{F}$. Since $\mathcal{F}$ is a family, no ancestors of $P_{3,2} \in \mathcal{F}$. The only polyominoes that are not ancestors of $P_{3,2}$ are skinny. Therefore if $P_{3,2} \in \mathcal{F}$, then the size of $\mathcal{F}$ is at most two. $\square$

Remark 4.15 The previous proposition gives us that if $\mathcal{F}$ is a family, then if $P_{3,1} \in \mathcal{F}$ then $|\mathcal{F}| = 3$ and if $P_{3,2} \in \mathcal{F}$ then $|\mathcal{F}| = 2$.

Chapter 5

Classification of Families

For the remainder of the thesis we will only be considering the (1,2) game. We will classify all families of size n, for $1 \leq n \leq 4$. In each section we will describe a characterizing set of families $\mathcal{C}_n$ containing winners which are less simple than any size n winner and losers which are simpler than any size n losers. Thus we will show that for any family $\mathcal{F}$ of size $n, \mathcal{F}$ is either simpler than a winner from $\mathcal{C}_n$ or a losing family from $\mathcal{C}_n$ is simpler than $\mathcal{F}$.

The characterizing set of families will be listed in a table with names and then as polyominoes to help understand why the families are important. For the families of size n, the winning families are all size n, while the losing families are at most size n. Note that these might not necessarily be the simplest families that classify the size n families. Rather they are the families that are easiest to compare to the size n families.

5.1 Size One Families

The characterizing set of families for size one families is listed in Table 5.1.

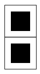
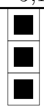
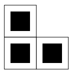
$\mathcal{W}_1$	$\mathcal{L}_1$	
$\mathcal{W}_{1,1}$	$\mathcal{L}_{1,1}$	$\mathcal{L}_{1,2}$
$P_{2,1}$	$P_{3,1}$	$P_{3,2}$
		

Table 5.1: Characterizing set $\mathcal{C}_1$ of winners and losers for size one families.

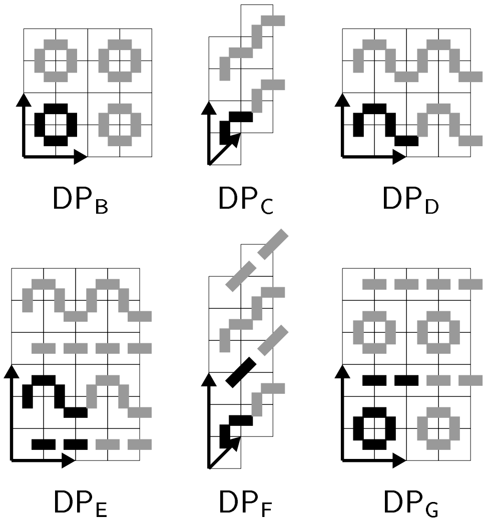


Figure 5.1: Double pavings that are used to classify certain families as losers in this chapter

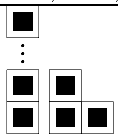
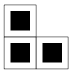
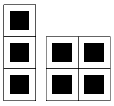
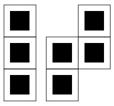
$\mathcal{W}_2$	$\mathcal{L}_2$		
$\mathcal{W}_{2,n}$	$\mathcal{L}_{2,1}$	$\mathcal{L}_{2,2}$	$\mathcal{L}_{2,3}$
$P_{n+2,1}, P_{3,2}$	$P_{3,2}$	$P_{3,1}, P_{4,4}$	$P_{3,1}, P_{4,5}$
			

Table 5.2: Characterizing set $\mathcal{C}_2$ of winners and losers for size two families.

Size one families are completely determined for the (1,2)-achievement game in [36]. It states that the only winning animals are $P_{1,1}$ and $P_{2,1}$. These are both simpler than $\mathcal{W}_{1,1}$. Now $\mathcal{L}_1$ is the set of size three polyominoes. This means that anything of size three or larger has a member of $\mathcal{L}_1$ simpler than it. Therefore they are all losers by Proposition 4.5. Hence $\mathcal{W}_1$ and $\mathcal{L}_1$ classify all size one families.

5.2 Size Two Families

The characterizing set of families for size two families is listed in Table 5.2. Note that $\mathcal{L}_{2,1} = \mathcal{L}_{1,2}$ and so is a loser.

In Table 5.2 there are infinitely many winning families, one for each $n \geq 3$. This topic is discussed and explored further in Chapter 7.

Proposition 5.1 *The family* $\mathcal{W}_{2,n} = \{P_{n+2,1}, P_{3,2}\}$ *is a winner for all* n*. The families* $\mathcal{L}_{2,2} = \{P_{3,1}, P_{4,4}\}$ *and* $\mathcal{L}_{2,3} = \{P_{3,1}, P_{4,5}\}$ *are losing families.*

Proof: $\mathcal{W}_{2,n}$ is winning by the strategy in Figure 5.2. The families $\mathcal{L}_{2,2}$ and $\mathcal{L}_{2,3}$ are defeated by $\mathsf{DP_D}$ and $\mathsf{DP_B}$ respectively, see Figure 5.1. □

Proposition 5.2 *Every family of size two is completely determined as winning or losing by comparison to* $\mathcal{C}_2$*.*

Proof: Let $\mathcal{F}$ be a family of size two. First note that if $P_{n,1} \notin \mathcal{F}$ for some $n \in \mathbb{N}$, then $\mathcal{F}$ is a loser by Proposition 4.13. Therefore let us assume that $\mathcal{F} = \{P_{n,1}, Q\}$ for some $n \geq 3$.

We will consider cases based on the size of the animals in $\mathcal{F}$.

Case 1: $|Q| \leq 4$.

Then $Q \in \{P_{3,2}, P_{4,2}, P_{4,3}, P_{4,4}, P_{4,5}\}$ since the animals in $\{P_{1,1}, P_{2,1}, P_{3,1}, P_{4,1}\}$ are related to $P_{n,1}$ (see Figure B.1).

Case 1.a: If $Q = P_{3,2}$ then $\mathcal{F} = \mathcal{W}_{2,n-2}$.

Case 1.b: If $Q \in \{P_{4,2}, P_{4,3}\}$ then $\mathcal{L}_{2,2}, \mathcal{L}_{2,3} \preceq \mathcal{F}$.

Case 1.c: If $Q = P_{4,4}$ then $\mathcal{L}_{2,2} \preceq \mathcal{F}$.

Case 1.d: If $Q = P_{4,5}$ then $\mathcal{L}_{2,3} \preceq \mathcal{F}$.

Case 2: $|Q| > 4$.

Then $Q \neq P_{k,1}$ for $k \geq 5$. So $P_{4,i} \sqsubseteq Q$ for some $i \geq 2$. Hence $\{P_{n,1}, P_{4,i}\} \preceq \mathcal{F}$ and so $\mathcal{F}$ is a loser by Case 1.

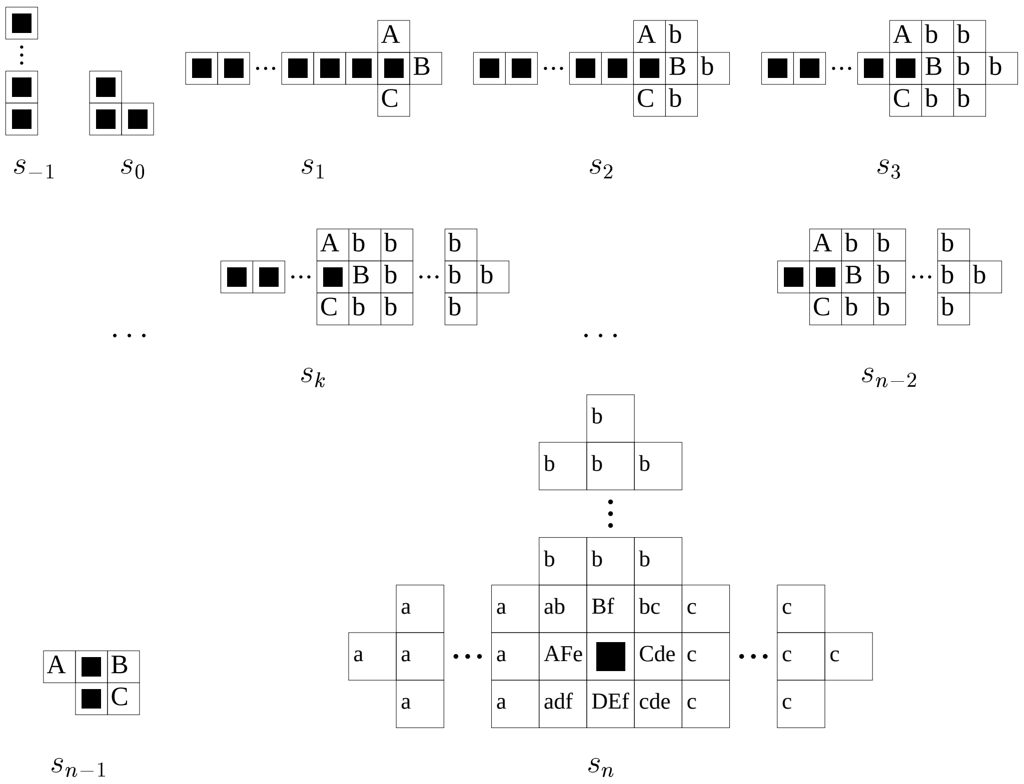
□

5.3 Size Three Families

The characterizing set of families for size three families is listed in Table 5.3. In the table, $\mathcal{L}_{3,1} = \mathcal{L}_{1,2}$ and $\mathcal{L}_{3,2} = \mathcal{L}_{2,3}$ and so they are both losers.

Proposition 5.3 $\mathcal{W}_{3,1} = \{P_{3,1}, P_{4,4}, P_{4,5}\}$ *is a winning family.*

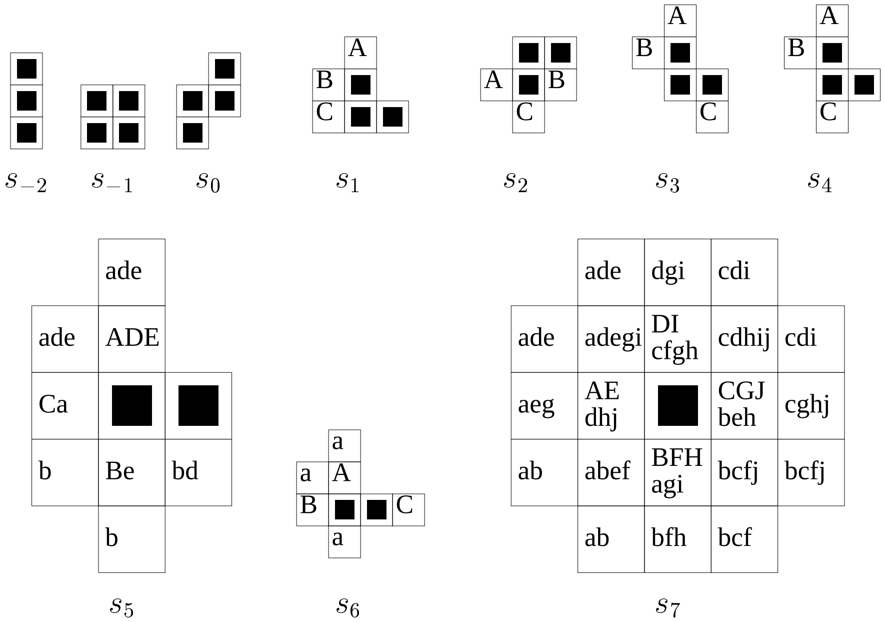
Proof: The winning strategy in Figure 5.3 shows that $\mathcal{W}_{3,1}$ is a winning family. □



	A	B	C	D	E	F
s_1	s_0	s_{-1}	s_0			
s_2	s_0	s_1	s_0			
$\vdots$	$\vdots$	$\vdots$	$\vdots$			
s_k	s_0	s_{k-1}	s_0			
$\vdots$	$\vdots$	$\vdots$	$\vdots$			
s_{n-2}	s_0	s_{n-3}	s_0			
s_{n-1}	s_0	s_0	s_0			
s_n	s_{n-2}	s_{n-2}	s_{n-2}	s_{n-1}	s_{n-1}	s_{n-1}

$$\begin{tikzcd}
s_n \arrow[r] \arrow[dr] & s_{n-2} \arrow[r] \arrow[drrrrrr, bend right] & \cdots \arrow[r] & s_k \arrow[r] \arrow[drrrr, bend right] & \cdots \arrow[r] & s_2 \arrow[r] \arrow[drr, bend right] & s_1 \arrow[r] \arrow[dr] & s_{-1} \\
 & s_{n-1} \arrow[rrrrrr] & & & & & & s_0
\end{tikzcd}$$

Figure 5.2: Winning strategy for the family $\mathcal{F} = \{P_{n,1}, P_{3,2}\}$.



	A	B	C	D	E	F	G	H	I	J
s_1	s_{-2}	s_{-1}	s_{-2}							
s_2	s_{-1}	s_0	s_{-2}							
s_3	s_{-2}	s_{-1}	s_{-1}							
s_4	s_{-2}	s_{-1}	s_{-2}							
s_5	s_4	s_{-2}	s_{-2}							
s_6	s_1	s_2	s_{-2}	s_3	s_4					
s_7	s_5	s_5	s_5	s_5	s_6	s_6	s_6	s_6	s_6	s_6

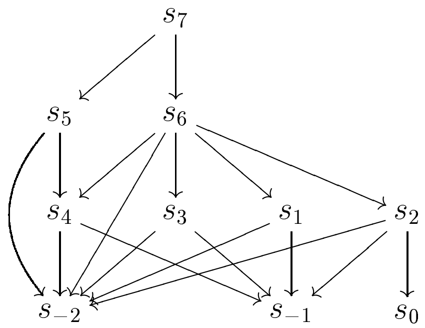


Figure 5.3: Winning strategy for $\mathcal{W}_{3,1} = \{P_{3,1}, P_{4,4}, P_{4,5}\}$

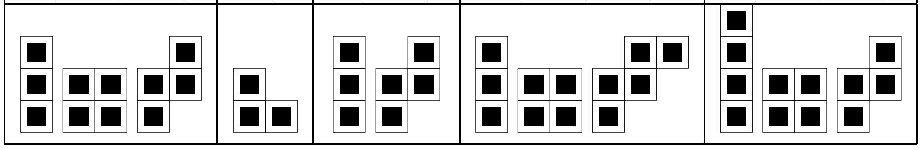
$\mathcal{W}_3$	$\mathcal{L}_3$			
$\mathcal{W}_{3,1}$	$\mathcal{L}_{3,1}$	$\mathcal{L}_{3,2}$	$\mathcal{L}_{3,3}$	$\mathcal{L}_{3,4}$
$P_{3,1}, P_{4,4}, P_{4,5}$	$P_{3,2}$	$P_{3,1}, P_{4,5}$	$P_{3,1}, P_{4,4}, P_{5,10}$	$P_{4,1}, P_{4,4}, P_{4,5}$
				

Table 5.3: Characterizing set $\mathcal{C}_3$ of winners and losers for size three families.

Proposition 5.4 *The families $\mathcal{L}_{3,3} = \{P_{3,1}, P_{4,4}, P_{5,10}\}$ and $\mathcal{L}_{3,4} = \{P_{4,1}, P_{4,4}, P_{4,5}\}$ are losing families.*

Proof: $\mathcal{L}_{3,3}$ and $\mathcal{L}_{3,4}$ are defeated by $\mathsf{DP_C}$ and $\mathsf{DP_E}$ respectively, see Figure 5.1. Therefore they are both losing families.
$\square$

Proposition 5.5 *Every family of size three is completely determined as winning or losing by comparison to $\mathcal{C}_3$.*

Proof: Let $\mathcal{F}$ be a family of size three. If $P_{n,1} \notin \mathcal{F}$ for some $n \in \mathbb{N}$, then $\mathcal{L}_{3,1} \preceq \mathcal{F}$. So let us assume that $\mathcal{F} = \{P_{n,1}, Q, R\}$ for some $n \geq 3$.

We will consider cases based on the sizes of Q and R. Note that by Remark 4.15, Q and $R \neq P_{3,2}$.

Case 1: $|Q| = 4$ and $|R| = 4$

Then $Q, R \subset \{P_{4,2}, P_{4,3}, P_{4,4}, P_{4,5}\}$ since the animals in $\{P_{1,1}, P_{2,1}, P_{3,1}, P_{4,1}\}$ are related to $P_{n,1}$ (see Figure B.1).

Case 1.a: If $Q = P_{4,2}$ and $R = P_{4,3}$ then $\mathcal{L}_{3,2} \preceq \{P_{3,1}\} \preceq \mathcal{F}$.

Case 1.b: If $Q \in \{P_{4,2}, P_{4,3}\}$ and $R = P_{4,4}$ then $\mathcal{L}_{3,3} \preceq \{P_{3,1}, P_{4,4}\} \preceq \mathcal{F}$.

Case 1.c: If $Q \in \{P_{4,2}, P_{4,3}\}$ and $R = P_{4,5}$ then $\mathcal{L}_{3,2} \preceq \mathcal{F}$.

Case 1.d: If $Q = P_{4,4}$ and $R = P_{4,5}$ then if $n = 3$ we have $\mathcal{F} = \mathcal{W}_{3,1}$. If $n \geq 4$ then $\mathcal{L}_{3,4} \preceq \mathcal{F}$.

Case 2: $|Q| \geq 4$ and $|R| \geq 5$

Since Q and R are not skinny, there is a $\mathcal{S} \subset \{P_{4,2}, P_{4,3}, P_{4,4}, P_{4,5}\}$ with $|\mathcal{S}| \leq 2$ such that $\mathcal{S} \preceq \{Q, R\}$. Then $\mathcal{E} = \mathcal{L}(\{P_{n,1}\} \cup \mathcal{S}) \preceq \{P_{n,1}\} \cup \mathcal{S} \preceq \mathcal{F}$ with $1 \leq \mathcal{E} \leq 3$.

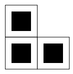
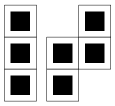
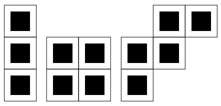
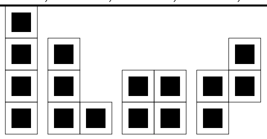
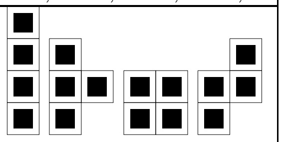
$\mathcal{L}_4$				
$\mathcal{L}_{4,1}$	$\mathcal{L}_{4,2}$	$\mathcal{L}_{4,3}$	$\mathcal{L}_{4,4}$	$\mathcal{L}_{4,5}$
$P_{3,2}$	$P_{3,1}, P_{4,5}$	$P_{3,1}, P_{4,4}, P_{5,10}$	$P_{4,1}, P_{4,2}, P_{4,4}, P_{4,5}$	$P_{4,1}, P_{4,3}, P_{4,4}, P_{4,5}$
				

Table 5.4: Characterizing set of winners and losers for size four families.

Case 2a: $|\mathcal{E}| = 1$

Then $\mathcal{L}_{3,2} \preceq \mathcal{E} \preceq \mathcal{F}$.

Case 2b: $|\mathcal{E}| = 2$

Then by Proposition 5.2 and the fact that $\mathcal{E}$ has a polyomino of size four, $\mathcal{L}_{2,1}, \mathcal{L}_{2,2}$ or $\mathcal{L}_{2,3} \preceq \mathcal{E}$. Then $\mathcal{L}_{3,1} = \mathcal{L}_{2,1}, \mathcal{L}_{3,3} = \mathcal{L}_{2,2}$ and $\mathcal{L}_{3,2} = \mathcal{L}_{2,3}$. Therefore $\mathcal{L}_{3,i} \preceq \mathcal{L}_{2,j} \preceq \mathcal{E} \preceq \mathcal{F}$ for some i and j.

Case 2c: $|\mathcal{E}| = 3$

If $\mathcal{E} \neq \mathcal{W}_{3,1}$ then $\mathcal{L}_{3,i} \preceq \mathcal{E} \preceq \mathcal{F}$ for some i by Case 1. Then let us consider when $\mathcal{E} = \mathcal{W}_{3,1}$. Then there exists a $Q' \sqsubseteq Q$ and $R' \sqsubseteq R$ such that $|Q'| = 4$ and $|R'| = 5$. From Figures B.1, B.2, B.3 we can see that either $Q' = P_{4,5}$ and $R' = P_{5,4}$ or $Q' = P_{4,4}$ and $R' \in \{P_{5,4}, P_{5,8}, P_{5,9}, P_{5,10}\}$. In the first case $\mathcal{L}_{3,2} \preceq \{P_{n,1}, P_{4,5}, P_{5,4}\} \preceq \mathcal{F}$. In the second case, one of the following occurs:

$\mathcal{L}_{3,3}, \mathcal{L}_{3,4} \preceq \{P_{n,1}, P_{4,4}, P_{5,4}\} \preceq \mathcal{F}$
$\mathcal{L}_{3,4} \preceq \{P_{n,1}, P_{4,4}, P_{5,8}\} \preceq \mathcal{F}$ ($n \geq 4$ since $P_{n,1} \preceq P_{5,8}$)
$\mathcal{L}_{3,4} \preceq \{P_{n,1}, P_{4,4}, P_{5,9}\} \preceq \mathcal{F}$ ($n \geq 4$ since $P_{n,1} \preceq P_{5,9}$)
$\mathcal{L}_{3,3} \preceq \{P_{n,1}, P_{4,4}, P_{5,10}\} \preceq \mathcal{F}$.

$\square$

5.4 Size Four Families

The characterizing set of families for size four families is listed in Table 5.4. There are no winners of size four, thus the characterizing set consists of only losers. Note that $\mathcal{L}_{4,1} = \mathcal{L}_{3,1}, \mathcal{L}_{4,2} = \mathcal{L}_{3,2}$ and $\mathcal{L}_{4,3} = \mathcal{L}_{3,3}$ and so are losers.

Proposition 5.6 *The families*

$$\mathcal{L}_{4,4} = \{P_{4,1}, P_{4,2}, P_{4,4}, P_{4,5}\}, \mathcal{L}_{4,5} = \{P_{4,1}, P_{4,3}, P_{4,4}, P_{4,5}\}$$

are losing families.

Proof: $\mathcal{L}_{4,4}$ and $\mathcal{L}_{4,5}$ are defeated by $\mathsf{DP_F}$ and $\mathsf{DP_G}$ respectively, see Figure 5.1. Therefore they are losing families.
□

Proposition 5.7 *Every family of size four is a losing family by comparison to $\mathcal{C}_4$.*

Proof: Let $\mathcal{F}$ be a family of size four. If $P_{n,1} \notin \mathcal{F}$ for some $n \in \mathbb{N}$, then $\mathcal{L}_{3,1} \preceq \mathcal{F}$. So let us assume that $\mathcal{F} = \{P_{n,1}, P, Q, R\}$ for some $n \geq 3$.

By Proposition 4.14 we can assume that $n, |P|, |Q|, |R| \geq 4$. Then there is an $\mathcal{S} \subset \{P_{4,2}, P_{4,3}, P_{4,4}, P_{4,5}\}$ with $|\mathcal{S}| \leq 3$ such that $\mathcal{S} \preceq \{P, Q, R\}$. Then $\mathcal{E} = \mathcal{L}(\{P_{n,1}\} \cup \mathcal{S}) \preceq \{P_{n,1}\} \cup \mathcal{S} \preceq \mathcal{F}$ and $1 \leq |\mathcal{E}| \leq 4$.

Case 1: $|\mathcal{E}| = 1$.

Then $\mathcal{L}_{4,2}, \mathcal{L}_{4,3} \preceq \mathcal{E} \preceq \mathcal{F}$.

Case 2: $|\mathcal{E}| = 2$.

Then one of the following holds:

$$\begin{aligned}
&\mathcal{L}_{4,2}, \mathcal{L}_{4,3}, \mathcal{L}_{4,4} \preceq \{P_{n,1}, P_{4,2}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,2}, \mathcal{L}_{4,3}, \mathcal{L}_{4,5} \preceq \{P_{n,1}, P_{4,3}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,3}, \mathcal{L}_{4,4}, \mathcal{L}_{4,5} \preceq \{P_{n,1}, P_{4,4}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,2}, \mathcal{L}_{4,4}, \mathcal{L}_{4,5} \preceq \{P_{n,1}, P_{4,5}\} = \mathcal{E} \preceq \mathcal{F}.
\end{aligned}$$

Case 3: $|\mathcal{E}| = 3$.

Then one of the following holds:

$$\begin{aligned}
&\mathcal{L}_{4,2}, \mathcal{L}_{4,3} \preceq \{P_{n,1}, P_{4,2}, P_{4,3}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,3}, \mathcal{L}_{4,4} \preceq \{P_{n,1}, P_{4,2}, P_{4,4}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,2}, \mathcal{L}_{4,4} \preceq \{P_{n,1}, P_{4,2}, P_{4,5}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,3}, \mathcal{L}_{4,5} \preceq \{P_{n,1}, P_{4,3}, P_{4,4}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,2}, \mathcal{L}_{4,5} \preceq \{P_{n,1}, P_{4,3}, P_{4,5}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,4}, \mathcal{L}_{4,5} \preceq \{P_{n,1}, P_{4,4}, P_{4,5}\} = \mathcal{E} \preceq \mathcal{F}.
\end{aligned}$$

Case 4: $|\mathcal{E}| = 4$.

Then one of the following holds:

$$\begin{aligned}
&\mathcal{L}_{4,3} \preceq \{P_{n,1}, P_{4,2}, P_{4,3}, P_{4,4}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,2} \preceq \{P_{n,1}, P_{4,2}, P_{4,3}, P_{4,5}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,4} \preceq \{P_{n,1}, P_{4,2}, P_{4,4}, P_{4,5}\} = \mathcal{E} \preceq \mathcal{F}\\
&\mathcal{L}_{4,5} \preceq \{P_{n,1}, P_{4,3}, P_{4,4}, P_{4,5}\} = \mathcal{E} \preceq \mathcal{F}.
\end{aligned}$$

□

Chapter 6

Further Results

6.1 Size Five Families

Definition 6.1 An *exterior boundary cell* of an animal is an empty cell that is adjacent to a cell in the animal. The *boundary* $\partial(P)$ (which is called the exterior boundary in the literature) is the set of boundary cells. The *perimeter* of an animal is the size $|\partial(P)|$ of the animal's boundary. For $\mathcal{P}_n$, the family of polyominoes of size n, we use the notation $p_\star(n) = \min\{|\partial(P)| \mid P \in \mathcal{P}_n\}$.

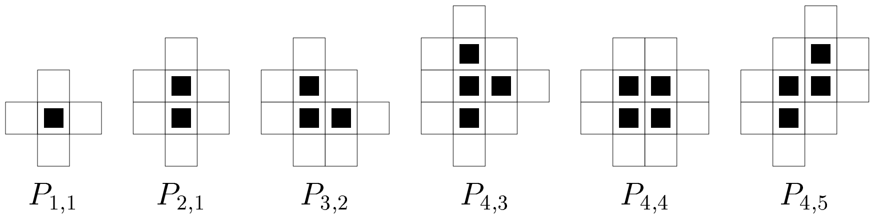


Figure 6.1: All polyominoes $P \in \mathcal{P}_n$ such that $|\partial(P)| = p_\star(n)$ for $n \leq 4$. The boundary consists of the empty cells.

Definition 6.2 A polyomino is an *economical* winner if the maker can achieve the polyomino in as many moves as the size of the polyomino.

Definition 6.3 A family is an *economical* winner if the maker can win within as many moves as the size of the largest polyomino in the family.

Proposition 6.4 *The family $\mathcal{P}_4$ is an economical winner for the $(1, 2)$ game.*

Proof: The maker's strategy is to place his mark adjacent to a previous mark of his own. Let M_k represent the animal achieved by the maker after the k^{th} move. We will show that $|\partial(M_k)|$ is greater than the number of possible marks available to the breaker. Thus a polyomino of size $k+1$ can be achieved, see Figure 6.1. Table 6.1 has the polyominoes $P \in \mathcal{P}_n$ such that $|\partial(P)| = p_\star(n)$ for $n \leq 4$, along with the number of possible marks of the breaker. From the table we can see that a size four polyomino is always achievable. □

k	$p_\star(k)$	Breaker's marks
1	4	2
2	6	4
3	7	6
4	8	8

Table 6.1: $p_\star(k)$ and the number of possible marks of the breaker in the (1,2)-achievement game for $1 \leq k \leq 4$.

Proposition 6.5 *The family* $\mathcal{F}_n = \{P_{n,1} \cup (\mathcal{P}_4 \backslash P_{4,1})\} = \{P_{n,1}, P_{4,2}, P_{4,3}, P_{4,4}, P_{4,5}\}$ *is a winning family for* $n \geq 4$.

Proof: Let the maker's strategy through the first four marks be to mark cells adjacent to a previous mark of his own. Since $\mathcal{P}_4$ is a winner, the maker the can achieve one of $P_{4,2}, P_{4,3}, P_{4,4}$ or $P_{4,5}$ and win or achieve $P_{4,1}$ in four moves. If the breaker places her marks in such a way as to leave an open space beside $P_{4,1}$ then the maker should mark in that open space to win. If there is no place to mark, then the marks look like Figure 6.2. Note that the cells with the white boxes represent the marks of the breaker. Then the maker can achieve the situation in Figure 6.3. If either A or B are not marked by the breaker, the maker can then achieve $P_{4,2}$ and thus will win. If the breaker marks in both A and B then the maker can continue to achieve the situation in Figure 6.3 and will eventually achieve n-Skinny after n moves or $P_{4,2}$ at some time when the breaker leaves an opening. □

Note that for $n \leq 3, \mathcal{F}_n$ is not a family.

6.2 General Results

These results are all for the (1,2) achievement game on the rectangular board.

Proposition 6.6 *If a family $\mathcal{G}$ consists of animals of size five or greater, then it is a losing family.*

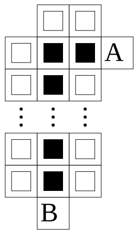


Figure 6.2: Game board position after being able to achieve only $P_{4,1}$.

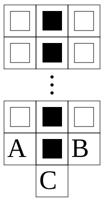


Figure 6.3: Infinite board situation for a winning strategy of $\mathcal{F}_n$.

Proof: If $\mathcal{F} = \{P_{3,1}, P_{4,5}\}$, then $\mathcal{F} \preceq \mathcal{G}$. Since $\mathcal{F}$ is a loser, by Proposition 5.1, $\mathcal{G}$ is a loser by Proposition 4.5. □

This limitation at size five helps reduce the number of possibly different winning families. Now the only cases to consider are families that have some of the animals with size less that five. We will study cases for families that have a specific number of animals with size less than five.

Proposition 6.7 *If $\mathcal{F}$ is a family of size greater than one that has only one polyomino P such that $|P| < 5$ then $\mathcal{F}$ is classified.*

Proof: In our cases we will determine that $\mathcal{F}$ is winning or we will find a simpler family $\mathcal{S}$ that is losing to classify $\mathcal{F}$ as a loser. Note that $P_{1,1}$ and $P_{2,1}$ are not candidates for any families of size greater than one.

If $P_{m,1} \notin \mathcal{F}$ for some m, then $\mathcal{F}$ is a loser by Proposition 4.13. Thus assume $P_{m,1} \in \mathcal{F}$ for some m. We have the following cases:

Case A: $P = P_{3,1}$.

Then $\mathcal{F} = \{P_{3,1}, n\text{-squiggle}\}$. So $\mathcal{S} = \{P_{3,1}, P_{4,5}\} \preceq \mathcal{F}$.

Case B: $P = P_{3,2}$.

Then $\mathcal{F} = \{P_{n,1}, P_{3,2}\}$ and therefore $\mathcal{F}$ is a winner.

Case C: $P \in \{P_{4,1}, P_{4,2}, P_{4,3}\}$.
Then if n-Squiggle $\notin \mathcal{F}$, $\mathcal{S} = \{P_{3,1}\} \preceq \mathcal{F}$. If n-Squiggle $\in \mathcal{F}$, then $\mathcal{S} = \{P_{3,1}, P_{4,5}\} \preceq \mathcal{F}$.

Case D: $P = P_{4,4}$.
Then if n-Squiggle $\notin \mathcal{F}$ then $\mathcal{S} = \{P_{3,1}, P_{4,4}\} \preceq \mathcal{F}$. If n-Squiggle $\in \mathcal{F}$, then $\mathcal{S} = \{P_{3,1}, P_{4,4}, P_{5,10}\} \preceq \mathcal{F}$.

Case E: $P = P_{4,5}$.
Then $\mathcal{S} = \{P_{3,1}, P_{4,5}\} \preceq \mathcal{F}$.

In each case where $\mathcal{S} \preceq \mathcal{F}, \mathcal{S}$ is a loser, and therefore $\mathcal{F}$ is a loser by Proposition 4.5. Every possible case for $\mathcal{F}$ has been determined. $\square$

Chapter 7

Infinitude of Animals and Sets

In this chapter we extend the definition of animal to include polyominoes with infinitely many cells. We also extend the defintion of families to allow for these infinite animals. The relation $\sqsubseteq$ is extended in this setting, but as seen in Figure 4.1 this leads to some unexpected consequences. For example, the infinite animals in the figure are surprisingly ancestors of each other.

7.1 Transfamilies

Definition 7.1 A *transfamily* is a family with at least one infinite animal. A family that has no infinite animals is called a *regular* family.

The relation $\preceq$ is also extended to this more general setting.

Definition 7.2 A transfamily $\mathcal{T}$ is a *winner* if $\mathcal{S}$ is a winner for all regular families $\mathcal{S}$ such that $\mathcal{S} \preceq \mathcal{T}$.

Definition 7.3 Let $\mathcal{T}$ be a transfamily. For each infinite animal $T \in \mathcal{T}$ pick R_T to be a finite animal such that $R_T \sqsubseteq T$. The set $\mathcal{R} = \{R_T \mid T \text{ is an infinite animal in } \mathcal{T}\} \cup \{P \in \mathcal{T} \mid P \text{ is a finite animal}\}$ is called a *finite restriction* of $\mathcal{T}$.

Proposition 7.4 *A transfamily $\mathcal{T}$ is a winner if and only if every finite restriction of $\mathcal{T}$ is a winner.*

Proof: If $\mathcal{T}$ is a winner, then every simpler regular family is a winner. Now every finite restriction $\mathcal{R}$ is simpler than $\mathcal{T}$, and therefore is a winner.

Let us assume that every finite restriction is a winner and let $\mathcal{S}$ be a regular family such that $\mathcal{S} \preceq \mathcal{T}$. Then for each $T \in \mathcal{T}$ define $R_T = T$ if T is finite otherwise define R_T to be an element of $\mathcal{S}$ such that $R_T \sqsubseteq T$. If $\mathcal{R} = \{R_T \mid T \in \mathcal{T}\}$, then $\mathcal{S} \preceq \mathcal{R}$.

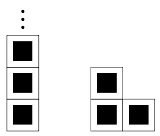


Figure 7.1: A winning transfamily.

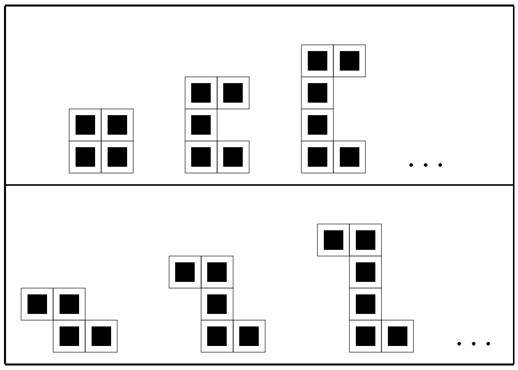


Figure 7.2: Two infinite losing families.

Now $\mathcal{R}$ is a finite restriction of $\mathcal{T}$ and therefore is a winner. Thus by Proposition 4.5, $\mathcal{S}$ is a winner and hence $\mathcal{T}$ is a winner. □

Example 7.5 The family in Figure 7.1 is a winning transfamily by Theorem 5.1 and Proposition 7.4.

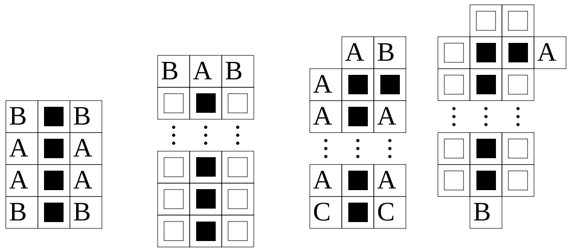
7.2 Infinite Families

Example 7.6 The families in Figure 7.2 are infinite losing families by DP_A.

Definition 7.7 In Figure 7.2 the polyominoes in the first row are called C_k and the polyominoes in the second row are called Z_k. Their sizes are defined as $|Z_k| = |C_k| = k + 2$.

Proposition 7.8 *If $\mathcal{F}_n$ is a finite restriction of $\mathcal{I}_4$, see Figure 7.4 with $P_{n,1} \in \mathcal{F}_n$ then $\mathcal{F}_n$ is a winner.*

Proof: From Proposition 6.4, $\mathcal{F}_4$ is a winner. Therefore for each $\mathcal{F}_n$ for $n \geq 4$ we can either achieve $P_{4,1}$ or $P_{4,2}$ or win with $P_{4,3}, P_{4,4}$ or $P_{4,5}$.



(a) (b) (c) (d)

Figure 7.3: Positions for the infinite transfamily winning strategy.

Let us first consider the case when we have achieved $P_{4,1}$. Using induction we will show that we can either achieve $P_{n+1,1}$ or L_k for some $4 \leq k \leq n$. Figure 7.3(a) shows the situation before the fifth move of the maker. If the breaker has not marked a cell containing the letter A, then the maker can mark that cell and achieve $P_{4,3}$. If the breaker has not marked a cell containing the letter B, then the maker can mark that cell and achieve L_4. Thus we can assume the eight marks by the breaker are in the cells with the letters A and B. Now let us assume that we are in situation Figure 7.3(b) where the maker has marked $P_{j-1,1}$ and the empty squares denote the marks of the breaker. The maker should now mark the cell containing A. If the breaker does not then mark the cells containing B, the maker can achieve L_j by marking one of these. However, if the breaker does mark both B's, we are again in situation Figure 7.3(b) but with $P_{j,1}$ now achieved. Thus we will either achieve $P_{n+1,1}$ or L_k.

Let us now consider the case in Figure 7.3(c) where the maker has achieved L_k. If the breaker has no mark in a cell containing the letter A, then the maker can mark that cell and achieve $P_{4,3}$.If the breaker has no mark in the cell containing the letter B, then the maker can mark that cell and achieve $P_{4,5}$ or Z_2.If the breaker has no mark in a cell containing the letter C, then the maker can mark that cell and achieve C_k or Z_k. Thus we can assume we are in the situation in Figure 7.3(d). Notice that the breaker has $2k+2$ marks to place on the board while only $2k+1$ marks are forced moves. Thus the breaker cannot stop the maker from marking either A or B since cells A and B are disjoint. In both cases, the maker marks cells to the right of A or below B, depending on which cell he marked, until he can turn. An inductive argument similar to the one above shows that in either case he will achieve $P_{n+1,1}$ if he cannot turn, or he will achieve C_k or Z_k for some $4 \leq k \leq n$. $\square$

Corollary 7.9 *The infinite transfamily $\mathcal{I}_4$ is a winner.*

Since a family exists that is an infinite winning family, the goal of listing all finite winning families is not obtainable. Instead, we should try and find a way to

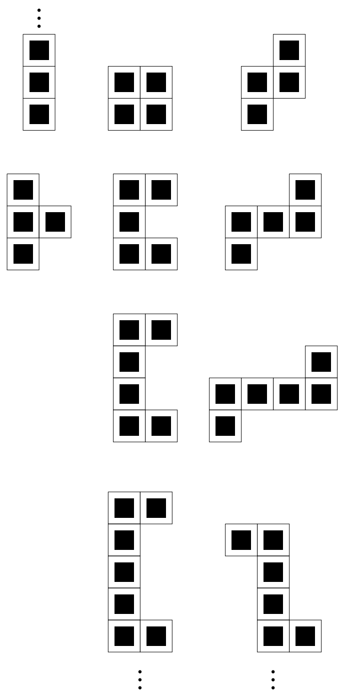


Figure 7.4: A winning infinite transfamily, $\mathcal{I}_4$.

characterize them. This prompts the following definition.

Definition 7.10 A family $\mathcal{I}$ is a *super n-winning family* if for all winning families $\mathcal{F}$ with $|\mathcal{F}| \leq n$, we have $\mathcal{F} \preceq \mathcal{I}$.

Proposition 7.11 *The transfamily $\mathcal{I}_4$ in Figure 7.4 is a super 4-winning family.*

Proof: The winning families of size 1 are ancestors of every polyomino greater than size 1 and thus are simpler than any winning or losing family. The size 2 winning family $\mathcal{F}_2$, in Figure 5.2, is clearly simpler than $\mathcal{I}_4$. The size 3 winning family $\mathcal{F}_3$, in Figure 5.3, is also clearly simpler than $\mathcal{I}_4$. Since there are no winning size 4 families, $\mathcal{I}_4$ is thus a super 4-winning family. $\square$

Proposition 7.12 *There is a winning family for each size $n \in \mathbb{N}$ except for $n = 4$.*

Proof: Let us first note that the previous chapters determine winning families for sizes 1,2,3 and 5. Now from Proposition 7.8 we can see that $\mathcal{F}_n = \{P_{n,1}, P_{4,3}\} \cup \{U_k, Z_k \mid 1 \leq k \leq n-2\}$. Define $\mathcal{G}_n = \mathcal{F}_n \cup \{P_{5,6}\}$.

It is clear that $P_{5,6}$ is not related to any animal in $\mathcal{F}_n$. Thus $\mathcal{G}_n$is a family. Furthermore $\mathcal{G}_n$ is a winning family because $\mathcal{G}_n \preceq \mathcal{F}_n$. Note that for $n = 3, \mathcal{F}_3 = \mathcal{G}_3 = \mathcal{W}_{3,1}$. Therefore we are going to focus on those families where $n \geq 4$.

It is easy to see that $|\mathcal{F}_n| = 2(n-1)$ and that $|\mathcal{G}_n| = 2(n-1)+1$ for $n \geq 4$. Thus $\mathcal{F}_n$ is a winning family of even size for all even numbers greater than or equal to 6. Similarly $\mathcal{G}_n$ is a winning family of odd size for all odd numbers greater than or equal to 7. Therefore the only family size for which there is no winner is size 4. $\square$

Chapter 8

Programs

8.1 Polyomino Creation

This program is a support function that is used by other programs. It generates a list of the polyominoes up to a given size, n. These are ordered by lexicographic order within each size.

To generate all the polyominoes up to size n, we merely need to add a cell onto a polyomino of size $n-1$. However, we need to consider all possible polyominoes, so we can't just add a cell in a single location to each polyomino. Instead we must add a cell to all possible places that might generate a different polyomino. However, we don't want to include too many polyominoes, so we need to normalize each one that is created and see if we have acquired a new one.

The polyominoes are stored in a vector where each element of the vector is a duple of coordinates. Each duple signifies the location of each cell in the polyomino. For the list of polyominoes, there is a vector of the polyomino vectors ordered by lexicographic order within each size.

To create a new polyomino, the program takes a polyomino of size k and for each cell in the polyomino, adds a cell adjacent to it. On the rectangular board, there are four ways to be adjacent and each of them is considered. In some cases, a cell was added in the same location as a previously existing cell. When this happens, the size of the polyomino has not changed, and therefore a new polyomino could not have been created.

Let us assume that the cell was placed in a location that did not contain another cell. This possibly new polyomino is put into standard position and then inserted into a set. If there is already a polyomino with that standard position then there is only one copy because of the properties of sets. This allows us to not worry about generating copies of the same polyomino.

The program begins with $P_{1,1}$, follows the indicated procedure until all the animals of size $n-1$ have been considered as generators.

There are a few things that could be improved with this program. It currently does not check to make sure that the polyominoes created do not have holes. This is not a major concern for this thesis because the polyominoes considered are all size 6 or less. Size 7 is the first size for which there is a polyomino with a hole in it.

Also many of the polyominoes have some symmetry. However, this program does not take that into account. Therefore many of the adjacent placements could be ignored if we could in some way use the symmetry to narrow down the options. Since the polyominoes used here are smaller, this has not been a major concern. Yet if it could be sped up, it would be easier to check different things.

8.2 Paving Creation

The most important program that we created searches for a paving to a set of polyominoes. The program attempts to generate a double paving based on a size parameter and a list of polyominoes under consideration. The paving that is generated is output to a data file that can be used to generate a picture.

Required for the program to run is input that is read from the console. We use a file that contains all the data to increase the speed of input and simplify the process. The file consists of integers in a specific order. First are two integers that represent the size of the board the x size then y size. After that is a list of the polyominoes under consideration. These integers represent the polyominoes in the list generated by a separate class described in the previous section.

The polyominoes are retrieved from the program and all flips, rotations and reflections for each polyomino are stored in a vector, say Fam. Then there is a pair placed in the center of the board. Each of these cells is then placed in a vector called Single which stores all the cells that are related to a single other cell. Since this is a double paving, there is also a vector called Double that contains the cells related to two other cells.

Now the program considers all placements of the polyominoes in Fam that share a paired cell. This ensures that we consider only placements that relate to the portion of the paving that has already been created. For each of these placements, the program calculates the number of pairs that can be placed within the polyomino and thus kill it. It also calculates the distance from the center of the board to the center of the polyomino. Then the placement that has the least number of options for pairs and is closest to the center is considered. That is, the first of the killing pairs is suggested for placement in the paving.

When a pair is in line for placement in the paving, the cells of the pairs are going to be associated with another cell. Thus if either cell is in the Double vector, they are already associated with two cells and can not have a third association. Therefore that pair would be invalidated. If neither of the cells are in Double, then they are moved from Single to Double or into Single, whichever is appropriate.

If there are no pairs, then there is a placement of a polynomial from Fam within the boarders of the paving which has no pairs which will kill it. This means that the paving will not defeat the polynomial and therefore the paving is no good. At this time, the program goes back a level and tries a different pair from the most recent set of possible pairs.

If there are no placements which do not have a killing pair in them, then the paving is done and the program exits. However, the outer border of the paving that is generated is not always killing for every polyomino. This is because the program exits at the first opportunity and so does not check all of the polyominoes from Fam. In Figure 8.1 there is an example of output from the program designed to defeat $P_{4,3}, P_{4,4}, P_{4,5}, P_{5,1}$ and $P_{5,5}$. Notice that there are different patterns within the paving that was created. Some of them are the same pattern with just a rotation or skewed and some are completely different. After a paving of this sort is created, we then go through, look for a pattern that is within the center of the paving (ignoring about 2 boundary cells) and try to generate a double paving from the idea in the paving.

In Figure 8.2 there are two pavings that came from Figure 8.1. The first is $\mathsf{DP_E}$ which defeats the family that was used to generate the paving, see Table B.1. The other paving is not given a name because it is not as useful as $\mathsf{DP_E}$. When both of the pavings are run through the paving checking program (see next section), $\mathsf{DP_E}$ defeats all but two animals of size four to five. The other one defeats all but five, and two of the five not defeated are the ones that $\mathsf{DP_E}$ didn't defeat. Therefore, the other paving is not as powerful as $\mathsf{DP_E}$.

8.3 Paving Checking

The code in this section is used to determine whether a paving defeats a specific set.

Notation 8.1 $|v_i| := (|x_i|, |y_i|)$
$(v_x, v_y) := |v_1| + |v_2|$
$|P|_x :=$ width of a polyomino P in standard position.
$|P|_y :=$ height of a polyomino P in standard position.

In Figure 8.3 we see vectors that represent the fundamental vectors of a paving. A paving can be represented by any of 4 pairs of fundamental vectors, a positive and negative for each of two vectors. Thus there are 4 candidates for the pair of

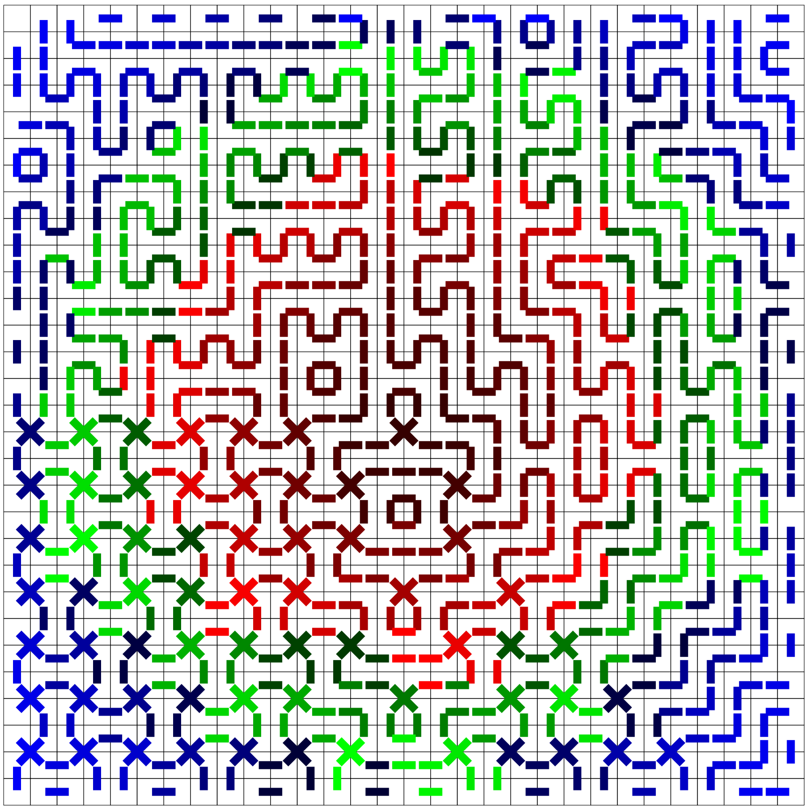


Figure 8.1: A paving generated by the paving program on a 30x30 board for $P_{4,3}, P_{4,4}, P_{4,5}, P_{5,1}$ and $P_{5,5}$

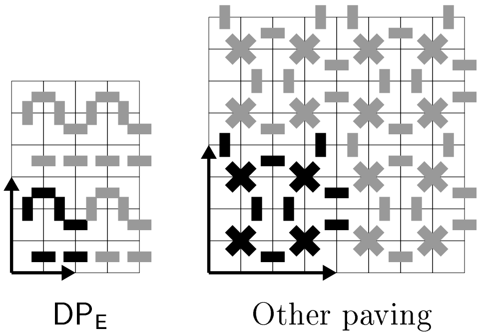


Figure 8.2: Two pavings that can be extracted from Figure 8.1

fundamental vectors for any paving. I restrict the consideration for fundamental

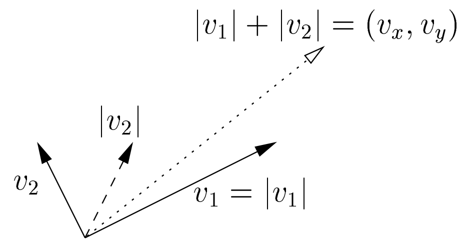


Figure 8.3: The fundamental vectors of a 2-paving and their relation to constructs in the paving checking program. Note that $|v_k| = (|x_k|, |y_k|)$

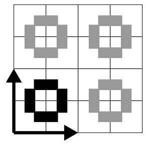


Figure 8.4: 2-paving as reference for problem locations in a paving.

vectors to those vectors whose angles α are such that $0 \leq \alpha < \frac{\pi}{2}$. This restriction leaves only two vectors. The vector with the smallest angle α is v_1 and the other v_2. Therefore there is a unique representation of the fundamental vectors of a paving that is used by this program.

The size of the board is critical to the success of the program. The board must be large enough to accommodate all placements of all configurations of each polyomino under consideration. A problem area with a paving is often found in the boundaries and corners of adjacent copies of a fundamental region of the paving. In Figure 8.4 the fundamental region of the paving is apparently strong, but in the corner of four copies, $P_{4,4}$ can be placed and therefore the paving does not kill $P_{4,4}$. Thus the board must be at least twice as tall and twice as wide as any paving to include the corners and the boundaries in the search region.

To determine the size of the board, a variable max is created such that $max = \max\{v_x, v_y, |P|_x, |P|_y\}$ for each polyomino P in the set under consideration. Thus the largest measure of any polyomino is a factor in determining the size of the board. Once this max is generated, the board is generated as a square with sides of length $2 * max + 1$. This is at least twice as tall as the largest polyomino in the set and the largest measure on the paving. The addition of one more unit allows a little extra room along the edges for checking, but not so much that time is wasted in redundant placements.

Once the size has been determined, the paving is copied along its fundamental vectors an appropriate number of times in appropriate places so that every cell on the board is related as defined by the fundamental set. Through this process, cells outside the required region could have been related. To help with process time, the program removes any related cells that fall outside the defined square board.

After this clean up process is completed this program proceeds to determine if the paving under consideration kills the polyominoes under consideration. Each of the polyominoes is processed singly with all its flips, rotations and reflections. Each of these transformations is shifted around the board. If there is a placement of a transformation that does not contain a pair, then the paving does not kill this polyomino. A data file is output which contains the board and the placement that was not killed. Then the next polyomino in the set is considered.

If a transformation is killed, the next transformation is considered and shifted across the board. When all transformations have been shifted and killed, then the paving kills the polyomino and the next one in the set is considered.

Bibliography

[1] M. Anderson and F. Harary. Achievement and avoidance games for generating abelian groups. *Internat. J. Game Theory*, 16(4):321–325, 1987.

[2] J. Beck. On 3-chromatic hypergraphs. *Discrete Math.*, 24(2):127–137, 1978.

[3] J. Beck. Remarks on positional games. I. *Acta Math. Acad. Sci. Hungar.*, 40(1-2):65–71, 1982.

[4] J. Beck. Tic-Tac-Toe. In *Contemporary combinatorics*, volume 10 of *Bolyai Soc. Math. Stud.*, pages 93–137. János Bolyai Math. Soc., Budapest, 2002.

[5] Jens-P. Bode. *Strategien Føur Aufbauspiele mit mosaik-polyominos*. PhD thesis, Technichen Universitøat Braunschweig, 2000.

[6] Jens-P. Bode and Heiko Harborth. Achievement games on Platonic solids. *Bull. Inst. Combin. Appl.*, 23:23–32, 1998.

[7] Jens-P. Bode and Heiko Harborth. Hexagonal polyomino achievement. *Discrete Math.*, 212(1-2):5–18, 2000. Graph theory (Dörnfeld, 1997).

[8] Jens-P. Bode and Heiko Harborth. Triangular mosaic polyomino achievement. In *Proceedings of the Thirty-first Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, FL, 2000)*, volume 144, pages 143–152, 2000.

[9] Jens-P. Bode and Heiko Harborth. Triangle polyomino set achievement. In *Proceedings of the Thirty-second Southeastern International Conference on Combinatorics, Graph Theory and Computing (Baton Rouge, LA, 2001)*, volume 148, pages 97–101, 2001.

[10] Jens-P. Bode and Heiko Harborth. Achievement games for polyominoes on Archimedean tessellations. In *Mathematical properties of sequences and other combinatorial structures (Los Angeles, CA, 2002)*, pages 101–112. Kluwer Acad. Publ., Boston, MA, 2003.

[11] Fred Buckley and Frank Harary. Diameter avoidance games for graphs. *Bull. Malaysian Math. Soc. (2)*, 7(1):29–33, 1984.

[12] Steven C. Cater, Frank Harary, and Robert W. Robinson. One-color triangle avoidance games. In *Proceedings of the Thirty-second Southeastern International Conference on Combinatorics, Graph Theory and Computing (Baton Rouge, LA, 2001)*, volume 153, pages 211–221, 2001.

[13] Gary Chartrand, Frank Harary, Michelle Schultz, and Donald W. VanderJagt. Achievement and avoidance of a strong orientation of a graph. In *Proceedings of the Twenty-sixth Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, FL, 1995)*, volume 108, pages 193–203, 1995.

[14] P. Erdős and J. L. Selfridge. On a combinatorial game. *J. Combinatorial Theory Ser. A*, 14:298–301, 1973.

[15] Martin Erickson and Frank Harary. Picasso animal achievement games. *Bull. Malaysian Math. Soc. (2)*, 6(2):37–44, 1983.

[16] Martin Erickson and Frank Harary. Generalized Ramsey theory. XV. Achievement and avoidance games for bipartite graphs. In *Graph theory, Singapore 1983*, volume 1073 of *Lecture Notes in Math.*, pages 212–216. Springer, Berlin, 1984.

[17] Solomon G. Golomb. *Polyominoes: Puzzles, Patterns, Problem and Packings.* Princeton University Press, 1965.

[18] Ronald L. Graham, Bruce L. Rothschild, and Joel H. Spencer. *Ramsey theory.* John Wiley & Sons Inc., New York, 1980. Wiley-Interscience Series in Discrete Mathematics, A Wiley-Interscience Publication.

[19] Frank Harary. Achievement and avoidance games designed from theorems. *Rend. Sem. Mat. Fis. Milano*, 51:163–172 (1983), 1981.

[20] Frank Harary. Achievement and avoidance games for graphs. In *Graph theory (Cambridge, 1981)*, volume 13 of *Ann. Discrete Math.*, pages 111–119. North-Holland, Amsterdam, 1982.

[21] Frank Harary. An achievement game on a toroidal board. In *Graph theory (Łagów, 1981)*, volume 1018 of *Lecture Notes in Math.*, pages 55–59. Springer, Berlin, 1983.

[22] Frank Harary. Achievement and avoidance games on finite configurations. *J. Recreational Math.*, 16(3):182–187, 1983/84.

[23] Frank Harary. Achievement and avoidance games on finite configurations with one color. *J. Recreational Math.*, 17(4):253–260, 1984/85.

[24] Frank Harary. Arithmetic progression achievement and avoidance games. *Menemui Mat.*, 7(3):105–113, 1985.

[25] Frank Harary. Is Snaky a winner? *Geombinatorics*, 2(4):79–82, 1993.

[26] Frank Harary and Heiko Harborth. Extremal animals. *J. Combinatorics Information Syst. Sci.*, 1(1):1–8, 1976.

[27] Frank Harary, Heiko Harborth, and Markus Seemann. Handicap achievement for polyominoes. In *Proceedings of the Thirty-first Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, FL, 2000)*, volume 145, pages 65–80, 2000.

[28] Frank Harary and Christopher Leary. Latin square achievement games. *J. Recreational Math.*, 16(4):241–246, 1983/84.

[29] Frank Harary and Ken Plochinski. On degree achievement and avoidance games for graphs. *Math. Mag.*, 60(5):316–321, 1987.

[30] Frank Harary, Wolfgang Slany, and Oleg Verbitsky. A symmetric strategy in graph avoidance games. In *More games of no chance (Berkeley, CA, 2000)*, volume 42 of *Math. Sci. Res. Inst. Publ.*, pages 369–381. Cambridge Univ. Press, Cambridge, 2002.

[31] Heiko Harborth and Markus Seemann. Snaky is an edge-to-edge looser. *Geombinatorics*, 5(4):132–136, 1996.

[32] Heiko Harborth and Markus Seemann. Snaky is a paving winner. *Bull. Inst. Combin. Appl.*, 19:71–78, 1997.

[33] Heiko Harborth and Markus Seemann. Handicap achievement for squares. *J. Combin. Math. Combin. Comput.*, 46:47–52, 2003. 15th MCCCC (Las Vegas, NV, 2001).

[34] Heiko Harborth and Hartmut Weiss. Minimum sets of partial polyominoes. *Australas. J. Combin.*, 4:261–268, 1991. Combinatorial mathematics and combinatorial computing (Palmerston North, 1990).

[35] András Pluhár. Generalized Harary games. *Acta Cybernet.*, 13(1):77–83, 1997.

[36] Nándor Sieben. Wild polyomino weak (1,2)-achievement games. Preprint.

[37] Nándor Sieben. Hexagonal polyomino weak $(1, 2)$-achievement games. *Acta Cybernet.*, 16(4):579–585, 2004.

[38] Nándor Sieben. Snaky is a 41-dimensional winner. *Integers*, 4:G5, 6 pp. (electronic), 2004.

[39] Nándor Sieben and Elaina Deabay. Polyomino weak achievement games on 3-dimensional rectangular boards. *Discrete Mathematics*, 290:61–78, 2005.

Appendix A

C++ Code

A.1 Postscript Generating Code

This code creates the postscript files for polyominoes and k-pavings. The code reads in an input file that has locations of cells to be created and designations for different basic cells.

The first value specifies the size of the boxes to be created. Then there is either a code for a paving to be created or a list of commands. If it is the paving code, the file has the vectors and the number of copies to generate. The rest of the file contains the pairs and any further marks in the picture.

If it is not a paving, then there is a code followed by location values "x y" and if there is another parameter it is on the end.

```
// compile with
// g++ -O boxa.C -lg2

/*
This program takes a file in special format and creates a
postscript graphic's file

The file starts with an integer for magnification.
   Used for spacing to make letters fit within each box.
   The default value of 12 is given for an integer value of 0.
   17 is usually used when any text in to be inside the box.

The next lines need to start with a number 0-7 for the
following functions
   0: An empty box with integer coordinates following in x,y
   1: A tiling of one color with integer coordinates of the
        two boxes to be linked in x,y and x,y
   2: A 2nd tiling color with integer coordinates as previous
   3: A box that has been marked by the maker with integer
      coordinates x,y
   4: A box that has an empty square (to represent the
      breaker's mark) with integer coordinates x,y
   5: A box with letters in it with integer coordinates x,y
```

```
        followed by the letters to be placed within the box
   6: A horizontal ellipsis with integer coordinates x,y
   7: A vertical ellipsis with integer coordinates x,y
   8:   A paving is to be created
There is no terminating line
*/

#include <g2.h>
#include <g2_PS.h>
#include <math.h>
#include <set>
#include <string>
#include <vector>
#include <stdio.h>
#include <iostream>
#include <algorithm>

using namespace std;

int dev, size;
int color[3];
bool Multi_Paving = false;
double X[2], Y[2];

vector <int> pairs;
vector <double> boxes;

const double num = 1;
const double denom = 2;
const double xx = 10;
const double yy = 10;

void boxf(double x, double y, double s, bool filled)
  {
  double points[8];

  points[0] = xx + x - s; //Upper left corner of box
  points[1] = yy + y + s;

  points[2] = xx + x + s; //Upper right corner of box
  points[3] = yy + y + s;

  points[4] = xx + x + s; //Lower right corner of box
  points[5] = yy + y - s;

  points[6] = xx + x - s; //Lower left corner of box
  points[7] = yy + y - s;

  if (!filled)
    g2_polygon(dev, 4, points);
  else
    g2_filled_polygon(dev, 4, points);
  }


void Write(double a, double b, char *t)
  {
  int i = 0; //counter for length of t
  int j = 0; //counter for Capital string
  int k = 0; //counter for lower case string
```

```
  float shift; //amount to raise or lower the second row

  char g[6];
  char h[6];

  for (int m = 0; m < 7; m++)
    {
    g[m] = 0; //clear uppercase string
    h[m] = 0; //clear lowercase string
    }

  if (t[0] == 0)
    cout << "Empty string";

  for (i; t[i+1] > '@'; i++); //count size of t

  if (i < 3) //if less that 4 characters
    g2_string(dev, xx + a - .4, yy + b - .25, t);
  else
    {
    for (int m = 0; m < i + 1; m++)
      if ((t[m] >= 'A') && (t[m] <= 'Z')) //If a capital
        {
        g[j] = t[m];
        j++;
        }
      else if ((t[m] >= 'a') && (t[m] <='z'))
        {
        h[k] = t[m];
        k++;
        }
      else {}
        //nothing
    if (j != 0)
      {
      g2_string(dev, xx + a - .4, yy + b + .1, g);
      shift = -.3; //There are caps, shift lower case down
      }
    else
      shift = -.1; //No caps, only shift lower case
    g2_string(dev, xx + a - .4, yy + b + shift, h);
    }
  }


void boxi(int i, int j, char *t)
  {
  double x = i;
  double y = j;

  boxf(x, y, .5, 0); //blank box

  if (t[0] == '!') //mark in box
    boxf(x, y, .3, 1);
  else if (t[0] == '?') //breaker's mark in box
    boxf(x, y, .3, 0);
  else if (t[0] != 0) //box with writing
    Write(x, y, t);
  }
```

```
void domino(double i, double j, double k, double l)
  {
  double m = (i + k) / 2.0; // midpoint x-value
  double n = (j + l) / 2.0; // midpoint y-value
  double x = xx + num*(i + m) / denom; // end points
  double y = yy + num*(j + n) / denom;
  double w = xx + num*(k + m) / denom;
  double z = yy + num*(l + n) / denom;

  g2_line(dev, x, y, w, z);
  }


void MakeDominoes(int colored)
  {
  double p1, p2, p3, p4;
  int i, n, m;

  g2_set_line_width(dev, 4);

  if (colored)
    g2_pen(dev, color[2]);

  for (i = 0; i < pairs.size(); i+=4)
    domino(pairs[i], pairs[i+1], pairs[i+2], pairs[i+3]);

  if (Multi_Paving)
    {
    g2_pen(dev, color[2]);
    n = 1;

    for (m = 0; m < 2; m++)
      {
      for (n; n < size/2; n++)
        for (i = 0; i < pairs.size(); i+=4)
          {
          p1 = pairs[ i ] + m * X[0] + n * Y[0];
          p2 = pairs[i+1] + m * X[1] + n * Y[1];
          p3 = pairs[i+2] + m * X[0] + n * Y[0];
          p4 = pairs[i+3] + m * X[1] + n * Y[1];
          domino(p1, p2, p3, p4);
          }
        n = 0;
        }
    g2_pen(dev, color[1]);
    }

  g2_set_line_width(dev, 1);
  }


void Arrow(double a, double b)
  {
  double four_d, first, secnd, middle, j, k, m, n, s, df;

  df = xx - .5;
  four_d = 4.0 * sqrt(a*a + b*b);
  middle = 1 - sqrt(3.0)/four_d;
  first = a / four_d;
```

```
  secnd = b / four_d;

  m = a * middle + df;
  n = b * middle + df;
  j = a + df;
  k = b + df;
  s = abs(a) + abs(b);

  g2_filled_triangle(dev, j, k, m + secnd, n - first, m - secnd,
                     n + first);
  g2_line(dev, df, df, j - (a * .3)/s, k - (b * .3)/s);
  }


void MakeRegion()
  {
  int x_min, x_max, n;
  double c1, c2;
  double m_x;

  x_min = 0;
  x_max = int(X[0] + Y[0]);

  if (Y[0] < 0)
    {
    x_min = int(Y[0]);
    x_max = int(X[0]);
    }

  m_x = X[1] / X[0];

  for (int b = 0; b < X[1] + Y[1]; b++)
    for (int a = x_min; a < x_max; a++)
      if ((b >= m_x * a) && (b < m_x * (a - Y[0]) + Y[1])
          && (Y[0] * b <= Y[1] * a)
          && (Y[0] * (b  - X[1]) > Y[1] * (a - X[0])))
        {
        boxes.push_back(a);
        boxes.push_back(b);
        }

  for (int i = 0; i < boxes.size(); i+=2)
    boxf(boxes[i], boxes[i+1], .5, 0);

  n = 1;

    if (size > 1)
      for (int m = 0; m < 2; m++)
        {
        for (n; n < size/2; n++)
          for (int i = 0; i < boxes.size(); i+=2)
            {
            c1 = boxes[ i ] + m * X[0] + n * Y[0];
            c2 = boxes[i+1] + m * X[1] + n * Y[1];
            boxf(c1, c2, .5, 0);
            }
        n = 0;
        }

  g2_set_line_width(dev, 1.2);
```

```
  Arrow(X[0], X[1]);
  Arrow(Y[0], Y[1]);
  g2_set_line_width(dev, 1);
  }


void ellipsis(double a, double b, int dir)
  {
  double l, m, x, y;

  x = xx + a;
  y = yy + b;

  switch (dir)
    {
    case 6: //horizontal ellipsis
      l = x + .25;
      m = x - .25;
      g2_filled_circle(dev, l, y, .05);
      g2_filled_circle(dev, m, y, .05);
      break;
    case 7: //vertical ellipsis
      l = y + .25;
      m = y - .25;
      g2_filled_circle(dev, x, l, .05);
      g2_filled_circle(dev, x, m, .05);
      break;
    }
  g2_filled_circle(dev, x, y, .05);
  }


void readin()
  {
  typedef set<char> Situations;
  vector <Situations> Cells;

  int i, j, k, l, a;
  int colored = 0;
  char text[20];
  double s = 0.5;
  double points[8];
  double var1, var2;
  bool failure = false;
  Situations CurCell, All, temp;

  while (cin >> a)
    {
    switch (a)
      {
      case 0: //empty box
        cin >> i >> j;
        boxi(i, j, "");
        break;
      case 2: //1st tiling marker
colored = 1;
      case 1: //2nd tiling marker
        for (j = 0; j < 4; j++)
          {
          cin >> i;
```

```
        pairs.push_back(i);
        }
      break;
    case 3: //Box with maker's mark
      cin >> i >> j;
      boxi(i, j, "!");
      break;
    case 4: //Box with breaker's mark
      cin >> i >> j;
      boxi(i, j, "?");
      break;
    case 5: //Box with letters
      cin >> i >> j >> text;
      boxi(i, j, text);
      CurCell.clear(); //empty the set
      for (int n = 0; text[n]; n++)
        {
        text[n] = tolower(text[n]);
        CurCell.insert(text[n]);
        }
      Cells.push_back(CurCell);
      set_union(CurCell.begin(), CurCell.end(), All.begin(), All.end(),
                inserter(temp, temp.begin()));
      All = temp;
      temp.clear(); //empty temporary set
      break;
    case 6: //horizontal ellipsis
    case 7: //vertical ellipsis
      cin >> var1 >> var2;
      ellipsis(var1, var2, a);
      break;
    case 8: //paving
      cin >> size >> X[0] >> X[1]  >> Y[0] >> Y[1];
      if (size > 1)
        Multi_Paving = true;
      MakeRegion();
      break;
    default:
      cout << "Error, incorrectly formatted file.\n";
    }
  }
MakeDominoes(colored);

int count = All.size(); //total number of letters

for (i = 0; i < Cells.size(); i++)
  for (j = i + 1; j < Cells.size(); j++)
    {
    set_union(Cells[i].begin(), Cells[i].end(),
              Cells[j].begin(), Cells[j].end(),
              inserter(temp,temp.begin()));
    if (count == temp.size())
      cout << "Not a winning strategy: " << i << " " << j << endl;
    temp.clear();
    }
}

int main()
{
dev=g2_open_EPSF("boxa.ps");
```

```
color[0]=g2_ink(dev, 0, 1, 1);//teal tile color
color[1]=g2_ink(dev, 0, 0, 0);//black tile color
color[2]=g2_ink(dev, .6, .6, .6); //grey tile color
g2_set_font_size(dev, 10);
double magnif;

cin >> magnif;
if (magnif == 0) magnif = 12;

g2_set_coordinate_system(dev, 0, 0, magnif, magnif);

g2_pen(dev, color[1]);
readin();

g2_close(dev);
return 0;
}
```

A.2 Paving Code for a Specific Family

This code generates a 2-paving for a family.

```
/*
The program is designed to create a double paving that will
establish a strategy for the breaker that will defeat all the
animals in a given family.  This program allows for triples
of pavings, that is, three mutually joined cells.  A file with
the following structure is needed.

Dimensions of region that are desired to be paved and the
number of the animals in the family that are going to be
considered given in size then lexicographic order.
dim_x dim_y

If a paving is found, the program outputs a file (tile1.dat)
with the dimensions dim_x dim_y then the size of the paving
in number of pairs and a list of ordered pairs that coincide
with the paired cells of a paving.
*/

#include "state.h"
#include "fstream"
#include "createanimals.h"
#include "STLmore.h"
#include <iostream>

const int xmin = 0;
const int ymin = 0;
int xmax;
int ymax;
int levelmax;

int level = 0;
int tilenum = 0;

Tanimal paving;
Tanimal Double;
```

```
Tanimal Single;

Tstate state;
Tstates transfers;
Tstatesset shiftedtransfersset;
Tstates shiftedtransfers;

void writetile (void)
  {
  tilenum++;
  ofstream os ("tile1.dat");
  os << xmax + 1 << " " << ymax + 1 << " " << paving.size();
  for (int i = 0; i < paving.size (); i++)
    if ( i % 6 == 0)
      os << endl;
    os << paving[i][0] << " " << paving[i][1] << "  ";
  os.close ();
  }


void PavePrint()
  {
  string filename = "level-" + all2string(level) + ".dat";

  ofstream os (filename.c_str());
  os << xmax + 1 << " " << ymax + 1 << " " << paving.size();
  for (int i = 0; i < paving.size (); i++)
    if ( i % 6 == 0)
      os << endl;
    os << paving[i][0] << " " << paving[i][1] << "  ";
  os.close ();

  string command1 = "./paving/Shader < " + filename;
  string command2 = "mv Pave.ps Level-" + all2string(level) + ".ps";

  system(command1.c_str());
  system(command2.c_str());
  cout << level << endl;
  }


bool inside (const Tstate & state)
  {
  Tcell ll = llcorner(state);
  if (ll[0] < xmin) return false;
  if (ll[1] < ymin) return false;

  Tcell ur = urcorner(state);
  if (ur[0] > xmax) return false;
  if (ur[1] > ymax) return false;

  return true;
  }


bool killed (const Tstate & state)
  {
  for (int i = 0; i < paving.size(); i++)
    if (binary_search(BE(state.core), paving[i]))
      {
```

```
      i++;
      if (binary_search(BE(state.core), paving[i]))
        return true;
      }
    else
      i++;
  return false;
  }


void findshifts (Tstate & state)
  {
  Tstate shiftstate;
  int i, j;

  if (paving.size() == 0)
    {
    Tcell mid;
    mid.push_back((xmin + xmax) / 2);
    mid.push_back((ymin + ymax) / 2);
    shiftstate = state;
    shift(shiftstate, mid);
    shiftedtransfersset.insert(shiftstate);
    return;
    }

  set < Tcell> collect;
  Tanimal::iterator it;
  Tanimal::iterator jt;

  collect.clear ();
  for (i = 0; i < paving.size(); i++)
    for (j = 0; j < state.core.size(); j++)
      {
      Tcell cell = make_cell (paving[i][0] - state.core[j][0],
                     paving[i][1] - state.core[j][1]);
      collect.insert (cell);
      }

  set < Tcell >::iterator itt;

  for (itt = collect.begin (); itt != collect.end (); itt++)
    { // go through the shifts
    shiftstate = state;
    shift (shiftstate, *itt);
    if (inside(shiftstate) and ! killed(shiftstate))
      shiftedtransfersset.insert (shiftstate);
    }
  }


bool InSquare(Tcell one, Tcell two)
  {
  //ordered cells, ie two > one
  int diff1 = two[0] - one[0];
  int diff2 = two[1] - one[1];

  if (diff1 > 2 || diff2 > 2)
    return false;
  return true;
```

```
  }


bool Pavable(Tcell first, Tcell second)
  {
  if (binary_search(BE(Double), first) || binary_search(BE(Double), second))
    return false;
  return true;
  }


double distance(Tanimal & Poly)
  {
  double d;
  double mid_x, mid_y;
  double cnt_x, cnt_y;

  int n = Poly.size() - 1;

  Tcell ll = Poly[0];
  Tcell ur = Poly[n];
  mid_x = (ll[0] + ur[0])/2.0;
  mid_y = (ll[1] + ur[1])/2.0;
  cnt_x = xmax/2.0;
  cnt_y = ymax/2.0;

  d = (mid_x - cnt_x)*(mid_x - cnt_x) + (mid_y - cnt_y) * (mid_y - cnt_y);

  return d;
  }


void findkillers(const Tstate & animal, Tanimal & pairs,
double & dist)
  {
  int i, j, size;
  Tanimal diff, spectre;

  set_difference(BE(animal.core), BE(Double), INS(diff));
  size = diff.size();
  spectre = animal.core;

  dist = distance(spectre);

  for (i = 0; i < size - 1; i++)
    for ( j = i + 1; j < size; j++)
      if (InSquare(diff[i],diff[j]))
        {
        pairs.push_back(diff[i]);
        pairs.push_back(diff[j]);
        }
  }


void Remove(Tcell Cell)
  {
  Tanimal::iterator it;

  it = lower_bound(BE(Double),Cell);
  if (it != Double.end() && *it == Cell)
```

```
    {
    Double.erase(it);
    insertsorted(Single, Cell);
    }
  else
    {
    it = lower_bound(BE(Single), Cell);
    Single.erase(it);
    }
  }


void Store(Tcell Cell)
  {
  Tanimal::iterator it;

  it = lower_bound(BE(Single), Cell);
  if (it != Single.end() && *it == Cell)
    {
    Single.erase(it);
    insertsorted(Double, Cell);
    }
  else
    insertsorted(Single,Cell);
  }


void reduce(Tanimal & bestdom)
  {
  int i, j;
  Tstate state1, state2;

  for (i = 0; i < bestdom.size() - 3; i++)
    {
    state1.core.push_back(bestdom[i]);
    state1.core.push_back(bestdom[i + 1]);
    normal(state1);
    i++;
    for (j = i + 1; j < bestdom.size() - 1; j++)
      {
      state2.core.push_back(bestdom[j]);
      state2.core.push_back(bestdom[j + 1]);
      normal(state2);
      j++;
      if (state1 == state2)
        {
        j--;
        bestdom.erase(bestdom.begin() + j);
        bestdom.erase(bestdom.begin() + j);
        j--;
        }
      }
    }
  }


void add_domino ()
  {
  int i;
```

```
  level++;
  if (level > levelmax)
    levelmax = level;
  double dist, best_dist;
  Tanimal bestdominoes(100);
  Tanimal pairs;
  Tstate bestposition;
  shiftedtransfersset.clear ();

  for (i = 0; i < transfers.size (); i++)
    findshifts (transfers[i]);
  shiftedtransfers.clear();
  set2vector (shiftedtransfersset, shiftedtransfers);

  if (shiftedtransfers.size() == 0)
    {
    writetile();
    //return;
    exit(1);
    }

  best_dist = xmax*xmax + ymax*ymax; //clear old distance
  for (i = 0; i < shiftedtransfers.size(); i++)
    {
    pairs.clear();
    findkillers(shiftedtransfers[i], pairs, dist);

    if (pairs.size() > bestdominoes.size())
      continue;
    if (pairs.size() == bestdominoes.size() && dist >
        best_dist)
      continue;
    bestdominoes = pairs;
    best_dist = dist;

    if (bestdominoes.size() == 0)
      {
      if (level >= levelmax)
PavePrint();
      level--;
      return;
      }
    }

  if (paving.size() == 0){}
    //reduce(bestdominoes);

  for (i = 0; i < bestdominoes.size(); i+=2)
    {
    paving.push_back(bestdominoes[i]);
    paving.push_back(bestdominoes[i + 1]);
    Store(bestdominoes[i]);
    Store(bestdominoes[i + 1]);
    add_domino();

    paving.pop_back();
    paving.pop_back();
    Remove(bestdominoes[i]);
    Remove(bestdominoes[i + 1]);
    }
```

```
  if (level >= levelmax)
    PavePrint();
  level--;
  }

int main (void)
  {
  int dim_x, dim_y;
  int anim; //# for animal to pave
  Tstates animals;
  Tstatesset transferset;
  createanimals (6, animals);

  cin >> dim_x >> dim_y;

  xmax = dim_x - 1;
  ymax = dim_y - 1;

  while (cin >> anim)
    {
    state = animals[anim];
    transferall (state, transferset);
    set2vector (transferset, transfers);
    transferset.clear();
    }

  add_domino ();

  return 0;
  }
```

A.3 Created Paving Postscript Code

This code takes output from the paving creation code and generates a post-script file with special colors. These colors are used to determine the approximate time a pair was generated by the program. The colors start dark and become lighter as the pairs progress later into the paving. The colors are first red, then green and finally blue. This also helps understand if the program is actually creating a paving from the inside out or if it is instead going to the boundary.

```
// compile with
// g++ -O3 Zpave.C -lg2

/*
This program takes a file in special format and creates a
postscript graphic's file of a paving file created by the Paver
program.

The format is the dimensions of the region to be drawn in
integers in the following format:
dim_x dim_y

Followed by the number of pairs of cells.
```

```
Then a series of pairs of cells.

There is no terminating line
*/

#include <g2.h>
#include <g2_PS.h>
#include <math.h>
#include <set>
#include <stdio.h>
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

int dev;

const double num = 7;
const double denom = 8;
const double xx = 10;
const double yy = 10;

void box(double x, double y, double s)
  {
  double points[8];

  points[0] = xx + x - s; //Upper left corner of box
  points[1] = yy + y + s;

  points[2] = xx + x + s; //Upper right corner of box
  points[3] = yy + y + s;

  points[4] = xx + x + s; //Lower right corner of box
  points[5] = yy + y - s;

  points[6] = xx + x - s; //Lower left corner of box
  points[7] = yy + y - s;

  g2_polygon(dev, 4, points);
  }

void domino(int i, int j, int k, int l)
  {
  double m = (i + k) / 2.0; //midpoint x-value
  double n = (j + l) / 2.0; //midpoint y-value
  double x = xx + (num*i + m) / denom; //end points
  double y = yy + (num*j + n) / denom;
  double w = xx + (num*k + m) / denom;
  double z = yy + (num*l + n) / denom;

  g2_line(dev, x, y, w, z);
  }

void readin()
  {
  int color, i, j, k, l;
  int width, height, length, count;
  float col[3], dc;
  int loc = 0;
```

```
  cin >> width >> height >> length;

  count = length / 6 + 1;
  dc = .8 / double(count);

  for (int m = 0; m < width; m++)
    for (int n = 0; n < height; n++)
      box(m, n, .5);

  col[0] = .2 - dc;
  col[1] = 0;
  col[2] = 0;

  g2_set_line_width(dev, 3);

  while (cin >> i)
    {
    col[loc]+=dc;

    cin >> j >> k >> l;

    color = g2_ink(dev, col[0], col[1], col[2]);
    g2_pen(dev, color);

    domino(i, j, k, l);

    if (col[loc] >= 1)
      {
      col[loc] = 0;
      loc++;
      col[loc] = .2 - dc;
      }
    }
  }

int main()
  {
  dev=g2_open_EPSF("Pave.ps");
  g2_set_font_size(dev, 10);
  double magnif;

  magnif = 10;

  g2_set_coordinate_system(dev, 0, 0, magnif, magnif);

  readin();

  g2_close(dev);
  return 0;
  }
```

A.4 Paving Checking Code

This code checks a particular 2-paving with a specific animal to see if the animal is defeated by the 2-paving.

A.4.1 Checking Code

```
#include <iostream>
#include <fstream>
#include <set>
#include <string>
#include <vector>
#include <algorithm>
#include "state.h"
#include "STLmore.h"

using namespace std;

set < Tstate > transs;
vector < Tcell > pairs;
vector < Tcell > paving;

int R, x[2], y[2];
int num = 0;

void FailPic(Tstate & state)
  {
  string name;
  num++;
  name = "fail" + all2string(num) + ".dat";
  ofstream os (name.c_str());

  os << "0" << endl;

  for (int i = 0; i < state.core.size(); i++)
    os << "3 " << state.core[i][0] << " " << state.core[i][1] << endl;

  for (int i = 0; i < paving.size(); i+=2)
    os << "1 " << paving[i][0] << " " << paving[i][1] << "  "
       << paving[i+1][0] << " " << paving[i+1][1] << endl;

  os.close ();
  }


void CurBoard(void)
  {
  ofstream os ("board.dat");

  os << "0\n"; //boxa

  for (int i = 0; i < paving.size(); i+=2)
    os << "1 " << paving[i][0] << " " << paving[i][1] << "  "
       << paving[i+1][0] << " " << paving[i+1][1] << endl;
  }


void CleanUp()
  {
  vector < Tcell >::iterator i;
  Tcell temp1, temp2;

  for (i = paving.begin(); i != paving.end(); i++)
    {
    temp1 = (*i);
```

```
    temp2 = (*(i+1));

    if ((temp1[0] < 0 && temp1[1] < 0 && temp2[0] < 0 && temp2[1] < 0) ||
        (temp1[0] > R && temp1[1] > R && temp2[0] > R && temp2[1] > R))
      {
      paving.erase(i);
      paving.erase(i);
      i--;
      }
    else
      i++;
    }
  }


void Region(void)
  {
  Tcell temp1, temp2;
  int M_x, M_y, m_y, denom;

  denom = x[0] * y[1] - x[1] * y[0];
  M_x = (R * (y[1] - y[0]))/ denom + 1;

  M_y = (R * x[0])/ denom + 1;

  m_y = - (R * x[1])/ denom - 1;

  for (int i = 0; i < pairs.size(); i+=2)
    for (int m = 0; m < M_x; m++)
      for (int n = m_y; n < M_y; n++)
{
temp1 = pairs[i];
temp2 = pairs[i+1];

temp1[0] = temp1[0] + m * x[0] + n * y[0];
temp1[1] = temp1[1] + m * x[1] + n * y[1];
paving.push_back(temp1);
temp2[0] = temp2[0] + m * x[0] + n * y[0];
temp2[1] = temp2[1] + m * x[1] + n * y[1];
paving.push_back(temp2);
    }
  }


bool tiled (Tstate state)
  { // return 1 if tile works for all shifts 0 if not
  Tcell C1, C2;
  Tstate shstate = state;

  for (int j = 0; j < R; j++) // shift the animal around
    for (int k = 0; k < R; k++)
      {
      shstate = state;
      shift (shstate, make_cell (k, j));
      bool hasinshift = false;
      for (int i = 0; i < paving.size (); i += 2)
        {
        C1 = paving[i];
        C2 = paving[i+1];
        if (binary_search (BE (shstate.core), C1) and
```

```
          binary_search (BE (shstate.core), C2))
         hasinshift = true;
        }
      if (!hasinshift)
        {
        FailPic(shstate);

        return false;
        }
      }
  return true;
  }


bool checkanimal (Tstate & state)
  {
  set < Tstate >::iterator i;
  transs.clear ();
  transferall (state, transs);

  for (i = transs.begin (); i != transs.end (); i++)
    if (!tiled ((*i)))
      return false; // the transform didn't have a domino
  return true;
  }


void ReadIn()
  {
  int a, b;
  ifstream file;
  file.open ("tile.dat");

  file >> x[0] >> x[1] >> y[0] >> y[1] >> R;

  while (file >> a)
    {
    file >> b;
    pairs.push_back (make_cell (a, b));
    }
  file.close();

  Region();
  }


bool loser (Tstate state)
  {
  bool soreloser = false;

  paving.clear();
  pairs.clear();

  ReadIn();

  if (checkanimal (state))
    soreloser = true;

  return soreloser; //true if paved
  }
```

```
void loserfam(Tfam & family)
  {
  bool soreloser = false;

  paving.clear();
  pairs.clear();

  ReadIn();

  for (int i = 0; i < family.size(); i++)
    if (checkanimal (family[i]))
      {
      soreloser = true;
      cout << "Paved" << endl;
      }
    else
      cout << "Not Paved " << i << endl;
  }
```

A.4.2 Paving File Generator

```
#include <iostream>
#include <fstream>
#include <vector>
#include "state.h"
#include "STLmore.h"
#include "Tilecheck.h"
#include "createanimals.h"

using namespace std;

Tstates animals;
vector <Tstate> family;

void MakeFile(char filename[])
  {
  int anim;
  int x[2], y[2];
  int temp, max, t1, t2, t3, t4;
  Tcell corner;
  Tstate state;

  ifstream infile (filename);
  ofstream outfile ("tile.dat");

  //ignore the values that are used to print the paving.
  infile >> temp >> temp >> temp >> x[0] >> x[1] >> y[0] >> y[1];

  if (x[0] + y[0] <= x[1] + y[1])
    max = x[1] + y[1];
  else
    max = x[0] + y[0];

  while (cin >> anim)
    {
    state = animals[anim];
```

```
    corner= urcorner(state);

    if (corner[0] > max)
      max = corner[0];
    else if (corner[1] > max)
      max = corner[1];

    family.push_back(state);
    }

  outfile << x[0] << " " << x[1] << endl;
  outfile << y[0] << " " << y[1] << endl;
  outfile << 2 * max + 1 << endl;

  while (infile >> temp)  //readin the 1
    {
    infile  >> t1 >> t2 >> t3 >> t4;
    outfile << t1 << " " << t2 << " " << t3 << " " << t4 << endl;
    }

  infile.close();
  outfile.close();
  }

int main(void)
  {
  char filename[50];
  bool paved;

  createanimals (6, animals);

  cin.getline(filename, 50);
  MakeFile(filename);

  for (int i =0; i < family.size(); i++)
    {
    paved = loser(family[i]);

    if (paved)
      cout << "Paved" << endl;
    else
      cout << "Not Paved " << i << endl;
    }

  return 0;
  }
```

A.5 PERL Code

This is the PERL code that implemented various other code for tile creation and checking purposes.

A.5.1 Paving Generation

This section contains code which is used to run the paving creating program. The script gives the program different sized boards to consider up to a certain size. When a the program finishes, the script creates a postscript file from the output and renames the file to correspond to the size of the board that had been considered.

```
#!/usr/bin/perl
$filename = "fam.txt";

for ($i = 20;$i <= 30;$i++)
  {
  open FAM, ">$filename";
  print FAM "$i $i 7 8 9 10 11 13 14 15 19 20";
  close (FAM);
  $a=qx[nice -15 ./paving/Paver < $filename];
  print "$i - $i:$a\n";
  if ($a = 1) {qx[mv tile1.dat Pave$i-$i.dat];}
  else {exit();}
  }
```

A.5.2 Polyominoes

This code is used to generate all the pictures of the polyominoes in the directory Poly. Most of the programs that create the other pictures is the thesis are structured similarly.

```
#!/usr/bin/perl
@names = <./Poly/*.txt>;

system "g++ -O4 ../boxa.C -lg2";

foreach $name (@names){
  $poly = $name;
  $poly = ~ s [\.txt]()g;

  $a=qx[./a.out < Poly/$poly.txt];
  print "$poly: $a\n";
  qx[mv boxa.ps poly-ps/$poly.ps];
}
`rm boxa.ps`;
```

A.5.3 Paving pictures

This code reads all the files in the Tiles directory and generates pictures of the pavings and pictures that have pavings. Note that the same program is used for tese pictures and the polyomino pictures.

```
#!/usr/bin/perl
@names = <./Tiles/*.txt>;
```

```
system "g++ -O4 ../boxa.C -lg2";

foreach $name (@names){
  $tile = $name;
  $tile = ~ s [\.txt]()g;

  $a=qx[./a.out < Tiles/$tile.txt];
  print "$tile: $a\n";
  qx[mv boxa.ps tile-ps/$tile.ps];
}

`rm boxa.ps`;
```

Appendix B

Polyomino information

$P_{n,i}$	$\mathsf{DP_A}$	$\mathsf{DP_B}$	$\mathsf{DP_C}$	$\mathsf{DP_D}$	$\mathsf{DP_E}$	$\mathsf{DP_F}$	$\mathsf{DP_G}$	$\mathsf{DP_H}$	$\mathsf{DP_I}$	$\mathsf{DP_J}$
$P_{3,1}$		•	•	•						
$P_{3,2}$	•									
$P_{4,1}$		•	•	•	•	•	•			
$P_{4,2}$	•	•	•	•		•				
$P_{4,3}$	•	•	•	•	•		•			
$P_{4,4}$	•		•	•	•	•	•	•	•	•
$P_{4,5}$	•	•			•	•	•	•	•	•
$P_{5,1}$		•	•	•	•	•	•	•	•	•
$P_{5,2}$	•	•	•	•	•	•	•	•		
$P_{5,3}$	•	•	•	•	•	•	•		•	•
$P_{5,4}$	•	•	•	•	•	•	•	•	•	•
$P_{5,5}$	•	•	•	•	•	•		•	•	•
$P_{5,6}$	•	•	•	•	•	•	•	•		•
$P_{5,7}$	•	•	•	•	•	•	•	•		•
$P_{5,8}$	•	•	•	•	•	•	•	•	•	•
$P_{5,9}$	•	•	•	•	•	•	•	•	•	•
$P_{5,10}$	•	•	•	•	•	•	•	•	•	•
$P_{5,11}$	•	•	•	•		•	•	•	•	•
$P_{5,12}$	•	•	•	•	•		•	•	•	•

Table B.1: Polyominoes and the double pavings that defeat them.

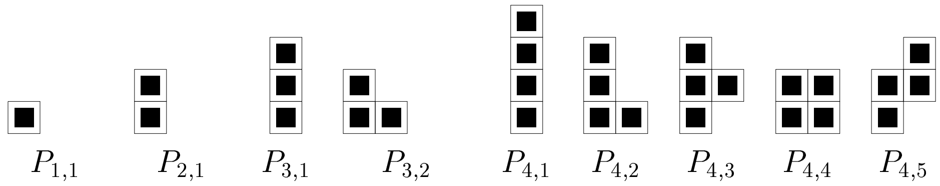


Figure B.1: All congruence classes of polyominoes up to size 4, ordered by size and then by lexicographic order.

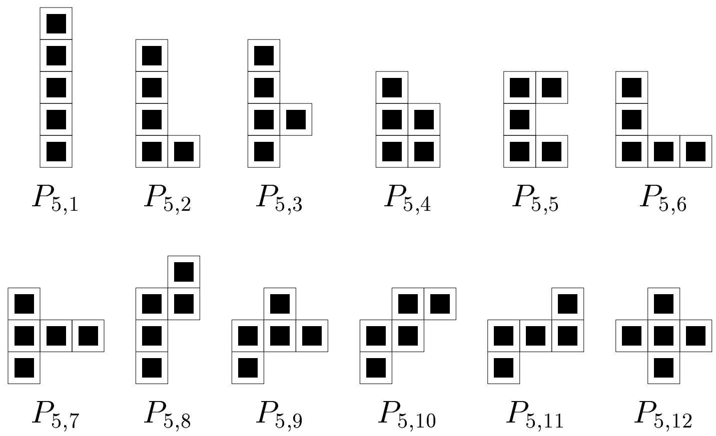


Figure B.2: Congruence classes of polyominoes of size 5, ordered by lexicographic order.

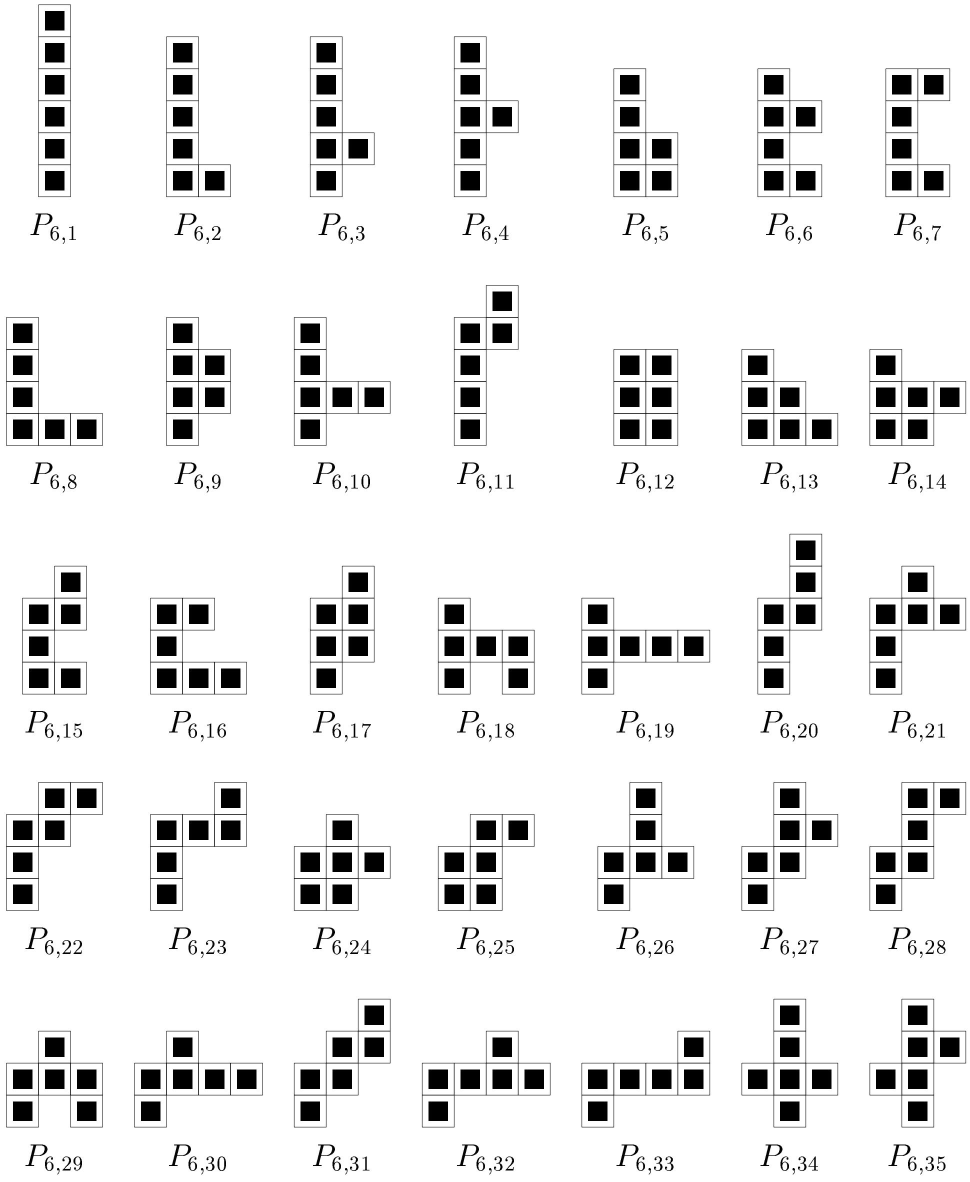


Figure B.3: Congruence classes of polyominoes of size 6, ordered by lexicographic order.

$P_{3,1}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$									
$P_{3,2}$		$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$							
$P_{4,1}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$									
$P_{4,2}$		$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$	$P_{5,8}$	$P_{5,9}$		$P_{5,11}$	
$P_{4,3}$			$P_{5,3}$	$P_{5,4}$			$P_{5,7}$		$P_{5,9}$			$P_{5,12}$
$P_{4,4}$				$P_{5,4}$								
$P_{4,5}$				$P_{5,4}$				$P_{5,8}$	$P_{5,9}$	$P_{5,10}$		

Table B.2: Polyomino ancestry for next immediately sized polyominoes.