WINNING DOTS-AND-BOXES IN TWO AND THREE DIMENSIONS

by Abel Romer

Submitted in partial fulfillment of the requirements for the
Degree of
Bachelor of Arts and Sciences
Quest University Canada

and pertaining to the Question

How can I graduate without actually choosing a Question?

April 8, 2017

ABSTRACT

This paper is divided into three sections. The first section provides basic grounding and mathematical theory behind the children's game of Dots-and-Boxes. It covers basic concepts in combinatorial game theory, including the game of Nim and nimbers, other simple games and the Sprague-Grundy Theorem. We then provide an overview of how these concepts are applied to the game of Dots-and-Boxes, and end with a description of the current mathematical theory on the game. The second section describes the author's design of a Java computer program to play Dots-and-Boxes. Similarly to chess, most Dots-and-Boxes games remain unsolved due to their enormous number of possible game states. This section addresses potential remedies to this time-space problem, and describes their implementation in the author's program. The third section extends the game of Dots-and-Boxes to three-dimensional space and examines three possible ways to play. For each of these cases, the author develops basic equivalencies to the existing strategies for two-dimensional Dots-and-Boxes and examines any significant differences between the games.

CONTENTS

COMBINATORIAL GAME THEORY

THE BASICS

Combinatorial game theory is the area of mathematics devoted to understanding a special class of games called *combinatorial games*. A combinatorial game[1]

1. is played between two players;
2. has perfect information (each player knows everything about the game; there are no chance devices nor hidden pieces);
3. must end (typically when there are no remaining moves; the player unable to make a move loses);[2]
4. cannot end in a draw;

The archetypal example of a combinatorial game is Nim (which we will introduce later). Games that do not fulfill these criteria include poker (no perfect information), Tic-Tac-Toe (may end in a tie), chess (may end in a draw or may not end) and Dots-and-Boxes (does not follow normal play convention). However, we shall see that the results of combinatorial game theory can still help us understand these games. It is also common to see games constrained by impartiality. In an impartial game, both players have access to the same set of moves. Dots-and-Boxes is considered impartial, because any line which may be drawn by one player may also be drawn by his opponent. In contrast, chess is not impartial, since one player may only move white pieces while the other may only move black. Although we do not include impartiality in

[1]There are some exceptions, but for the purposes of our discussion this list will suffice.

[2]This is called *normal play*. In *misere play*, the player unable to move is considered the winner.

our definition of combinatorial games, our discussion will focus on impartial games.

Let us take a moment to play a very simple combinatorial game, which we shall call

Socks-and-Boxes. This game is played between two players, whom we shall refer to as *Left* and *Right*. A cardboard box contains some number of blue and red socks. Left may only remove bLue socks from the box and Right may only remove Red socks. The two players alternate removing any number of their appropriately colored socks. The game follows the normal play convention, so the first player unable to move is declared the loser.

We can easily characterize any Socks-and-Boxes game by the difference $d = blue - red$. This number gives us the move advantage for Left (when positive) or Right (when negative). For example, in a game beginning with 3 blue socks and 1 red sock, Left has two more available moves than Right and may easily win, independent of who moves first. In a game where $d = 0$, the first player to move will inevitably lose the game. Let us take a moment to work through a game G beginning with 2 blue socks and 3 red socks. Supposing Left moves first, the possible sub-positions are either 1 blue and 3 red or 0 blue and 3 red, with advantages of $d = -2$ and $d = -3$, respectively. If Right moves first, the outcomes are either 2 blue and 2 red, or 2 blue and 1 red, or 2 blue and 0 red. These have advantages $d = 0$, $d = 1$ and $d = 2$. We might then express G as the set of all the positions that Left may move to and the set of all positions that Right may move to, or

$$\{-2, -3 \mid 0, 1, 2\}.$$

Clearly removing one sock is always the best move, so we might simplify this result to $\{-2 \mid 0\}$. We already know that the evaluation of the 2 blue, 3 red game is $d = -1$, so $\{-2 \mid 0\}$ somehow equals -1. We shall see why below.

Simple combinatorial games, like Socks-and-Boxes, are conventionally described in the following notation:

$$\{a, b, c, d, ... \mid e, f, g, h, ...\},$$

where $X_L = \{a, b, c, d, ...\}$ represents the numerical evaluations of all possible positions that Left can move to and $X_R = \{e, f, g, h, ...\}$ represents the same for Right. Scores are defined in terms of Left, so Left will always choose the

maximum of X_L and Right will always choose the minimum of X_R. Supposing that $\mathbf{max}\{X_L\} = x_L$ and $\mathbf{min}\{X_R\} = x_R$, we may simplify our notation to

$$\{x_L \mid x_R\},$$

the value of which represents the move advantage (value) of the game. But what is this value? Let us begin with the simplest possible case:

$$\{ \mid \}.$$

Here, neither Left nor Right have any available moves, which means that, by the normal play convention, whomever moves first loses. Any game where the first player to move is guaranteed a loss is considered to have value 0. Therefore, $\{ \mid \} = 0$. Suppose now that we have the game

$$\{0 \mid \}.$$

If Right moves first, he shall lose. On the other hand, if Left moves first, he shall move to a game with value 0, which will cause Right to lose. In either case, Left wins, so we assign this position a value of 1.[3] Building this up recursively, we call

$$\{1 \mid \} = 2, \{2 \mid \} = 3, ..., \{n \mid \} = n + 1.$$

For an intuitive understanding of this result, consider again Socks-and-Boxes. If the game starts with $n+1$ blue socks, Left's best move is to remove 1 blue sock, whereas Right cannot move. We may model this position with the expression $\{n \mid \} = n+1$. Negative integer game-values may be constructed using the same technique, corresponding to advantages for Right.

We may also encounter games with fractional advantages, which may be described by

$$\frac{2p+1}{2^{q+1}} = \left\{ \frac{p}{2^q} \mid \frac{p+1}{2^q} \right\}.$$

To develop a bit of intuition about this result, consider a game with position $\{0 \mid 1\}$. Unfortunately, this situation cannot be modeled by Socks-and-Boxes. Any position where the optimal move by Left leads to a game with value 0 must have $d = 1$. However, the best move by Right from a position with value $d = 1$ is to a position with $d = 2$. To understand $\{0 \mid 1\}$, we must add a new rule to Socks-and-Boxes.

[3]See Donald Knuth's *Surreal Numbers*[3] and Berlekamp, Conway and Guy's *Winning Ways for you Mathematical Plays Vol. 1*[2] for a more detailed examination.

Stacked Socks-and-Boxes. Suppose that inside the cardboard box, socks are arranged in neat little stacks. Now if a player wishes to remove a sock of his own color, he must also remove all of the socks above it as well.

Imagine a one-pile game of Stacked Socks-and-Boxes with a blue sock on the bottom and a red sock on the top. The best move for Left is to a position of value 0, and the best move for Right is to a position of value 1, or $\{0 \mid 1\}$. Let's try to understand what this position evaluates to by adding another blue-red stack and a lone red sock. This new game has value $\{0 \mid 1\} + \{0 \mid 1\} - 1$. Who wins? If Left moves first, he must removes one of the blue-red stacks. Right will respond by removing the top red sock of the remaining stack, leaving a position of value 0, which Left will lose. If Right moves first, he will remove the top red sock of one of the stacks. Left will respond by removing the entirety of the remaining stack, leaving a position of value 0, which Right will lose. We see that whomever moves first shall lose, and so $\{0 \mid 1\} + \{0 \mid 1\} - 1 = 0$. Therefore, $\{0 \mid 1\}$ must equal $\frac{1}{2}$. By a similar argument, we may show that each game of one-pile Stacked Socks-and-Boxes with one blue sock buried beneath q red socks has value $\frac{1}{2^q}$. From this result, it is easy to characterize a game with value $\frac{p}{2^q}$ as p piles consisting of one blue sock buried beneath q red socks. What position, then, would be evaluated to $\{\frac{p}{2^q} \mid \frac{p+1}{2^q}\}$? This could be a game beginning with p piles of one blue sock buried beneath q red socks, and 1 pile of one blue sock buried beneath $q+1$ red socks, which has a value of $\frac{p}{2^q} + \frac{1}{2^{q+1}}$. Some simple algebraic manipulation will show us that this equals $\frac{2p+1}{2^{q+1}}$, and so

$$\frac{2p+1}{2^{q+1}} = \left\{ \frac{p}{2^q} \mid \frac{p+1}{2^q} \right\}.$$

Now, naturally not all games $\{x_L \mid x_R\}$ will be written in one of these canonical forms:

1. $\{ \mid \} = 0$
2. $\{n \mid \} = n + 1$
3. $\{ \mid -n\} = -n - 1$
4. $\{\frac{p}{2^q} \mid \frac{p+1}{2^q}\} = \frac{2p+1}{2^{q+1}}$

To evaluate games not of these forms, we apply the Simplicity Rule.

Simplicity Rule. *For any game*

$$G = \{a, b, c, d, ... \mid e, f, g, h, ...\} = \{X_L \mid X_R\}$$

*G has game-value x where x is the **simplest** number such that $x > x_L$ and $x < x_R$.*

Suppose that each new number is constructed from previously formed numbers. For example, $0 = \{ \mid \}$ was the first number created and is therefore the simplest. Second-level numbers are all those that may be formed from 0, namely 1 and -1. Third-level numbers are those which may be created from 0, 1 and -1: -2, $-\frac{1}{2}$, $\frac{1}{2}$ and 2, respectively. Then, the **simplicity** of a number is its ranking in this order of creation. For example, the Simplicity Rule implies that

$$\{-3 \mid 5\} = 0$$

and

$$\{0 \mid 1\} = \frac{1}{2}.$$

The observant reader will notice that we have so far failed to address the value of the game $\{0 \mid 0\}$. We cannot apply the Simplicity Rule, since there are no numbers strictly between 0 and itself. Let's try to understand what happens in this game. Whoever moves first will move to a 0 position, which loses for his opponent. Therefore, whoever moves first shall win. This scenario does not exist in our games of Socks-and-Boxes and Stacked Socks-and-Boxes. Again, we shall have to make a modification. We shall call this new game Capitalist Socks-and-Boxes, and rather than blue and red socks, we will now play with white socks, which both Left and Right may remove. Our enigmatic game $\{0 \mid 0\}$ now reveals itself as a one-white-sock version of Capitalist Socks-and-Boxes. What number shall we assign to $\{0 \mid 0\}$? Since the first player to move wins, this position is almost the *opposite* of the game $0 = \{ \mid \}$. But what is the opposite of 0? Since there is no obvious answer to this question, we shall define a new type of number, which we shall write as $*$ (pronounced *star*).

This new value $*$ has some unusual properties that take us out of the real number line. Let's explore some of these properties. A quick case analysis of moves shows us that the sum of two games of value $*$ (i.e. a game of Capitalist Socks-and-Boxes played with two stacks of one white sock each) is

$$* + * = \{0 \mid 0\} + \{0 \mid 0\} = 0$$

So it appears our new number is somehow half of zero, but not zero. It is as though $*$ is a cousin of zero on a parallel number line. What if we try to combine these number lines by looking at the value of $x + *$ for some integer x? It turns out that

$$x + * = \{x \mid x\} = x*\,^{4} \tag{1.1}$$

Explanation. Let us invent yet another variation of Socks-and-Boxes. We shall call it Socialist Socks-and-Boxes, and it will be a combination of Stacked Socks-and-Boxes and Capitalist Socks-and-Boxes (i.e. we will play with blue, red and white socks). Consider the game of Socialist Socks-and-Boxes played with one stack of one white sock and one stack of x blue socks (the Finnish game, if you will). We may describe this game as $* + x$. Suppose Left moves first. He shall remove the one white sock, eliminating all possible moves for Right. Now suppose Right moves first. He has only one available move, and that is to remove the one white sock. In both cases, all that remains is the stack of x blue socks, and so $x + * = \{x \mid x\}$. □

It appears that by adding a game of value $*$ to a game of Stacked Socks-and-Boxes, we end up with a game of Capitalist Socks-and-Boxes. Capitalist Socks-and-Boxes has the property that both players can remove any sock they wish; in other words, Capitalist Dots-and-Boxes is an impartial game. Let us take some time to examine Capitalist Socks-and-Boxes in greater detail.

IMPARTIAL GAMES

Quite intentionally, it turns out that Capitalist Socks-and-Boxes is precisely equivalent to the archetypal impartial game: Nim.

Nim. Nim is played between two players. In front of both players are a number of differently-sized stacks of coins (rather than white socks). Players alternate turns removing any positive number of coins (including all) from one stack. The player to remove the final coin is declared the winner.

We should first note that a stack of n coins in a game of Nim has value $\{n \mid n\} = n*$. This is because the optimal move for both players is to remove the entire stack and win immediately. Therefore, any given game of Nim may

[4] It is conventional to write $x + *$ as $x*$.

be described by the sum of the heights of its various stacks. The question that we must now resolve is how to calculate this sum.

Remark. *Any game consisting of two equally-sized stacks of coins has a value of 0.*

Proof. Suppose we play a game with two stacks of one coin each. The first player must remove the first stack and the second player must remove the second stack. This is a second player win, and therefore has a value of zero. Assume that all games with equal stacks up to size n have value zero. Now consider a game with two stacks of height $n+1$. The first player must remove some number k coins from one stack. His opponent responds by removing k coins from the other stack. We are left with two stacks of equal height $n-k$, which, by our induction hypothesis, must have value zero. □

This strategy of mimicking one's opponent's moves is formally known as the *tit-for-tat* technique. Before we go about finding the value of the sum of unequal stacks of coins, we must introduce the new concept of the $\mathbf{mex}(S)$.

Definition. *The* $\boldsymbol{mex}(S)$ *is the minimum excluded number of the set* S*.*

For example, the $\mathbf{mex}(0, 1, 2, 4, 7) = 3$. We might think of S as the set of values of possible positions that a player may move to in a somewhat unusual game of Nim. In the previous example, $S = \{0, 1, 2, 4, 7\}$. Let us look at the game of Nim

$$\{0*, 1*, 2*, 4*, 7* \mid 0*, 1*, 2*, 4*, 7*\}.$$

This is like a normal game of Nim with one stack of 3 coins, except in addition to removing coins, a player may also add 1 or 4 coins. However, this ability to add coins turns out to be meaningless. Because one of the players will have a winning strategy that does not require adding coins, any move made by his opponent to counteract his winning strategy by adding coins can simply be negated by removing the same number of coins. Therefore, any game represented by a set S of nimbers is really just a disguised version of a game of normal Nim with one stack of $\mathbf{mex}(S)$ coins. Now, suppose we are playing a game of Nim with two stacks of coins of heights 1 and 2. On Left's turn, he may move either to position $0* + 2*$, $1* + 1*$ or $1* + 0*$, which leaves Right with

$$1* + 2* = \{0*, 1*, 2* \mid 0*, 1*, 2*\} = G = \mathbf{mex}(0*, 1*, 2*)$$

Of course, we can see that G is simply our game notation for $3*$. Let's generalize this result a bit:

Theorem 1.2.1. [5] The *nim-sum* of two nimbers $a*$ and $b*$ is given by the $\mathbf{mex}(a' * + b*, b' * + a*)$ where $a'*$ and $b'*$ represent the sets of all nimbers less than $a*$ and $b*$, respectively.

An easier technique for computing the sum of nimbers is to write each nimber as a sum of distinct powers of two, cancel all pairs of equivalent numbers, and calculate the sum of the remaining numbers. For example, if we wanted to calculated $5* + 7*$, we would re-write this sum as $1* + 4* + 1* + 2* + 4*$, cancel the pairs of $1*$'s and $4*$'s, and be left with $2*$. This process is the same as calculating the binary **XOR** of the numbers, but is simpler than writing each number out in binary notation.

THE SPRAGUE-GRUNDY THEOREM

We have now arrived at a point where we can understand the Sprague-Grundy Theorem, which states that all impartial games may be reduced to the sums of games of Nim. This result will be invaluable in our analysis of Dots-and-Boxes.

Sprague-Grundy Theorem. *Every impartial game may be characterized as the sum of games of Nim with additive moves (as we saw above in our explanation of the* ***mex****). Because every game with additive moves is equivalent to a normal game of Nim (due to the* ***mex*** *rule), each impartial game may be written as the sum of nimbers. Furthermore, because nimbers are closed under addition, every impartial game may be characterized as a one-stack game of Nim.*

Explanation. Suppose we are playing the simplest version of some arbitrary impartial game G. In this simplest version, there are no available moves to be made; whoever moves first immediately loses. This is equivalent to the game of Nim with no coins. Now suppose that we are playing a slightly more complex version of G. Here, a player may end the game in one move, or extend the game by making some sort of additive move. This version is equivalent to the game of Nim with one stack of one coin. By continuing

[5] A formal proof of this result may be found in *Winning Ways*.

to increment the size of G and applying the **mex** rule to account for the additive moves, we can recursively construct nimbers for each position. We shall see examples of this in the following section. □

TWO MORE COMPLICATED GAMES

Let us look at two examples of how nimbers can help us solve other impartial games.

KAYLES[6]

Kayles. Kayles is a two-player impartial game played with a row of bowling pins and a ball. Players take turns rolling the ball at the pins. In our contrived universe, players are talented enough to either knock down one pin or any two adjacent pins. As soon as someone is unable to knock down a pin, that player is declared the loser.

Suppose we decide to play a game of Kayles with 11 pins (K_{11}). Let us consider the possible outcomes after the first player rolls his ball. Either he knocks over a pair of adjacent pins, leaving positions

$$K_9,\ K_1 + K_8,\ K_2 + K_7,\ K_3 + K_6,\ \text{or}\ \ K_4 + K_5;$$

or he knocks over one pin, leaving positions

$$K_{10},\ K_9 + K_1,\ K_8 + K_2,\ K_7 + K_3,\ K_6 + K_4,\ \text{or}\ \ K_5 + K_5.$$

Our analysis now requires that we determine nimbers for each of these smaller positions, and calculate the **mex** of all these nimbers. The resulting nimber will represent the value of K_{11}. How might we tackle this daunting task?

Our only solution is to build up nim-values of progressively larger boards recursively. We begin with positions $K_0 = 0*$ and $K_1 = 1*$. From here, way may determine the value of

$$K_2 = \mathbf{mex}(K_0,\ K_1) = \mathbf{mex}(0*, 1*) = 2*$$

[6]In this section, we will provide an overview of the analysis of Kayles. Certain calculations are proofs are skipped, as they tend to direct attention away from our ultimate goal of understanding Dots-and-Boxes. As in previous sections, the interested reader is directed to *Winning Ways for your Mathematical Plays Vol. 1* for a more detailed explanation.

n	0	1	2	3	4	5	6	7	8	9	10	11
	0	1	2	**3**	1	4	**3**	2	1	**4**	2	**6**
12	4	1	2	**7**	1	4	**3**	2	1	**4**	**6**	7
24	4	1	2	8	**5**	4	7	2	1	8	**6**	7
36	4	1	2	**3**	1	4	7	2	1	8	2	7
48	4	1	2	8	1	4	7	2	1	**4**	2	7
60	4	1	2	8	1	4	7	2	1	8	**6**	7
72	4	1	2	8	1	4	7	2	1	8	2	7
84	4	1	2	8	1	4	7	2	1	8	2	7
96	4	1	2	8	1	4	7	...				

Table 1.1: The periodicity of Kayles

$$K_3 = \mathbf{mex}(K_1,\ K_2,\ K_1 + K_1) = \mathbf{mex}(1*, 2*, 0*) = 3*.$$

Eventually, we arrive at

$$K_{11} = \mathbf{mex}(2*, 4*, 5*, 0*, 3*, 0*, 1*, 0*, 2*, 5*, 0*) = 6*.^{7}$$

This tells us that K_{11} is equivalent to the game of Nim with one stack of 6 coins. The first player may win this game by removing the entire stack; he must reduce $6*$ to $0*$. What Kayles move accomplishes this? We have seen the nim-values for all subpositions of K_{11} in our calculation of the **mex** above. Had we included the calculations of K_4 through K_{10}, we would know that the nim-values $0*$ refer to positions $K_8 + K_1$, $K_7 + K_2$, $K_5 + K_5$ and $K_6 + K_3$.

Although it is convenient to know the winning moves for K_{11}, it would be far more useful to know the winning values for K_n. Just as we built up a table for K_{11}, we might also build up a larger table. This is exactly what we have done in Table 1.1, borrowed from *Winning Ways for your Mathematical Plays*, which illustrates the nim-values of Kayles for K_0 through K_{102} and shows a regular periodicity of 12, starting from $n = 70$.

DAWSON'S KAYLES[8]

Dawson's Kayles. Dawson's Kayles is a simpler version of Kayles where players are only permitted to knock down two adjacent pins.

[7] Further explanation and more detailed calculations may be found in *Winning Ways.*

[8] This name comes from another game, called Dawson's Chess. We shall not discuss it here, but its rule and analysis may be found in *Winning Ways.*

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	0	**0**	1	1	2	0	3	1	1	0	3	3	2	2	4	**0**	5
17	**2**	**2**	3	3	0	1	1	3	0	2	1	1	0	4	5	**2**	7
34	4	**0**	1	1	2	0	3	1	1	0	3	3	2	2	4	4	5
51	5	**2**	3	3	0	1	1	3	0	2	1	1	0	4	5	3	7
68	4	8	1	1	2	0	3	1	1	0	3	3	2	2	4	4	5
102	5	9	3	3	0	1	1	3	0	2	1	1	0	4	5	3	...

Table 1.2: The periodicity of Dawson's Kayles

In our analysis of Dawson's Kayles, we shall proceed precisely as we did in regular Kayles, first by building up nim-values one by one, and then by looking for some emergent periodic pattern. We begin with D_0 and D_1 (that is, games consisting of 0 and 1 pins, respectively), both of which equal $0*$, since only two pins may knocked over. From here, we may calculate

$$D_2 = \mathbf{mex}(D_0) = 1*$$

$$D_3 = \mathbf{mex}(D_1) = 1*$$

$$D_4 = \mathbf{mex}(D_2,\ D_1 + D_2) = 2*$$

and so on, eventually building Table 1.2.

As we can see, Dawson's Kayles, after correcting for a few extraneous values (written in **bold**), settles down into a period of 34. The explanation for our certainty in assigning this periodicity is essentially the same as it was for Kayles (see above).

We are now equipped to tackle Dots-and-Boxes!

DOTS-AND-BOXES

Dots-and-Boxes. Dots-and-Boxes was a very popular game amongst my primary school friends, and maybe yours as well. It is played between two players on a square grid of dots. Players alternate turns connecting vertically or horizontally adjacent dots. Whenever a square is formed, the player to form it tallies a point, writes their initial in the square and moves again. The player with the majority of points (initialized squares) is declared the winner[9].

[9]This section relies heavily on the work done by Dr. Elwyn Berlekamp. A more detailed analysis of the game can be found in his book *The Dots and Boxes Game: Sophisticated*

How might we go about winning this game? There are various levels of strategic complexity at which Dots-and-Boxes may be played. We shall analyze these strategies from the most basic to what might seem like the intractably complicated.

THE NAIVE APPROACH

Suppose we are playing a 4-by-4, 16 box game. The beginning play is rather dull, as players alternate connecting dots without forming the fatal third edge that will give away a square to their opponent. Inevitably, the game eventually reaches a state where one poor soul must form the third edge, providing his opponent with a cascading chain of moves, as shown below. Once these boxes have been formed, his opponent is then forced to form the third edge of some other box, and the roles switch. This process continues until all of the boxes are completed.

Each player, when required to draw the third edge of a box, elects to draw the one that supplies his opponent with the fewest number of boxes possible. For example, in the game in Figure 1.1, the first chain of boxes has length 4, the second has length 5, and the last has length 7. This gives the first player to capture a chain (whom we shall later refer to as the *controlling player*) a victory with a score of 11 to 5. Can the controlling player do better than this?

DOUBLE-DEALING AND LONG CHAINS

Let us consider the same game as in Figure 1.1, now played against a more conniving controlling player (Figure 1.2).

We see that rather than take all four available boxes in the first chain, the conniving controlling player took just two, and left two for his opponent. This move forced his opponent to open the next chain, where again the conniving player left the final two boxes. Once more, his opponent was forced to open the final chain, which was entirely taken by the conniving player. Let's tally up the scores:

Child's Play[1].

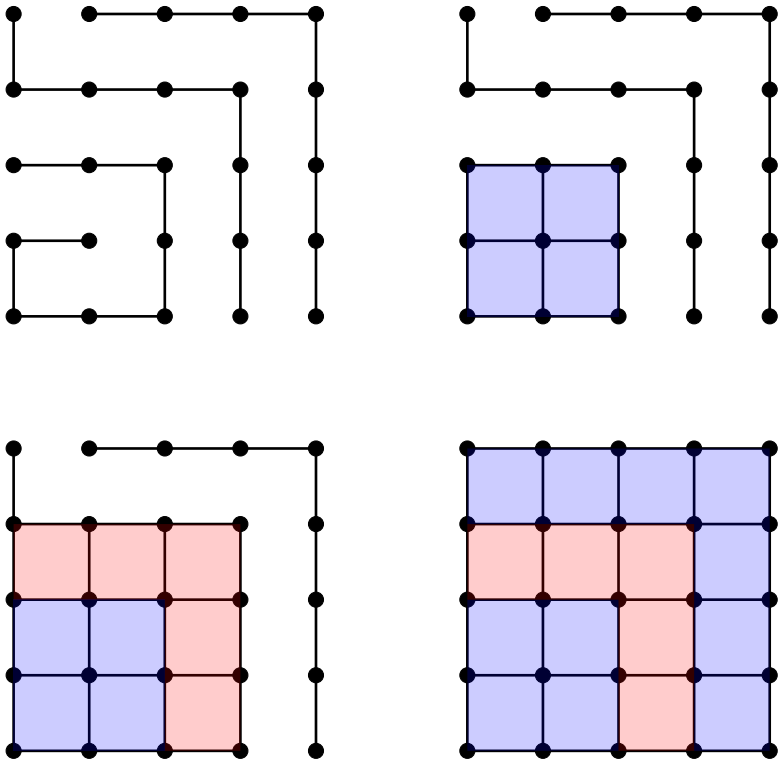

Figure 1.1: A naive game of Dots-and-Boxes

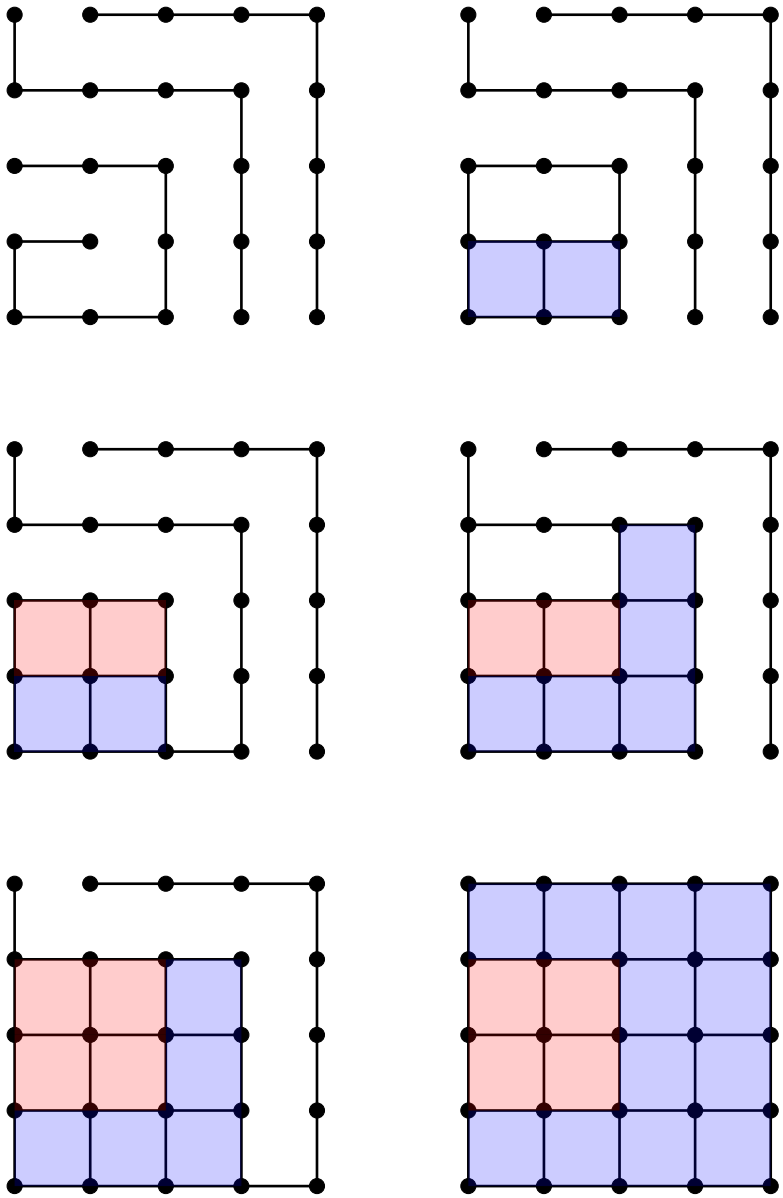

Figure 1.2: The conniving player's double-dealing strategy

Conniving player	Opponent
$4-2$	2
$5-2$	2
7	0
12	4

The conniving player has improved his overall score by one point. More importantly, however, the controlling player who follows this strategy will serve up a mere $2(n-1)$ boxes to his opponent from a position with n chains.

Let us take a moment to define more precisely this notion of chains. From here on out, we shall use *long chains* to refer to chains of 3 or more boxes that may be captured by a single player in one turn. We limit the length of long chains to at least 3 because these are all the chains to which a conniving player might apply the double-dealing strategy we saw above.

Clearly, it is advantageous to be the controlling player. What strategies can we use to gain control? The most simple strategy (which I have found helpful when playing some online Dots-and-Boxes AIs) is the

Long Chain Rule. *In order to become the controlling player, try to make the number of initial dots plus eventual long chains even if you are Player 1 and odd if you are Player 2.*

Proof. In Dots-and-Boxes, the player who has control in the endgame will also be the player to complete the final box. By calculating the number of turns in a given game, we may determine who will make the last move and, consequently, who has control in the endgame. This number happens to be a function of the number of starting dots and the number of eventual long chains. Suppose we are playing a game on a grid of $n \times m$ dots (that is, an $(n-1)(m-1)$-box game). The number of turns in the game depends on the number of moves in the game, the number of boxes in the game (since each move that completes a box does not end a turn), and moves that complete two boxes simultaneously (double-dealt moves). More specifically,

$$\text{Turns} = \text{Moves} - \text{Boxes} + \text{Double-dealt moves}.$$

But we already know that there are $(n-1)(m-1) = mn - n - m + 1$ boxes in the game and we may easily calculate the number of moves to be $n(m-1) + m(n-1) = 2nm - n - m$. We now have

$$\text{Turns} = 2nm - n - m - (mn - n - m + 1) + \text{Double-dealt moves}$$

$$= mn + 1 + \text{Double-dealt moves.}$$

Since we started with nm initial dots, the final result is

$$\text{Turns} = \text{Dots} + \text{Double-dealt moves} + 1.$$

Finally, recall that in a typical game, the number of double-dealt moves is precisely one less than the number of long chains[10]. But if this is true, then

$$\text{Turns} = \text{Dots} + \text{Long chains.}$$

□

However, this is a bit of a vague piece of advice. How can we articulate this strategy in more precise terms?

STRINGS-AND-COINS

Let us begin by presenting an alternative way to describe Dots-and-Boxes. In this dual form, called Strings-and-Coins, each prospective box is replaced by a coin, and each undrawn edge is replaced by a string. A move in Dots-and-Boxes corresponds to cutting a string in Strings-and-Coins, and the capture of a box corresponds to the removal of a completely detached coin. All the same rules of Dots-and-Boxes apply. Translating Dots-and-Boxes into Strings-and-Coins can be useful for assessing the location of long chains and, as we shall see, invaluable once we develop a more complex theory of the game. Additionally, Strings-and-Coins removes the restrictions that the Dots-and-Boxes grid places upon us. Any graph[11] can be translated into a game of Strings-and-Coins and analyzed.

NIMSTRING

As you may recall, our toolkit for solving games is really only suited for impartial combinatorial games, and unfortunately, Dots-and-Boxes (and Strings-and-Coins) do not quite apply. Although impartial, they do not follow the normal play convention. It order to satisfy this constraint, we must tweak them ever so slightly and hope that whatever information we glean from this

[10]The double-dealing strategy leaves a double-dealt move after every long chain except the final one.

[11]For those unfamiliar, a graph is a number of nodes connected by lines.

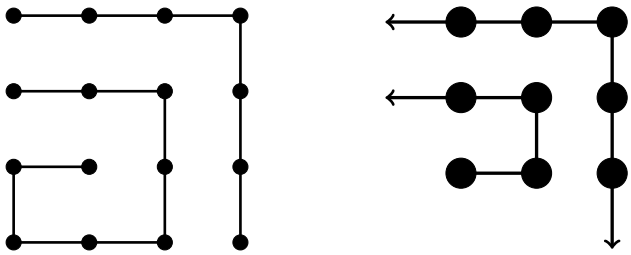

Figure 1.3: A Dots-and-Boxes position and its Strings-and-Coins dual

new game will still provide us with helpful insight. This new game is called Nimstring, and differs from Strings-and-Coins only in its manner of scoring. Rather than claiming the minority of coins, the loser in Nimstring is the player who cannot make a legal move on his turn. In most cases, this happens immediately after a player removes the final coin, as the rules require that he make another turn when there are no remaining strings to be cut. (It is conceivable that a Strings-and-Coins game might contain a string that is unattached to any coin, but as this has no parallel position in Dots-and-Boxes, we shall ignore this possibility.) As our first order of business, we shall show that

Claim. *Nimstring is just a special case of Strings-and-Coins.*

Proof. Suppose we are presented with a Nimstring position G for which we have some winning strategy. We may then construct a Strings-and-Coins position G' by adding a long chain of coins to G such that the number of coins in the long chain exceeds the number of coins in G. The winning strategy for the game G' is exactly that of G, because the loser of Nimstring must cut the first string of the long chain, thereby providing us with the majority of the coins. □

For wider application, we might augment this proof to allow for n long chains and m long loops, rather than one giant chain. In this case, as long as the number of coins in these chains and loops exceeds the number of coins in G plus $4n + 8m$, the winning Nimstring strategy applies. In general, when applying Nimstring analysis to a game of Dots-and-Boxes, one should simultaneously aim to win the corresponding Nimstring game, and leave a good number of long chains laying about. As it stands, we shall direct our focus on winning at Nimstring.

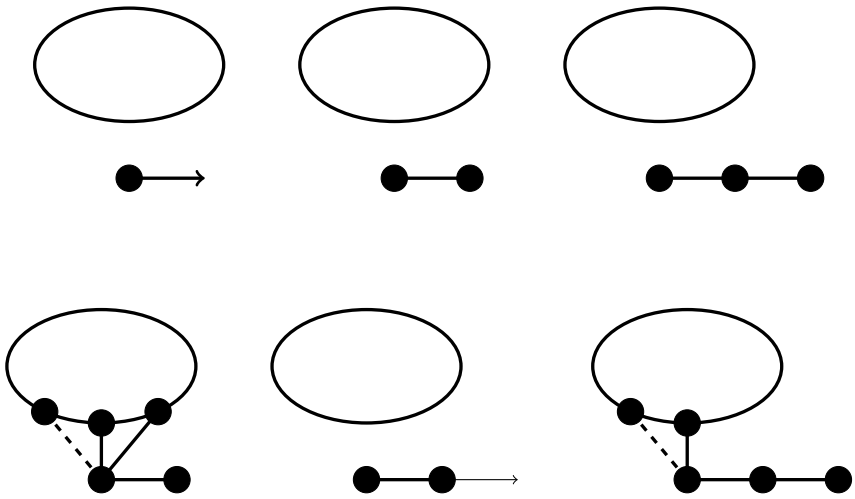

Figure 1.4: The six unique types of capturable coins

Our goal here, as it was with other impartial games, is to determine nim-values for every Nimstring position, and hopefully discover some sort of periodicity. As we shall see, this is going to be a monumental task.

In Nimstring, as in Dots-and-Boxes and Strings-and-Coins, there are two different types of moves: those that capture a coin (or box) and those that do not. Let us take a look at the possible circumstances of capturing moves (Figure 1.4).

In Figure 1.4, each ellipse above the pictured coins represents the rest of the game (G) of Nimstring, and dashed lines indicate additional strings that may or may not be present.

Claim. *In the first four positions, the player to move might as well capture the available coins, as doing so will in no way adversely affect his strategy.*

Explanation. Suppose the player to move has a winning strategy that involves moving somewhere in the game of Nimstring without capturing the available coins. If he chooses to capture these coins, this winning move with still be available to him, so there is no reason for him not to do so. □

Claim. *In games with the last two types of capturable coins, the player to move is guaranteed a winning strategy.*

Proof. Suppose the player to move has a winning strategy in G. In this case, he would do well to capture the free coins before moving on to his

winning strategy in G. His winning move in G will be equally available before and after these captures. Now suppose that the player to move will lose by moving in G. He can avoid this circumstance by cleverly stealing his opponent's strategy and forcing him to move in G first. He does this by cutting the string of the coin with degree 2 and thereby double-crossing his opponent. □

From this point onwards, we shall refer certainly winning positions and moves that lead to such positions as *loony* and label them with a ☽.

Recall that our technique for solving other impartial games was to find the **mex** of all subpositions after a move was made. This technique is no different with Nimstring, but is made far more complex by the sheer number of possible subpositions after a given move. Nevertheless, we shall present the algorithm for finding a position's nim-value, before moving to more practicably applicable options.

1. *The Nimstring position with no available strings to cut has value* $0*$.
2. *The position with capturable coins of the first type has the same value as the position after those coins are captured.*
3. *The position with capturable coins of the second type has value* ☽.
4. *The position with no capturable coins has value equal to the* ***mex*** *of the values of its subpositions.*

Explanation. If a position has no strings to cut, then the player whose turn it is cannot move, and subsequently loses. As we have shown above, there is no reason not to take capturable coins of the first type, so any move in a position with these coins ends in the same position as a move made without these capturable coins. As defined above, positions with capturable coins of the second type are loony. □

At this point, you might ask,"Now what? We've solved Nimstring, so let's just translate these results to Dots-and-Boxes and be done." Unfortunately, this is not quite accurate. Although we have a formula for calculating the nim-values of any Nimstring position, there is no current research that has developed a complete documentation of all nim-values for all positions. This is likely due to the fact that the calculation of nim-values runs in $O(n!)$ time on games with n strings, which is simply not feasible for large n. Despite this disappointing setback, we shall persevere, now with the help of computers!

THE PROGRAM

OTHER GAMES

Apart from our cursory overview of combinatorial games in the beginning of this paper, our focus has been all but entirely directed on the theory of impartial games. For the sake of balance, we shall spend a moment here discussing other types of games including, of course, Dots-and-Boxes. Complementing impartial games, the other major class of combinatorial games are those for which opposing players move unique pieces. These games are called *partizan games*, and include many well-known classics like chess, checkers, go, and tic-tac-toe. The theory of partizan games is not nearly as well-developed and clean as that of impartial games, nor is it the focus of this paper. However, there are certain partizan games that have garnered the attention of computer scientists and programmers, and it is the structure of programs designed to play these games that we shall imitate as we design our program to play Dots-and-Boxes. Chess is the archetypal example of such a game.

Chess is notoriously difficult to fully analyze, as there are unfathomably many possible games to trudge through in order to dig up a winning move. Mathematician Claude Shannon, in his influential 1950 paper *Programming a Computer for Playing Chess*, placed a conservative lower bound on the number of possible chess games at 10^{120}, well beyond the practical limitations of computing. As a result, chess programs were designed to look a number of moves ahead and then employ a *heuristic function* to assess the position, before passing that heuristic value back to the start. As this area of study increased in popularity, a number of improvements were made to this basic structure, allowing for a greater depth of exploration before the heuristic was called.

As it happens, the game of Dots-and-Boxes presents an similar challenge when it comes to analysis. A starting board with $n \times m$ dots has, at the

beginning, a total of $2nm - n - m$ possible moves, with this value decreasing by 1 for every cut string. This means that the number of possible games to be played is $(2nm - n - m)!$, a ghastly number for even moderately small n, m.[1] It is for precisely this reason that we employ the same techniques as early chess programmers in our Dots-and-Boxes program.

Before we dive in, we should make a brief note about some terminology. The following algorithms are designed to efficiently search and *prune* a *game tree*. The game tree is a structure used to define a particular game. It begins at a certain game state, and branches off for every possible sub-game after a move is made. The *leaf nodes* are the final game states, in this case when all boxes are claimed. To prune the game tree is to remove a certain branch of moves by determining their outcome to be less desirable. We will describe the efficiency of some algorithms using the notation $O(n)$. This is conventional in computer science and can be thought of as the number of options the program must explore on a given input size n.

THE BASIC STRUCTURE

The most simple algorithm for searching ahead in a game tree is the *minimax algorithm*. It is described in pseudocode below:

[1] To be perfectly clear, a game played on a 4×4 board has $24! = 6.2 \times 10^{23}$ possible game states.

input : CurrentBoard, depth, true
output: heuristic value of CurrentBoard

```
1  Function Minimax(node, depth, MaximizingPlayer)
2      if depth is 0 or node is terminal then
3          return heuristic value of node;
4      end
5      if MaximizingPlayer then
6          BestValue ← −∞;
7          foreach child of node do
8              v ← Minimax(child, depth − 1, false);
9              BestValue ← max(v, BestValue);
10         end
11         return BestValue;
12     end
13     else
14         BestValue ← ∞;
15         foreach child of node do
16             v ← Minimax(child, depth − 1, true);
17             BestValue ← min(v, BestValue);
18         end
19         return BestValue;
20     end
```

This algorithm explores all game states down to a given depth, heuristically evaluates each one, and then works backwards to determine what move will direct the player towards the optimal game state. Although this works, it is clearly not ideal, as it must search $\frac{n!}{(n-depth)!}$ total game states before coming to a conclusion. One way to cleverly modify the minimax algorithm is to implement *alpha-beta pruning*.

input : CurrentBoard, depth, true, -∞, ∞
output: heuristic value of CurrentBoard

```
1  Function AlphaBeta(node, depth, MaximizingPlayer, α, β)
2      if depth is 0 or node is terminal then
3          return heuristic value of node;
4      end
5      if MaximizingPlayer then
6          v ← −∞;
7          foreach child of node do
8              v ← max(v, AlphaBeta(child, depth − 1, false, α, β));
9              α ← max(α, v);
10             if β ≤ α then
11                 break;
12             end
13         end
14         return v;
15     end
16     else
17         v ← ∞;
18         foreach child of node do
19             v ← min(v, AlphaBeta(child, depth − 1, true, α, β));
20             β ← min(β, v);
21             if β ≤ α then
22                 break;
23             end
24         end
25         return v;
26     end
```

The idea here is to avoid looking down branches that will inevitably be ignored due to better available options. For example, suppose the maximizingPlayer can choose between one game state evaluated at 6 and another game state for which the minimizingPlayer can evaluate to be either 2 or some yet unexplored option n (Figure 2.1). We needn't investigate option n because we know that the minimizingPlayer will provide no option better than 2, and the maximizingPlayer already has the better choice of 6. In the worst case scenario, if game states are searched so that no search options are cut off, alpha-beta pruning works exactly as efficiently as minimax.

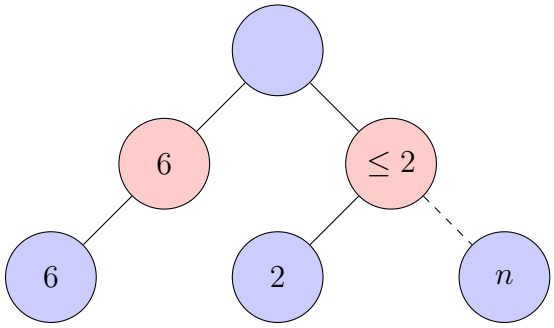

Figure 2.1: A simple game tree with $\alpha\beta$ pruning.

However, in an optimal scenario, where game states are searched in an order that promotes the most cut-offs, roughly half of the game states at each level are ignored, leaving us with an operating time of approximately $O(\frac{\frac{n}{2}!}{(n-depth)!})$. We shall use an implementation of alpha-beta pruning in our Dots-and-Boxes program.

Why stop here? We are still at the whim of the order in which we examine game states. How can we ensure that we take full advantage of minimax algorithm?

ACCOUNTING FOR SPACE AND TIME

Here we will introduce two of the techniques used to maximize time efficiency in game algorithms. The author's implementation of these techniques, along with all other code, is included in the appendix.

TRANSPOSITION TABLES

In many games, and especially Dots-and-Boxes, it commonly happens that previously explored game states show up in multiple places in a search tree. One can easily imagine a circumstance, for example, where a Dots-and-Boxes player first draws a horizontal line in the upper left of the playing board, and his opponent responds by drawing a vertical line in the same upper left corner. These two moves might easily be reversed and still lead to the same outcome, requiring the final game state to be explored an additional time. More generally, a game state reached after n moves could have been reached in $(n-1)!-1$ other ways, leading to a great deal of redundancy in our

alpha-beta algorithm. To avoid this situation, we add a sort of dictionary at the beginning of our algorithm that contains all of the previously explored game states. Prior to beginning any search, the algorithm checks if the game state that it is currently examining is in the table. If so, it simply spits out that state's associated heuristic value and moves on.

This dictionary is called a Transposition Table, and it uses an ingenious trick to reference its entries. When a new board is created at the beginning of a game, each possible move is assigned a unique 64-bit string, called a Zobrist hash. Every time a move is made, its corresponding 64-bit string is **XOR**ed[2] onto the current value of the board. This new board hash is then used to index the board's heuristic value (along with some other information) in a HashMap. When we want to check whether a board position has been previously explored, we simply call the HashMap with the board position's Zobrist hash. If we return a value, it is the value of that board position. Otherwise, it has not yet been explored.

In the author's implementation, there was a small hiccup in this process. It happens that in Dots-and-Boxes, identical game states may not have identical values, as the overall scores may differ. To prevent the wrong heuristic value from being pulled from the Transposition Table, the author added another condition that required not only the Zobrist hash to match, but also the current score. This reduced the efficacy of the Transposition Table, but not too significantly.

ANALYSIS TECHNIQUES

At this point, we still need to implement a heuristic function. How might we have the computer determine which game state is the most preferable? More directly, how can we quantify in a simple series of steps the thought process behind choosing what move to make? When do we know if a position is "good" or "bad?" In the previous section, we found that the Long Chain Rule was a pretty simple metric that worked surprising well. We also learned about Strings-and-Coins and Nimstring and nimbers, but nimber calculation turned out to take $O(n!)$ time, which we simply cannot handle. The Long Chain Rule worked well, though. Is there a way for us to apply it?

[2] **XOR** is a binary operation that acts like addition without carrying. It is also its own inverse.

There are a few significant issues with implementing a heuristic based on the Long Chain Rule:

1. Oftentimes we, as humans, see the inevitable formation of a long chain long before it is completed. How can we communicate these nuances to a computer program?

2. Suppose we reach a position where the parity of long chains is certain, but the precise number is not. How can we holistically determine the parity without simply counting long chains?

3. What type of algorithm can effectively find the parity of long chains without wasting so much time that the benefit of using a heuristic is lost?

The first two points can, to an extent, be shunted aside, if we are willing to have an imperfect heuristic[3]. Even in the most brutish of techniques, where we choose only to evaluate positions that are strictly composed of long chains, we will occasionally benefit from our heuristic. The truly distressing question is whether the time we use to actually determine the heuristic pays off (in the form of a more competitive AI), or whether it is better spent on a deeper search. We shall attempt to quantify the difference between these scenarios in the following paragraphs.

One simple way to search for long chains is to start on a coin with degree 1 and follow the path of coins it is connected to until either (a) we reach a coin with degree > 2 or (b) we reach another coin with degree 1. If (a), then we discard all visited coins and move on to the next coin with degree 1. If (b), then we discard all visited coins, count one long chain (if we looked at ≥ 3 coins), and move on to the next coin with degree 1. Fortunately, this is not a particularly time-intensive process; if we start with a board of n coins, we need only look at each one once, leaving us with a worst case scenario of $O(n)$. However, we must run the heuristic for every node at some depth d, so we must actually evaluate (in a worst case scenario) $\frac{n!}{(n-d)!}$ total boards. This puts the upper bounded cost of our heuristic at roughly $n(\frac{n!}{(n-d)!})$; not a negligible price.

How much deeper can we search for the price of our heuristic? This depends heavily on the starting depth. Each additional increase in depth d

[3]Which we should be, since a heuristic by definition need only be adequate.

adds

$$\frac{n!}{(n-d-1)!} - \frac{n!}{(n-d)!}$$

game states to check. If $d \approx \frac{1}{n}$, then this number is quite large, but for values of d either near n or near 0, this number is small. Clearly, the benefit of this sort of heuristic is up-in-the-air.

The author elected to use a simple heuristic that returns the difference in score between the computer and its opponent. This heuristic functions well, although it results in some very predictable behavior during the opening moves. Because the author has not yet implemented a move-ordering heuristic, if no points are scored during the computer's assessment of the game, it simply goes about cutting strings clockwise around coins, starting in the SW corner of the board and moving northwards row by row. This does not appear to be a significant impediment, although it does make the computer look rather foolish. In order to augment this simple heuristic, the author has also included a function that increases the depth of search depending upon the number of moves remaining in the game. This allows the computer to play quickly during the low-scoring opening, and still search deep into the tree in the later stages of the game.

WHAT ABOUT 3D?

HOW SHOULD WE PLAY?

There are three possible interpretations of what it means to play three-dimensional Dots-and-Boxes. We shall introduce all three and discuss how each differs from traditional Dots-and-Boxes.

1 string, $\geq$ 2 coins. We play on a square three-dimensional grid of dots. Players alternate turns connecting adjacent dots. Whenever a square is completed, the player to complete the square tallies a point and moves again. Whoever scores the most points wins.

This version appears trivially equivalent to traditional Dots-and-Boxes at first glance. However, it is complicated by the presence of single moves that complete more than two boxes. We shall see that this makes the process of gaining control significantly more difficult.

1 string, 2 coins. Again, we play on a square three-dimensional grid of dots. However, rather than connecting adjacent dots, players take turns shading in squares (or faces). Whoever completes shading in all six faces of a cube tallies a point and moves again. The game ends when all faces are shaded. Whoever has the most points wins.

This version can be reduced to a simple game of Strings-and-Coins in three dimensions. There are no significant changes.

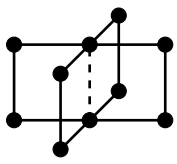

Figure 3.1: Scoring four points in one move.

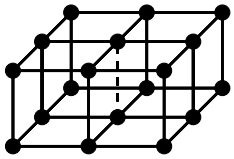

Figure 3.2: Four birds, one stone.

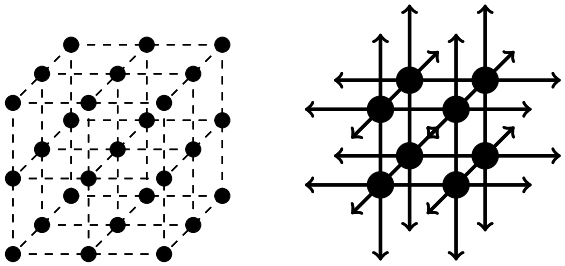

Figure 3.3: Mapping simple 3D Dots-and-Boxes to Strings-and-Coins.

$\geq$ 1 string, $\geq$ 2 coins. We play on a three-dimensional grid of dots. Players alternate turns connecting adjacent dots. When all twelve edges of a cube are completed, the player to do so tallies a point and moves again. The game ends when all edges are drawn. The player with the most points wins.

This version is the most complicated. If we try to create an analogue Strings-and-Coins game, we find many examples of coins that are connected by up to four strings. We may also find circumstances where cutting one string corresponds to capturing four coins. An example of this scenario is shown below.

THE TRIVIAL CASE

Let us take a moment to fully understand the **1 string, 2 coins** version of 3D Dots-and-Boxes. This is easiest to comprehend as an analogue Strings-and-Coins game (Figure 3.3). The mapping we shall use is cubes to coins and faces to strings.

As in traditional Dots-and-Boxes, our objective is to be the controlling player when only long chains remain. We may accomplish this by implementing a slightly modified version of the Long Chain Rule.

Long Chain Rule (version 2). *Suppose we are playing 3D Dots-and-Boxes*

on an $n \times m \times l$ rectangular prism. If exactly two of $\{n, m, l\}$ are odd, then Player 1 gains control by making the number of eventual long chains even, and Player 2 gains control by making this number odd. Otherwise, Player 1 gains control by making the eventual number of long chains odd, and Player 2 gains control by making this number even.

Proof. As was the case in our original proof of the Long Chain Rule, we set out to determine who shall make the final move. The expression for the number of turns in this version of 3D Dots-and-Boxes is

$$\text{Turns} = \text{Moves} - \text{Cubes} + \text{Double-dealt moves}.$$

The number of moves is given by the number of faces, or

$$3nml - 2nl - 2nm - 2ml + n + m + l.$$

The number of cubes is simply

$$(n-1)(m-1)(l-1) = nml - nm - nl - ml + n + m + l - 1.$$

Taking the difference,

$$\text{Turns} = 2nml - nm - nl - ml + 1 + \text{Double-dealt moves}.$$

Recall that each long chain in the endgame corresponds to a double-dealt move, except the last one, which the controlling player takes completely. Therefore, double-dealt moves = long chains $- 1$, and so

$$\text{Turns} = 2nml - nm - nl - ml - \text{Long Chains}.$$

We want to find the parity of this number. If all of $\{n, m, l\}$ are even, then the parity of turns depends on the parity of long chains. This is the same if just one of $\{n, m, l\}$ is odd. If two of $\{n, m, l\}$ are odd, then the parity of turns is the opposite of the parity of long chains. Finally, if all of $\{n, m, l\}$ are even, then the parity of turns is again the same as that of long chains. To gain control, we wish to make the final move, so as the first player, we want the parity of moves to be odd, and as the second player we want the parity of moves to be even. This is accomplished by arranging the proper number of long chains. □

Of course, we should also examine whether our constructions for Nimstring apply to this version of 3D Dots-and-Boxes. Recall that in traditional Nimstring, there are six types of moves with capturable coins. If these moves are the same in the Nimstring analogue of this version of 3D Dots-and-Boxes, then our algorithm for determining the nimber of a specific position will remain the same, and the strategies we have previously discussed will apply.

Claim. *All capturable coins in the Strings-and-Coins version of* ***1 string, 2 coins*** *3D Dots-and-Boxes are of one of the types pictured in Figure 1.4.*

Proof. Suppose we have a capturable coin. That coin may either be isolated (Figure 1.4 (a)) or connected to another coin. If it is connected to another coin, the coin to which it is connected may have one (Figure 1.4 (b)), two (Figure 1.4 (c),(e) and (f)), or three or more (Figure 1.4 (d)) strings attached to it. If it is attached to a coin with two strings, one of those strings may either go to the ground (Figure 1.4 (e)), to a coin with one string (Figure 1.4 (c)), or to a coin with at least two strings (Figure 1.4 (f)). These are all the possibilities, and each is depicted in Figure 1.4. □

DEALING WITH MULTIPLES

Let us try to unravel the two complex versions of 3D Dots-and-Boxes. We shall begin with the **1 string, $\geq$ 2 coins** version.

1 STRING, $\geq$ 2 COINS

It will be helpful in our analysis to have a Strings-and-Coins analogue for this version. Unfortunately, it is rather difficult to convey the idea of one string connecting multiple coins in a visually aesthetic manner. In Figure 3.4 we have given the Strings-and-Coins analogue for a game played on a cube. Generally speaking, a larger game is essentially equivalent to a field of octahedrons stacked on top of each other (Figure 3.5). This game could easily have more dimensionality by extending backwards as well, but at that point the entire image becomes a jumbled, illegible mess.

There is one significant problem with this analogue. Recall from Figure 3.1 that certain moves in this version complete up to four squares. Such a move is analogous to a string that connects more than two coins. In Figure 3.5, these strings are represented by the red, blue, green and pink

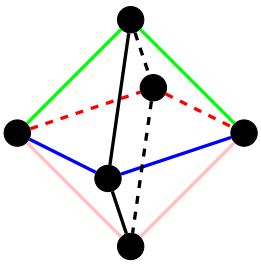

Figure 3.4: Another Strings-and-Coins analogue.

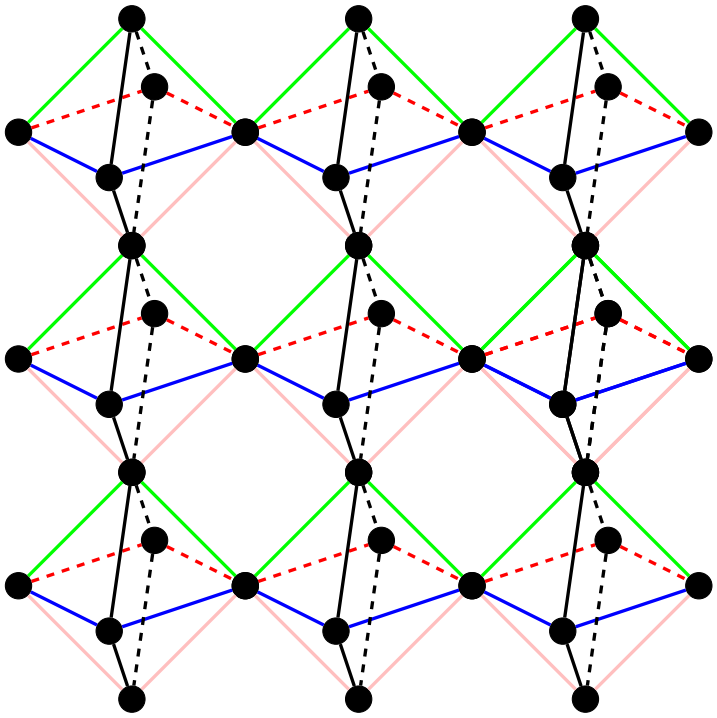

Figure 3.5: A field of octahedrons.

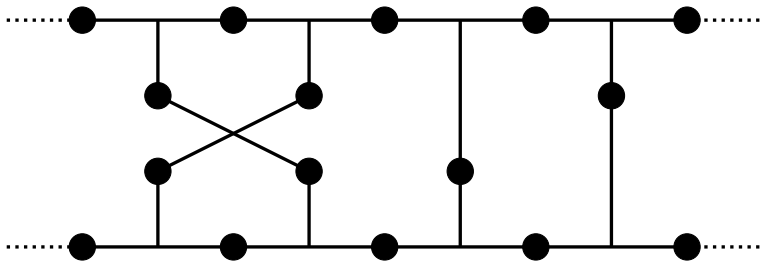

Figure 3.6: A diagram of a loopy engame.

lines emanating from the junction coin between two octahedra. For example, cutting a string attached to the coin just above the "9" in "Figure 9" would correspond to removing both lines of a given color (say, orange). Also, note that there are no strings attached to the ground. This means that there are no moves that affect only one face.

Is there an equivalent Long Chain Rule for **1 string, $\geq$ 2 coins** 3D Dots-and-Boxes? Some difficulties arise when determining the number of turns in this game. Because we have moves that remove both 3 and 4 coins, the expression for the number of turns becomes a bit more complicated. We now have

$$\text{Turns} = \text{Moves} - \text{Faces} + \text{2-dealt} + 2(\text{3-dealt}) + 3(\text{4-dealt})$$

where n-dealt refers to a move that removes n coins. We may calculate the value of moves $-$ faces and simplify to

$$\text{Turns} = nm + nl + ml - n - m - l + (\text{multiple-coin strings})$$

where *multiple-coin strings* = 2-dealt+2(3-dealt)+3(4-dealt), but we haven't a way to put this expression in terms of long chains.

Let us take a step back and examine what the endgame of **1 string, $\geq$ 2 coins** 3D Dots-and-Boxes might look like. Because no strings connect to the ground, the endgame is comprised not of long chains, but of a network of long loops. Figure 3.6 presents a schematic illustration of a possible loopy endgame.

Suppose we are playing a game on a $4 \times 4 \times 4$ cube (shown in Figure 3.7). We might think of this game as two nested games: one played on the outer surface of the $4 \times 4 \times 4$ cube (shaded blue) and the other on the outer surface of the internal $2 \times 2 \times 2$ cube (shaded red). Both of these

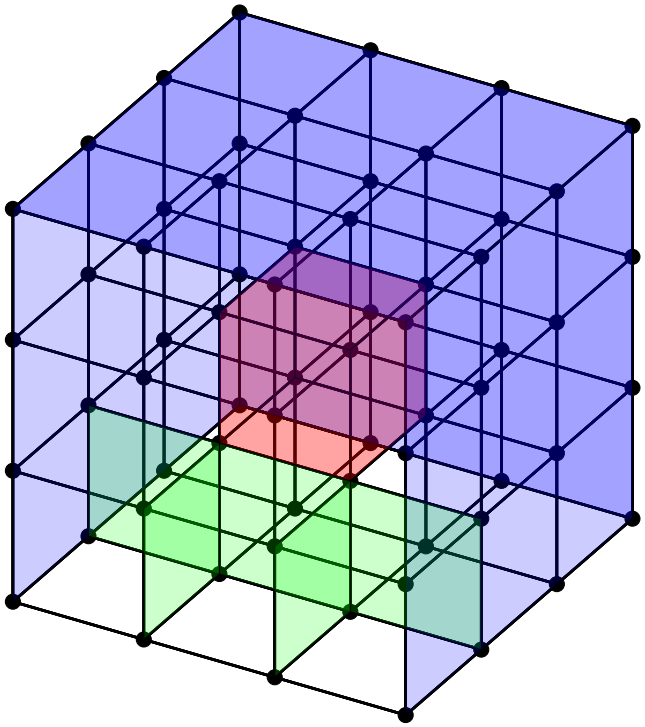

Figure 3.7: The different parts of a game.

games have endgames which are essentially equivalent to traditional Dots-and-Boxes, with the exception that rather than ending in long chains, they end in long loops. Two of these long loops are represented by the horizontal chains in Figure 3.6. Of course, the internal $2 \times 2 \times 2$ cube and the external $4 \times 4 \times 4$ cube are connected by other faces. Some of these faces, which we shall refer to as *supports*, are illustrated in green in Figure 3.7. They correspond to the coins in between the two long loops in Figure 3.6.

Claim. *The only 2-coin strings in* ***1 string,*** $\geq$ ***2 coins*** *3D Dots-and-Boxes are those that connect coins corresponding to adjacent squares on different faces of the outer board.*

Proof. All faces in **1 string,** $\geq$ **2 coins** 3D Dots-and-Boxes are either in-line with or orthogonal to each other. In three dimensions, the greatest number of orthogonal faces around an edge is $4 = 360 \div 90$. In this scenario, each face points in one of the cardinal directions (N, E, S, W). All other arrangements of faces about an edge are:

1. NE (or SE, SW, NW)
2. EW (or NS)
3. NES (or ESW, SWN, WNE)

Of these scenarios, only cases 1 and 2 represent circumstances where two coins are attached by one string (i.e. two faces are adjoined by one edge). Case 1 is what we find in traditional Dots-and-Boxes. It does not extend to three dimensions, since the z-axis will tack on an additional face and turn it into case 3. We are left with case 2, which only exists between adjacent squares on different faces of the outer board. □

Claim. *All connections from support coins to the outer board are 3-coin strings, and two of these three coins lie on the outer board.*

Proof. This is an example of the modification that takes place when we add the third dimensional axis to a 2D game of Dots-and-Boxes (i.e. the shift from case 1 to case 3). □

Claim. *All connections between any two coins not on the outer board are 4-coin strings.*

Proof. We have seen that case 1 does not exist in three dimensions and case 2 is only found between adjacent squares on different faces of the outer board. Case 3 must have 180 degrees between two faces, a situation which only exists on the outer face. By process of elimination, the only remaining scenario is to have four faces adjoined about one edge (i.e a 4-coin string). □

What do these results tell us about the types of n-dealt moves, and consequently number of turns, in a game? Obviously, a 2-dealt move (previously referred to as a double-dealing move) is the most preferable of all n-dealt moves, as it provides your opponent with the fewest number of boxes. We should therefore ask whether there exists any circumstance where a player might be forced to make a 3 or 4-dealt move in order to maintain control during the endgame. What would such a scenario look like?

Recall that the endgame is comprised of a tangled web of interconnected loops. Some of these loops are attached by 3 and 4-coin strings, while others are attached at junction coins (i.e. coins with degree > 2). The distinction between these two cases is important. A junction coin is the terminating point of a long loop. When one of its strings is cut, no coin is removed, and

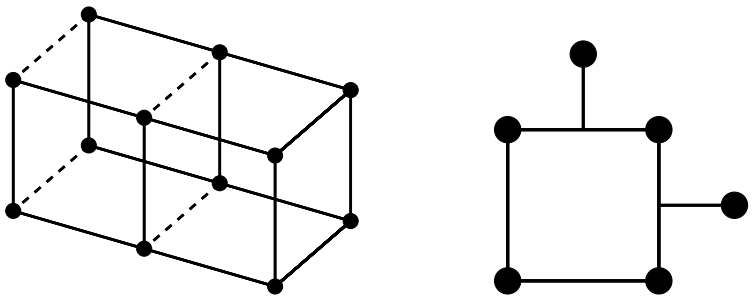

Figure 3.8: A 3-coin forfeit.

the turn ends. A 3 or 4-coin string, however, is like an unusual fork in the road where the traveller can choose to walk down both paths simultaneously.[1] This means that a move cascades along 3 and 4-coin strings until it eventually bumps into junction coins (or itself, if it is one long loop) and terminates.

Because of 3 and 4-coin chains, a long chain in a given loopy endgame might thought of as a branching tree, where rather than having leaves, the terminus of each branch is either grafted back onto itself at the base of the trunk (case 1) or grafted onto an adjacent tree (case 2). Let's examine the first of these scenarios:

Case 1. *We immediately encounter a scenario that would necessitate the forfeit of a 3-coin chain in order to maintain control. Suppose we are playing a game on the board in Figure 3.8 (a). This has a corresponding Strings-and-Coins game, shown in Figure 3.8 (b). Whatever move our opponent makes, we wish to maintain control by forcing him to make the final move in this loop and open the next loop. We accomplish this by cutting whatever string is opposite the string he cuts (i.e. if he cuts the top string, we cut the bottom, and if he cuts the left string, we cut the right). Unfortunately, this move guarantees that our opponent will cut a 3-coin string and get 3 points.*

There is no simple way to determine the precise number of circumstances that would force a controlling player to make 3 or 4-dealt moves in the endgame. Consequently, the number of turns in a game is no longer exclusively dependent on the number of long chains and starting dots, and the Long Chain Rule, if it exists, is non-trivial to deduce and apply.

[1] Much to the chagrin of Robert Frost.

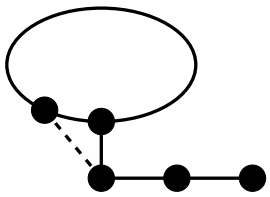

Figure 3.9: A loony move.

NIMSTRING: 1 STRING, $\geq$ 2 COINS

Let's determine an algorithm for finding the nim-value of a **1 string, $\geq$ 2 coins** game of Dots-and-Boxes. There are two type of capturable boxes (coins) in a game of Dots-and-Boxes (Strings-and-Coins):

1. canonical captures.

2. loony options.

A canonically capturable coin is a coin whose capture in no way adversely affects the capturing player's strategy. We saw an example of such moves in Figure 1.4 (a), (b), (c) and (d). A capturable coin with a loony option provides a winning strategy for the player to move; examples of such moves may be found in Figure 1.4 (e) and (f). Let's consider all coins in **1 string, $\geq$ 2 coins** 3D Dots-and-Boxes that have a loony option. Recall that the loony moves in traditional Strings-and-Coins either had a string going to the ground (illegal in **1 string, $\geq$ 2 coins** 3D Dots-and-Boxes) or a pair of connected coins that could be pawned off on one's opponent if necessary (reproduced in Figure 3.9).

In order for a position to be loony, the player to move must have the option to either capture at least one coin, or make a move that forces his opponent to capture at least one coin. Figure 3.10 provides examples of how this might occur with a 2-coin string, a 3-coin string and a 4-coin string. Each red coin may be attached to other coins not shown in the figure, and each red string may be transformed into either a 3 or 4-coin string.

Claim. *A position in **1 string, $\geq$ 2 coins** 3D Dots-and-Boxes is loony if and only if it takes one of the three forms shown in Figure 3.10.*

Proof. Suppose we have an arbitrary loony position. Furthermore, suppose that this loony position is as simple as possible (i.e. there are no extraneous

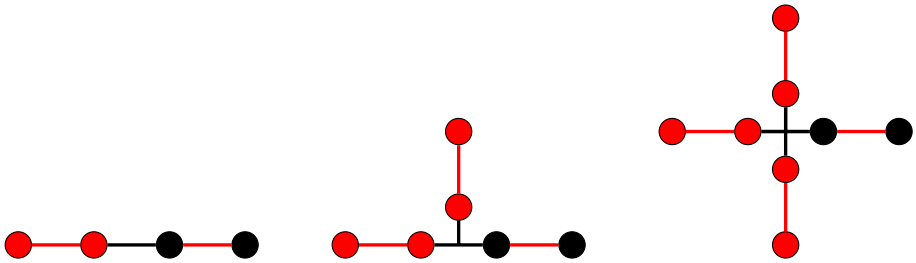

Figure 3.10: 2, 3 and 4-coin loony positions.

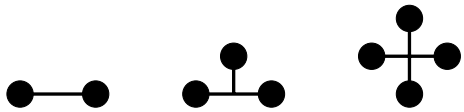

Figure 3.11: 2, 3 and 4-dealt moves.

canonically capturable coins). By definition, we have the option to either remove all capturable coins or force our opponent to remove all capturable coins. If we wish to force our opponent to remove all capturable coins, then we must make a move that does not capture any coins. This may be accomplished by cutting either a 2, 3 or 4-coin string (shown in black in Figure 3.10) and leaving our opponent with a number of 2, 3 or 4-dealt moves (see Figure 3.11). If our opponent were left with anything other than 2, 3 or 4-dealt moves, that would contradict our assumption that the loony position was a simple as possible. All possible combinations of 2, 3 and 4-dealt moves are described in Figure 3.10 (consider the red strings). Alternatively, if we wish to remove all capturable coins, we may remove all potential 2, 3 and 4-dealt moves (described by the red strings) and end our turn by cutting one of the black strings in Figure 3.10. Now suppose that we have one of the positions shown in Figure 3.10. Each of these positions has at least one capturable coin (the black coin attached to just one red string) and one move that leaves only n-dealt moves (cutting the black string). These are the two criteria for a loony position. □

Recall that all capturable coins must either be canonically capturable or have a loony option. We have shown that all coins with loony options must exist in one of the forms pictured in Figure 3.10. Therefore, all other capturable coins in **1 string, $\geq$ 2 coins** 3D Dots-and-Boxes must be canonically capturable. We now have enough information to present an algorithm

for determining the nim-value of an arbitrary game of **1 string, ≥ 2 coins** 3D Dots-and-Boxes:

1. *The Nimstring position with no available strings to cut has nim-value* $0*$.
2. *The position with canonically capturable coins has the same value as the position after those coins are captured.*
3. *The position with loony options shown in Figure 3.10 has value* $\mathfrak{D}$.
4. *The position with no capturable coins has value equal to the* ***mex*** *of the values of its subpositions.*

≥ 1 STRING, ≥ 2 COINS

≥ 1 string, ≥ 2 coins 3D Dots-and-Boxes is the most complicated version. We will now be faced with both n-coin strings and n-string coins (that is, a pair of coins connected by more than one string). As we have done before, let's begin by creating a Strings-and-Coins analogue. We shall map cubes to coins and edges to strings. The number written on each string represents the number of strings connecting those two coins (Figure 3.12). It should be noted that if we limit the shape of our game board to a rectangular prism (that is, some $n \times m \times l$ 3D grid), only 1, 2 and 4-coin strings exist.[2]

Is there an equivalent Long Chain Rule for **≥ 1 string, ≥ 2 coins** 3D Dots-and-Boxes? We should expect to run into the same problem that we did with the **1 string, ≥ 2 coins** version; namely, the number of n-dealt moves in the endgame varies from one match to another. Let us therefore move straight ahead and try to understand the Nimstring equivalent of **≥ 1 string, ≥ 2 coins** 3D Dots-and-Boxes.

NIMSTRING: ≥ 1 STRING, ≥ 2 COINS

Again, our ultimate goal is to construct an algorithm that will theoretically allow us to compute the nim-value of any position. From a functional perspective, the existence of such an algorithm will not enable us to solve most

[2] A 3-coin string requires a L-shaped board (think a 3D version of the L-shaped Tetris piece).

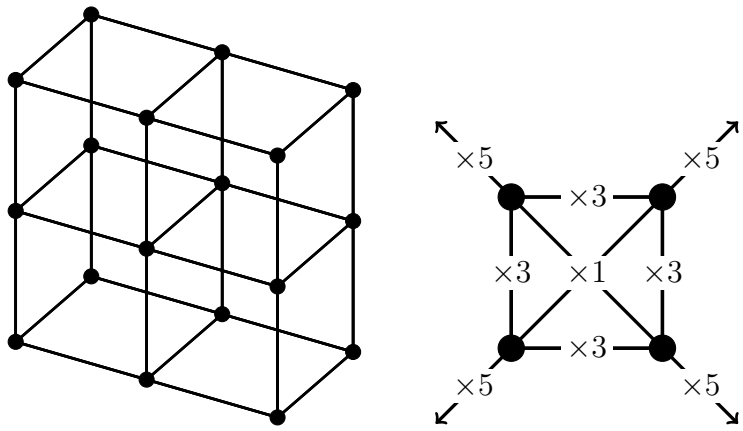

Figure 3.12: Mapping complicated 3D Dots-and-Boxes to Strings-and-Coins

games; the search space is far to large to build up a useful library of nimbers. Nevertheless, it forms a foundation upon which we may construct a more complex and detailed analysis.

As before, we shall separate all positions with capturable coins into two classes: canonical captures and loony options. Because the number of canonical captures far exceeds that of loony options, we shall proceed by constructing a list of all loony positions in $\geq$ **1 string,** $\geq$ **2 coins** 3D Dots-and-Boxes. Part of this list is illustrated schematically in Figure 3.13.

The first two loony positions roughly correspond to the 2 and 4-coin string positions in Figure 3.10. There are two noteworthy distinctions. In Figure 3.13, the red strings indicate a 1-coin string, a 2-coin string or a 4-coin string, and the dashed red strings indicate a 2-coin string or a 4-coin string. The remaining four positions represent specific scenarios; while the first three positions are relatively self-explanatory, the last requires some further explanation. This position is considered loony because the player to move has the choice of either removing all coins (by cutting the 4-coin string), or alternating turns cutting 1-coin strings until his opponent is forced to cut the central 4-coin string. In either case, the player to move has the capacity to dictate the outcome of the position.

It is uncertain whether the diagrams in Figure 3.13 include all possible loony positions. Unlike in Figure 3.10, there are a significant number of specific positions that are loony (e.g. the last four positions in Figure 3.13), and these do not fit easily into particular schema. The author hesitantly proposes

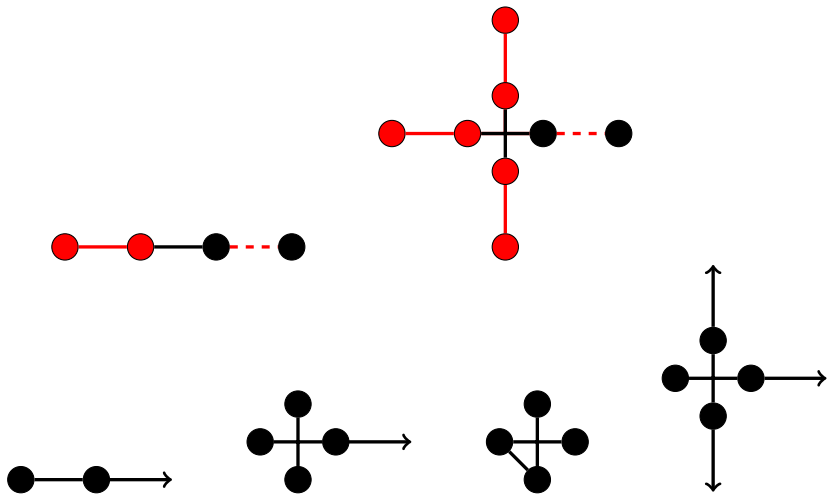

Figure 3.13: More loony positions.

the following hypothesis, acknowledging the possibility of some missing cases:

Claim. *All loony positions in* $\geq$ ***1 string,*** $\geq$ ***2 coins*** *3D Dots-and-Boxes are illustrated in Figure 3.13.*

Independent of the veracity of this claim, we may still apply the generalized algorithm for determining Nimstring nim-values:

1. *The Nimstring position with no available strings to cut has nim-value* $0*$.
2. *The position with canonically capturable coins has the same value as the position after those coins are captured.*
3. *The position with loony options shown in (but not necessarily limited to) Figure 3.13 has value* $\supset$.
4. *The position with no capturable coins has value equal to the* ***mex*** *of the values of its subpositions.*

WRAPPING UP

We have reached the end of this paper, although we have really only scrambled up into the lower canopy of what feels like an ever-expanding tree of questions and problems relating to Dots-and-Boxes. In brief, I hope that this paper has opened your eyes to the complexities of what appear to be[1] the banalities (or banality; I shouldn't get carried away) of everyday life. For me, this paper is, in a sense, precisely that. I never expected to struggle with the questions I did, nor did I expect to end 50+ pages feeling like I was finally prepared to really start exploring Dots-and-Boxes. To an extent, I suspect that that is simply the nature of this type of project, and perhaps leaving these open questions is more indicative of a success than a failure. But honestly, that decision is up to you.

[1] and perhaps are, depending on your scope

Bibliography

[1] Elwyn R. Berlekamp. *The Dots-and-Boxes Game: Sophisticated Child's Play*. 1st ed. A K Peters, Ltd., 2000.

[2] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays: Volume 1*. 2nd ed. A K Peters, Ltd., 2001.

[3] Donald E. Knuth. *Surreal Numbers*. Ed. by 1. Addison-Wesley Professional, 1974.

CONNECTIONS TO GRAPH THEORY

The different versions of Nimstring that we have explored in this paper show some compelling similarities to graph theoretical concepts. Let us take a moment to illustrate these connections.

ORIGINAL NIMSTRING

The original game of Nimstring is played on a graph. Each coin is equivalent to a vertex and each string is equivalent to an edge. 1-coin strings may be thought to connect to a special vertex called the "ground." Readers familiar with graph-coloring and Ramsey theory might find interest in examining possible crossover between cutting strings and coloring edges.

1 STRING, $\geq$ 2 COINS NIMSTRING

This is Nimstring played on a hypergraph. Hypergraphs allow edges to be joined between more than two vertices.

$\geq$ 1 STRING, $\geq$ 2 COINS NIMSTRING

This is Nimstring played on a hypergraph-multigraph hybrid. Multigraphs allow for multiple edges between the same two vertices. One area of interest might be what a multigraph game of Nimstring looks like when translated back into Dots-and-Boxes.

GENERALITIES

Graph theory examines a number of classes of graph. These include, but are in no way limited to:

1. bipartite graphs
2. complete graphs
3. cyclic and acyclic graphs
4. star graphs
5. wheel graphs

It could be quite enlightening to determine the nim-values of games of Nimstring played on these various graphs. What if we look at comparable structures in hypergraphs and multigraphs? It may also be worth examining whether there is an obvious connection between the nim-value of a graph and the nim-value of its complement. To the extent of the author's knowledge, this is an essentially unexplored area of mathematics.

THE CODE

MAIN CLASS

```
1  import java.util.Hashtable;
2  import java.util.Scanner;
3
4  public class Main {
5
6    public static void main(String[] args) {
7      // TODO Auto-generated method stub
8
9      Scanner input = new Scanner(System.in);
10     Strategy helper = new Strategy(1);
11
12     System.out.println("Player 1 or player 2?");
13     int id = input.nextInt();
14     System.out.println("How many rows?");
15     int rows = input.nextInt();
16     System.out.println("How many columns?");
17     int columns = input.nextInt();
18
19     int movesRemaining = 2*rows*columns + rows + columns;
20     int depth = 5;
21
22     Board startingBoard = new Board(rows, columns, 1);
23     Board afterMove = startingBoard;
24
25     //getTotalDegree function is inefficient and WRONG
26     while (movesRemaining != 0) {
27       //if (movesRemaining < 12) depth = 9;
28       if (afterMove.getID() == id) {
29         afterMove = makeOppMove(afterMove);
30         movesRemaining--;
31         depth = (int) Math.pow(1.25, 2*rows*columns + rows + columns -
     ↪ movesRemaining);
32       }
33       //if (movesRemaining < 12) depth = 15; //REDUNDANT
34       if (afterMove.getID() != id) {
35         helper.alphaBeta(afterMove, depth, Double.NEGATIVE_INFINITY,
     ↪ Double.POSITIVE_INFINITY, true);
36         afterMove = helper.bestBoard;
37         movesRemaining--;
38         System.out.println("The computer removed the following string:
```

```
      ↪ \nString ID = " + helper.bestMove[2] +
39              "\nx = " + helper.bestMove[0] + "\ny = " + helper.bestMove[1]);
40        }
41      }
42
43      //scoring is wrong!
44      System.out.println("Game over! Final score:\nComputer: " +
      ↪ helper.computerScore + "\nYou: " + helper.opponentScore);
45    }
46
47    public static Board makeOppMove(Board board) {
48      Scanner input = new Scanner(System.in);
49      System.out.println("Please enter the x-coordinate of the coin whose
      ↪ string you wish to cut:");
50      int x = input.nextInt();
51      System.out.println("Please enter the y-coordinate of the coin whose
      ↪ string you wish to cut:");
52      int y = input.nextInt();
53      System.out.println("Please enter the ID of the string you wish to
      ↪ cut:");
54      int stringID = input.nextInt();
55      //Add error statement here!!!
56      return DBUtilities.nextMove(board, board.boardCoins[x][y], stringID);
57    }
58
59  }
```

STRATEGY CLASS

```
1  import java.util.ArrayList;
2  import java.util.Hashtable;
3
4  public class Strategy {
5    int id;
6    Board bestBoard;
7    int[] bestMove = new int[3];
8    Hashtable<Long, Transposition> transpositionTable = new Hashtable<Long,
       ↪ Transposition>();
9    int computerScore = 0;
10   int opponentScore = 0;
11
12   public Strategy(int playerID) {
13     this.id = playerID;
14   }
15
16   public double alphaBeta(Board board, int depth, double alpha, double
       ↪ beta, boolean maximizingPlayer) {
17     //System.out.println("Computer Score: " + computerScore + " & Opponent
       ↪ Score: " + opponentScore + " & Depth: " + depth);
18     //this isn't working b/c it sets external variable equal to null in
       ↪ each recursion. !!!Temp solution might be sketchy
19     Board tempBestBoard = null;
20     //temp solution might be sketchy!!!!!
21     int[] tempBestMove = {-1,-1,-1};
22     double value;
23     Transposition entry = transpositionTable.get(board.zobristKey);
```

```
24      //if the board zobrist key in the TT equals the current board key
25      if (entry != null) {
26        if (entry.zobristKey == board.zobristKey) {
27          if (entry.depth >= depth) {
28            if (entry.compScore == computerScore && entry.oppScore ==
        ↪ opponentScore) {
29              //Get bestBoard from TT
30              tempBestBoard = entry.bestBoard;
31              tempBestMove = entry.bestMove;
32              if (entry.flag == 0) {
33                value = entry.value;
34                //Somehow encode the score here...???
35                return value;
36              }
37              if (entry.flag == 1 & entry.value >= alpha) {
38                //alpha, beta are fns of the score...
39                alpha = entry.value;
40              }
41              if (entry.flag == 2 & entry.value <= beta) {
42                beta = entry.value;
43              }
44            }
45          }
46        }
47      }
48      //getTotalDegree function is inefficient
49      if (board.getTotalDegree() == 0) {
50        value = nimEval(board);
51        //puts the value in the TT. Evaluates zobrist key of entire board. 0
        ↪ means that we store the real value
52        storeHash(board, depth, 0, value, false, tempBestBoard, tempBestMove);
53        return value;
54      }
55      else if (depth == 0) {
56        value = heuristicEarly(board);
57        //puts the value in the TT. Evaluates zobrist key of entire board. 0
        ↪ means that we store the real value
58        storeHash(board, depth, 0, value, false, tempBestBoard, tempBestMove);
59        return value;
60      }
61      else if (maximizingPlayer == true) {
62        value = Double.NEGATIVE_INFINITY;
63        ArrayList<int[]> moves = DBUtilities.orderMoves(board);
64        for (int[] array : moves) {
65          if (board.boardCoins[array[0]][array[1]].getString(array[2]) == 1) {
66            //PROBLEM! No deep copying
67            Board newBoard = DBUtilities.nextMove(board,
        ↪ board.boardCoins[array[0]][array[1]], array[2]);
68            if (newBoard.boardCoins[array[0]][array[1]].getDegree() == 0){
69              computerScore = computerScore + 1;
70              value = Math.max(value, alphaBeta(newBoard, depth - 1, alpha,
        ↪ beta, true));
71              computerScore = computerScore - 1;
72            } else {
73              value = Math.max(value, alphaBeta(newBoard, depth - 1, alpha,
        ↪ beta, false));
74            }
```

```
75            if (value > alpha) {
76              alpha = value;
77              tempBestBoard = newBoard;
78              tempBestMove = array;
79              //stores real value of board
80              storeHash(board, depth, 0, value, false, tempBestBoard,
    ↪ tempBestMove);
81            }
82            if (beta <= alpha) {
83              //In case of a TT-invoked alpha-change
84              value = Math.max(value, alpha);
85              //In case of a TT-invoked alpha-change
86              storeHash(board, depth, 1, value, false, tempBestBoard,
    ↪ tempBestMove);
87              break;
88            }
89          }
90        }
91        //finally sets the proper best board. Worth rechecking later!!!!!
92        bestBoard = tempBestBoard;
93        bestMove = tempBestMove;
94        //System.out.println("************VALUE: " + value);
95        return value;
96      }
97      else {
98        value = Double.POSITIVE_INFINITY;
99        ArrayList<int[]> moves = DBUtilities.orderMoves(board);
100       for (int[] array : moves) {
101         if (board.boardCoins[array[0]][array[1]].getString(array[2]) == 1) {
102           Board newBoard = DBUtilities.nextMove(board,
    ↪ board.boardCoins[array[0]][array[1]], array[2]);
103           if (newBoard.boardCoins[array[0]][array[1]].getDegree() == 0){
104             opponentScore = opponentScore + 1;
105             value = Math.min(value, alphaBeta(newBoard, depth - 1, alpha,
    ↪ beta, false));
106             opponentScore = opponentScore - 1;
107           } else {
108             value = Math.min(value, alphaBeta(newBoard, depth - 1, alpha,
    ↪ beta, true));
109           }
110           if (value < beta) {
111             beta = value;
112             tempBestBoard = newBoard;
113             tempBestMove = array;
114             //stores real value of board
115             storeHash(board, depth, 0, value, false, tempBestBoard,
    ↪ tempBestMove);
116           }
117           if (beta <= alpha) {
118             //In case of a TT-invoked beta-change
119             value = Math.min(value, beta);
120             //In case of a TT-invoked beta-change
121             storeHash(board, depth, 2, value, false, tempBestBoard,
    ↪ tempBestMove);
122             break;
123           }
124         }
```

```
125         }
126         return value;
127       }
128     }
129
130     public void storeHash(Board board, int depth, int flag, double value,
        ↪ boolean ancient, Board tempBestBoard, int[] tempBestMove) {
131       Transposition tableEntry = new Transposition(board.zobristKey, depth,
        ↪ flag, value, ancient, computerScore, opponentScore, tempBestBoard,
        ↪ tempBestMove);
132       transpositionTable.put(board.zobristKey, tableEntry);
133     }
134
135     public double iterativeDeepening(Board board, int d) {
136       bestBoard = board;
137       double value = 0;
138       for (int depth = 1; depth <= d; depth++) {
139         value = alphaBeta(bestBoard, depth, Double.NEGATIVE_INFINITY,
        ↪ Double.POSITIVE_INFINITY, true);
140       }
141       return value;
142     }
143
144     public double nimEval(Board board) {
145       double result = computerScore - opponentScore;
146       return result;
147     }
148
149     public double heuristicEarly(Board board) {
150       double result = computerScore - opponentScore;
151       return result;
152     }
153 }
```

BOARD CLASS

```
1  import java.util.ArrayList;
2  import java.util.Random;
3
4  public class Board {
5    int id;
6    int rows;
7    int columns;
8    Coin[][] boardCoins;
9    int counter = 1;
10   boolean[][] wasHere;
11   long[][][] zobristHashes;
12   long zobristKey;
13   ArrayList<ArrayList<Coin>> components = new ArrayList<ArrayList<Coin>>();
14   boolean[][] wasHereAgain;
15
16   Random rdm = new Random();
17
18   public Board(Board clone) {
19     this.id = clone.id;
20     this.rows = clone.rows;
```

```
21      this.columns = clone.columns;
22      this.components = clone.components;
23      this.zobristKey = clone.zobristKey;
24      this.wasHere = clone.wasHere;
25      this.wasHereAgain = clone.wasHereAgain;
26      this.zobristHashes = clone.zobristHashes;
27      this.boardCoins = new Coin[columns][rows];
28      for (int j = 0; j < rows; j++) {
29        for (int i = 0; i < columns; i++) {
30          boardCoins[i][j] = new
      ↪ Coin(i,j,clone.boardCoins[i][j].getString(0),clone.boardCoins[i][j].getString(1),
31
      ↪ clone.boardCoins[i][j].getString(2),clone.boardCoins[i][j].getString(3));
32        }
33      }
34    }
35
36    public Board(int rows, int columns, int id) {
37      this.id = id;
38      this.rows = rows;
39      this.columns = columns;
40      this.boardCoins = new Coin[columns][rows];
41      this.wasHere = new boolean[columns][rows];
42      this.wasHereAgain = new boolean[columns][rows];
43      this.zobristHashes = new long[columns][rows][4];
44      for (int j = 0; j < rows; j++) {
45        for (int i = 0; i < columns; i++) {
46          boardCoins[i][j] = new Coin(i,j);
47          for (int k = 0; k < 4; k++) {
48            zobristHashes[i][j][k] = Math.abs(rdm.nextLong());
49            zobristKey = zobristKey ^= zobristHashes[i][j][k];
50          }
51        }
52      }
53    }
54
55    public long getZobristKey() {
56      this.zobristKey = 0;
57      for (int i = 0; i < columns; i++) {
58        for (int j = 0; j < rows; j++) {
59          for (int k = 0; k < 4; k++) {
60            if (boardCoins[i][j].getString(k) == 1) {
61              zobristKey = zobristKey ^= zobristHashes[i][j][k];
62            }
63          }
64        }
65      }
66      return zobristKey;
67    }
68
69    public void updateKey(int x, int y, int stringID) {
70      zobristKey = zobristKey ^= zobristHashes[x][y][stringID];
71    }
72
73    public int getTotalDegree() {
74      int totalDegree = 0;
75      for (int j = 0; j < rows; j++) {
```

```
76        for (int i = 0; i < columns; i++) {
77          totalDegree = totalDegree + boardCoins[i][j].degree;
78        }
79      }
80      return totalDegree;
81    }
82
83    public int getRows() {
84      return this.rows;
85    }
86
87    public int getColumns() {
88      return this.columns;
89    }
90
91    public int getID() {
92      return this.id;
93    }
94
95    public void setID(int id) {
96      this.id = id;
97    }
98
99    public int countChains(Coin coin) {
100     if (coin.getDegree() > 2) {
101       counter = 0;
102       return counter;
103     }
104     if (wasHere[coin.getX()][coin.getY()]) {
105       counter = counter - 1;
106       return counter;
107     }
108     wasHere[coin.getX()][coin.getY()] = true;
109     if (coin.getString(0) == 1 & coin.getY() < rows - 1) {
110       counter++;
111       countChains(boardCoins[coin.getX()][coin.getY() + 1]);
112     }
113     if (coin.getString(1) == 1 & coin.getX() < columns - 1) {
114       counter++;
115       countChains(boardCoins[coin.getX() + 1][coin.getY()]);
116     }
117     if (coin.getString(2) == 1 & coin.getY() > 0) {
118       counter++;
119       countChains(boardCoins[coin.getX()][coin.getY() - 1]);
120     }
121     if (coin.getString(3) == 1 & coin.getX() > 0) {
122       counter++;
123       countChains(boardCoins[coin.getX() - 1][coin.getY()]);
124     }
125     return counter;
126   }
127
128   public void getConnectedCoins(Coin coin, int x) { //THERE COULD BE DEEP
      ↪ COPYING PROBLEMS!!!
129     //Coin cloneCoin = new Coin(coin.getX(),coin.getY(),coin.getString(0),
      ↪ coin.getString(1),coin.getString(2),coin.getString(3));
130     if (wasHereAgain[coin.getX()][coin.getY()] == false) { //If not
```

```
      ↪ visited...
131       wasHereAgain[coin.getX()][coin.getY()] = true;
132       components.get(x).add(coin);
133       if (coin.getString(0) == 1) {
134         if (coin.getY() + 1 < rows) {
135           getConnectedCoins(boardCoins[coin.getX()][coin.getY() + 1],x);
136         }
137       }
138       if (coin.getString(1) == 1){
139         if (coin.getX() + 1 < columns) {
140           getConnectedCoins(boardCoins[coin.getX() + 1][coin.getY()],x);
141         }
142       }
143       if (coin.getString(2) == 1) {
144         if (coin.getY() - 1 >= 0) {
145           getConnectedCoins(boardCoins[coin.getX()][coin.getY() - 1],x);
146         }
147       }
148       if (coin.getString(3) == 1){
149         if (coin.getX() - 1 >= 0) {
150           getConnectedCoins(boardCoins[coin.getX() - 1][coin.getY()],x);
151         }
152       }
153     }
154   }
155
156   public void setComponents() {
157     int x = 0;
158     for (int j = 0; j < rows; j++) {
159       for (int i = 0; i < columns; i++) {
160         if (wasHereAgain[i][j] == false) {
161           ArrayList<Coin> component = new ArrayList<Coin>();
162           components.add(component);
163           getConnectedCoins(boardCoins[i][j], x);
164           x++;
165         }
166       }
167     }
168   }
169 }
```

COIN CLASS

```
1
2  public class Coin {
3    int x;
4    int y;
5    int[] strings = new int[4];
6    int degree;
7
8    public Coin(int x, int y) {
9      this.x = x;
10     this.y = y;
11     strings[0] = 1;
12     strings[1] = 1;
13     strings[2] = 1;
```

```
14      strings[3] = 1;
15      this.degree = strings[0] + strings[1] + strings[2] + strings[3];
16    }
17
18    public Coin(int x, int y, int a, int b, int c, int d) {
19      this.x = x;
20      this.y = y;
21      strings[0] = a;
22      strings[1] = b;
23      strings[2] = c;
24      strings[3] = d;
25      this.degree = strings[0] + strings[1] + strings[2] + strings[3];
26    }
27
28    public int getX() {
29      return this.x;
30    }
31    public int getY() {
32      return this.y;
33    }
34    public int getString(int x) {
35      return this.strings[x];
36    }
37    public void setString(int x) {
38      this.strings[x] = 0;
39      this.degree = strings[0] + strings[1] + strings[2] + strings[3];
40    }
41    public int getDegree() {
42      return this.degree;
43    }
44  }
```

TRANSPOSITION CLASS

```
1
2  public class Transposition {
3
4    long zobristKey;
5    int depth;
6    int flag;
7    double value;
8    boolean ancient;
9    int compScore;
10   int oppScore;
11   Board bestBoard;
12   int[] bestMove;
13
14   public Transposition(long zobristKey, int depth, int flag, double value,
       ↪ boolean ancient, int compScore, int oppScore, Board bestBoard, int[]
       ↪ bestMove) {
15     this.zobristKey = zobristKey;
16     this.depth = depth;
17     this.flag = flag;
18     this.value = value;
19     this.ancient = ancient;
20     this.compScore = compScore;
```

```
21      this.oppScore = oppScore;
22      this.bestBoard = bestBoard;
23      this.bestMove = bestMove;
24    }
25  }
```

UTILITIES CLASS

```
1  import java.util.ArrayList;
2
3  public class DBUtilities {
4
5    //this could use coin.x and coin.y as parameters instead of coin
6    public static Board nextMove(Board board, Coin coin, int stringID) {
7      Board nextMove = new Board(board);
8      nextMove.updateKey(coin.getX(), coin.getY(), stringID);
9      nextMove.boardCoins[coin.getX()][coin.getY()].setString(stringID);
10
11     if (stringID == 0 & coin.getY() < nextMove.rows - 1) {
12       nextMove.boardCoins[coin.getX()][coin.getY() + 1].setString((stringID
       ↪ + 2) % 4);
13       nextMove.updateKey(coin.getX(), coin.getY() + 1, (stringID + 2) % 4);
14     }
15     else if (stringID == 1 & coin.getX() < nextMove.columns - 1) {
16       nextMove.boardCoins[coin.getX() + 1][coin.getY()].setString((stringID
       ↪ + 2) % 4);
17       nextMove.updateKey(coin.getX() + 1, coin.getY(), (stringID + 2) % 4);
18     }
19     else if (stringID == 2 & coin.getY() > 0) {
20       nextMove.boardCoins[coin.getX()][coin.getY() - 1].setString((stringID
       ↪ + 2) % 4);
21       nextMove.updateKey(coin.getX(), coin.getY() - 1, (stringID + 2) % 4);
22     }
23     else if (stringID == 3 & coin.getX() > 0) {
24       nextMove.boardCoins[coin.getX() - 1][coin.getY()].setString((stringID
       ↪ + 2) % 4);
25       nextMove.updateKey(coin.getX() - 1, coin.getY(), (stringID + 2) % 4);
26     }
27
28     //if removes coin, keeps same ID
29     if (nextMove.boardCoins[coin.getX()][coin.getY()].degree == 0) {
30       nextMove.setID(board.getID());
31     } else {
32       //otherwise, changes ID from one player to other
33       if (board.getID() == 1) nextMove.setID(2);
34       if (board.getID() == 2) nextMove.setID(1);
35     }
36
37     return nextMove;
38   }
39
40   public static ArrayList<int[]> orderMoves(Board board) {
41     ArrayList<int[]> moves = new ArrayList<int[]>();
42     for (int j = 0; j < board.rows; j++) {
43       for (int i = 0; i < board.columns; i++) {
44         for (int k = 0; k < 4; k++) {
```

```
45              int[] toMove = {i,j,k};
46              moves.add(toMove);
47            }
48          }
49        }
50        return moves;
51    }
52 }
```