Spring 2016

Public Key Cryptography with the Brin-Thompson Group 2V

Cyril Xavier Kuhns
Bard College

Public Key Cryptography with the Brin-Thompson Group 2V

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Cyril Kuhns

Annandale-on-Hudson, New York
May, 2016

Abstract

The Brin-Thompson group $2V$ is a simple, finitely presented group of functions with solvable word problem and unsolvable torsion problem, which makes it a promising platform group for the Anshel-Anshel-Goldfeld key agreement protocol. The primary results of this project are an implementation of $2V$ and the AAG protocol in Java, which is shown to be susceptible to the heuristic length based attack.

Contents

Dedication

To Sarah Harrelson, who called this years ago.

Acknowledgments

First and foremost, I thank my parents, Anna and John, whose unconditional support is my constant motivation to achieve great things. I know that you would stand by me no matter what, so for you, I try to do my very best.

Secondly, I thank my adviser Jim Belk for being an incredible wealth of knowledge, and my adviser Bob McGrail, for always being available when I needed help. I also want to thank John Cullinan, Becky Thomas, Keith O'Hara, Greg Landweber, Japheth Wood, Maria Belk, and Sven Anderson, each of whom deepened my love of their subjects with their infectious passion and practical advice.

Lastly, I've got to thank Fraiser Kansteiner for his unending patience and friendship through the whole thing.

Introduction

The Brin-Thompson group $2V$ is the a higher dimensional Thompson's group discovered by Matthew Brin in 2004 [6]. This group is one member of the family of groups $\{nV\}_{n=1}^{\infty}$ each of which acts on the product of n copies of the Cantor set, where the group $1V$ is the group V discovered by Richard Thompson in the 1960s [3]. Elements of $2V$ are functions that describe homeomorphisms of the Cantor square, and can be represented with a pair of pictures. An example of a function in $2V$ is given in Figure 0.0.1.

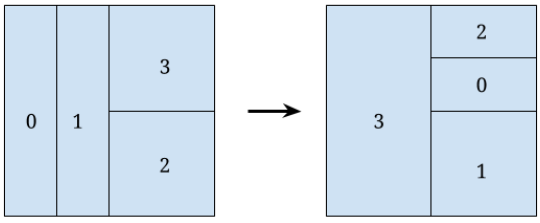


Figure 0.0.1: This function maps the rectangle labeled 0 on the left to the rectangle labeled 0 on the right, the rectangle labeled 1 on the left to the rectangle labeled 1 on the right, and so forth.

$2V$ is an infinite, simple [8] group with a finite presentation [7]. Furthermore, $2V$ has decidable word problem and was proved in 2014 to have undecidable torsion problem [3]. For these reasons, $2V$ is a promising candidate for group-based public key cryptography.

A cryptographic scheme consists of two users, Alice and Bob, who wish to exchange private information over a public channel. One way to accomplish this goal is for Alice and Bob to agree on a secret encryption key beforehand, and use that to encrypt messages sent over the public channel. If Alice and Bob did not or could not privately share an encryption key prior to their need to exchange private information, one option is to implement a key agreement protocol, such as the Diffie-Hellman protocol [11], which relies on the difficulty of something known as the discrete logarithm problem (DLP).

Early attempts at implementing a group based key agreement protocol focused on abelian groups for which the DLP is hard [4]. More recent protocols focusing on non-abelian groups include the Ko-Lee-Cheon-Han-Kang-Park key agreement protocol [12] and Anshel-Anshel-Goldfeld (AAG) key agreement protocol [1]. Both of these protocols rely on the difficulty of the conjugacy search problem (CSP), which is an analogue of the DLP [4].

The difficulty of the CSP is unknown for the group $2V$, but the fact that its torsion problem is undecidable suggests that the group also has undecidable conjugacy problem, which would make the CSP hard. On the other hand, its decidable word problem makes it easy for two users to verify that they have the same group element, even if they obtain the element in different forms, as they often do in a key agreement situation.

The primary goal of this project was to implement the AAG protocol with $2V$ as a platform group. The result is a robust Java implementation of $2V$, as well as AAG, given in the appendix. The authors of the protocol suggested braid groups as a promising family of possible platform groups [1], but these were shown to be susceptible to an heuristic

cryptographic attack known as the length based attack (LBA) [13], so we also implement a simple version of the LBA to assess this platform group's security, also in the appendix.

We use our implementation of $2V$ to collect data about the number of rectangles in a product of n generators. This data suggests that any algorithm for multiplying elements of $2V$ has a quadratic worst case runtime, making it intractably slow for encryption purposes, unless we create a back door to escape such cases. In general, multiplication runs quickly enough to feasibly implement AAG, but this fact makes it difficult to test the LBA against "large" instances of the protocol, which is unfortunate as Myasnikov and Ushakov assert that the LBA is most successful against long words of group generators [13]. We collect other data, however, which suggests that $2V$ fits the criteria to be vulnerable to this kind of attack.

In the interest of creating a practical encryption key for the two users, we also collected data about the orbit of the origin in keys generated by AAG. What we discover is that functions of a computationally feasible size have remarkably un-chaotic orbits, which presents another flaw in the security of this key agreement protocol.

All this considered, $2V$ is most likely not a secure platform group for AAG, at least as it is implemented here, but it may still be of use to other key agreement protocols based on the difficulty of the CSP, like Ko-Lee-Cheon-Han-Kang-Park [12]. It was very recently proven that group extensions shown by Bogopolski, Martino, and Ventura to have unsolvable conjugacy problem [5] also have solvable word problem [2], so these may also be an area for future exploration with AAG.

Chapter 1 discusses Thompson's group V and the Brin-Thompson group $2V$ at length, and Chapter 2 explains in detail the AAG protocol, including the fundamental basis of its security, as well as the LBA. In Chapter 3, we define a data structure called an augmented binary tree, which we use to represent dyadic separations of the unit square. We also present our Java implementation of $2V$, and many of the algorithms used by the class'

methods. Chapter 4 discusses our implementation of the AAG key agreement protocol in Java, as well as the LBA, proving that AAG with $2V$ as a platform group is not secure. We discuss raw runtimes of AAG, and present data collected about the orbit of the origin in keys generated by our protocol.

1
Group Theory Background

This chapter covers a thorough background of the Brin-Thompson group $2V$ by first examining its one-dimensional relative V. The first section defines dyadic separations of the unit interval, the following section establishes the concept of defining a function in V using dyadic separations, and the section after that utilizes this notion to present an algorithm for multiplying elements of V. The following three sections do the same with the unit square and the group $2V$, and the final section gives a finite presentation of $2V$. Each section builds from the previous to arrive at a full understanding of the functions in the group $2V$.

1.1 The Unit Interval

The functions in Thompsons group V are homeomorphisms of something known as the Cantor set, which itself is related to separations of unit interval $[0, 1]$ into dyadic intervals. This section defines all of these terms, proves properties of each, and establishes relations between them, that are helpful for understanding V.

Dyadic Intervals	Non-Dyadic Intervals
$[0,1]$	$\left[\frac{1}{4},\frac{3}{4}\right]$
$\left[\frac{1}{2},\frac{3}{4}\right]$	$\left[\frac{1}{8},1\right]$
$\left[\frac{3}{16},\frac{1}{4}\right]$	$\left[\frac{1}{4},\frac{1}{3}\right]$
$\left[\frac{1}{2},1\right]$	$\left[\frac{1}{7},\frac{2}{3}\right]$

Table 1.1.1: Examples and non-examples of dyadic intervals.

Definition 1.1.1. A **dyadic** number is a rational number whose denominator is a power of 2. A **dyadic interval** is an interval of the form $I = \left[\frac{k}{2^m}, \frac{k+1}{2^m}\right]$ such that $0 \leq \frac{k}{2^m}$ and $\frac{k+1}{2^m} \leq 1$, and where $k, m \in \mathbb{Z}_{\geq 0}$. △

Some examples of what does and does not constitute a dyadic interval are given in Table 1.1.1.

Definition 1.1.2. We say that an interval I is a **left interval** if k is even and a **right interval** if k is odd. Two adjacent intervals I and J are **reducible** if I is a left interval, J is a right interval, $I = \left[\frac{k}{2^m}, \frac{k+1}{2^m}\right]$, and $J = \left[\frac{k+1}{2^m}, \frac{k+2}{2^m}\right]$. △

Specifically, the pair reduce to $H = I \cup J$. Since I is a left interval, we know that $k = 2l$ for some $l \in \mathbb{Z}_{\geq 0}$. We know that the two are adjacent, so taken all together, we can show algebraically that

$$\begin{aligned} H = I \cup J &= \left[\frac{k}{2^m}, \frac{k+2}{2^m}\right] \\ &= \left[\frac{2l}{2^m}, \frac{2l+2}{2^m}\right] \\ &= \left[\frac{l}{2^{m-1}}, \frac{l+1}{2^{m-1}}\right] \end{aligned}$$

which is itself a dyadic interval.

Theorem 1.1.3. *Given any two dyadic intervals I_1 and I_2, either I_1 is contained in I_2, or I_2 is contained in I_1, or the two do not intersect except at perhaps an endpoint.*

Proof. Let $I_1 = \left[\frac{k}{2^m}, \frac{k+1}{2^m}\right]$ and let $I_2 = \left[\frac{j}{2^n}, \frac{j+1}{2^n}\right]$. For ease of notation, designate

$$s_1 = \frac{k}{2^m} \qquad s_2 = \frac{j}{2^n}$$
$$e_1 = \frac{k+1}{2^m} \qquad e_2 = \frac{j+1}{2^n}$$

so that $I_1 = [s_1, e_1]$ and $I_2 = [s_2, e_2]$. Without loss of generality, let $s_1 < s_2$. Suppose I_1 and I_2 have more than one point in their intersection, but neither is properly contained in the other. Then we know that $s_2 < e_1$, since if $s_2 > e_1$, the two intervals do not intersect, and if $s_2 = e_1$, the two only intersect at that point. We also know that $e_1 < e_2$, since anything else would mean that $I_2 \subset I_1$. So we have the following relation:

$$s_1 < s_2 < e_1 < e_2$$
$$\frac{k}{2^m} < \frac{j}{2^n} < \frac{k+1}{2^m} < \frac{j+1}{2^n}$$
$$k \cdot 2^{n-m} < j < (k+1) \cdot 2^{n-m} < j+1$$
and
$$k < j \cdot 2^{m-n} < k+1 < (j+1) \cdot 2^{m-n}$$

This, however, is a contradiction, since $j, k, m, n \in \mathbb{Z}$. If $m \leq n$, then

$$j < (k+1) \cdot 2^{n-m} < j+1$$

implies the existence of an integer between j and $j+1$, and if $m > n$, then

$$k < j \cdot 2^{m-n} < k+1$$

implies the existence of an integer between k and $k+1$. □

Corollary 1.1.4. *The intersection of two overlapping dyadic intervals is another dyadic interval.*

Proof. Two dyadic intervals I_1 and I_2 overlap each other if they share a common interior point. If they intersect at more than a single point, either $I_1 \cap I_2 = I_1$ or $I_1 \cap I_2 = I_2$, both of which are dyadic intervals by hypothesis. □

Definition 1.1.5. In the context of a single dimension, a **dyadic separation** of an interval I is a set S of non-overlapping dyadic intervals whose union is I. △

Dyadic separations have many properties that are crucial for later proofs about V and $2V$.

Lemma 1.1.6. *Given a dyadic separation S of a dyadic interval $[a, b]$, no interval in S contains $\frac{a+b}{2}$ in its interior, unless S is non-trivial.*

Another way of stating Lemma 1.1.6 is to say that every non-trivial dyadic separation of a dyadic interval $[a, b]$ has a cut at its halfway point $\frac{a+b}{2}$.

Proof by contradiction. . Let $[a, b]$ be a dyadic interval and let S be a non-trivial separation of $[a, b]$ into dyadic intervals. Suppose that in S there exists some $I = \left[\frac{k}{2^m}, \frac{k+1}{2^m}\right]$ that contains $\frac{a+b}{2}$. Since $[a, b]$ is dyadic, we know that $a = \frac{j}{2^m}$ and $b = \frac{j+1}{2^m}$, so $\frac{a+b}{2} = \frac{2j+1}{2^{n+1}}$. We also know that $m > n$, since $I \subsetneq [a, b]$, as S is non-trivial. All these facts taken together give us the following inequality

$$\begin{array}{ccccc} \frac{k}{2^m} & < & \frac{a+b}{2} & < & \frac{k+1}{2^m} \\ \frac{k}{2^m} & < & \frac{2j+1}{2^{n+1}} & < & \frac{k+1}{2^m} \\ k & < & (2j+1)2^{m-n-1} & < & k+1 \end{array}$$

which implies the existence of an integer between k and $k + 1$. Thus, we arrive at a contradiction. □

Theorem 1.1.7. *Every nontrivial separation S of the unit interval into dyadic intervals contains a pair of adjacent intervals which can be reduced.*

This theorem can be proved directly, or by induction. The direct proof is quicker, but the inductive proof can be generalized to $\mathbb{R}^2$, which we will need to do in Section 1.4. Both proofs are below.

One Proof of Theorem 1.1.7. Let S be a separation of the unit interval into dyadic intervals. The first interval in S must be a left interval (since $k = 0$), and the last must

be a right (since $k+1$ must equal 2^m for some m). It follows that somewhere in S there are an adjacent left and right intervals such that $I_1 = \left[\frac{k}{2^m}, \frac{k+1}{2^m}\right]$ and $I_2 = \left[\frac{j}{2^n}, \frac{j+1}{2^n}\right]$.

Since I_1 and I_2 are adjacent, we know that $\frac{k+1}{2^m} = \frac{j}{2^n}$, and that $k+1$ and j are both odd. Since $2 \nmid k+1$ and $2 \nmid j$, we know that $m = n$ and hence $k+1 = j$. Thus I_1 and I_2 can be reduced. □

Another Proof of Theorem 1.1.7. Let our induction hypothesis be as follows: Any separation of the unit interval into k pieces, where $1 < k < n$ has two adjacent intervals that can be reduced.

Let $k = 2$. Then $S = \left\{\left[0, \frac{1}{2}\right], \left[\frac{1}{2}, 1\right]\right\}$, and we can reduce those two intervals.

Let S be a separation of the unit interval into n dyadic intervals. Then $n \geq 3$, so S is nontrivial, so by Lemma 1.1.6 there is a cut in S at $\frac{1}{2}$. Furthermore, we know that at least one half of the S contains at least 2 intervals.

Let S' be the dyadic separation of the unit interval created by taking a half of S containing at least 2 intervals, and then multiplying all of the intervals contained in that half by a factor of 2. S' has k intervals, where $1 < k < n$, so by our hypothesis, it contains two intervals I_1 and I_2, of the form $\left[\frac{j}{2^m}, \frac{j+1}{2^m}\right]$ and $\left[\frac{j+1}{2^m}, \frac{j+2}{2^m}\right]$ that can be reduced. It follows that S contains two intervals of the form $\left[\frac{j}{2^{m+1}}, \frac{j+1}{2^{m+1}}\right]$ and $\left[\frac{j+1}{2^{m+1}}, \frac{j+2}{2^{m+1}}\right]$. Therefore S contains an adjacent pair of intervals which can be reduced. □

Corollary 1.1.8. *Every non-trivial dyadic separation S of a dyadic interval $I = [\frac{k}{2^m}, \frac{k+1}{2^m}]$ can be reduced to I itself, because every such S has a reducible pair of left and right intervals.*

We can scale every interval in S by 2^m and shift it by $-k$, to obtain a dyadic separation T of the unit interval, which we have just proven contains a reducible pair of left and right intervals. It follows that S has a reducible pair of left and right intervals, so reduce them. We showed after Theorem 1.1.2 that this results in another dyadic interval, so S reduced

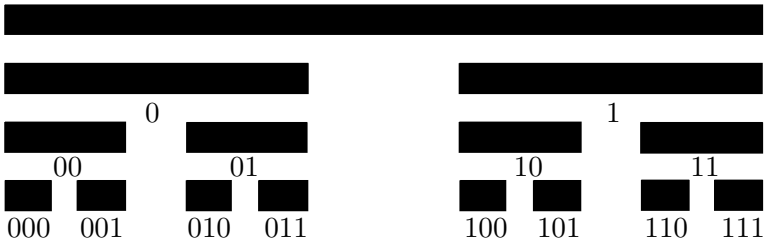


Figure 1.1.1: The first few iterations of constructing the Cantor set.[1]

is another dyadic separation S'. If S' is non-trivial, we can repeat the process until we arrive at a trivial separation, in which case we have reduced all the way to I.

Dyadic intervals are closely related to the Cantor set, as we will see below.

Definition 1.1.9. The **Cantor set** $\mathcal{C}$ is constructed by taking the unit interval, and removing the open middle third, $(\frac{1}{3}, \frac{2}{3})$, and then recursively removing the middle thirds of the resultant intervals ad infinitum. Every point that is not removed in this process is an element of the Cantor set. The first few steps of this process are shown in Figure 1.1.1 $\triangle$

The nth step of this process yields a set of intervals C_n, each of which are the left or right part of an interval in C_{n-1}. Note that $C_0 = [0, 1]$. Each of the intervals in the set C_n can be given by a finite binary string as in Figure 1.1.1, where a 0 as the final digit indicates that it is the left part of an interval in C_{n-1} and a 1 indicates that it is the right.

Because of this binary choice between left and right, every dyadic interval corresponds to an interval in some set C_n, and can also be represented with a binary string, as in Figure 1.1.2.

So we now have two ways of representing dyadic separations of the unit interval, namely a set of dyadic intervals, and a set of binary strings. We now introduce a third way of representing dyadic separations that will be useful for a few final proofs: the binary tree.

[1]Image by Eugen Anitas (Own work) [Public domain], via Wikimedia Commons

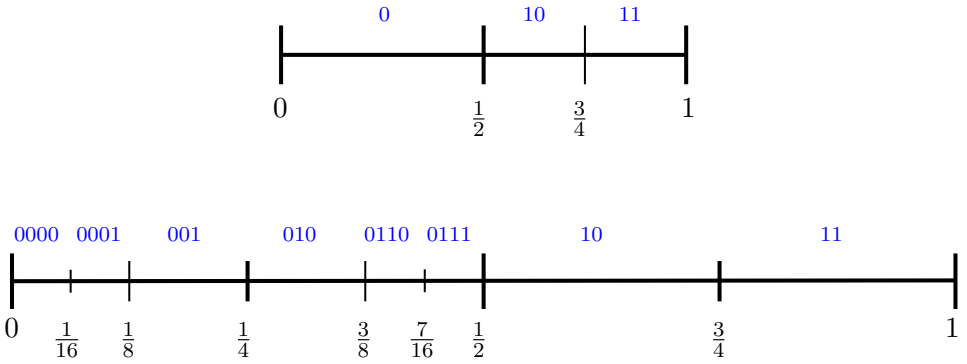


Figure 1.1.2: Two examples of translating dyadic separations into binary strings.

Lemma 1.1.6 tells us that every dyadic separation has a cut at $\frac{1}{2}$, and that either of those halves, if cut, is cut at $\frac{1}{2}$ that interval's length, and so forth. So we can use a binary tree to represent this sequence of cuts, where the root node corresponds the interval $[0, 1]$ and each left and right child of a node represent the left and right halves of its corresponding interval, if that interval is cut in half. The leaves of such a tree represent the intervals in a dyadic separation of the unit interval. An example of this correspondence is given in Figure 1.1.3.

Theorem 1.1.10. *Any two separations of the unit interval into dyadic intervals have a common refinement, which is also a dyadic separation.*

Proof. Let S_1 and S_2 be two dyadic separations of the unit interval, and let T_1 and T_2 be their corresponding trees. Let $T = T_1 \cup T_2$, and let S be the corresponding separation. We claim that S is a common refinement of S_1 and S_2. Each interval in S corresponds to a leaf in T, which is a leaf in either T_1 or T_2, and therefore corresponds to an interval in S_1 or S_2. Thus, every interval in S is an interval in S_1 or S_2, and so it is a common refinement of the two. □

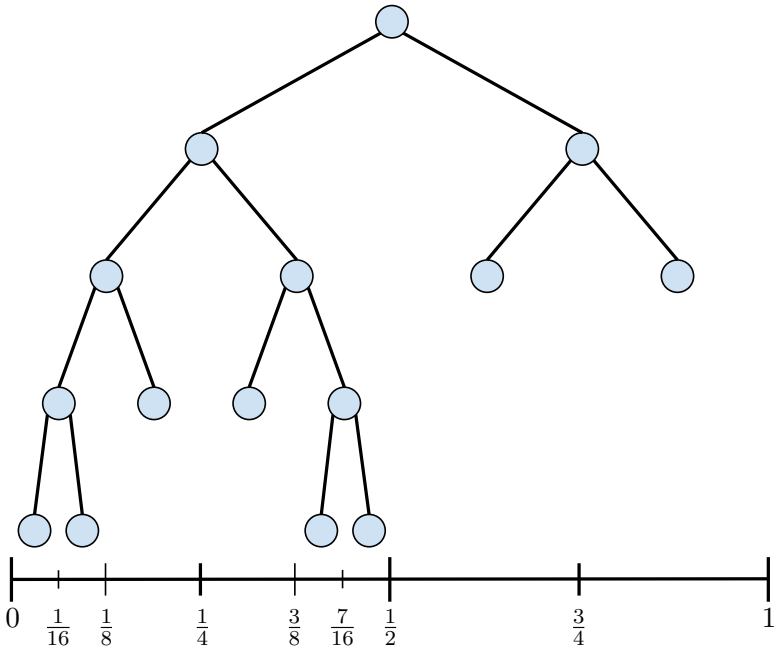


Figure 1.1.3: An example of representing a dyadic separation of the unit interval using a binary tree.

This tree representation will be used in one final proof about dyadic intervals, namely the following.

Theorem 1.1.11. *Given an interval $I \subseteq [0,1]$ with dyadic endpoints, there exists a unique reduced separation S of I into standard dyadic intervals.*

Before we prove this theorem, we need to lay some groundwork first. Let $I = \left[\frac{k}{2^m}, \frac{j}{2^n}\right]$ be an interval with dyadic endpoints contained in $[0,1]$, and let T be the complete binary tree of height $N = \max\{m,n\}$. Color all the leaves in T that are contained in I. Next, iteratively color the parents of the nodes you last colored if both children of that parent are colored, until there are no new nodes to color. This is guaranteed to be the case eventually, since T has a finite height, N. An example of this process is given in Figure 1.1.4.

Claim 1.1.12. *Every colored node whose parent is uncolored in T represents a maximal standard dyadic interval contained in I.*

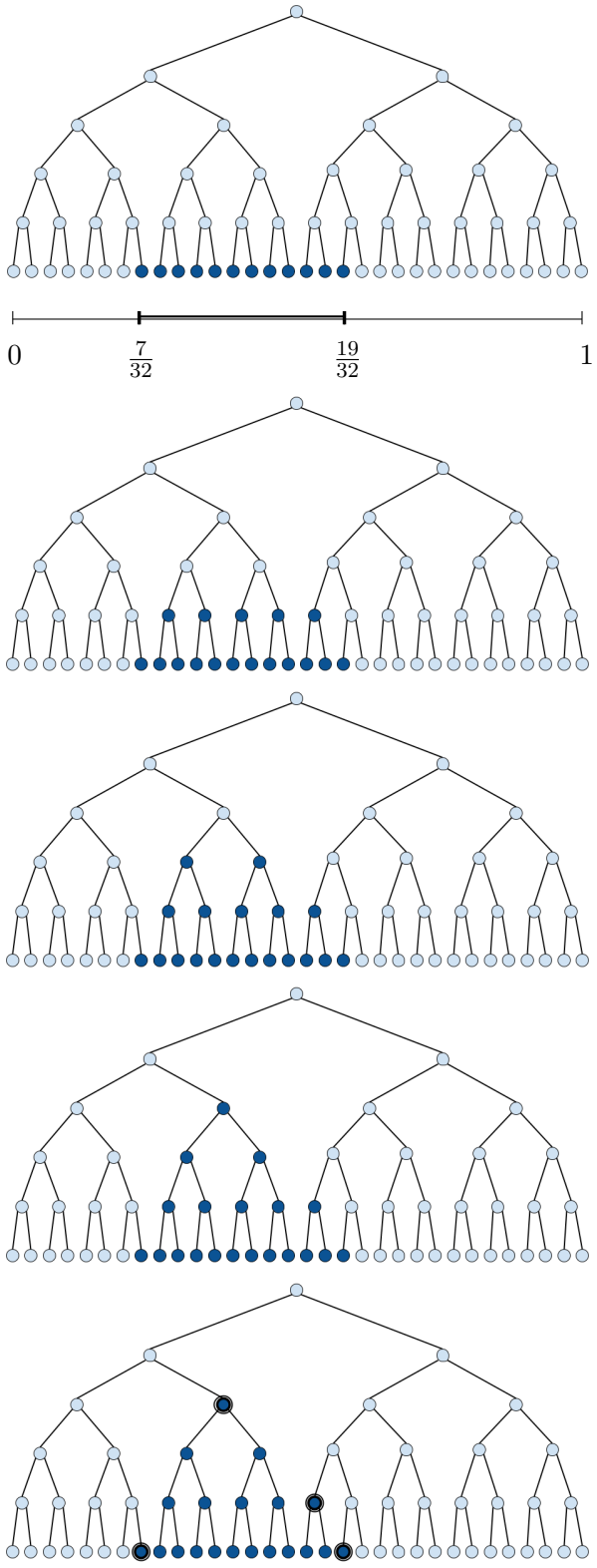


Figure 1.1.4: Finding the maximal standard dyadic intervals contained in $\left[\frac{7}{32}, \frac{19}{32}\right]$, namely $\left[\frac{7}{32}, \frac{1}{4}\right]$, $\left[\frac{1}{4}, \frac{1}{2}\right]$, $\left[\frac{1}{2}, \frac{9}{16}\right]$, and $\left[\frac{9}{16}, \frac{19}{32}\right]$.

Proof. Let S be the proposed set of intervals, and suppose $J \in S$ is an interval whose parent node in T was uncolored, but is not maximal. Then there exists $K \in S$ such that $J \subset K$. Since $K \in S$, we know that K's node in T is colored, and since $J \subset K$, we know that K's node is an ancestor of J's node in T. However, the fact that J's parent is uncolored in T implies that all of J's ancestors are uncolored. Thus we arrive at a contradiction. □

Proof of Theorem 1.1.11. Let $I = \left[\frac{j}{2^m}, \frac{k}{2^n}\right]$ be such an interval, and let S be the set of maximal standard dyadic intervals contained in I.

The proof will follow this list of criteria that S must satisfy:

i. S is a separation of I.

ii. S is reduced.

iii. There does not exist another reduced separation of I besides S.

Part i. Let $\mathcal{H} = \left\{H = \left[\frac{h}{2^N}, \frac{h+1}{2^N}\right] \mid N = \max\{n, m\}, \frac{j}{2^m} \leq \frac{h}{2^N}, \frac{h+1}{2^N} \leq \frac{k}{2^n}\right\}$, and let $x \in I$.

It should be fairly obvious that $\bigcup \mathcal{H} = I$, since each $H \in \mathcal{H}$ corresponds to one of the nodes we initially colored in T to represent I. Thus $x \in I$ implies $x \in H$ for some $H \in \mathcal{H}$. Furthermore, each $J \in S$ was constructed by reducing left and right pairs of intervals in $\mathcal{H}$, as modeled by coloring nodes whose children were both colored in T. So for all $H \in \mathcal{H}$, $H \subseteq J$ for some $J \in S$. (This also follows from the fact that all $J \in S$ are maximal, and that H is a dyadic subinterval of I.) Thus we see simultaneously that $\bigcup S = \bigcup \mathcal{H} = I$ and that $x \in I$ implies $x \in H$ for some $H \in \mathcal{H}$, which implies $x \in J$ for some $J \in S$.

Part ii. Suppose S is not reduced. Then there exist a left and right pair of intervals $J, K \in S$ that can be reduced, contradicting the fact that all $J, K \in S$ are maximal.

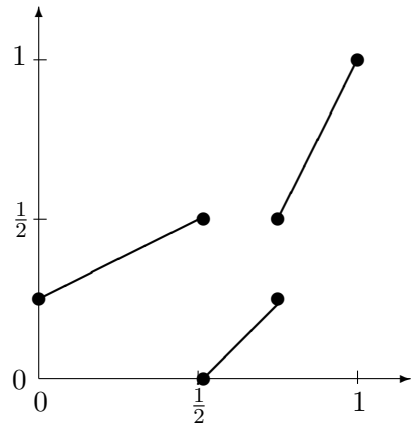


Figure 1.2.1: A function $f(x)$ in V. This function is not well-defined at $\frac{1}{2}$ and $\frac{3}{4}$, but we are not concerned with what the value of $f(x)$ is at these points.

Part iii. Suppose R is another reduced separation of I into dyadic intervals. Then there is at least one point, x, at which R differs from S. Let J_R and J_S be the intervals in R and S respectively. Since R and S differ at x, $J_R \neq J_S$. Since J_R and J_S are both dyadic intervals, it follows that either $J_R \subset J_S$, $J_S \subset J_R$, or $J_R \cap J_S = \emptyset$. We know that the last scenario is not the case, since $x \in J_R \cap J_S$. If $J_S \subset J_R$, this contradicts the fact that J_S is maximal. If $J_R \subset J_S$, then since J_S is maximal, we can conclude that J_R is a left or right subinterval whose mate is also contained in J_S. This contradicts the supposition that R is reduced, and we have exhausted all possibilities. □

1.2 Thompson's Group V

Functions in V are piece-wise, linear bijections of the unit interval that are differentiable at all but finitely many dyadic points, such that $f'(x) = 2^k$ for some $k \in \mathbb{Z}$, for each maximal interval for which $f(x)$ is differentiable. An example of a function in V is given in Figure 1.2.1.

Definition 1.2.1. Using the approach taken in Cannon, Floyd, and Parry [9], let $f(x) \in V$ and let

$$0 = x_0 < x_1 < x_2 < \ldots < x_n = 1$$

be all the points for which $f(x)$ is not differentiable. Then we can define $f(x) = a_i x + A_i$ when $x_{i-1} \leq x \leq x_i$, where $a_i = 2^k$ for some $k \in \mathbb{Z}$, and A_i is a dyadic number. △

We use this definition of the elements of V to prove that V is a group.

Proof. We will show that V taken with function composition has closure, associativity, inverses, and an identity element.

1. *Closure:* Let$f(x), g(x) \in V$. Then $f(x) = a_i x + A_i$ if $x_{i-1} \leq x \leq x_i$, and $g(x) = b_j x + B_j$ if $x_{j-1} \leq x \leq x_j$, where a_i and b_j are 2^k and 2^l respectively, and A_i and B_j are both dyadic numbers. That is, $A_i = \frac{p}{2^q}$ and $B_j = \frac{s}{2^t}$. Then

$$\begin{aligned} fg = f(g(x)) &= f(b_i x + B_i) \\ &= a_i(b_j x + B_j) + A_i \\ &= a_i b_j x + a_i B_j + A_i \\ &= 2^k 2^l x + \frac{2^k s}{2^t} + \frac{p}{2^q} \\ &= 2^{k+l} x + \frac{2^{k+q} s + 2^t p}{2^{q+t}} \end{aligned}$$

   which is an element of V.

2. *Associativity:* Function composition is associative.

3. *Inverses:* Let $f(x) \in V$, and let $x \in [0, 1]$. Then $f(x) = a_i x + A_i$ if $x_{i-1} \leq x \leq x_i$, where a_i is 2^k, and A_i and B_j is a dyadic number. That is, $A_i = \frac{p}{2^q}$. Then

$$\begin{aligned} f^{-1}(x) &= \frac{x - A_i}{a_i} \\ &= a_i^{-1} x - a_i^{-1} A_i \\ &= 2^{-k} x + \frac{p}{2^{k+q}} \end{aligned}$$

   which is an element of V.

4. *Identity:* The identity function $f(x) = x$ is an element of V.

Thus, V forms a group. □

Let $f(x) \in V$, and consider the set $S = \{[x_0, x_1], [x_1, x_2], \ldots, [x_{n-1}, x_n]\}$, where $x_0, \ldots, x_n$ are the breakpoints of $f(x)$. Since $x_0 = 0$ and $x_n = 1$, and each x_i is a dyadic number, S is a dyadic separation of the unit interval. Furthermore, consider $f([x_{i-1}, x_i])$. For all $x \in [x_{i-1}, x_i]$, we stated above that $f(x) = a_i x + A_i$, where $a_i = 2^k$ and $A_i = \frac{p}{2^q}$ is a dyadic number. We also know that $x_{i-1} = \frac{s}{2^t}$ and $x_i = \frac{u}{2^v}$, since both are dyadic numbers. All this taken together gives us

$$\begin{aligned} f([x_{i-1}, x_i]) &= [a_i x_{i-1} + A_i, a_i x_i + A_i] \\ &= [\frac{2^k s}{2^t} + \frac{p}{2^q}, \frac{2^k u}{2^v} + \frac{p}{2^q}] \\ &= [\frac{2^{k+q} s + 2^t p}{2^{q+t}}, \frac{2^{k+q} u + 2^v p}{2^{q+v}}] \end{aligned}$$

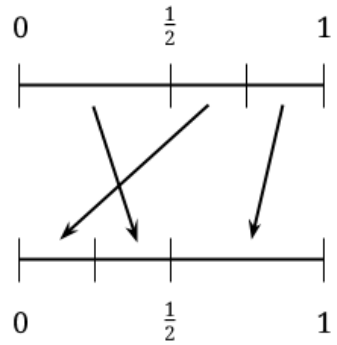
which, unsightly though it may be, is another dyadic interval. Since these functions are homeomorphisms, the union of the images of all the intervals in S is the entire unit interval. Furthermore, Theorem 1.1.3 tells us that $f(x) \in f(I)$ and $f(x) \in f(J)$ implies $I = J$. Thus the image of S is another dyadic separation of the unit interval.

Taking all this into consideration, we move to the idea of representing a function $f \in V$ with a tuple (S_1, S_2, φ), where S_1 and S_2 are dyadic separations of equal size, representing the domain and range of f, respectively, and $\varphi : S_1 \to S_2$ is a bijection, such that $x \in I$ where $I \in S_1$, implies that $f(x) \in \varphi(I)$, where $\varphi(I) \in S_2$. This abstract representation has a number of implementations, given below, and can also be generalized to $2V$.

One easy way of implementing the tuple of a function in V uses two pictures of unit intervals cut into S_1 and S_2, and arrows to indicate φ. For example, the function

$$f(x) = \begin{cases} \frac{1}{2}x + \frac{1}{4} & x \in \left[0, \frac{1}{2}\right] \\ x - \frac{1}{2} & x \in \left[\frac{1}{2}, \frac{3}{4}\right] \\ 2x - \frac{1}{4} & x \in \left[\frac{3}{4}, 1\right] \end{cases}$$

from Figure 1.2.1 can be described with this picture:



Another way of implementing function tuples in V is to use the binary string representations of the intervals in S_1 and S_2, and use φ to describe prefix replacement of any input $x \in [0, 1]$, translated to an infinite binary string. In the example from Figure 1.2.1,

$$\begin{aligned} \varphi(0) &= 01 & f(0w) &= 01w \\ \varphi(10) &= 00 & f(10w) &= 00w \\ \varphi(11) &= 1 & f(11w) &= 1w \end{aligned}$$

where $w \in \{0, 1\}^\infty$.

The last way of representing elements of V that we will discuss uses the binary tree representations of S_1 and S_2, and implements φ with a labeling, as in Figure 1.2.2.

Definition 1.2.2. A tuple (S_1, S_2, φ) of a function $f \in V$ is **reducible** if:

1. There exist intervals $L_1, R_1 \in S_1$ that are a reducible pair of left and right intervals.

2. There exist intervals $L_2, R_2 \in S_2$ that are a reducible pair of left and right intervals.

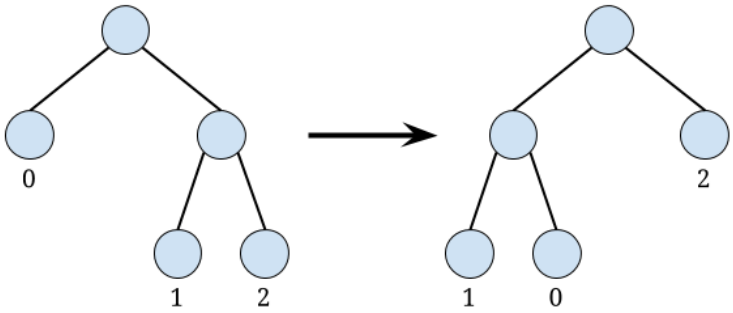


Figure 1.2.2: $f(x)$ described using binary trees.

3. Both $\varphi(L_1) = L_2$ and $\varphi(R_1) = R_2$. $\triangle$

Remark 1.2.3. Requirement 3 of Definition 1.2.2 can be shown algebraically to be equivalent to the statement that if $x \in L_1 \cup R_1$, then $f(x) = ax + A$. $\Diamond$

Let $I_1 = L_1 \cup R_1$ and let $I_2 = L_2 \cup R_2$. Observe that $x \in I_1$ implies

$$f(x) \in \varphi(L_1) \cup \varphi(R_1) = L_2 \cup R_2 = I_2.$$

Create the dyadic separation T_1 from S_1 by reducing L_1 and R_1 to I_1, and the dyadic separation T_2 from S_2 by reducing L_2 and R_2 to I_2. We showed after Corollary 1.1.8 that T_1 and T_2 are assured to be dyadic separations. Construct a bijection $\psi : T_1 \to T_2$ such that $\psi(I) = \varphi(I)$ for all $I \in S_1$ not equal to L_1 or R_1, and let $\psi(I_1) = I_2$. Observe that for all $J \in T_1$, if $x \in J$, the new bijection ψ holds that $f(x) \in \psi(J)$. That is, the tuple (T_1, T_2, ψ) describes the same function as the tuple (S_1, S_2, φ)

Notice that we can enact this process backwards as well. That is, we can "un-reduce" a tuple (S_1, S_2, φ) of a function f where $\varphi(I_1) = I_2$. We create T_1 by cutting $I_1 \in S_1$, and T_2 by cutting $I_2 \in S_2$. Cutting I_1 yields the pair of left and right intervals L_1 and R_1, and cutting I_2 yields L_2 and R_2. We construct a new bijection ψ such that $\psi(I) = \varphi(I)$ for all $I \in S_1$ except I_1, and we also let $\psi(L_1) = L_2$ and $\psi(R_1) = R_2$. One can verify that (T_1, T_2, ψ) describes the same function as (S_1, S_2, φ) by reducing $L_{1,2}$ and $R_{1,2}$

Theorem 1.2.4. *A function $f \in V$ with breakpoints $0 = x_0 < x_1 < \ldots < x_n = 1$ can be described with the tuple (S_1, S_2, φ), where $m = |S_1| = |S_2|$ can be arbitrarily large, provided that $m \geq n$.*

Proof. Let $f \in V$ be given by the tuple (S_1, S_2, φ), and without loss of generality, let $I \in S_1$, and let T be a non-trivial, dyadic separation of I with which we would like to replace I. (We could also want to replace some interval $J \in S_2$, and a similar proof would result.) Since T is non-trivial, then by Lemma 1.1.6 we know that there is a cut at the halfway point of T, so we can make that cut to I and "un-reduce" the tuple of f in the manner described above. This creates L and R, the left and right halves of I. which we can now recursively replace with $L \cap T$ and $R \cap T$, respectively, until the separation with which we're replacing I becomes trivial. □

Theorem 1.2.5. *Every element of V has a unique reduced tuple (S_1, S_2, φ).*

Proof by Contradiction. Let $f, g \in V$ be given by the reduced tuples (S_1, S_2, φ) and (T_1, T_2, ψ), respectively. Suppose that $f(x) = g(x)$ for all $x \in [0, 1]$, but there exists some $c \in [0, 1]$ such that c is in the interior of two intervals $I \in S_1$ and $J \in T_1$, and $I \neq J$.

By Theorem 1.1.3 we know that $I \subset J$ or $J \subset I$. Without loss of generality, suppose $J \subset I$. Intersect I with every interval in T_1 to create a set

$$U = \left\{ J' \cap I \mid J' \in T_1, \operatorname{Int}(J' \cap I) \neq \emptyset \right\}.$$

Notice that for each $J' \in T_1$, the intersection of J' and I is J' itself. This is again a result of Theorem 1.1.3, since I and J' overlap, so the only other option for $J' \cap I$ would be I. This would imply that J is contained in some J', which would contradict that both are intervals in a dyadic separation.

So, now we have a set U, which a separation of the interval I into dyadic intervals. By Theorem 1.1.7, U contains a reducible pair of left and right intervals, L and R. Further-

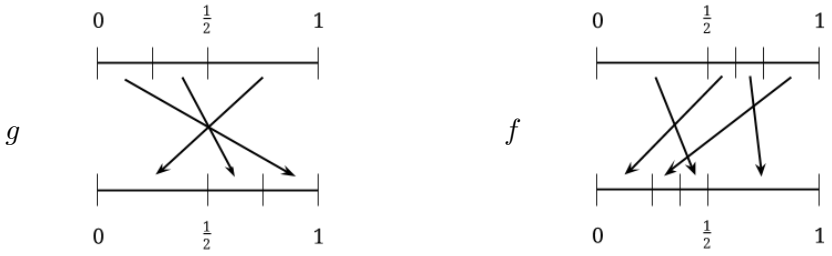
more, we know that $f(x) = a_I x + A_I$ for all $x \in I$, and that $f(x) = g(x)$ for all $x \in [0, 1]$. That is,

$$g(x) = \begin{cases} a_I x + A_I & x \in L \\ a_I x + A_I & x \in R \end{cases}$$

since L and R are both contained in I. So the tuple (T_1, T_2, ψ) is reducible, which is a contradiction. □

1.3 Multiplying Elements of V

Suppose we want to compose the following functions f, given by (S_1, S_2, φ), and g, given by (T_1, T_2, ψ) by first applying g, and then f to the unit interval:



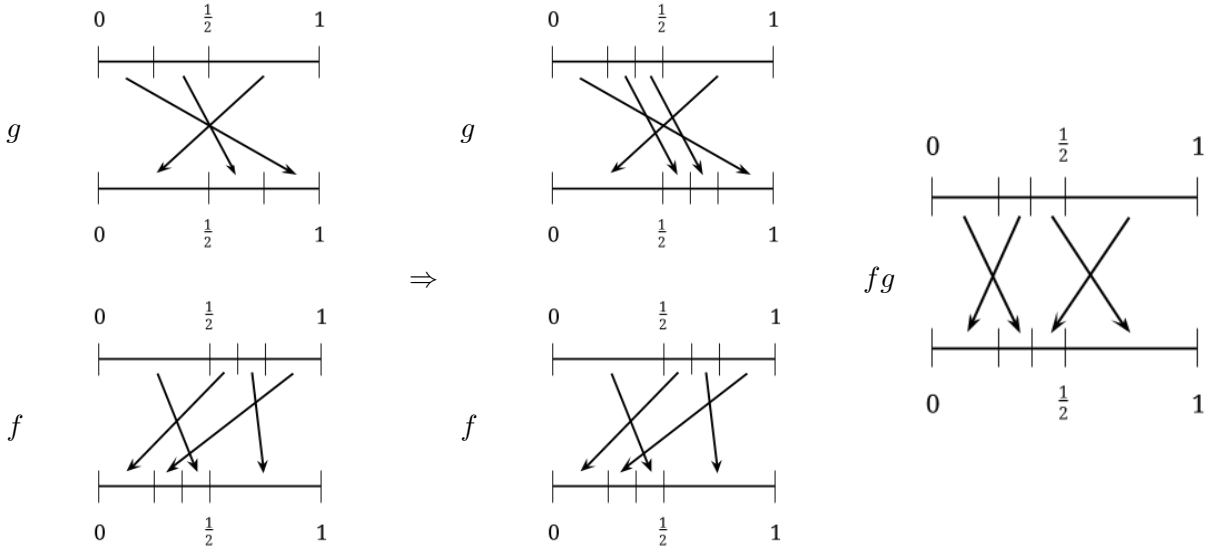
After we apply g, we are faced with a problem, namely that $\left[\frac{1}{2}, \frac{3}{4}\right] \in T_2$ but $\psi(\left[\frac{1}{2}, \frac{3}{4}\right])$ is undefined. To solve this problem, we must find a common refinement of T_2 and S_1. The following algorithm for finding a common refinement is similar to the technique used in the proof of Theorem 1.1.10, but with some slight changes, to accommodate the fact that the two dyadic separations are now parts of function tuples.

Algorithm 1.3.1. *Finding a common refinement of S_1 and T_2, given two functions $f, g \in V$, with tuples (S_1, S_2, φ) and (T_1, T_2, ψ), respectively.*

```
for J ∈ T_2 do
    for I ∈ S_1 do
        if I ⊂ J then
            Intersect J with S_1 to create a set U, which is a dyadic separation of J.
            Replace J with U as in the proof of Theorem 1.2.4.
        else if J ⊂ I then
            Intersect I with T_2 to create a set U, which is a dyadic separation of I.
            Replace I with U as in the proof of Theorem 1.2.4.
    end
end
```

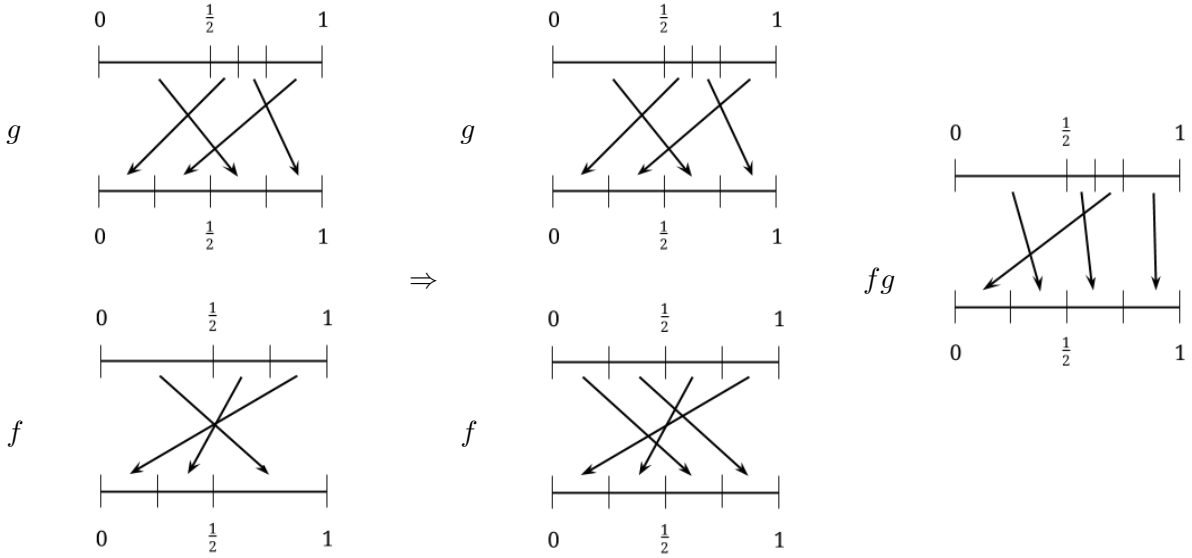
When we replace J or I with U, what we are doing is replacing a leaf node in either T_2 or S_1 with a subtree from the other, to create the union of the two trees as we did before in the proof of Theorem 1.1.10. Thus, we have created a common refinement of S_1 and T_2. Additionally, every interval $K \in T_1$ now maps to an interval $J \in T_2$ which is equal to some $I \in S_1$, which in turn maps to some interval $L \in S_2$. Knowing this, we can now create a third tuple (T_1, S_2, σ) to represent fg, where $\sigma(K) = \varphi(\psi(K))$ for all $K \in T_1$.

Example 1.3.2.



◊

Example 1.3.3.



◊

1.4 The Unit Square

In a few words, $2V$ is V in two dimensions. Instead of describing transformations of the unit interval separated into dyadic intervals, $2V$ describes transformations of the unit square, separated into dyadic rectangles. As such, we give definitions and theorems about the unit square in this section that are helpful for understanding $2V$.

Definition 1.4.1. A **dyadic rectangle** is the product of two dyadic intervals. That is, a dyadic rectangle is a rectangle of the form $R = \left[\frac{k}{2^m}, \frac{k+1}{2^m}\right] \times \left[\frac{j}{2^n}, \frac{j+1}{2^n}\right]$, where $j, k, m, n \in \mathbb{Z}_{\geq 0}$. In the context of two dimensions, a **dyadic separation** refers to a set of non-overlapping dyadic rectangles whose union is the unit square. △

Each separation of the unit square into dyadic rectangles can be constructed by cutting the unit square in half, either horizontally or vertically, then cutting any of the resultant rectangles in half vertically or horizontally, then cutting any of those resultant rectangles, and so forth. An example of a dyadic separation with cuts made in this manner is given in Figure 1.4.1.

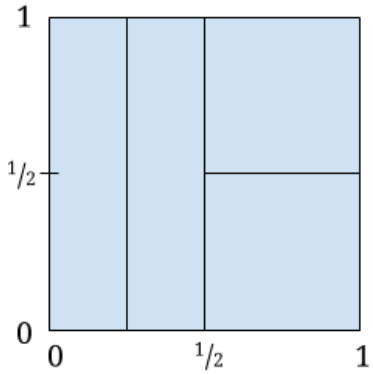


Figure 1.4.1: A separation of the unit square into the rectangles $\left[0, \frac{1}{4}\right] \times [0,1]$, $\left[\frac{1}{4}, \frac{1}{2}\right] \times [0,1]$, $\left[\frac{1}{2}, 1\right] \times \left[0, \frac{1}{2}\right]$, and $\left[\frac{1}{2}, 1\right] \times \left[\frac{1}{2}, 1\right]$. This separation was constructed by cutting the unit square in half vertically, then cutting its left half vertically, and finally, cutting the right-most of those three rectangles horizontally.

Observe that every time a cut is made, the x- or y-interval of the rectangle is split into a pair of dyadic intervals. Each rectangle in a separation of the unit square constructed in this way, then, is the product of two dyadic intervals. That is, it is a dyadic rectangle. A related statement is Lemma 1.4.2, which will be used for another proof about dyadic separations of the unit square.

Lemma 1.4.2. *Any nontrivial separation of the unit square into dyadic rectangles has either a horizontal cut or a vertical cut all the way across it. That is, either none of the rectangles in S contains $\frac{1}{2}$ in the interior of its x range, or none of the rectangles in S contains $\frac{1}{2}$ in the interior of its y range.*

Proof. Suppose the contrary. Then there exists a nontrivial separation S of the unit square into dyadic rectangles which has neither a horizontal nor a vertical cut all the way across it.

No horizontal cut implies there is at least one rectangle, call it R_h, which contains $\frac{1}{2}$ in the interior of its y range. By Lemma 1.1.6, R_h is a rectangle of the form $\left[\frac{k}{2^m}, \frac{k+1}{2^m}\right] \times [0,1]$, since $[0,1]$ is the only dyadic interval which contains $\frac{1}{2}$ in its interior.

By the same token, no vertical cut implies there is a rectangle R_v of the form $[0,1] \times \left[\frac{j}{2^n}, \frac{j+1}{2^n}\right]$.

Consider R_h. We know that $\left[\frac{k}{2^m}, \frac{k+1}{2^m}\right] \neq [0,1]$, otherwise R_h would be the unit square, and S is non-trivial. Similarly, if we consider R_v, we see that $\left[\frac{j}{2^n}, \frac{j+1}{2^n}\right] \neq [0,1]$.

Let (x,y) be in the interior of $\left[\frac{k}{2^m}, \frac{k+1}{2^m}\right] \times \left[\frac{j}{2^n}, \frac{j+1}{2^n}\right]$. Then (x,y) is contained in both R_h and R_v, which is a contradiction, since S is a separation of the unit square. □

Definition 1.4.3. Two dyadic rectangles, L and R, are **left** and **right** rectangles if they have the form

$$L = [\frac{k}{2^m}, \frac{k+1}{2^m}] \times [x,y] \text{ and } R = [\frac{k+1}{2^m}, \frac{k+2}{2^m}] \times [x,y]$$

where k is even and $[x,y]$ is some dyadic interval.

Two dyadic rectangles, B and T, are **bottom** and **top** rectangles if they have the form

$$B = [x,y] \times [\frac{k}{2^m}, \frac{k+1}{2^m}] \text{ and } T = [x,y] \times [\frac{k+1}{2^m}, \frac{k+2}{2^m}].$$

where k is even and $[x,y]$ is some dyadic interval.

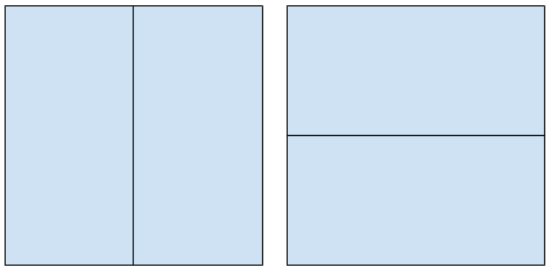
Two rectangles are **reducible** if they are matching left and right, or bottom and top rectangles. △

With some algebra, one can verify that L and R reduce to $[\frac{l}{2^{m-1}}, \frac{l+1}{2^{m-1}}] \times [x,y]$, and B and T reduce to $[x,y] \times [\frac{l}{2^{m-1}}, \frac{l+1}{2^{m-1}}]$, where $l = \frac{k}{2}$.

Now, as we were able to use Lemma 1.1.6 to show that every dyadic separation of the unit interval can be reduced to the unit interval itself, we can use Lemma 1.4.2 to prove the equivalent statement in $\mathbb{R}^2$.

Theorem 1.4.4. *Every nontrivial separation S of the unit square into dyadic rectangles can be reduced to the unit square itself, because every such S contains a pair of adjacent rectangles which can be reduced.*

Proof. As a base case, let $k = 2$. There are only two ways to separate the unit square into two dyadic rectangles:



Both of these can be reduced to the unit square.

Next, suppose that any separation of the unit square into k dyadic rectangles, where $1 < k < n$, has two adjacent rectangles that can be reduced, and let S be a separation of the unit square into n dyadic rectangles. By Lemma 1.4.2, we know that S has either full horizontal cut or a full vertical cut.

Without loss of generality, suppose it is a horizontal cut. Since $1 < k < n$, we know that at least one half of S contains at least two rectangles. Take a half of S containing at least two rectangles, and scale the y coordinates of all of the rectangles by a factor of 2.

In this way, we have constructed S', a separation of the unit square into k dyadic rectangles, where $1 < k < n$. By our induction hypothesis, we know that S' contains a pair of rectangles

$$\left[\frac{k_1}{2^{m_1}}, \frac{k_1+1}{2^{m_1}}\right] \times \left[\frac{j_1}{2^{n_1}}, \frac{j_1+1}{2^{n_1}}\right], \left[\frac{k_2}{2^{m_2}}, \frac{k_2+1}{2^{m_2}}\right] \times \left[\frac{j_2}{2^{n_2}}, \frac{j_2+1}{2^{n_2}}\right]$$

that can be reduced.

It follows that S has a pair of rectangles

$$\left[\frac{k_1}{2^{m_1+1}}, \frac{k_1+1}{2^{m_1+1}}\right] \times \left[\frac{j_1}{2^{n_1+1}}, \frac{j_1+1}{2^{n_1+1}}\right], \left[\frac{k_2}{2^{m_2+1}}, \frac{k_2+1}{2^{m_2+1}}\right] \times \left[\frac{j_2}{2^{n_2+1}}, \frac{j_2+1}{2^{n_2+1}}\right]$$

that can be reduced. If S had had a horizontal cut, the only difference in this proof would be that we scale the x coordinates of one half of S. □

Theorem 1.4.5. *Any two separations S_1 and S_2 of the unit square into dyadic rectangles have a common refinement S_3 which is also a dyadic separation of the unit square.*

Algorithm 1.4.6. *Finding a common refinement, S_3, of two dyadic separations, S_1 and S_2, of the unit square.*

Let $S_1 = \{R_1, R_2, \ldots, R_n\}$, let $S_2 = \{T_1, T_2, \ldots, T_m\}$, and let $S_3 = \{\}$.
for $R_i \in S_1$ **do**
  for $T_j \in S_2$ **do**
    By definition, $R_i = I_1 \times I_2$, and $T_j = J_1 \times J_2$, where $I_{1,2}$ and $J_{1,2}$ are dyadic intervals.
    if $I_1 \subset J_1$ **then**
      if $I_2 \subset J_2$ **then**
        Add $I_1 \times I_2$ to S_3.
      else if $J_2 \subset I_2$ **then**
        Add $I_1 \times J_2$ to S_3.
    else if $J_1 \subset I_1$ **then**
      if $I_2 \subset J_2$ **then**
        Add $J_1 \times I_2$ to S_3.
      else if $J_2 \subset I_2$ **then**
        Add $J_1 \times J_2$ to S_3.
  end
end
return S_3

Claim 1.4.7. *S_3 is a dyadic separation of the unit square.*

Proof. Every rectangle in S_3 is the Cartesian product of the intersection of the x and y intervals of two dyadic rectangles $R_i \in S_1$ and $T_j \in S_2$. We know that R_i and T_i are dyadic rectangles, so their x and y intervals are dyadic intervals. By Theorem 1.1.3, the intersection of the x intervals will be one of the x intervals of R_i or T_j, and the intersection of the y intervals will be one of the y intervals of R_i or T_j. Thus, every rectangle in S_3 is a Cartesian product of two dyadic intervals, and is itself dyadic.

Let $p = (x, y) \in [0, 1] \times [0, 1]$. Since S_1 and S_2 are separations of the unit square, we know there exists $R \in S_1$ and $T \in S_2$ such that $p \in R$ and $p \in T$. Furthermore, we know that $R = I_1 \times I_2$ and $T = J_1 \times J_2$, where $I_{1,2}$ and $J_{1,2}$ are dyadic intervals. From this, we can gather that $x \in I_1 \cap J_1$ and $y \in I_2 \cap J_2$.

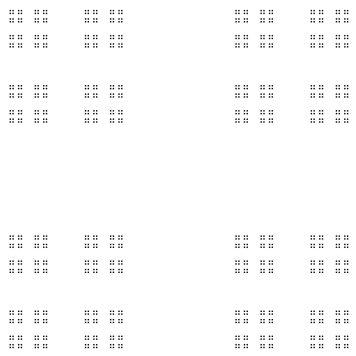


Figure 1.4.2: The Cantor square.[2]

Remark 1.4.8. If $I_1 \cap J_1 = \{x\}$ then x is an endpoint of both I_1 and J_1. If x is a left endpoint of I_1, then S_1 must contain some other rectangle $R' = I_3 \times I_4$, such that x is a right endpoint of I_3, and I_4 contains $I_2 \cap J_2$, otherwise no point (x_0, y) where $x_0 < x$ would be contained in any rectangle in S_1. Similarly, if x is a right endpoint, then S_1 must contain some other rectangle $R' = I_3 \times I_4$, such that x is a left endpoint of I_3, and I_4 contains $I_2 \cap J_2$. Likewise, if $I_2 \cap J_2 = \{y\}$, then y is a left or right endpoint of I_2, and we can find $R' = I_3 \times I_4 \in S_1$ such that y is the opposite endpoint of I_4 and I_3 contains $I_1 \cap J_1$. In any case, we let $R = R'$. ◊

Now we have $R = I_1 \times I_2$ and $T = J_1 \times J_2$ such that $p \in R \cap T$, and $|I_1 \cap J_1|, |I_2 \cap J_2| > 1$. Thus, the x and y intervals of the two rectangles overlap, and our algorithm adds the Cartesian product of their intersections to S_3. Therefore, for any point $p \in [0,1] \times [0,1]$, there exists a rectangle, call it Q, in S_3 such that $p \in Q$. Thus, we have proven Theorem 1.4.5. □

We saw previously a relationship between the Cantor set, $\mathcal{C}$, and dyadic intervals. There exists a similar relationship between dyadic rectangles and something called the Cantor square.

[2]By Hferee (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

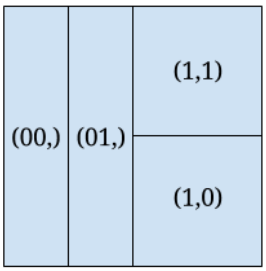


Figure 1.4.3: Figure 1.4.1 represented as ordered pairs of binary strings.

Definition 1.4.9. The **Cantor square**, $\mathcal{C} \times \mathcal{C}$, is the Cartesian product of the Cantor set with itself. A picture of the Cantor square, sometimes called Cantor dust, is given in Figure 1.4.2. △

For the same reasons that we could describe dyadic intervals with finite binary strings, we can describe dyadic rectangles using an ordered pair of finite binary strings, since each string corresponds to a dyadic interval, and every dyadic rectangle is the cartesian product of two dyadic intervals. An example is given Figure 1.4.3.

1.5 Brin-Thompson Group $2V$

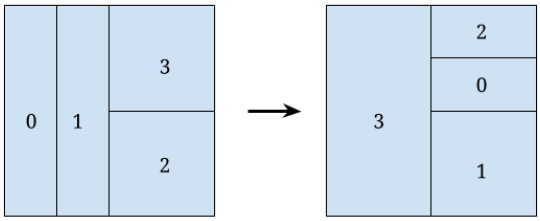
The group $2V$ is a finitely presented, simple group of functions which describe homeomorphisms of the Cantor square [6]. In V, functions were defined by dyadic breakpoints. Since they act on two dimensions, the functions in $2V$ can be thought of as having dyadic break-rectangles. A function $f : [0,1] \times [0,1] \to [0,1] \times [0,1]$ in $2V$ is differentiable except at the boundaries of finitely many dyadic rectangles $R_1, \ldots, R_n$, and f is defined piecewise so that $f(x,y) = (a_i x + A_i, b_i y + B_i)$ if $(x,y) \in R_i$, where a_i, b_i are both powers of 2 and A_i, B_i are dyadic numbers. The rectangles $R_1, \ldots, R_n$ form a separation of the unit square, and we can do the same algebra we did in Section 1.2 for both the x and y intervals of an arbitrary rectangle R_i in the domain of f, to show that $f(R_i)$ is another dyadic rectangle, and draw the conclusion that $f(R_1), \ldots, f(R_n)$ is another dyadic separation of the unit

square. Thus functions in $2V$ can be given by a tuple (S_1, S_2, φ), where S_1 and S_2 are two separations of the unit square into an equal number of dyadic rectangles, and $\varphi : S_1 \to S_2$ is a bijection, such that $(x, y) \in R$ where $R \in S_1$, implies that $f(x, y) \in \varphi(R)$, where $\varphi(R) \in S_2$.

As before, the most intuitive implementation uses two pictures of the unit square, cut into the separations S_1 and S_2. This time φ is given as a labeling of S_1 and S_2 with the numbers $0, \ldots, n-1$, where $n = |S_1| = |S_2|$. For example, the function

$$f(x,y) = \begin{cases} (2x + \frac{1}{2}, \frac{1}{4}y + \frac{1}{2}) & x \in [0, \frac{1}{4}], y \in [0,1] \\ (2x + \frac{1}{4}, \frac{1}{2}y) & x \in [\frac{1}{4}, \frac{1}{2}], y \in [0,1] \\ (x, \frac{1}{2}y + \frac{3}{4}) & x \in [\frac{1}{2}, 1], y \in [0, \frac{1}{2}] \\ (x - \frac{1}{2}, 2y - \frac{1}{2}) & x \in [\frac{1}{2}, 1], y \in [\frac{1}{2}, 1] \end{cases}$$

can be described with this picture:



We could also use binary strings associated with the x and y intervals of the rectangles in S_1 and S_2, translate any input point (x, y) to a pair of infinite binary strings, and describe φ in terms of prefix replacement in both coordinates, like so:

$$\varphi(00,) = (1, 10) \qquad \varphi(01,) = (1, 0)$$

$$\varphi(1, 0) = (1, 11) \qquad \varphi(1, 1) = (0,)$$

so that

$$f(00w_1, w_2) = (1w_1, 10w_2) \qquad f(01w_1, w_2) = (1w_1, 0w_2)$$

$$f(1w_1, 0w_2) = (1w_1, 11w_2) \qquad f(1w_1, 1w_2) = (0w_1, w_2)$$

where $w_1, w_2 \in \{0,1\}^\infty$.

Definition 1.5.1. A tuple (S_1, S_2, φ) for a function $f \in 2V$ is **reducible** if:

1. There exist rectangles $L_1, R_1 \in S_1$ that are a reducible pair of left and right rectangles.

2. There exist rectangles $L_2, R_2 \in S_2$ that are a reducible pair of left and right rectangles.

3. Both $\varphi(L_1) = L_2$ and $\varphi(R_1) = R_2$

or if:

1. There exist rectangles $B_1, T_1 \in S_1$ that are a reducible pair of left and right rectangles.

2. There exist rectangles $B_2, T_2 \in S_2$ that are a reducible pair of left and right rectangles.

3. Both $\varphi(B_1) = B_2$ and $\varphi(T_1) = T_2$ △

Just as before, we can also "un-reduce" a tuple (S_1, S_2, φ) for a function $f \in 2V$ by making identical cuts to corresponding pairs of rectangles in S_1 and S_2, and mapping the resultant left and right, or top and bottom, rectangles in the domain to the resultant left and right, or top and bottom, rectangles in the range.

Theorem 1.5.2. *A function $f \in 2V$ that is differentiable except at the boundaries of finitely many dyadic rectangles $R_1, \ldots, R_n$ can be described with the tuple (S_1, S_2, φ), where $m = |S_1| = |S_2|$ can be arbitrarily large, provided that $m \geq n$.*

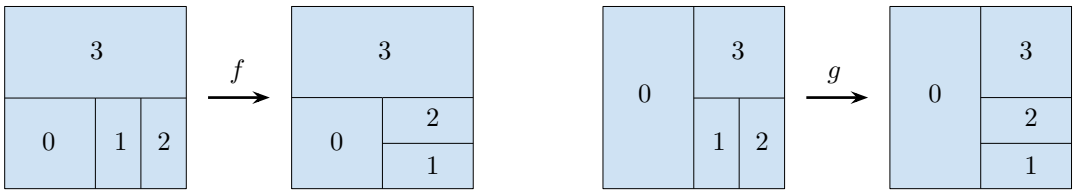
Proof. Let $f \in 2V$ be given by the tuple (S_1, S_2, φ), and without loss of generality, let $Q \in S_1$, and let U be a non-trivial, dyadic separation of Q with which we would like to

replace Q. (We could also want to replace some other rectangle $Q \in S_2$, and a similar proof would result.) Since U is non-trivial, then by Lemma 1.4.2 we know that there is vertical cut or a horizontal cut across U, so we can make the same cut to Q and "un-reduce" the tuple of f in the manner described above. This creates L and R, the left and right or top and bottom halves of Q, which we can now recursively replace with $L \cap U$ and $R \cap U$, respectively, until the separation with which we're replacing Q becomes trivial. □

We saw in Theorem 1.2.5 that for $f, g \in V$ with tuples (S_1, S_2, φ) and (T_1, T_2, ψ), respectively, that if $f(x) = g(x)$ for all $x \in [0, 1]$, then the two tuples can be reduced to be identical. This is not the case for $2V$.

Theorem 1.5.3. *A function $f \in 2V$ is not guaranteed to have a unique reduced tuple (S_1, S_2, φ).*

Example 1.5.4. Two functions $f, g \in 2V$ with respective tuples (S_1, S_2, φ) and (T_1, T_2, ψ) are given by the pictures below.



Both tuples are reduced, and for all $(x, y) \in [0, 1] \times [0, 1]$, it is the case that $f(x, y) = g(x, y)$, but $(S_1, S_2, \varphi) \neq (T_1, T_2, \psi)$. ◊

In Example 1.5.4 the rectangles labeled 0 and 3 form an L-shaped region that maps as the identity. This is a problem because an L-shaped region can be reduced two different ways, as the example shows. Unfortunately, this in not the only such troublesome shape. A cross-shaped region in the center of the unit square can be reduced any of the sixteen ways given in Figure 1.5.1. Any function which maps a cross-shaped region in the center of the unit square as the identity can be fully reduced and have any formation of the cross

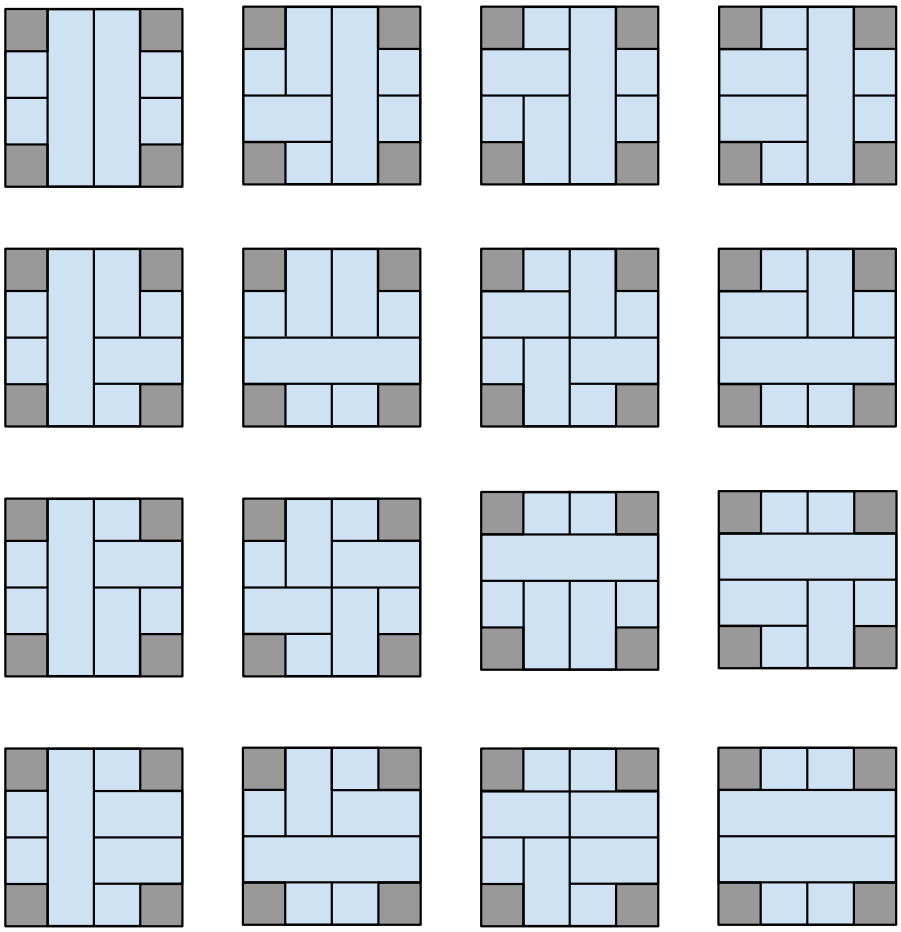


Figure 1.5.1: Sixteen reduced ways to represent a cross shaped area in the center of the unit square.

in its domain, and any formation of the cross in its range, meaning that any such function has at least 256 reduced tuples that represent it.

1.6 Multiplying Elements of $2V$

To multiply two functions $f, g \in 2V$ with respective tuples (S_1, S_2, φ) and (T_1, T_2, ψ) by first applying g, then f, we take the same approach we took with V. That is, we find a common refinement of S_1 and T_2. We again use a modified version the algorithm we used to find a common refinement of two dyadic separations of the unit square.

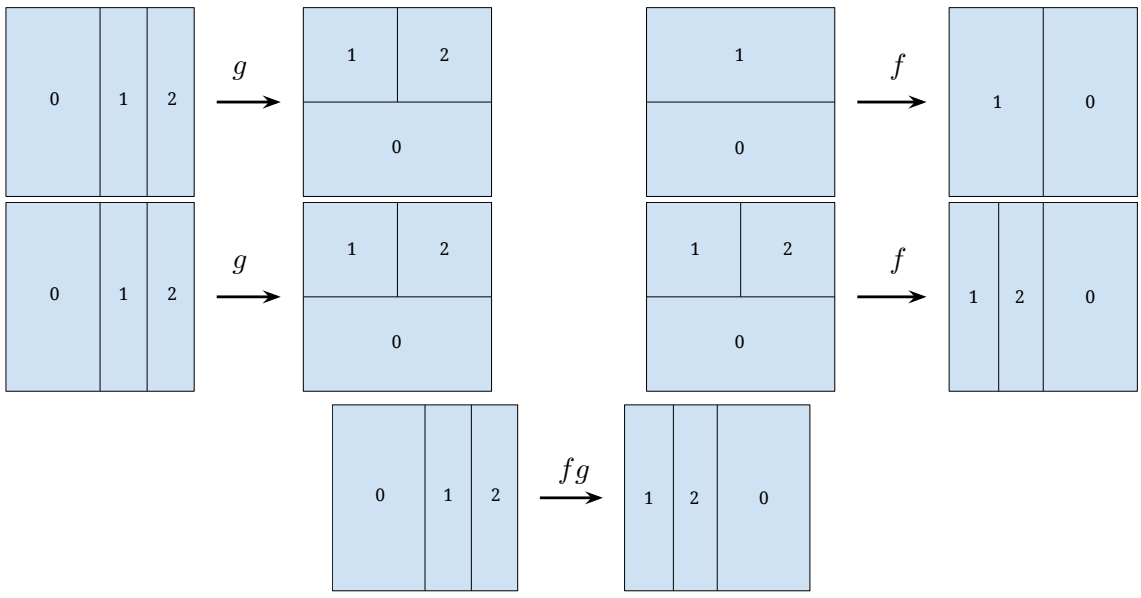
Algorithm 1.6.1. *Finding a common refinement of S_1 and T_2, given two functions $f, g \in 2V$, with respective tuples (S_1, S_2, φ) and (T_1, T_2, ψ).*

```
Let S = [{}_1, {}_2, ..., {}_n], where n = |S_1| = |S_2|.
for Q_i ∈ T_2 do
    Let T = {}.
    for R_j ∈ S_1 do
        if Q_i and R_j overlap then
            Add Q_i ∩ R_j to S[j] and to T.
    end
    Replace Q_i with T, according to Theorem 1.5.2.
end
for R_j ∈ S_1 do
    Replace R_j with S[j], according to Theorem 1.5.2.
end
```

Thus we have created a common refinement of S_1 and T_2, and for all $Q \in T_1$, we now can be assured that $\psi(Q) = R$, where $R \in S_1$. Thus we can construct a tuple (T_1, S_2, σ) to represent fg, where $\sigma(Q) = \varphi(\psi(Q))$ for all $Q \in T_1$.

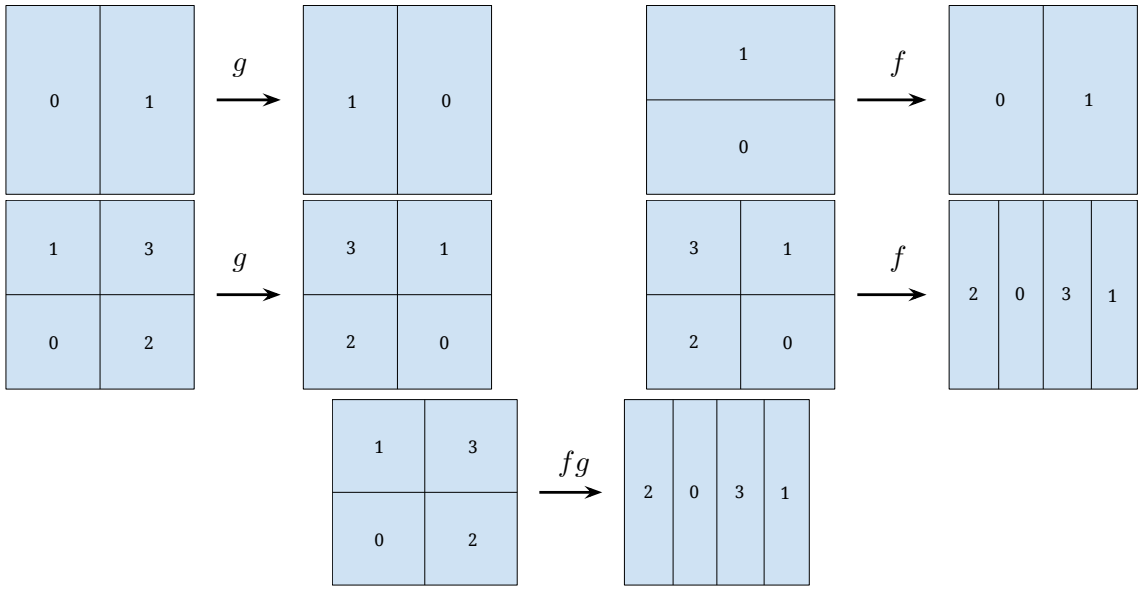
Some examples of the process are given below.

Example 1.6.2.



◊

Example 1.6.3.



◊

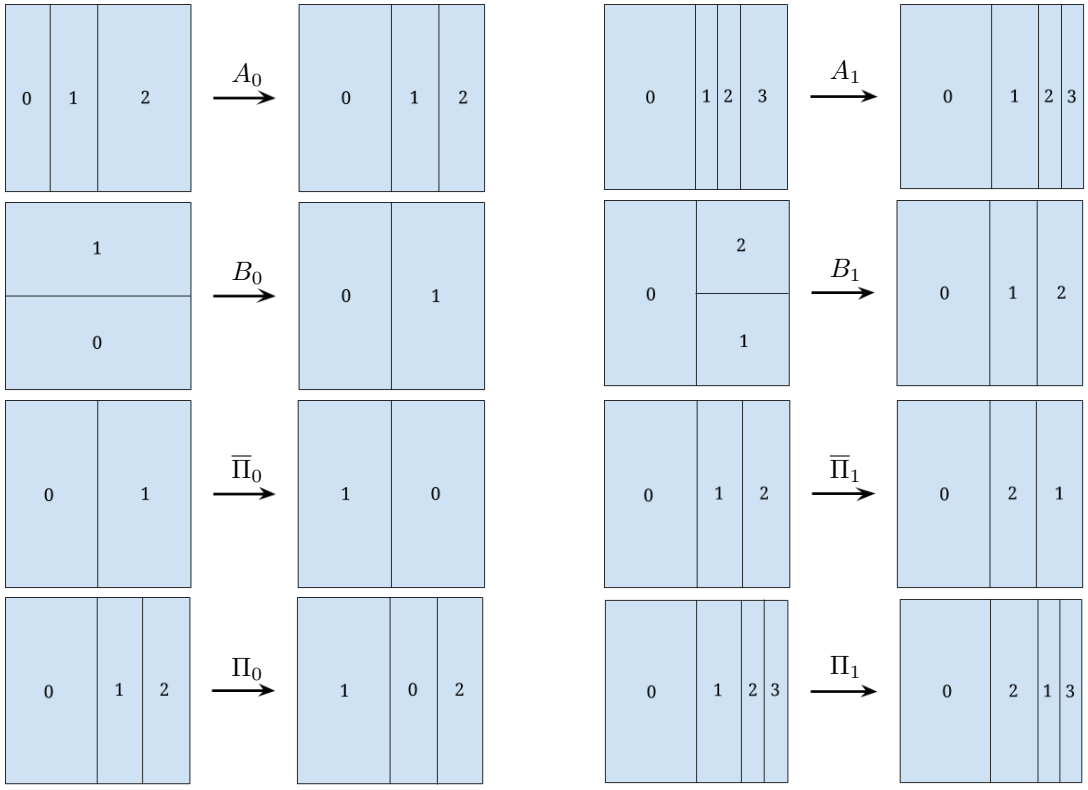
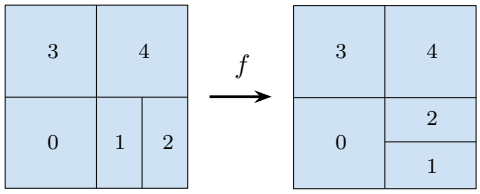


Figure 1.7.1: These eight functions, together with their inverses, form a generating set for $2V$. Notice that all the Π functions are their own inverses, and are therefore not included twice.

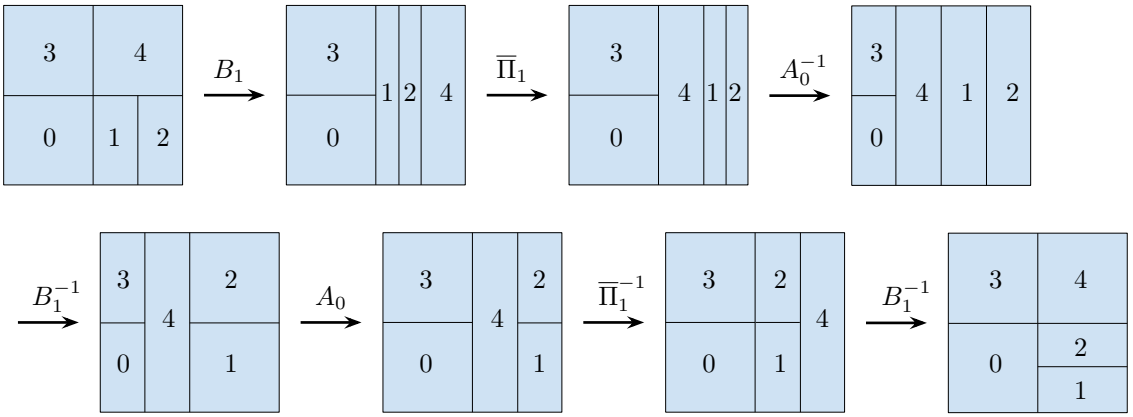
1.7 Generators of $2V$

As mentioned in the previous section, $2V$ is finitely presented and generated. Any function in $2V$ can be written as a word of the eight functions given in Figure 1.7.1 and their inverses [7].

Example 1.7.1. Let $f \in 2V$ be the function given by the drawing below.



We can write f as a word of generators.



Thus, $f = B_1^{-1}\overline{\Pi}_1^{-1}A_0B_1^{-1}A_0^{-1}\overline{\Pi}_1B_1$. ◊

2

Computational Problems and Group Based Cryptography

This chapter covers a small portion of the research already done in the field of group based cryptography. The first section defines the basis of the security of existing group based key agreement protocols, and the second details the Anshel-Anshel-Goldfeld key agreement protocol, as well the length based attack, which has been used to break other implementations of AAG.

2.1 Decision Problems for Groups

In 1911, Max Dehn formulated three decision problems for groups [10] which are now fundamental problems to consider for any group.

Definition 2.1.1. Let G be a finitely presented group. **Dehn's problems** (also called **decision problems**) for G are the following:

1. **Word problem**: Given a word w in the generators of G, does $w = e$, where e is the identity element of G?

2. **Conjugacy problem**: Given an arbitrary pair of words u and v in the generators of G, does there exist a word x in the generators of G such that $x^{-1}ux = v$?

3. **Isomorphism problem**: Given a finite presentation of a group H, does $H = G$?

If a computable algorithm exists to solve a decision problem for G, we say that problem is **decidable**. Otherwise, we say that it is **undecidable**. △

The word problem for $2V$ is decidable, since given a tuple (S_1, S_2, φ) for a function f, we can determine whether $f = e$ in linear time by asking whether $\varphi(R) = R$ for all $R \in S_1$.

Definition 2.1.2. Given a finitely presented group G, and an element $x \in G$, the **torsion problem** asks, does there exist $n \in \mathbb{N}$ such that $x^n = e$ where e is the identity element of G? △

The torsion problem is another decision problem, which was shown to be undecidable for $2V$ by Belk and Bleak in 2014 [3]. The decidability of the conjugacy problem is unknown for $2V$, but given Belk and Bleak's results about the torsion problem, it is likely that it is also undecidable. What this does say for certain, however, is that $2V$ is not isomorphic to V, since the torsion problem is decidable for V [6].

These problems are called decision problems because they require only a yes or no answer. The following are other problems we can ask about groups, but they are not decison problems, since both ask for an answer of one or more elements in G.

Definition 2.1.3. Given a group G, and two elements $g, h \in G$, the **conjugacy search problem (CSP)** tells us that there exists some $x \in G$ such that $x^{-1}gx = h$, and asks us to find at least one such x. △

Because the conjugacy problem is most likely undecidable for $2V$, the CSP is most likely infeasible. If some solution $c : G \times G \to G$ for the CSP has a computable runtime, then for any $g, h \in G$ we know the upper bound on the runtime of $c(g, h)$, and can simply

wait that long for a result. If we get a result x in that time, then we know that g and h are conjugate, otherwise we know that they are not. Thus, a computable runtime for a solution to the CSP implies that the conjugacy problem is decidable for that group.

Definition 2.1.4. The **simultaneous conjugacy search problem (SCSP)** is the CSP over multiple g and h. That is, given a group G, and two sets of elements $\{g_1, g_2, ..., g_n\}$ and $\{h_1, h_2, ..., h_n\}$, the SCSP tells us that there exists $x \in G$ such that $x^{-1}g_i x = h_i$ for all g_i, h_i, and asks us to find at least one such x. △

The difficulty of the SCSP is the basis of the security of AAG, as we see in the next section.

2.2 Anshel-Anshel-Goldfeld and the Length Based Attack

Suppose there exist two users, A and B, that wish to exchange private information despite having only a public means of communication. The two might like a way to encrypt their information. They need a way of establishing an encryption key between the two of them that no observer can calculate, despite being able to observe all the information A and B exchange to establish said key. In 1999, Anshel, Anshel, and Goldfeld proposed such a method, using groups [1].

Definition 2.2.1. (Notation follows that in [13]) The **Anshel-Anshel-Goldfeld key exchange protocol** requires a finite presentation of a group G, two users Alice and Bob, and a set of integers $\{N_{1,2}, L_{1,2}, L\}$ such that $N_{1,2}, L \in \mathbb{N}$ and $1 \leq L_1 \leq L_2$.

1. Alice randomly generates the public set $\overline{a} = \{a_1, \ldots, a_{N_1}\}$, where each a_i is a word in generators of G, of length between L_1 and L_2, such that each generator of G appears in $\overline{a}$.

	Alice	Bob
Public:	$\overline{a}$ $\overline{b}'$	$\overline{b}$ $\overline{a}'$
Private:	A K_A	B K_B

Table 2.2.1

2. Bob randomly generates the public set $\overline{b} = \{b_1, \ldots, b_{N_2}\}$, where each b_i is a word in generators of G, of length between L_1 and L_2, such that each generator of G appears in $\overline{b}$.

3. Alice randomly generates the secret element $A = a_{s_1}^{\epsilon_1} \cdots a_{s_L}^{\epsilon_L}$, where $1 \leq s_i \leq N_1$ and $\epsilon_i = \pm 1$, for each $1 \leq i \leq L$.

4. Bob randomly generates the secret element $B = b_{t_1}^{\delta_1} \cdots b_{t_L}^{\delta_L}$, where $1 \leq t_i \leq N_2$ and $\delta_i = \pm 1$, for each $1 \leq i \leq L$.

5. Alice computes and publicly transmits to Bob the set $\overline{b}' = \left\{b'_1, \ldots, b'_{N_2}\right\}$, where $b'_i = A^{-1} b_i A$ for all $1 \leq i \leq N_2$.

6. Bob computes and publicly transmits to Alice the set $\overline{a}' = \left\{a'_1, \ldots, a'_{N_1}\right\}$, where $a'_i = B^{-1} a_i B$ for all $1 \leq i \leq N_1$.

7. Alice computes $K_A = A^{-1} a'^{\epsilon_1}_{s_1} \cdots a'^{\epsilon_L}_{s_L} = A^{-1} B^{-1} A B = K$.

8. Bob computes $K_B = b'^{-\delta_L}_{t_L} \cdots b'^{-\delta_1}_{t_1} B = A^{-1} B^{-1} A B = K$. △

Once the two users have recovered a common element $K = K_A = K_B$, there are numerous ways to use K to create a shared encryption key. If there exists a feasible algorithm for putting elements of the group G (which we call the **platform group**) into a canonical form, that algorithm can be applied, and K can be used as the encryption key. Otherwise, there may be a quicker algorithm for solving the group's word problem,

in which case Bob can send rewritten forms of K, or other random group elements, to Alice, who then determines whether the element sent was K (denoting the bit 1), or another element (denoting 0). This method can be used to establish an encryption key of an arbitrary length.

The hope is that in order for an adversary to calculate K, they must calculate A and B by solving the SCSP over the sets $\overline{a}$ and $\overline{a}'$, and $\overline{b}$ and $\overline{b}'$. When braid groups were under consideration as a platform group, the Dehornoy form was proposed by Anshel, Anshel, and Goldfeld [1] as a standard form in which to rewrite K. Viable platform braid groups, however, were found by to be insecure for certain values of $N_{1,2}$, $L_{1,2}$, and L, due to their susceptibility to something known as the Length Based Attack [13].

Definition 2.2.2. The **Length Based Attack (LBA)**, keeping notation from Definition 2.2.1, takes as input the sets $\overline{a}$, $\overline{b}$, and $\overline{a}'$, and outputs a guess for Bob's secret element B, if it is successful. The chief requirement of the LBA is that there exists a length function $l : G \to \mathbb{R}$ for elements in the platform group G, such that for most $x, y \in G$ it is usually the case that $l(y) < l(x^{-1}yx)$. △

The LBA uses the knowledge that for each $a'_i \in \overline{a}'$

$$a'_i = B^{-1}a_iB = b_{t_L}^{-\delta_L} \cdots b_{t_1}^{-\delta_1}\ a_i\ b_{t_1}^{\delta_1} \cdots b_{t_L}^{\delta_L}$$

and tries to guess B by figuring out each of its conjugating factors in reverse order, starting with $b_{t_L}^{\delta_L}$ and working down to $b_{t_1}^{\delta_1}$. The attack tries every possible b_j^{δ} where $b_j \in \overline{b}$ and $\delta = \pm 1$. In general, it is usually the case that $l(b_j^{-\delta}a'_ib_j^{\delta}) > l(a'_i)$, unless we guess the inverse of the outermost b_t^{δ}, in which case b_t^{δ} and $b_t^{-\delta}$ cancel out at both ends, and the resultant length is less than the length before conjugation. If this process is unclear, see Example 2.2.3. The attack is described step-by-step in Algorithm 2.2.4.

Example 2.2.3. Let G be a group with only 2 generators, a and b. Let $x, y \in G$ and suppose $y = a^{-1}bxb^{-1}a$. Using the LBA, we can guess what x was conjugated by without knowing x. We compute:

$$a^{-1}ya = a^{-1}a^{-1}bxb^{-1}aa \qquad b^{-1}yb = b^{-1}a^{-1}bxb^{-1}ab$$

$$byb^{-1} = ba^{-1}bxb^{-1}ab^{-1} \qquad aya^{-1} = aa^{-1}bxb^{-1}aa^{-1} = bxb^{-1}$$

Presumably, since the first three add conjugating factors, their length will be greater than $l(y)$. The hope of the LBA is that $l(aya^{-1})$ will be the only one of these four with length less than $l(y)$, since a and a^{-1} will cancel out at both ends. If the LBA works as intended, we now know the outermost generator by which x was conjugated. In AAG, x will be an element of one of the users' public sets, so at this point, we can check if aya^{-1} is an element of one of those sets. Since it's not, we continue the process and try to guess the outermost conjugating factor of $aya^{-1} = bxb^{-1}$. We compute:

$$a^{-1}aya^{-1}a = a^{-1}aa^{-1}bxb^{-1}aa^{-1}a \qquad b^{-1}aya^{-1}b = b^{-1}bxb^{-1}b = x$$

$$baya^{-1}b^{-1} = baa^{-1}bxb^{-1}aa^{-1}b^{-1} \qquad aaya^{-1}a^{-1} = baa^{-1}bxb^{-1}aa^{-1}b^{-1}$$

This time, presumably $l(b^{-1}aya^{-1}b)$ is the only length that decreases. We check if this product is an element in a user set. Since it is, we know that we have recovered all the conjugating factors of x, and can calculate the element by which x was conjugated to arrive at y. $\Diamond$

L_1, L_2	$10, 13$	$20, 23$	$30, 33$	$40, 43$
Success rate	00%	51%	97%	96%

Table 2.2.2: LBA success rates from [13].

Algorithm 2.2.4. *Algorithm for the LBA.*

Input: $\overline{a}$, $\overline{a}'$, $\overline{b}$

Let $\overline{\alpha} = \{\alpha_1, \ldots, \alpha_{N_1}\} = \overline{a}'$, and $x = e$, the identity element in the platform group G.
while $\overline{\alpha} \neq \overline{a}$ **do** // If this loop terminates, $x = B^{-1}$.
  for b_j^{δ} where $b_j \in \overline{b}$ and $\delta = \pm 1$ **do**
    Compute $\Gamma(b_j^{\delta}) = \sum_{i=1}^{N_1} l(b_j^{-\delta} a_i b_j^{\delta})$.
  end
  Let b_k be the b_j^{δ} with the smallest $\Gamma(b_j^{\delta})$.
  $x \leftarrow x b_k^{-1}$.
  $\overline{\alpha} \leftarrow \left\{b_k^{-1}\alpha_1 b_k, \ldots, b_k^{-1}\alpha_{N_1} b_k\right\}$.
end
return x^{-1}

As mentioned in the introduction, Anshel, Anshel, and Goldfeld proposed braid groups as a platform group. See the background of Ko et al. for a good introduction to these groups [12]. Using the group B_{80} (the group of braids on 80 strands) as a platform group, with the values $N_{1,2} = 20$ and $L = 50$, Myasnikov and Ushakov found the LBA successful at the rates given in Table 2.2.2 [13].

3
Implementing $2V$: Data Structures and Algorithms

In order to use $2V$ as a platform group AAG, we needed an implementation with a very fast multiplication algorithm. Divide-and-conquer algorithms work well to this end, and by Lemma 1.4.2, we know that any nontrivial dyadic separation of the unit square has either a horizontal or vertical cut at $\frac{1}{2}$, so we have a very natural way to split the work of multiplication in half. In the first section of this chapter, we define a structure called an augmented binary tree (which we call `AugTree`) to represent dyadic separations of the unit square. Each subtree of an augmented binary tree also represents a dyadic separation, so that during multiplication, when we wish to find a common refinement of two trees, we can find a common refinement of their left and right subtrees recursively. In the second section, we give our implementation of an element of $2V$ (which we call `V2Function`, as Java does not allow a class name to start with a numerical character), including a description of how this multiplication algorithm works. We also pseudocode methods used by both of these classes in their respective sections. All verbatim code is available in the appendix. In the last section, we present data collected with our implementation, and posit that the worst case runtime for any multiplication algorithm is inherently quadratic.

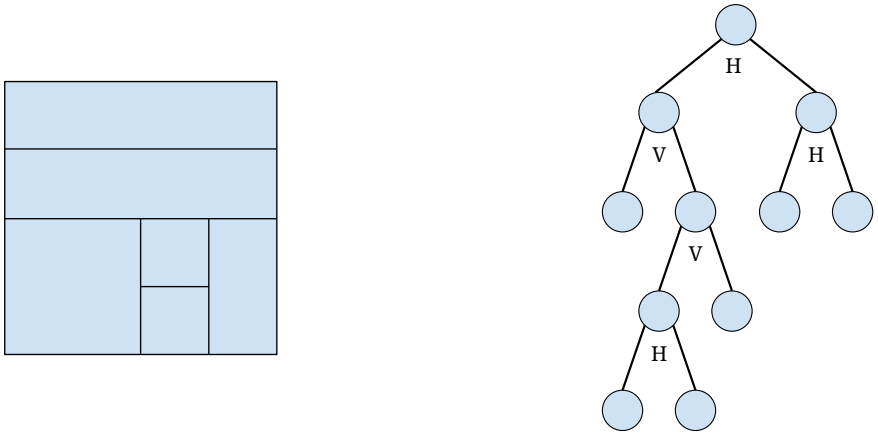


Figure 3.1.1: A dyadic separation S of the unit square, and a corresponding augmented binary tree T.

3.1 Augmented Binary Trees

In Section 1.2, we saw three ways of representing dyadic separations of the unit interval — pictures, binary strings, and trees — but for the unit square we only saw two representations — pictures and binary strings. We can also use an augmented version of the binary trees we used for dyadic separations of the unit interval. The only modification needed is a specification of whether a cut is horizontal or vertical. Instead of representing dyadic intervals, the nodes of the tree now represent dyadic rectangles. The left and right children of vertically cut nodes are, logically, the left and right rectangles that result from the cut. The left and right children of a horizontally cut node represent, respectively, the bottom and top rectangles that result from the cut. An example of a dyadic separation of the unit square and a corresponding augmented binary tree is given in Figure 3.1.1.

Implementing such a data structure is quite simple if one is familiar with the standard binary tree data structure, which consists of a value and two children which are also binary trees, and the class invariant that exactly zero or two of the children are non-null at any given time. An overview of our implementation is given below.

3.1.1 Class: AugTree

An **augmented binary tree** is a rooted, binary tree with the following fields:

- An `id` field holding a unique, random integer.
- Two fields, `cutHorz` and `cutVert`, holding boolean values.
- Six fields, `left` and `right`, `bottom` and `top`, `parent`, and `partner`, all of which are themselves augmented binary trees.

Class invariants:

- Exactly one or zero of the two fields `cutHorz` and `cutVert` is true at any given time.
- If `cutHorz` is `true`, then both the `bottom` and `top` fields are non-null.
- If `cutVert` is `true`, then both the `left` and `right` fields are non-null.
- If an augmented tree is not the root of the structure that it is in, then its `parent` field is non-null.
- The `partner` field is null until it is assigned by a `V2Function`, which we will see in the following section. If `partner` is non-null, it is always the case that the `partner` of the `partner` of an augmented binary tree T is T itself.

Our default constructor creates an `AugTree` object with a single node. We also include the obvious mutator methods `setLeft`, `setRight`, `setBot`, `setTop`, and `setPartner`, all of which maintain the class invariants.

Another, private constructor we include takes an `AugTree` parameter `p`, and creates a single node whose `parent` is `p`. This constructor is called by two methods `cutHorz()` and `cutVert()`, which maintain the class invariants. The method `cutHorz()` sets the field `cutHorz` to `true` and sets the object's `bottom` and `top` fields to new `AugTree` objects

created by the private constructor. Likewise, `cutVert()` sets the field `cutVert` to `true` and sets the object's `left` and `right` fields to new `AugTree` objects created by the private constructor. Both methods maintain the class invariants.

A method `getRelation()` takes as input an augmented binary tree T, and returns a string indicating whether T is the left, right, bottom, or top child of the tree that called `getRelation()`, or not a child. A `reduce()` method sets both `cutHorz` and `cutVert` to `false`, and all its children to `null`. The last quick and easy method is `isTrivial()` which returns `true` if both `cutHorz` and `cutVert` are `false`.

The `toString()` method returns the binary string representations of the x- and y-intervals of the dyadic rectangle that a given node corresponds to. The method calculates a node's x- and y- binary strings by crawling up the tree to the root node to ascertain its relative position in the tree. The algorithm is given in Algorithm 3.1.1. Writing `toString()` this way eliminates the need to store and constantly update several thousand binary strings for a function.

Algorithm 3.1.1. *An algorithm for calculating the binary strings corresponding to the x- and y-intervals of a node u in an augmented binary tree T.*

```
Let x = " " and let y = " ". Let p = u_parent.
while p is non-null do
    if p is cut horizontally then
        if u == p_bottom then
            y ← "0" + y
        else                                  // Must be the case that u == p_top, since p is
                                                  // cut horizontally and u is a child of p.
            y ← "1" + y
    else                          // p must be cut vertically, since p has a non-null child, namely u.
        if u == p_left then
            x ← "0" + x
        else                                  // Must be the case that u == p_right, since p is
                                                  // cut vertically and u is a child of p.
            x ← "1" + x
    p ← p_parent
end
return [x, y]
```

This more abstract representation is actually quite helpful when devising a more efficient algorithm for finding a common refinement of two dyadic separations of the unit square, but presents some limitations as well. For example, consider the dyadic separation in Figure 3.1.2. This separation could be represented with either of the trees below it, and both would be valid.

As such, our implementation needs to recognize that these distinct structures as equal, and report that the parent node appears to be cut both horizontally and vertically, despite the class invariants. We include two methods, `looksCutVert` and `looksCutHorz`, to aid us in this effort. The algorithm used by `looksCutVert` is given in Algorithm 3.1.2, and the algorithm used by `looksCutHorz` is identical, except for superficial changes to variable names. Notice that this method is also highly recursive. The structure that we engineered for recursive multiplication turns out to lend itself to recursion in most situations, since we can usually split a problem for a binary tree into the two subproblems of the subtrees that are its children.

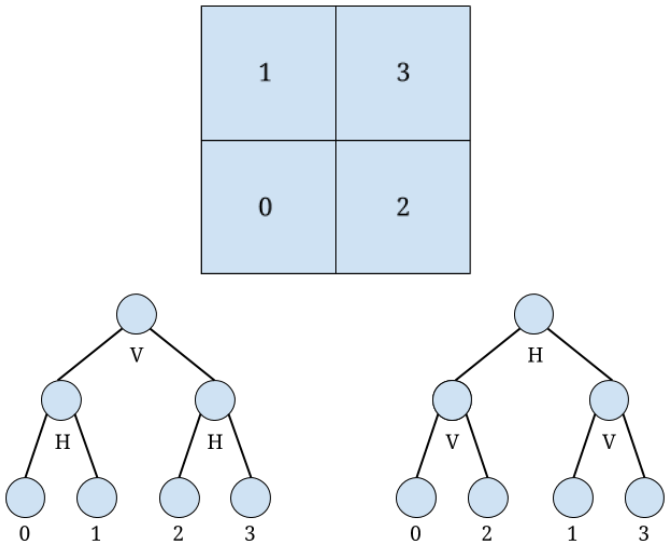


Figure 3.1.2: A troublesome dyadic separation of the unit square and its two valid tree representations.

Algorithm 3.1.2. *An algorithm for determining whether an augmented binary tree T represents a dyadic separation of the unit square that looks like it was cut vertically, which returns its left and right halves if so.*

```
1  Let u be the root node of T.
2  if u is cut vertically then          // Easy case. It looks cut vertically because it is.
3  |  return [u_left, u_right]
4  else if u is cut horizontally and both its children are non-trivial then
5  |  Let b = u_bottom.looksCutVert()   // Recursive call to this algorithm. Returns
   |                                    // left and right halves of u_bottom if so.
6  |  Let t = u_top.looksCutVert()      // Recursive call to this algorithm. Returns
   |                                    // left and right halves of u_top if so.
7  |  if both b and t are non-empty then
8  |  |  Let L and R be trivial augmented binary trees.
9  |  |  Cut both L and R horizontally.
10 |  |  L.setBot(b[0])
11 |  |  L.setTop(t[0])
12 |  |  R.setBot(b[1])
13 |  |  R.setTop(t[1])
14 |  |  return [L, R]
15 return [ ]
```

Now that we have a way to deal with the issue in Figure 3.1.2, we can use it to check the equality of two augmented binary trees S and T. The recursive equality algorithm given

in Algorithm 3.1.3 begins at the root nodes of both S and T, and crawls down both trees simultaneously, asking at each step if the two current nodes are cut in the same way (or at least appear to be, even if their labels H/V don't match), and if so, if both their children are equal. The algorithm returns true if we arrive at a leaf node at the same place in both trees for all leaves in both S and T.

Algorithm 3.1.3. *An algorithm for checking the equality of two augmented binary trees S and T.*

Let u be the root node of S and let v be the root node of T.
if u and v are both cut horizontally **then** // Easy case.
  return $u_{top} == v_{top}$ **and** $u_{\text{bottom}} == v_{\text{bottom}}$ // Recursive call to this algorithm.
else if u and v are both cut vertically **then** // Easy case.
  return $u_{left} == v_{left}$ **and** $u_{\text{right}} == v_{\text{right}}$ // Recursive call to this algorithm.
else if u is cut vertically and v is cut horizontally **then**
  Let $h = v$.`looksCutVert()` // Returns empty set if false,
    // else returns left and right halves of v.
  if h is empty **then**
    return false
  else
    return $u_{left} == h[0]$ **and** $u_{\text{right}} == h[1]$ // Recursive call to this algorithm.
else if u is cut horizontally and v is cut vertically **then**
  Let $h = v$.`looksCutHorz()` // Returns empty set if false,
    // else returns bottom and top halves of v.
  if h is empty **then**
    return false
  else
    return $u_{bottom} == h[0]$ **and** $u_{\text{top}} == h[1]$ // Recursive call to this algorithm.
else if u and v are both trivial **then** // We have crawled down both trees in the
    // same way and arrived at a leaf in both.
  return true
else // u is a leaf and v is not, or vice versa
  return false

3.1.2 Class: V2Function

Since we can represent a dyadic separation of the unit square with these augmented binary trees, any function in $2V$ can be described using two trees and a labeling, as we did for V in Section 1.2. An example of a tuple (S_1, S_2, φ) for a function $f \in 2V$ using augmented binary

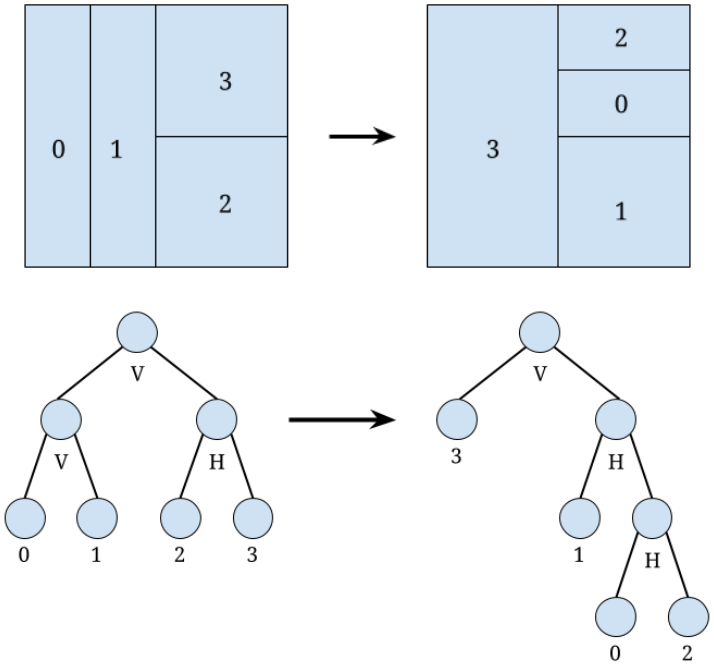


Figure 3.1.3: $f(x, y)$ represented as a pair of binary trees.

trees and a labeling is shown in Figure 3.1.3. As such, our implementation `V2Function` has the following:

- A `domain` and `range` field, which are the augmented binary tree representations of S_1 and S_2, respectively.
- A `domLeaves` and `ranLeaves` field, each of which are `ArrayList` data structures holding the leaves of `domain` and `range`, respectively.
- A `numLeaves` field, holding an integer indicating the size of this function's tuple.

Class invariant:

- At any given time, `numLeaves` = |`domLeaves`| = |`ranLeaves`| = the number of leaves in `domain` = the number of leaves in `range`.

Maintaining a carefully indexed list of the leaves of `domain` and `range` eliminates the need to crawl down the entire tree structure every time we need to access a leaf node.

The reader may notice that a φ function is conspicuously absent. The primary constructor for this class takes a parameter `phi` which is an `ArrayList` of integers used to initialize the `partner` relationships between the leaves of `domain` and the leaves of `range`, where `phi.get`$(i) = j$ means that φ maps the jth leaf of S_1 to the ith leaf of S_2. This eliminates the need for any sort of costly look up — for any leaf $u \in S_1$, we can know $\varphi(u)$ in constant time, since it will be u_{partner}. Should we ever need to recover φ, the `AugTree` class has a method `phi()` which constructs the proper `ArrayList` of integers by iterating over the leaves of S_2 and for each leaf node u, adding the index of the leaf u_{partner} in S_1.

We also include a default, zero-parameter constructor which creates the identity element of $2V$, and a private constructor with only `domain` and `range` parameters, which assumes that the partner relationships between the two are already initialized. This constructor is only called by the class' `multiply()` method, which we will arrive at after presenting some of the more obvious methods the `V2Function` class ought to have, as well as methods prerequisite for understanding the multiplication algorithm.

The `V2Function` class has a method `isIdentity()`, which solves the word problem for $2V$ using the short algorithm given after Definition 2.1.1.

Reducibility of function tuples is implemented with two methods `reduce()` and `reduceHelper()`. The `reduceHelper()` method uses Algorithm 3.1.4, and returns `true` if any reductions are made. The `reduce()` method calls `reduceHelper()` in a loop, until it returns `false`. This algorithm is extremely rudimentary in that it will only recognize a reducible pair of rectangles $L_1, R_1 \in S_1$ and $L_2, R_2 \in S_2$ if they are literally the same sort of child nodes in the `domain` and `range` trees, which may not always be the case due to the limitation illustrated in Figure 3.1.2. A possible solution could be a recursive algorithm down the `domain` and `range` trees with backtracking that figures out the most reducible structures for both.

Algorithm 3.1.4. *An algorithm for completing one pass of reductions over a tuple* (S_1, S_2, φ) *of a function* $f \in 2V$*, where* S_1 *and* S_2 *are given by augmented binary trees.*

```
1  Let b = false.
2  for u ∈ domLeaves do
3  |  if u_parent is null then    // Root is a leaf and f is the identity. Nothing to reduce.
4  |  |  return false
5  |  Let v = u_partner.
6  |  Let r_1 = u_parent.getRelation(u), and let r_2 = v_parent.getRelation(v).
   |                          // If u is not the root of S_1 then v cannot be the root of S_2,
   |                                  // since that would imply |S_2| = 1 < |S_1|.
7  |  if r_1 == r_2 then       // u and v are the same kind of child, go get their siblings.
8  |  |  Let p_1 = u_parent and let p_2 = v_parent.
9  |  |  Let x be the child of p_1 that is not u, and let y be the child of p_2 that is not v.
   |  |  if x_partner == y then                    // u & x reducible, v & y reducible,
   |  |  |                                                  // φ(u) = v, φ(x) = y.
10 |  |  |  Replace u in domLeaves with p_1 and replace v in ranLeaves with p_2.
11 |  |  |  Remove x from ranLeaves and remove y form domLeaves.
12 |  |  |  p_1.reduce()
13 |  |  |  p_2.reduce()
14 |  |  |  p_1.setParter(p_2)
15 |  |  |  numLeaves ← numLeaves − 1
16 |  |  |  b ←true
17 end
18 return b
```

Un-reducing is much easier to implement. In order for Theorem 1.5.2 to hold for our implementation, we create `cutHorz()` and `cutVert()` methods for the `V2Function` class. Algorithm 3.1.5 outlines the `cutVert()` method, and the `cutHorz()` functions analogously. Both take as parameters a leaf node u and a string `where` indicating whether u is in S_1 or S_2. The default value of `where` is "r" for "range," indicating S_2.

Algorithm 3.1.5. *An algorithm for cutting a leaf node u which, given a tuple (S_1, S_2, φ) for a function $f \in 2V$, may be in either S_1 or S_2.*

Input: `where` $\in$ {"d", "r"}

if `where` == "r" **then** // $u \in S_2$
  if $u \in$ `ranLeaves` **then** // Double-check it's really there.
    Let $v = u_{\text{partner}}$.
    u`.cutVert()`
    v`.cutVert()`
    u_{left}`.setPartner(`v_{left}`)`
    u_{right}`.setPartner(`v_{right}`)`
    Replace u in `ranLeaves` with u_{left}, and add u_{right} at the following index.
    Replace v in `domLeaves` with v_{left}, and add v_{right} at the following index.
    `numLeaves` $\leftarrow$ `numLeaves` $+ 1$
else if `where` == "d" **then** // $u \in S_1$
  if $u \in$ `domLeaves` **then** // Double-check it's really there.
    Let $v = u_{\text{partner}}$.
    u`.cutVert()`
    v`.cutVert()`
    u_{left}`.setPartner(`v_{left}`)`
    u_{right}`.setPartner(`v_{right}`)`
    Replace u in `domLeaves` with u_{left}, and add u_{right} at the following index.
    Replace v in `ranLeaves` with v_{left}, and add v_{right} at the following index.
    `numLeaves` $\leftarrow$ `numLeaves` $+ 1$

Before we can turn our attention to a multiplication algorithm, we define two more methods, `cutFullHorz()` and `cutFullVert()`, which again function analogously to each other. Both take as parameters a node u (in this case not usually a leaf) and a string `where` indicating if u is a node in `domain` or in `range`. The parameter `where` is unused until the base case, which cuts a leaf node with `cutVert()` or `cutHorz()`, and so requires that we know which tree u is located in. The method `cutFullVert()`, regardless of how u is currently cut, cuts u vertically, sets its left and right children to be the left and right halves of whatever dyadic rectangle u represented initially, and returns those halves as well. If this process seems unclear, see Figure 3.1.4 for an example.

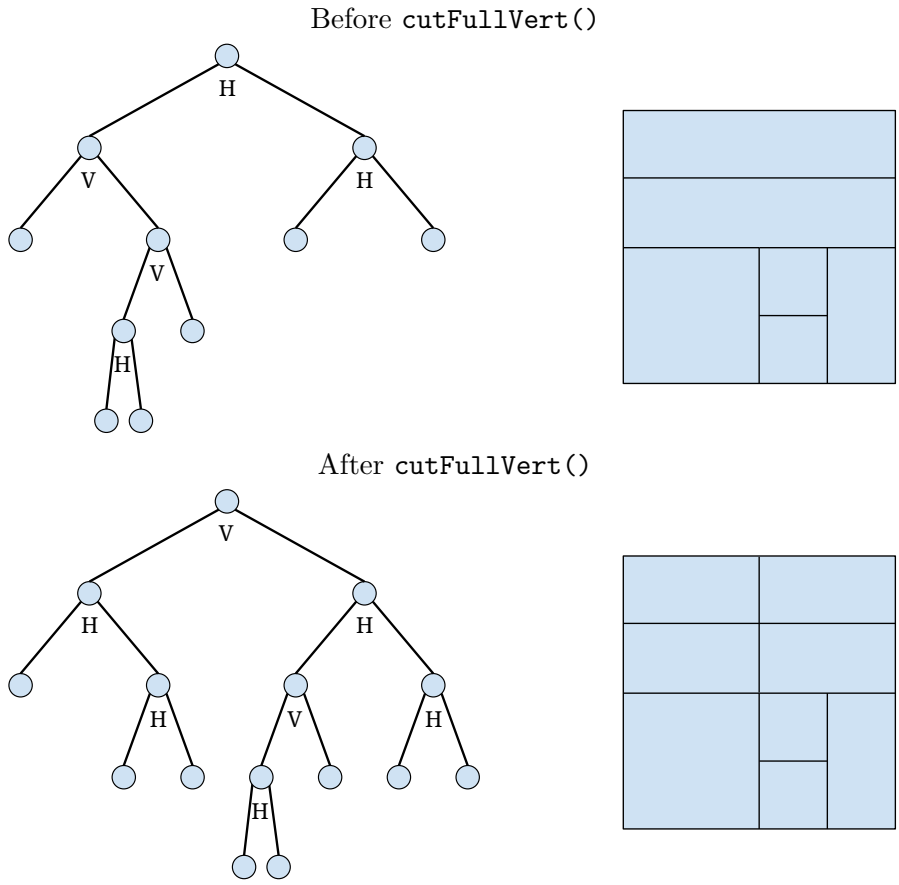


Figure 3.1.4: An augmented binary tree T and its corresponding dyadic separation S before and after `cutFullVert()` has been called on it.

Algorithm 3.1.6. An algorithm for fully cutting a node u vertically, regardless of whether or not u is a leaf.

Input: `where` $\in$ {"d", "r"}

if u is cut vertically **then** // Easy case
  return $[u_{\text{left}}, u_{\text{right}}]$
else if u is cut horizontally **then**
  Let $b =$ `cutFullVert`$(u_{\text{bottom}}, \texttt{where})$ // Recursive call to this algorithm
    // returns left and right halves of u_{bottom}.
  Let $t =$ `cutFullVert`$(u_{\text{top}}, \texttt{where})$ // Recursive call to this algorithm,
    // returns left and right halves of u_{top}.
  Let L and R be trivial augmented binary trees. Cut both L and R horizontally.
  L`.setBot`$(b[0])$
  L`.setTop`$(t[0])$
  R`.setBot`$(b[1])$
  R`.setTop`$(t[1])$
  return $[L, R]$
else // u is a leaf node.
  `cutVert`$(u, \texttt{where})$
  return $[u_{\text{left}}, u_{\text{right}}]$

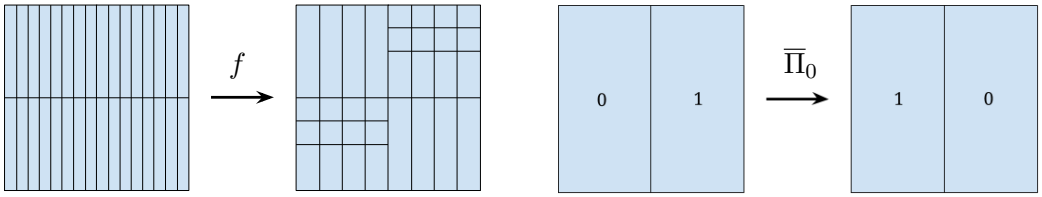


Figure 3.2.1: Some complex element $f \in 2V$ and a generator we would like to multiply it by.

3.2 An Algorithm for Multiplying Elements of $2V$

Given the tuple (S_1, S_2, φ) of a function f with m rectangles, and the tuple (T_1, T_2, ψ) of a function g with n rectangles, Algorithm 1.6.1 has a runtime of $\mathrm{O}(m \cdot n)$, i.e. quadratic if $m = n$. Algorithm 3.2.2 is a recursive multiplication algorithm that makes use of the tree structure. Its runtime is logarithmic in principle, and should be faster than quadratic in the average case.

One consideration that went into the design of this algorithm is that much of the multiplication that needs to take place to implement the Anshel-Anshel-Goldfeld protocol is multiplication by generators, which have very few rectangles. This algorithm is tailored for multiplying complex functions with many rectangles by a 2- to 4-rectangle generator. When faced with two functions as in Figure 3.2.1, rather than spend a lot of time cutting up $\overline{\Pi}_0$, we can make use of the tree representation of $2V$. What we'd like to be able to do in this example is pick up each half of the range of f by its root, and map those according to $\overline{\Pi}_0$, as in Figure 3.2.2.

Given two functions $f, g \in 2V$, with tuples (S_1, S_2, φ) and (T_1, T_2, ψ), respectively, to obtain the result fg, the `multiply()` method itself simply calls a method `intersect()`, and returns a new `V2Function` using the private constructor. The `intersect()` method recursively crawls down both trees S_1 and T_2 simultaneously, ensuring that for each pair u, v of nodes, where $u \in S_1$ and $v \in T_2$, that v is cut in the same way as u, until we reach a leaf in S_1, at which point `intersect()` calls a third method, `assign()`. The `intersect()`

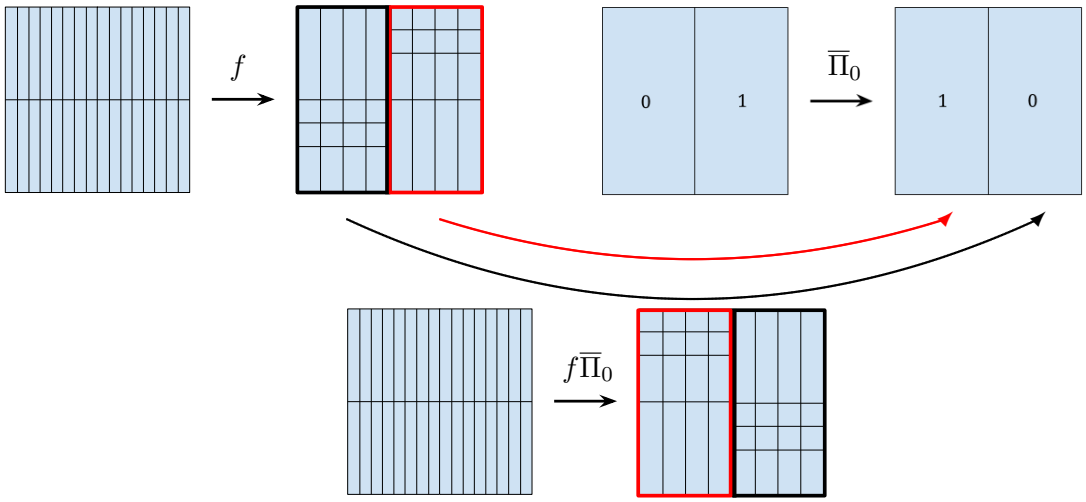


Figure 3.2.2

method is described in greater detail in Algorithm 3.2.2, and a description of `assign()` is given in Algorithm 3.2.1.

Algorithm 3.2.1. *The algorithm of the* `assign()` method used by Algorithm 3.2.2 to calculate the composition of functions fg, where f has the tuple (S_1, S_2, φ) and g has the tuple T_1, T_2, ψ.

Input: v is a node in T_2 and u is a leaf in S_1

Let $p = u_{\text{partner}}$, and let $P = p_{\text{parent}}$.
Let $r = P$.`getRelation`(p).
if r=="*bot*" **then**
  P.`setBot`(v)
else if r=="*top*" **then**
  P.`setTop`(v)
else if r=="*left*" **then**
  P.`setLeft`(v)
else if r=="*right*" **then**
  P.`setRight`(v)

Algorithm 3.2.2. *A recursive algorithm for finding a common refinement of S_1 and T_2, given $f, g \in 2V$, with tuples (S_1, S_2, φ) and (T_1, T_2, ψ), and asked to calculate fg.*

```
1  Let u be the root node of S_1, let v be the root node of T_2.
2  if u is a leaf then
3  |  assign(v, u)
4  else if both u and v are cut horizontally then
5  |  intersect(v_top, u_top)                    // Recursive call to this algorithm.
6  |  intersect(v_bottom, u_bottom)              // Recursive call to this algorithm.
7  else if both u and v are cut vertically then
8  |  intersect(v_left, u_left)                  // Recursive call to this algorithm.
9  |  intersect(v_right, u_right)                // Recursive call to this algorithm.
10 else if u is cut horizontally and v is a leaf then
11 |  cutHorz(v)
12 |  intersect(v_top, u_top)                    // Recursive call to this algorithm.
13 |  intersect(v_bottom, u_bottom)              // Recursive call to this algorithm.
14 else if u is cut vertically and v is a leaf then
15 |  cutVert(v)
16 |  intersect(v_left, u_left)                  // Recursive call to this algorithm.
17 |  intersect(v_right, u_right)                // Recursive call to this algorithm.
18 else if u is cut horizontally and v is cut vertically then
19 |  cutFullHorz(v)
20 |  intersect(v_top, u_top)                    // Recursive call to this algorithm.
21 |  intersect(v_bottom, u_bottom)              // Recursive call to this algorithm.
22 else if u is cut vertically and v is cut horizontally then
23 |  cutFullVert(v)
24 |  intersect(v_left, u_left)                  // Recursive call to this algorithm.
25 |  intersect(v_right, u_right)                // Recursive call to this algorithm.
```

The `assign` method makes leaf nodes of S_2 into subtrees whose leaves are already partnered up with the leaf nodes of T_1, so we don't need a `phi` parameter to assign partners, we can just call the private constructor with T_1 and S_2. Note that unless everything is copied beforehand, this destroys the original `V2Function` objects.

3.3 Resultant Rectangles when Multiplying n Generators

The resultant number of rectangles in a function tuple created by multiplying n generators of $2V$ is presented in Figure 3.3.1

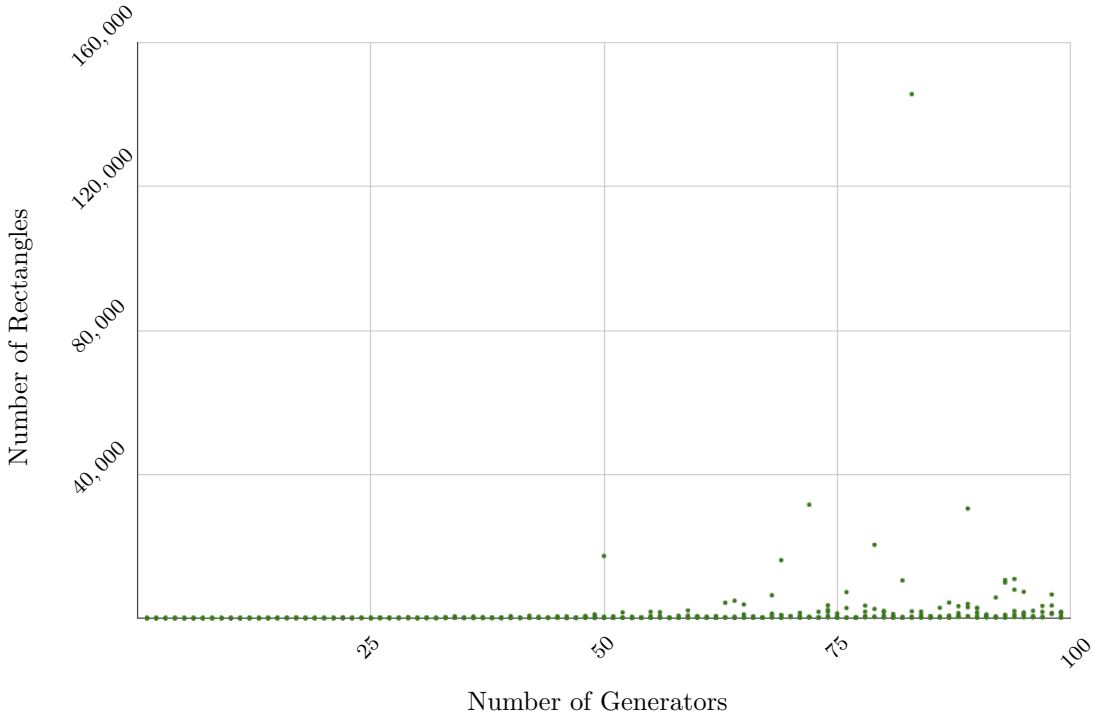


Figure 3.3.1

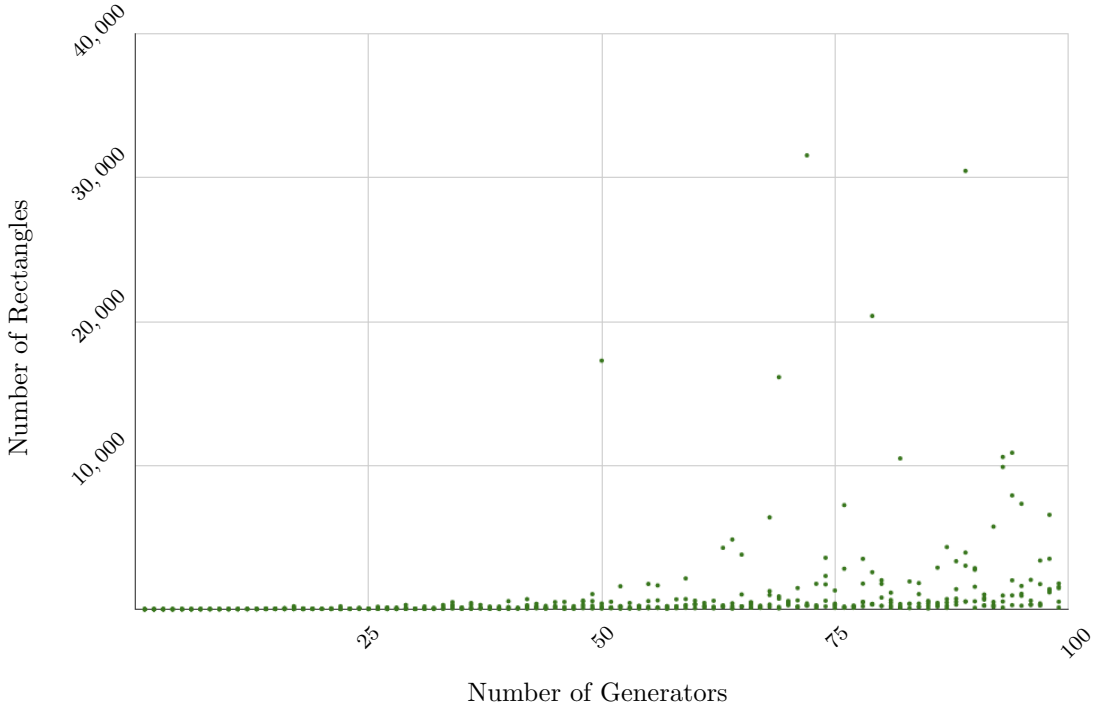


Figure 3.3.2

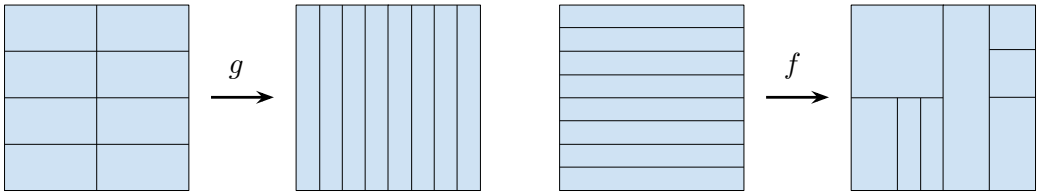


Figure 3.3.3: When composing g, given by the tuple (T_1, T_2, ψ), with f, given by the tuple (S_1, S_2, φ), this is the worst case scenario for the orientations of the cuts in S_1 and T_2.

Purely from a visual assessment, we can see that the average case does not increase especially quickly, though it's hard to tell due to some egregious outliers. Figure 3.3.2 is contains the same data with the most significant offender removed. This allows us to see the curvature of the growth more easily.

In order to analyze what is happening here, consider composing two arbitrary functions $f, g \in 2V$, given by tuples (S_1, S_2, φ) and (T_1, T_2, ψ), where $|S_1| = |S_2| = |T_1| = |T_2| = n$. The worst case scenario looks something like Figure 3.3.3, where the T_2 has only one sort of cut, and S_1 has only the other kind of cut. A common refinement of S_1 and T_2 has n^2 rectangles. Note that this means that if a subquadratic multiplication algorithm for elements of $2V$ exists, it's worst case runtime will still grow quadratically due to this fact.

The average generator of $2V$ has three rectangles. On average, then, each time we multiply a function f whose tuple has k rectangles by a generator, we potentially have up to $3k$ resultant rectangles. So, when we multiply together n generators, if the worst case orientation of S_1 and T_2 occurs every time, we would end up with around 3^n rectangles in our final product. This hypothesis has not been explored extensively, though I believe doing so would reveal that the worst case number of rectangles in a product of generators is exponential in the number of generators.

4

Implementation and Cryptanalysis of Anshel-Anshel-Goldfeld

The first section of this chapter discusses how we implement two distinct users, and the key agreement protocol between them. The second section outlines how we come up with an actual encryption key that is identical between the two users. The third section analyzes the runtime of AAG. Lastly, Section 4.4 discusses our implementation of the LBA and and our results, which show conclusively that AAG is not secure with $2V$ as a platform group.

4.1 Implementing the Anshel-Anshel-Goldfeld Key Agreement Protocol

To implement AAG, we create the `User` class. The fields a `User` needs, as well as the algorithms of the methods the class should have, are for the most part given in Definition 2.2.1. This section, as a result, only focuses on those things that are unique to our implementation.

Recall that AAG requires a set of integers, $\{N_{1,2}, L_{1,2}, L\}$ and a finite presentation of a group G. As such, our `User` constructor takes five parameters: four integers n, L_1, L_2,

and L, and a set of generators. For Alice, we pass N_1 as the n parameter, and for Bob, we pass N_2.

The `createUserSet()` method, which creates $\overline{a}$ for Alice, and $\overline{b}$ for Bob, generates a random number between L_1 and L_2, and then multiplies together that many randomly selected generators to create each element in the users' public sets, and returns a boolean value representing whether all the generators were used in this process. To ensure that all generators appear in a user's public set, the `User` constructor calls `createUserSet()` until it returns `true`. Even with relatively small choices for $N_{1,2}$, $L_{1,2}$, and L we can drive down the number of times the constructor will have to call `createUserSet()`. For example, if we use $N_{1,2} = 5$, $L_1 = 8$, $L_2 = 10$, and $L = 5$, each user's public set is a 5-tuple of words of 8 to 10 generators, meaning that at least 40 generators are used to create a user's public set. The probability that a generator does not get used during this process is at most $(\frac{11}{12})^{40}$ or a little over 3%. With these parameters, users occasionally had to try twice to create a valid public set, but never more than twice during our trials.

The next step in the constructor is to create each user's secret element. As we randomly select elements from a user's public set to create the secret element, we store their indices in a private `ArrayList` of integers called `eltsUsed`, so that we can remember their order later when we want to create K. For example, Alice's secret element $A = a_{s_1} \cdots a_{s_L}$, where each s_i is the ith entry in `eltsUsed` (In our implementation, inverses are given as distinct elements of the user's public set). Therefore, when Alice receives the set $\overline{a}'$, she can recover

$$B^{-1}AB \;=\; B^{-1}a_{s_1} \cdots a_{s_L}B \;=\; B^{-1}a_{s_1}B \cdots B^{-1}a_{s_L}B \;=\; a'_{s_1} \cdots a'_{s_L}.$$

Creating K is a two step process. The first step each user takes is identical. Alice calculates $C_A = B^{-1}AB$, and Bob computes $C_B = A^{-1}BA$. After this, Alice calculates $A^{-1}C_A$, and Bob calculates $C_B^{-1}B$. To accomplish this, the `User` class has two methods

`keyA()` and `keyB()`, each of which call a method `step1()`, and then call `step2A()` or `step2B()`, respectively.

I attempted to implement `UserA` and `UserB` as subclasses of a `User` superclass, but Java's variable protection made this difficult. If `userElt`, the field where we store a `User` object's secret element is declared `private`, then it is not accessible by subclasses, meaning each of `UserA` and `UserB` need their own private variables, which is unnecessarily specific. If class variables are declared `protected` in the User superclass, then they're public thoughout all subclasses, meaning that A is public to Bob, and B is public to Alice, which defeats the purpose of a key agreement protocol. As a result, writing the `User` class in this way, and then using the `KeyAgreement` class to designate each `User` as Alice or Bob, and instructing them to call `keyA()` or `keyB()` accordingly, seems to be the most faithful implementation at the present moment.

4.2 Establishing a Practical Encryption Key

Theorem 1.5.3 stated that a function $f \in 2V$ is not guaranteed to have a unique reduced tuple (S_1, S_2, φ). Since we are not guaranteed a standard form of the element K that Alice and Bob agree upon, they must carry out some other process to create an identical key that both can use for encryption. Since multiplication can still be computationally expensive, even with our revised algorithm, rather than send random group elements to establish a key, as proposed in Section 2.2, we use the orbit of the point $(0, 0)$ in K.

Definition 4.2.1. Given a set X and a function $f : X \to X$, the **orbit** of an input $x \in X$ is given by

$$x, f(x), f(f(x)), f(f(f(x))), \ldots$$

$\triangle$

Since $K_A = K_B = K$, we know that $K_A(x, y) = K_B(x, y)$ for all $(x, y) \in [0, 1] \times [0, 1]$. In other words, the orbit of $(0, 0)$ in both keys will be identical.

In order for a function $f \in 2V$, given by the tuple (S_1, S_2, φ) to map a point (x, y) we first need to know which leaf node $u \in S_1$ corresponds to the interval containing (x, y), so that we can find $\varphi(u) = u_{\text{partner}}$ which is a leaf in S_2. This is accomplished by a method `containsPoint()` in the `AugTree` class, which uses Algorithm 4.2.2.

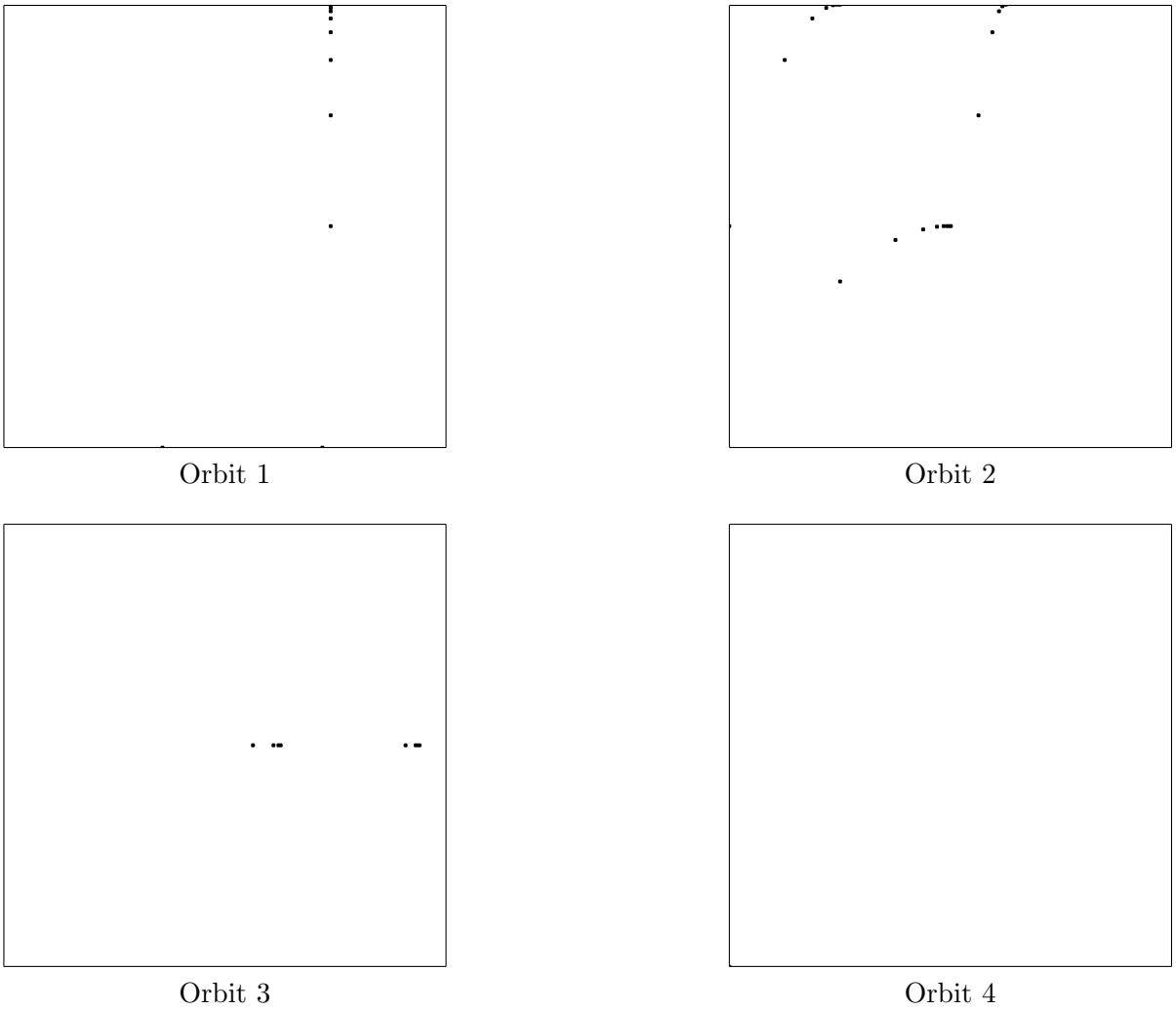
In theory, $x, y \in \{0, 1\}^\infty$. In practice, since it would be impossible to pass an infinite string to a method, x and y are both finite binary strings which are implied to be followed by an infinite string of zeros. Hopefully x and y are longer than the maximum length of the strings describing x- and y-intervals of the leaves of T, because in this case we can simply find the node that corresponds to the prefixes of both x and y, and effectively ignore the fact that x and y are finite. If we reach the end of either x or y as we read it before we reach a leaf node, we still need to get to a leaf node, so we cheat by padding the ends of x and y with zeros as necessary.

Algorithm 4.2.2. *An algorithm for finding which leaf node of an augmented binary tree T contains the point (x, y).*

```
Let u be the root node of T.
if u is cut horizontally then
    if the first digit of y is a 0 then
        return u_bottom.containsPoint(x, y[1 :])   // Recursive call to this algorithm.
    else if the first digit of y is a 1 then
        return u_top.containsPoint(x, y[1 :])      // Recursive call to this algorithm.
    else                                           // y must be empty
        return u_bottom.containsPoint(x, 0)        // Recursive call to this algorithm.
else if u is cut vertically then
    if the first digit of x is a 0 then
        return u_left.containsPoint(x[1 :], y)     // Recursive call to this algorithm.
    else if the first digit of x is a 1 then
        return u_right.containsPoint(x[1 :], y)    // Recursive call to this algorithm.
    else                                           // x must be empty
        return u_left.containsPoint(0, y)          // Recursive call to this algorithm.
else                                               // u is a leaf
    return u
```

Now, we can figure out what interval v corresponds to, and figure out the values of a, A, b, and B such that $f(x, y) = (ax + A, bx + B)$. This can be done algebraically, which requires lots of mathematical computations, or it can be done by means of binary string prefix replacement, thanks to the correspondence established between dyadic intervals in Section 1.5. This is the approach we take.

We provide a `mapPoint()` method in the `V2Function` class, which calls S_1.`containsPoint`(x, y) to get u, then replaces the prefixes given by u with the prefixes given by u_{partner} to determine $f(x, y)$ for a particular function. We also provide methods `bsToDbl()` and `dblToBS()` which allow the user to roughly convert from binary strings to floating point numbers, and back. The method `bsToDbl()` only reads to a certain number of digits, which the user can specify by editing the local method variable `maxStrLen`. Excessively long binary strings give an unnecessary level of specificity to floating point numbers, while significantly slowing computation time.



Orbit 1 Orbit 2

Orbit 3 Orbit 4

Figure 4.2.1: Orbit of the point $(0, 0)$ in the keys produced by our implementation of AAG with the parameters $N_{1,2} = 5$, $L_1 = 8$, $L_2 = 10$, and $L = 5$. Notice that in Orbit 4, the point $(0, 0)$ has period 1.

We calculate the orbit for a certain number of iterations (which can changed by editing the local method variable `keyLen` in the `User` class method `createBinStr()`), and concatenate the binary strings of each of the x- and y-coordinates to create a string called `encryptKey` which is identical for Alice and Bob.

Now, if we want to encrypt a message, we convert all its characters to their binary values, concatenate them, `XOR` the concatenated binary string with `encryptKey`, and send it. The other user then `XOR`s the encrypted message with their identical `encryptKey` to get

Parameters Trial	$5, 5, 8, 10, 5$	$5, 5, 8, 10, 10$	$10, 10, 8, 10, 10$	$10, 10, 5, 8, 20$
1	2.670 s	40.199 s	1.928 s	`OME`
2	1.008 s	`OME`	`OME`	5+ min
3	21.144 s	13.619 s	5+ min	5+ min
4	1.226 s	`OME`	5+ min	`OME`
5	206.049 s	5+ min	`OME`	5+ min
6	0.264 s	7.327 s	8.225 s	5+ min
7	10.873 s	409.652 s	`OME`	5+ min
8	0.642 s	`OME`	31.830 s	
9	0.116 s	`OME`	5+ min	
10	0.160 s	158.296 s	`OME`	

Table 4.3.1: The runtimes for various values of $N_{1,2}$, $L_{1,2}$, and L over ten trials. The label `OME`, for "out of memory error," indicates that the JVM ran out of heap space before establishing a key.

the message in binary back out, and then just converts each byte of the resultant message back to its corresponding character.

This would be a great method for establishing a random encryption key if these functions had chaotic orbits. Unfortunately, the orbits of the functions produced by a `KeyAgreement` object initialized with feasible parameters are remarkably unchaotic, as seen in the examples in Figure 4.2.1. Unchaotic orbits mean that an adversary wishing to intercept encrypted information can potentially guess the location of $f(x, y)$ and hence `encryptKey`.

4.3 AAG Runtime

The different parameters we tried with our implementation, and their runtimes are given in Table 4.3.1. Size, clearly, is an issue. We allocated 1.4GB of heap space for these trials, and still frequently ran out of memory before the users were able to agree upon a key, most likely due to the problem illustrated by Figures 3.3.1 and 3.3.2. One solution might be to

put in a check to catch instances of the worst case multiplication scenario we described (see Figure 3.3.3) in order to avoid them.

Using the parameters $N_{1,2} = 5$, $L_1 = 8$, $L_2 = 10$, and $L = 5$ had an average runtime of 24.415 seconds, which is far too slow for encryption purposes. Running the protocol with a profiler revealed that roughly 65% of calls are to `ArrayList.indexOf()`, so any improvements to our code should also aim to work around using this built-in Java data structure. The fact that the median runtime among these trials is only 1.117 s, however, is promising if we can find some way to avoid the worst case multiplication scenario.

4.4 The Length Based Attack

Our implementation of the LBA follows Algorithm 2.2.2. As a length function l, we use the `numLeaves` field of the `V2Function` class, which gives the number of leaves in S_1 (equal to the number of leaves in S_2) for the function it represents. We always call `reduce()` before checking the number of leaves, in order to get the most faithful representation of how much it has increased or decreased.

As one might guess from Table 4.3.1, we did not get to test the our LBA on especially large elements of $2V$, that is elements with tuples where $|S_1|$ and $|S_2|$ are on the order of $100,000$. Of the successfully completed trials where $L \geq 10$, none were broken by the LBA. In over 50 trials with the parameters $N_{1,2} = 5$, $L_1 = 8$, $L_2 = 10$, and $L = 5$, only one instance was broken by the LBA. The LBA is primarily successful against instances with very large parameters, so this information may not be of much use.

In twenty trials, we picked a random generator, and conjugated it by thirty more randomly selected generators. Over these trials, the length of the conjugated element decreased eight times, and of those eight, six were because we had conjugated by the inverse of the element by which we had just previously conjugated. What we can determine from this is that using the number of leaves in S_1 as our length function l fits the citeria listed

in Definition 2.2.2 that the LBA needs to be successful. Given the chance to test the LBA on instances of AAG with larger parameters, we will most likely see greater success rates.

Appendix: Code

```
public class AugTree{
  AugTree parent; //null if this is root
  AugTree left, right;
  AugTree bottom, top;
  AugTree partner; //partner is the augtree that this one maps. non-null if
                   //this is a leaf in an AugTree used in a fuction.
  Boolean cutHorz, cutVert; //only 0 or 1 of these is true
  int id;
  Random r = new Random();

  public AugTree(){
    this.parent = this.left = this.right = this.bottom
                                  = this.top = this.partner = null;
    this.cutHorz = this.cutVert = false;
    this.id = this.r.nextInt(1000000);
  }

  private AugTree(AugTree p){
    this.parent = p;
    this.left = this.right = this.bottom = this.top = this.partner = null;
    this.cutHorz = this.cutVert = false;
    this.id = this.r.nextInt(1000000);
  }

  public void setPartner(AugTree o){
    this.partner = o;
    o.partner = this;
```

```
  }

  public void setLeft(AugTree t){
    this.left = t;
    if (!this.cutVert){ this.cutHorz = false; this.cutVert = true; }
    t.parent = this;
  }

  public void setRight(AugTree t){
    this.right = t;
    if (!this.cutVert){ this.cutHorz = false; this.cutVert = true; }
    t.parent = this;
  }

  public void setBot(AugTree t){
    this.bottom = t;
    if (!this.cutHorz) { this.cutVert = false; this.cutHorz = true; }
    t.parent = this;
  }

  public void setTop(AugTree t){
    this.top = t;
    if (!this.cutHorz) { this.cutVert = false; this.cutHorz = true; }
    t.parent = this;
  }

  public void cutHorz(){
    this.bottom = new AugTree(this);
    this.top = new AugTree(this);
    if (this.cutVert) {this.cutVert = false; this.left = this.right =null;}
    this.cutHorz = true;
  }

  public void cutVert(){
    this.left = new AugTree(this);
    this.right = new AugTree(this);
    if (this.cutHorz) {this.cutHorz = false; this.bottom = this.top =null;}
    this.cutVert = true;
}

  public boolean isTrivial(){
    return (!this.cutHorz)&&(!this.cutVert);
  }

  public String getRelation(AugTree c){ //takes in a supposed child
```

```
    if (this.left==c){ return "left";}
    else if (this.right==c){ return "right";}
    else if (this.top==c){ return "top";}
    else if (this.bottom==c){ return "bot";}
    else { return "none";}
  }


  public ArrayList<AugTree> looksCutVert(){ //returns array containing left
                                                    //and right halves if so
    ArrayList<AugTree> res = new ArrayList<AugTree>();
    if (this.cutVert){
      res.add(this.left); res.add(this.right);
    }
    else if (this.cutHorz
              &&(!this.bottom.isTrivial())&&(!this.top.isTrivial())){
      ArrayList<AugTree> tops = this.top.looksCutVert();
      ArrayList<AugTree> bots = this.bottom.looksCutVert();
      if ((tops.size()>1)&&(bots.size()>1)){
        AugTree l = new AugTree(); AugTree r = new AugTree();
        l.cutHorz(); r.cutHorz();
        l.setTop(tops.get(0)); l.setBot(bots.get(0));
        r.setTop(tops.get(1)); r.setBot(bots.get(1));
        res.add(l); res.add(r);
      }
    }
    return res;
  }

  public ArrayList<AugTree> looksCutHorz(){ //returns array containing top
                                                  //and bottom halves if so
    ArrayList<AugTree> res = new ArrayList<AugTree>();
    if (this.cutHorz){
      res.add(this.top); res.add(this.bottom);
    }
    else if (this.cutVert
              &&(!this.left.isTrivial())&&(!this.right.isTrivial())){
      ArrayList<AugTree> lefts = this.left.looksCutHorz();
      ArrayList<AugTree> rites = this.right.looksCutHorz();
      if ((lefts.size()>1)&&(rites.size()>1)){
        AugTree t = new AugTree(); AugTree b = new AugTree();
        t.cutVert(); b.cutVert();
        t.setLeft(lefts.get(0)); t.setRight(rites.get(0));
        b.setLeft(lefts.get(1)); b.setRight(rites.get(1));
        res.add(b); res.add(t);
```

```
      }
    }
    return res;
  }

  public void reduce(){
    this.left = this.right = this.bottom = this.top = null;
    this.cutVert = this.cutHorz = false;
  }

  public ArrayList<AugTree> listLeaves(){
    if (this.cutHorz){
      ArrayList<AugTree> l1 = this.bottom.listLeaves();
      ArrayList<AugTree> l2 = this.top.listLeaves();
      l1.addAll(l2);
      return l1;
    }
    else if (this.cutVert){
      ArrayList<AugTree> l1 = this.left.listLeaves();
      ArrayList<AugTree> l2 = this.right.listLeaves();
      l1.addAll(l2);
      return l1;
    }
    else {
      ArrayList<AugTree> l = new ArrayList<AugTree>();
      l.add(this);
      return l;
    }
  }

  public ArrayList<String[]> listLeavesStr(){
    if (this.cutHorz){
      ArrayList<String[]> l1 = this.bottom.listLeavesStr();
      ArrayList<String[]> l2 = this.top.listLeavesStr();
      l1.addAll(l2);
      return l1;
    }
    else if (this.cutVert){
      ArrayList<String[]> l1 = this.left.listLeavesStr();
      ArrayList<String[]> l2 = this.right.listLeavesStr();
      l1.addAll(l2);
      return l1;
    }
    else {
      String s = this.toString();
```

```
    String ss = s.substring(1,s.length()-1);
    String[] xy = ss.split(", ");
    ArrayList<String[]> l = new ArrayList<String[]>(); l.add(xy);
    return l;
  }
}

public int size(){
  return this.listLeaves().size();
}

public ArrayList<String> toStringHelper(ArrayList<String> bS){
  String xBS = bS.get(0);
  String yBS = bS.get(1);
  String newX, newY;
  ArrayList<String> newBS = new ArrayList<String>();
  AugTree p = this.parent;
  if (p != null) {
    if (p.cutHorz){
      if (this == p.bottom) { newY = yBS.replaceFirst("","0"); }
      else { newY = yBS.replaceFirst("","1"); }
      newX = xBS;
    }
    else {
      if (this == p.left) { newX = xBS.replaceFirst("","0"); }
      else { newX = xBS.replaceFirst("","1"); }
      newY = yBS;
    }
    bS.set(0,newX); bS.set(1,newY);
    newBS = p.toStringHelper(bS);
    return newBS;
  }
  else { return bS;}
}

public String toString(){
  ArrayList<String> s = new ArrayList<String>(); s.add(""); s.add("");
  ArrayList<String> bS = this.toStringHelper(s);
  String res = "[";
  res+=bS.get(0);
  res+=", ";
  res+=bS.get(1);
  res+="]";
  return res;
}
```

```
public void showData(){
  System.out.print("(AugTree: "+this.id+" Parent: ");
  if (this.parent==null){ System.out.print("null"); }
  else {System.out.print(this.parent.id); }
  System.out.print(" cutHorz: "+this.cutHorz+" cutVert: "+this.cutVert);
  System.out.print(" Partner: ");
  if (this.partner==null){ System.out.print("null"); }
  else {System.out.print(this.partner.id); }
  System.out.print(" Left: ");
  if (this.left==null){ System.out.print("null"); }
  else {this.left.showData(); }
  System.out.print(" Right: ");
  if (this.right==null){ System.out.print("null"); }
  else {this.right.showData(); }
  System.out.print(" Top: ");
  if (this.top==null){ System.out.print("null"); }
  else {this.top.showData(); }
  System.out.print(" Bottom: ");
  if (this.bottom==null){ System.out.print("null"); }
  else {this.bottom.showData(); }
  System.out.print(")");
}

public boolean equals(AugTree o){
  if (this.cutHorz&&o.cutHorz){
    return (this.top.equals(o.top)&&this.bottom.equals(o.bottom));
  }
  else if (this.cutVert&&o.cutVert){
    return (this.left.equals(o.left)&&this.right.equals(o.right));
  }
  else if (this.cutVert&&o.cutHorz){
    ArrayList<AugTree> ohalves = o.looksCutVert();
    if (ohalves.size()<2){
      return false;
    }
    else{
      return (this.left.equals(ohalves.get(0))
                          &&this.right.equals(ohalves.get(1)));
    }
  }
  else if (this.cutHorz&&o.cutVert){
    ArrayList<AugTree> ohalves = o.looksCutHorz();
    if (ohalves.size()<2){
      return false;
```

```
      }
      else{
        return (this.bottom.equals(ohalves.get(0))
                              &&this.top.equals(ohalves.get(1)));
      }
    }
    else if (this.isTrivial()&&o.isTrivial()){
      return true;
    }
    else {
      return false;
    }
  }

  public AugTree copy(){
    AugTree r = new AugTree();
    if (this.cutHorz){
      r.cutHorz();
      r.setTop(this.top.copy());
      r.setBot(this.bottom.copy());
    }
    else if (this.cutVert){
      r.cutVert();
      r.setLeft(this.left.copy());
      r.setRight(this.right.copy());
    }
    return r;
  }

  public AugTree containsPoint(String xBS, String yBS){//returns which one
                    //of its leaf nodes contains point (xBS, yBS)
    if (cutHorz){
      if (yBS.startsWith("0")){
        return bottom.containsPoint(xBS,yBS.substring(1));
      }
      else if (yBS.startsWith("1")){
        return top.containsPoint(xBS,yBS.substring(1));
      }
      else {
        return bottom.containsPoint(xBS,"0");
      }
    }
    else if (cutVert){
      if (xBS.startsWith("0")){
        return left.containsPoint(xBS.substring(1),yBS);
```

```
      }
      else if (xBS.startsWith("1")){
        return right.containsPoint(xBS.substring(1),yBS);
      }
      else{
        return left.containsPoint("0",yBS);
      }
    }
    return this;
  }
}


public class V2Function{
  AugTree domain, range;
  ArrayList<AugTree> domLeaves, ranLeaves;
  int numLeaves;

  public V2Function(){
    this.domain = new AugTree();
    this.range = new AugTree();
    this.domLeaves = domain.listLeaves();
    this.ranLeaves = range.listLeaves();
    this.numLeaves = 1;
    this.domain.setPartner(this.range);
  }

  public V2Function(AugTree d, AugTree r, ArrayList<Integer> phi){
    //phi only used to initalize partner relationships
    if (d.size() == r.size()){
      this.numLeaves = d.size();
      this.domain = d;
      this.range = r;
      this.domLeaves = d.listLeaves();
      this.ranLeaves = r.listLeaves();
      for (int i = 0; i<numLeaves; i++){
        this.domLeaves.get(i).setPartner(
                    this.ranLeaves.get(phi.indexOf(i)));
      }
    }
  }

  private V2Function(AugTree d, AugTree r){
  //assumes partner relationships between d & r already initialized
  //only called by multiply().
```

```
    if (d.size() == r.size()){
      this.numLeaves = d.size();
      this.domain = d;
      this.range = r;
      this.domLeaves = d.listLeaves();
      this.ranLeaves = r.listLeaves();
    }
  }

  public boolean isIdentity(){
    for (int i=0; i<numLeaves; i++){
      if (!this.ranLeaves.get(i).toString().equals(
                  this.ranLeaves.get(i).partner.toString())) {
        return false;
      }
    }
    //might as well simplify
    this.domain.reduce();
    this.domLeaves.clear(); this.domLeaves.add(this.domain);
    this.range.reduce();
    this.ranLeaves.clear(); this.ranLeaves.add(this.range);
    this.numLeaves = 1;
    return true;
  }

  public V2Function copy(){
    ArrayList<Integer> phi = this.phi();
    return new V2Function(this.domain.copy(),this.range.copy(),phi);
  }

  public V2Function inverse(){
    ArrayList<Integer> phi = new ArrayList<Integer>();
    for (int i =0; i<numLeaves; i++){
      phi.add(this.ranLeaves.indexOf(this.domLeaves.get(i).partner));
    }
    return new V2Function(range.copy(),domain.copy(),phi);
  }

  public String toString(){
    String str = "[ ";
    for (int i =0; i<this.numLeaves;i++){
      str = str.concat("(");
      str = str.concat(this.domLeaves.get(i).toString());
      str = str.concat(":");
      str = str.concat(this.domLeaves.get(i).partner.toString());
```

```
      str = str.concat(") ");
    }
    str = str.concat("]");
    return str;
  }

  public void showData(){
    System.out.println("domain:");
    this.domain.showData();
    System.out.println();
    System.out.println("range:");
    this.range.showData();
    System.out.println();
  }

  public void reduce(){ //iterative process over leaf nodes,
  //leaf nodes after reducing, leaf nodes after that reduction,
  //etc till no more reductions to make
    boolean change = true;
    while (change){
      change = this.reduceHelper();
    }
  }

  public boolean reduceHelper(){
    boolean change = false;
    for (int i=0; i<this.numLeaves; i++){
      AugTree domLeaf = this.domLeaves.get(i); //get domain leaf we're on
      AugTree ranLeaf = domLeaf.partner; //and the one it maps to
      AugTree domSib, ranSib;
      if (domLeaf.parent==null){ return false; } //if we're all the way at
      //the root node, we cant reduce anymore, so return change=false
        String rel = domLeaf.parent.getRelation(domLeaf);
            //figure out what kind of leaf it is (left,right,bottom,top)
        String rel2 = ranLeaf.parent.getRelation(ranLeaf);
            //and what kind of leaf it maps to
        if (rel.equals(rel2)){ //if those aren't the same, then we already
            //know we can't reduce, so don't bother
        //if they are the same, get their "siblings"
        if (rel.equals("bot")){
          domSib = domLeaf.parent.top;
          ranSib = ranLeaf.parent.top;
        }
        else if (rel.equals("top")){
          domSib = domLeaf.parent.bottom;
```

```
        ranSib = ranLeaf.parent.bottom;
      }
      else if (rel.equals("left")){
        domSib = domLeaf.parent.right;
        ranSib = ranLeaf.parent.right;
      }
      else if (rel.equals("right")){
        domSib = domLeaf.parent.left;
        ranSib = ranLeaf.parent.left;
      }
      else { domSib = ranSib = null; } //just in case something weird
          //happens, we need to do this so the compiler doesn't yell
      if (domSib.partner==ranSib){ //if their siblings map to eachother,
                                   //then we can reduce
      //replace ranLeaf with its parent, and remove its sibling from
      //the list of leaf nodes
      //and do the same with the leaves they map to
        int d2 = domLeaves.indexOf(domSib);
        int r1 = ranLeaves.indexOf(ranLeaf);
        int r2 = ranLeaves.indexOf(ranSib);
        this.domLeaves.set(i,domLeaf.parent);
        this.domLeaves.remove(d2);
        this.ranLeaves.set(r1,ranLeaf.parent);
        this.ranLeaves.remove(r2);
        ranLeaf.parent.setPartner(domLeaf.parent);
        //lastly set their parents to map to each other
        ranLeaf.parent.reduce();
        domLeaf.parent.reduce();
        change = true;
        numLeaves--;
      }
    }
  }
  return change;
}

public void cutHorz(AugTree leaf){
  this.cutHorz(leaf,"r");
}

public void cutVert(AugTree leaf){
  this.cutVert(leaf,"r");
}

public void cutHorz(AugTree leaf, String where){
```

```
      //where specifies if leaf to cut is in domain or range
    if (where=="r"){
      if (this.ranLeaves.contains(leaf)){
        AugTree domLeaf = leaf.partner;
        domLeaf.cutHorz();
        leaf.cutHorz();
        leaf.bottom.setPartner(domLeaf.bottom);
        leaf.top.setPartner(domLeaf.top);
        int rI = ranLeaves.indexOf(leaf);
        int dI = domLeaves.indexOf(domLeaf);
        domLeaves.set(dI,domLeaf.bottom);
        domLeaves.add(dI+1,domLeaf.top);
        ranLeaves.set(rI,leaf.bottom);
        ranLeaves.add(rI+1,leaf.top);
        numLeaves++;
      }
    }
    else if (where=="d"){
      if (domLeaves.contains(leaf)){
        AugTree ranLeaf = leaf.partner;
        leaf.cutHorz();
        ranLeaf.cutHorz();
        leaf.bottom.setPartner(ranLeaf.bottom);
        leaf.top.setPartner(ranLeaf.top);
        int rI = ranLeaves.indexOf(ranLeaf);
        int dI = domLeaves.indexOf(leaf);
        domLeaves.set(dI,leaf.bottom);
        domLeaves.add(dI+1,leaf.top);
        ranLeaves.set(rI,ranLeaf.bottom);
        ranLeaves.add(rI+1,ranLeaf.top);
        numLeaves++;
      }
    }
  }

  public void cutVert(AugTree leaf, String where){
    if (where=="r"){
      if (ranLeaves.contains(leaf)){
        AugTree domLeaf = leaf.partner;
        domLeaf.cutVert();
        leaf.cutVert();
        leaf.left.setPartner(domLeaf.left);
        leaf.right.setPartner(domLeaf.right);
        int rI = ranLeaves.indexOf(leaf);
        int dI = domLeaves.indexOf(domLeaf);
```

```
        domLeaves.set(dI,domLeaf.left);
        domLeaves.add(dI+1,domLeaf.right);
        ranLeaves.set(rI,leaf.left);
        ranLeaves.add(rI+1,leaf.right);
        numLeaves++;
      }
    }
    else if (where=="d"){
      if (domLeaves.contains(leaf)){
        AugTree ranLeaf = leaf.partner;
        leaf.cutVert();
        ranLeaf.cutVert();
        leaf.left.setPartner(ranLeaf.left);
        leaf.right.setPartner(ranLeaf.right);
        int rI = ranLeaves.indexOf(ranLeaf);
        int dI = domLeaves.indexOf(leaf);
        domLeaves.set(dI,leaf.left);
        domLeaves.add(dI+1,leaf.right);
        ranLeaves.set(rI,ranLeaf.left);
        ranLeaves.add(rI+1,ranLeaf.right);
        numLeaves++;
      }
    }
  }

  public ArrayList<AugTree> cutFullHorz(AugTree tree){
    return this.cutFullHorz(tree,"r");
  }

  public ArrayList<AugTree> cutFullVert(AugTree tree){
    return this.cutFullVert(tree,"r");
  }

  public ArrayList<AugTree> cutFullHorz(AugTree tree, String where){
    ArrayList<AugTree> res = new ArrayList<AugTree>(2);
    if (tree.cutHorz){
      res.add(tree.bottom); res.add(tree.top);
    }
    else if (tree.cutVert){
      AugTree rBot = new AugTree();
      AugTree rTop = new AugTree();
      rBot.cutVert();
      rTop.cutVert();
      ArrayList<AugTree> lefts = cutFullHorz(tree.left,where);
      ArrayList<AugTree> rights = cutFullHorz(tree.right,where);
```

```
      rBot.setLeft(lefts.get(0));
      rBot.setRight(rights.get(0));
      rTop.setLeft(lefts.get(1));
      rTop.setRight(rights.get(1));
      tree.setBot(rBot); tree.setTop(rTop);
      res.add(rBot); res.add(rTop);
    }
    else{
      this.cutHorz(tree,where);
      res.add(tree.bottom); res.add(tree.top);
    }
    return res;
  }

  public ArrayList<AugTree> cutFullVert(AugTree tree, String where){
    ArrayList<AugTree> res = new ArrayList<AugTree>(2);
    if (tree.cutVert){
      res.add(tree.left); res.add(tree.right);
    }
    else if (tree.cutHorz){
      AugTree rLeft = new AugTree();
      AugTree rRight = new AugTree();
      rLeft.cutHorz();
      rRight.cutHorz();
      ArrayList<AugTree> tops = cutFullVert(tree.top,where);
      ArrayList<AugTree> bots = cutFullVert(tree.bottom,where);
      rLeft.setTop(tops.get(0));
      rLeft.setBot(bots.get(0));
      rRight.setTop(tops.get(1));
      rRight.setBot(bots.get(1));
      tree.setLeft(rLeft); tree.setRight(rRight);
      res.add(rLeft); res.add(rRight);
    }
    else{
      this.cutVert(tree,where);
      res.add(tree.left); res.add(tree.right);
    }
    return res;
  }

  public void intersect(AugTree ranTree, AugTree odomTree){
    boolean a = ranTree.cutHorz;
    boolean b = odomTree.cutHorz;
    boolean c = ranTree.cutVert;
    boolean d = odomTree.cutVert;
```

```
    if (!(a||b||c||d)){
      this.assign(ranTree,odomTree);
    }
    else if (a&&!(b||c||d)){
      this.assign(ranTree,odomTree);
    }
    else if (c&&!(a||b||d)){
      this.assign(ranTree,odomTree);
    }
    else if (b&&!(a||c||d)){
      this.cutHorz(ranTree);
      this.intersect(ranTree.top,odomTree.top);
      this.intersect(ranTree.bottom,odomTree.bottom);
    }
    else if (d&&!(a||b||c)){
      this.cutVert(ranTree);
      this.intersect(ranTree.left,odomTree.left);
      this.intersect(ranTree.right,odomTree.right);
    }
    else if (a&&b&&!(c||d)){
      this.intersect(ranTree.top,odomTree.top);
      this.intersect(ranTree.bottom,odomTree.bottom);
    }
    else if (c&&d&&!(a||b)){
      this.intersect(ranTree.left,odomTree.left);
      this.intersect(ranTree.right,odomTree.right);
    }
    else if (a&&d&&!(b||c)){
      this.cutFullVert(ranTree);
      this.intersect(ranTree.left,odomTree.left);
      this.intersect(ranTree.right,odomTree.right);
    }
    else if (b&&c&&!(a||d)){
      this.cutFullHorz(ranTree);
      this.intersect(ranTree.top,odomTree.top);
      this.intersect(ranTree.bottom,odomTree.bottom);
    }
  }

  public void assign(AugTree ranTree, AugTree odomTree){
    AugTree p = odomTree.partner;
    AugTree pparent = p.parent;
    String rel = pparent.getRelation(p);
    if (rel=="bot"){
```

```
      pparent.setBot(ranTree);
    }
    else if (rel=="top"){
      pparent.setTop(ranTree);
    }
    else if (rel=="left"){
      pparent.setLeft(ranTree);
    }
    else if (rel=="right"){
      pparent.setRight(ranTree);
    }
    else {
      System.out.println("p is not a child of its parent!");
      System.out.println("\tp: "); p.showData();
      System.out.println("\n\tpparent: "); pparent.showData();
      System.out.println("");
    }
  }

  public V2Function multiply(V2Function o){
    intersect(this.range,o.domain);
    V2Function res = new V2Function(this.domain,o.range);
    return res;
  }

  public Boolean equals(V2Function o){
    return this.copy().multiply(o.inverse()).isIdentity();
  }

  public ArrayList<Integer> phi(){
    ArrayList<Integer> phi = new ArrayList<Integer>();
    for (int i=0; i<numLeaves; i++){
      int n = domLeaves.indexOf(ranLeaves.get(i).partner);
      phi.add(n);
    }
    return phi;
  }

  public String[] mapPoint(String xBS, String yBS){
    String dX, dY, rX, rY;
    AugTree d = domain.containsPoint(xBS,yBS);
    AugTree r = d.partner;
    String sD = d.toString();
    String ssD = sD.substring(1,sD.length()-1);
    if (ssD.startsWith(",")){
```

```
      dX = "";
      String[] xyD = ssD.split(", ");
      if (xyD.length<1){
        dY = "";
      }
      else {
        dY = xyD[0];
      }
    }
    else {
      String[] xyD = ssD.split(", ");
      dX = xyD[0];
      if (xyD.length<2){
        dY = "";
      }
      else {
        dY = xyD[1];
      }
    }
    String sR = r.toString();
    String ssR = sR.substring(1,sR.length()-1);
    if (ssR.startsWith(",")){
      rX = "";
      String[] xyR = ssR.split(", ");
      if (xyR.length<1){
        rY = "";
      }
      else {
        rY = xyR[0];
      }
    }
    else {
      String[] xyR = ssR.split(", ");
      rX = xyR[0];
      if (xyR.length<2){
        rY = "";
      }
      else {
        rY = xyR[1];
      }
    }
    int diffX = dX.length()-xBS.length();
    if (diffX>0){
      for (int i=0; i<diffX; i++){
        xBS = xBS.concat("0");
```

```
      }
    }
    int diffY = dY.length()-yBS.length();
    if (diffY>0){
      for (int i=0; i<diffY; i++){
        yBS = yBS.concat("0");
      }
    }
    String x = xBS.substring(dX.length());
    String y = yBS.substring(dY.length());
    return new String[]{rX.concat(x),rY.concat(y)};
  }

  public double bsToDbl(String s){
    double start = 0.0;
    double end = 1.0;
    int maxStrLen = 20;
    int len = Math.max(maxStrLen,s.length());
    for (int i=0; i<s.length(); i++){
      if (s.charAt(i)=='0'){
        end -= (end-start)/2.0;
      }
      else if (s.charAt(i)=='1'){
        start += (end-start)/2.0;
      }
    }
    return start;
  }

  public String dblToBS(double d){
    String s = String.valueOf(d);
    s = s.replaceFirst("0.","");
    String res = "";
    double start = 0.0;
    double end = 1.0;
    for (int i=0; i<s.length(); i++){
      if (d<(start+((end-start)/2.0))){
        res = res.concat("0");
        end -= (end-start)/2.0;
      }
      else if (d>=(start+((end-start)/2.0))){
        res = res.concat("1");
        start += (end-start)/2.0;
      }
    }
```

```
    return res;
  }

  public ArrayList<String[]> orbitStr(int n){//n = # times to apply function
    ArrayList<String[]> res = new ArrayList<String[]>();
    String xBS = "0";
    String yBS = "0";
    for (int i=0; i<n; i++){
      String[] xy = mapPoint(xBS,yBS);
      res.add(xy);
      xBS = xy[0];
      yBS = xy[1];
    }
    return res;
  }

  public ArrayList<double[]> orbitDbl(int n){//n = # times to apply function
    ArrayList<double[]> res = new ArrayList<double[]>();
    String xBS = "0";
    String yBS = "0";
    for (int i=0; i<n; i++){
      String[] xy = mapPoint(xBS,yBS);
      double[] xyDbl = new double[]{ bsToDbl(xy[0]), bsToDbl(xy[1]) };
      res.add(xyDbl);
      xBS = xy[0];
      yBS = xy[1];
    }
    return res;
  }
}

public class User{
  private V2Function userElt; //word of l elts from userSet
  private V2Function key;
  private String encryptKey;
  Random r = new Random();
  ArrayList<V2Function> oSet, oConjSet, userSet, gens, conjSet;
  private ArrayList<Integer> eltsUsed;
  boolean sent;

  public User(int n, int l1, int l2,
                  int l, ArrayList<V2Function> generators){
    //n is size of user set
    //each elt of userset is a word b/t length l1 and l2
    //user elt is word of elts from user  set, of length 0<length<=l
```

```
    this.gens = generators;
    //oSet = null; //other user's set
    //initialized upon receival from other user
    //conjSet = null; //initialized first time send() is called.
    //oConjSet = null; //other users element conjugated with this.userSet
    //initializedupon receival from other user.
    this.sent = false; //set to true first time send() is called.
    this.userSet = new ArrayList<V2Function>();
    //n-tuple of words of generators
    this.eltsUsed = new ArrayList<Integer>(); //keeps track of elts from
            //userSet in order they were used to create userElt
    boolean usedAll = false; //keeps track of whether all generators
            //have been used in userSet
    while (!usedAll) {
      usedAll = this.createUserSet(n,l1,l2);
    }
    this.createUserElt(n,l);
  }

  private boolean createUserSet(int n, int l1, int l2){
    ArrayList<V2Function> gensLeft = new ArrayList<V2Function>();
    gensLeft.addAll(this.gens);
    for (int i = 0; i<n; i++){
      V2Function thisGen = gens.get(r.nextInt(gens.size()));
      if (gensLeft.size()>0){ if (gensLeft.contains(thisGen)) {
        gensLeft.remove(thisGen);}
      }
      V2Function thisElt = thisGen.copy();
      int thisLength = l1+r.nextInt(l2 - l1);
      for (int j = 1; j<thisLength; j++){
        int x = r.nextInt(this.gens.size());
        thisGen = this.gens.get(x);
        if (gensLeft.size()>0){
          if (gensLeft.contains(thisGen)) {
            gensLeft.remove(thisGen);
          }
        }
        thisElt = thisElt.multiply(thisGen.copy());
        thisElt.reduce();
      }
      this.userSet.add(thisElt);
      if(!(thisElt.inverse().equals(thisElt))) {
       this.userSet.add(thisElt.inverse());
       //add its inverse if its not its own inverse
      }
```

```
    }
  }

  private void createUserElt(int n, int l){
    int x = r.nextInt(n);
    this.userElt = userSet.get(x).copy();
    this.eltsUsed.add(x);
    for (int i=1; i<l; i++){
      x = r.nextInt(n);
      this.userElt = userElt.multiply(userSet.get(x).copy());
      this.eltsUsed.add(x);
      this.userElt.reduce();
    }
  }

  public ArrayList<V2Function> sendSet(){
    return this.userSet;
  }

  public ArrayList<V2Function> sendConjSet(){
    if (!this.sent){
      ArrayList<V2Function> conj = new ArrayList<V2Function>();
      for (int i=0; i<this.oSet.size(); i++){
        V2Function thisElt = ((this.userElt.inverse()).multiply(
                  oSet.get(i).copy())).multiply(this.userElt.copy());
        conj.add(thisElt);
      }
      this.sent = true;
      this.conjSet = conj;
      return conj;
    }
    else { return this.conjSet;}
  }

  public void receiveOSet(ArrayList<V2Function> o){
    this.oSet = o;
  }

  public void receiveConjSet(ArrayList<V2Function> o){
    this.oConjSet = o;
  }

  public V2Function step1(){
    V2Function res = new V2Function();
    for (int i=0; i<this.eltsUsed.size(); i++){
```

```
      V2Function oElt = this.oConjSet.get(this.eltsUsed.get(i)).copy();
      res = res.multiply(oElt);
    }
    return res;
  }

  public V2Function step2A(V2Function conj){
    return this.userElt.inverse().multiply(conj.copy());
  }

  public void keyA(){
    V2Function conj = this.step1();
    this.key = this.step2A(conj);
  }

  public V2Function step2B(V2Function conj){
    return conj.inverse().multiply(this.userElt.copy());
  }

  public void keyB(){
    V2Function conj = this.step1();
    this.key = this.step2B(conj);
  }

  public void createBinStr(){
    int keyLen = 50;
    String res = "";
    ArrayList<String[]> orbit = key.orbitStr(keyLen);
    for (int i=0;i<keyLen; i++){
      String[] pt = orbit.get(i);
      String ptX = pt[0];
      String ptY = pt[1];
      String x,y;
      int xIdx = ptX.lastIndexOf("1");
      int yIdx = ptY.lastIndexOf("1");
      if (xIdx>0)  x = ptX.substring(0,xIdx);
      else x = "";
      if (yIdx>0) y = ptY.substring(0,yIdx);
      else y = "";
      res = res.concat(x);
      res = res.concat(y);
    }
    encryptKey = res;
  }
```

```
public String encryptMsg(String msg){
  /*convert message to binary string*/
  String binMsg = "";
  byte[] x = msg.getBytes();
  for(int i=0; i<x.length;i++){
    binMsg+=(Integer.toBinaryString(0x100 + x[i]).substring(1));
  }

  /*make sure the lengths line up*/
  String thisKey = this.encryptKey;
  while (binMsg.length()>this.encryptKey.length()){
    thisKey+=this.encryptKey;
  }
  thisKey = thisKey.substring(0,binMsg.length());

  /*XOR binMsg with thisKey*/
  String msgNcrptd = "";
  for (int i=0;i<binMsg.length();i++){
    char b = binMsg.charAt(i);
    char k = thisKey.charAt(i);
    if (b==k) msgNcrptd+='0';
    else msgNcrptd+='1';
  }
  return msgNcrptd;
}

public String decryptMsg(String msg){
  /*make sure the lengths line up*/
  String thisKey = this.encryptKey;
  while (msg.length()>this.encryptKey.length()){
    thisKey+=this.encryptKey;
  }
  thisKey = thisKey.substring(0,msg.length());

  /*XOR binMsg with thisKey*/
  String msgDcrptd = "";
  for (int i=0;i<msg.length();i++){
    char m = msg.charAt(i);
    char k = thisKey.charAt(i);
    if (m==k) msgDcrptd+='0';
    else msgDcrptd+='1';
  }

  /*convert binary string to chars*/
  String solvedMsg = "";
```

```
    for (int i=0;i<msgDcrptd.length();i+=8){
      String s = msgDcrptd.substring(i,i+8);
      char c = (char)Integer.parseInt(s, 2);
      solvedMsg = solvedMsg+c;
    }
    return solvedMsg;
  }
}

public class KeyAgreement{
  User alice, bob;
  ArrayList<V2Function> gens;
  int n1, n2, l1, l2, l;

  public KeyAgreement(int n1, int n2, int l1, int l2, int l){

  AugTree domA0 = new AugTree();
  AugTree ranA0 = new AugTree();
  domA0.cutVert(); domA0.left.cutVert();
  ranA0.cutVert(); ranA0.right.cutVert();
  ArrayList<Integer> indA0 = new ArrayList<Integer>();
  indA0.add(0); indA0.add(1); indA0.add(2);
  V2Function A0 = new V2Function(domA0, ranA0, indA0);

  AugTree domA1 = new AugTree();
  AugTree ranA1 = new AugTree();
  domA1.cutVert(); domA1.right.cutVert(); domA1.right.left.cutVert();
  ranA1.cutVert(); ranA1.right.cutVert(); ranA1.right.right.cutVert();
  ArrayList<Integer> indA1 = new ArrayList<Integer>();
  indA1.add(0); indA1.add(1); indA1.add(2); indA1.add(3);
  V2Function A1 = new V2Function(domA1, ranA1, indA1);

  AugTree domB0 = new AugTree();
  AugTree ranB0 = new AugTree();
  domB0.cutHorz();
  ranB0.cutVert();
  ArrayList<Integer> indB0 = new ArrayList<Integer>();
  indB0.add(0); indB0.add(1);
  V2Function B0 = new V2Function(domB0, ranB0, indB0);

  AugTree domB1 = new AugTree();
  AugTree ranB1 = new AugTree();
  domB1.cutVert(); domB1.right.cutHorz();
  ranB1.cutVert(); ranB1.right.cutVert();
  ArrayList<Integer> indB1 = new ArrayList<Integer>();
```

```
indB1.add(0); indB1.add(1); indB1.add(2);
V2Function B1 = new V2Function(domB1, ranB1, indB1);

AugTree domPi_0 = new AugTree();
AugTree ranPi_0 = new AugTree();
domPi_0.cutVert();
ranPi_0.cutVert();
ArrayList<Integer> indPi_0 = new ArrayList<Integer>();
indPi_0.add(1); indPi_0.add(0);
V2Function Pi_0 = new V2Function(domPi_0, ranPi_0, indPi_0);

AugTree domPi_1 = new AugTree();
AugTree ranPi_1 = new AugTree();
domPi_1.cutVert(); domPi_1.right.cutVert();
ranPi_1.cutVert(); ranPi_1.right.cutVert();
ArrayList<Integer> indPi_1 = new ArrayList<Integer>();
indPi_1.add(0); indPi_1.add(2); indPi_1.add(1);
V2Function Pi_1 = new V2Function(domPi_1, ranPi_1, indPi_1);

AugTree domPi0 = new AugTree();
AugTree ranPi0 = new AugTree();
domPi0.cutVert(); domPi0.right.cutVert();
ranPi0.cutVert(); ranPi0.right.cutVert();
ArrayList<Integer> indPi0 = new ArrayList<Integer>();
indPi0.add(1); indPi0.add(0); indPi0.add(2);
V2Function Pi0 = new V2Function(domPi0, ranPi0, indPi0);

AugTree domPi1 = new AugTree();
AugTree ranPi1 = new AugTree();
domPi1.cutVert(); domPi1.right.cutVert(); domPi1.right.right.cutVert();
ranPi1.cutVert(); ranPi1.right.cutVert(); ranPi1.right.right.cutVert();
ArrayList<Integer> indPi1 = new ArrayList<Integer>();
indPi1.add(0); indPi1.add(2); indPi1.add(1); indPi1.add(3);
V2Function Pi1 = new V2Function(domPi1, ranPi1, indPi1);

gens =  new ArrayList<V2Function>();
gens.add(A0); gens.add(A1); gens.add(B0); gens.add(B1); gens.add(Pi_0);
gens.add(Pi_1); gens.add(Pi0); gens.add(Pi1);
gens.add(A0.inverse()); gens.add(A1.inverse()); gens.add(B0.inverse());
gens.add(B1.inverse());

alice = new User(n1, l1, l2, l, gens);
bob = new User(n2, l1, l2, l, gens);
}
```

```
public void createKey(){
  alice.receiveOSet(bob.sendSet());
  bob.receiveOSet(alice.sendSet());

  alice.receiveConjSet(bob.sendConjSet());
  bob.receiveConjSet(alice.sendConjSet());

  alice.keyA();
  bob.keyB();

  alice.createBinStr();
  bob.createBinStr();
}

public ArrayList<V2Function> aSet(){
  return alice.sendSet();
}

public ArrayList<V2Function> bSet(){
  return bob.sendSet();
}

public ArrayList<V2Function> aConjSet(){
  return alice.sendConjSet();
}

public ArrayList<V2Function> bConjSet(){
  return bob.sendConjSet();
}

public String aEncryptMsg(String msg){
  return alice.encryptMsg(msg);
}

public String bEncryptMsg(String msg){
  return bob.encryptMsg(msg);
}

public String aDecryptMsg(String msg){
  return alice.decryptMsg(msg);
}

public String bDecryptMsg(String msg){
  return bob.decryptMsg(msg);
}
```

```
}

public class LBA{

  public KeyAgreement k;
  public ArrayList<V2Function> aSet, bSet, bConjSet;

  public LBA(KeyAgreement keyAg){
    k = keyAg;
    aSet = k.aSet();
    bSet = k.bSet();
    bConjSet = k.bConjSet(); // = {b^(-1)(a_i)b} for all a_i in aSet.
  }

  public V2Function attack(){
    ArrayList<V2Function> alpha = bConjSet;
    V2Function x = new V2Function();
    int tries = 0;
    int maxTries = k.n1*k.l;
    while (tries<maxTries){
      int minIdx = 0;
      int g=0;
      for (int j=0; j<alpha.size();j++){ //initialize minG
        V2Function temp = (bSet.get(0).copy().multiply(
                    alpha.get(j).copy())).multiply(bSet.get(0).inverse());
        temp.reduce();
        g+= temp.numLeaves;
      }
      tries++;
      int minG = g;
      for (int i=1; i<bSet.size(); i++){
        System.out.println(minIdx);
        g=0;
        for (int j=0; j<alpha.size();j++){
          V2Function temp = (bSet.get(i).copy().multiply(
                        alpha.get(j).copy())).multiply(bSet.get(i).inverse());
        temp.reduce();
        g+= temp.numLeaves;
        }
        if (g<minG){
          minG = g;
          minIdx = i;
        }
        tries++;
      }
```

```
      x = x.multiply(bSet.get(minIdx).inverse());
      for (int k=0; k<alpha.size(); k++){
        alpha.set(k,(bSet.get(minIdx).copy().multiply(
                  alpha.get(k))).multiply(bSet.get(minIdx).inverse()));
      }
      for (int i=0; i<alpha.size(); i++){
        if (!(alpha.get(i).equals(aSet.get(i)))){
          break;
        }
        if (i==(alpha.size()-1)) {
          return x.inverse();
        }
      }
    }
    return null;
  }
}
```

Bibliography

[1] Iris Anshel, Michael Anshel, and Dorian Goldfeld, *An algebraic method for public-key cryptography*, Mathematical Research Letters **6** (1999), 287–292.

[2] Ali Abdallah, *Public key cryptography based on some extensions of group*, arXiv preprint **arXiv:1604.04474** (2016).

[3] James Belk and Collin Bleak, *Some undecidability results for asynchronous transducers and the Brin-Thompson group 2V*, arXiv preprint **arXiv:1405.0982** (2014).

[4] Simon R. Blackburn, Carlos Cid, and Ciar Mullan, *Group theory in cryptography*, Proceedings of Group St Andrews 2009 in Bath (2010), 133–149.

[5] Oleg Bogopolski, Armando Martino, and Enric Ventura, *Orbit decidability and the conjugacy problem for some extensions of groups*, Transactions of the American Mathematical Society **362(4)** (2010), 2003-2036.

[6] Matthew G. Brin, *Higher dimensional Thompson groups*, Geometriae Dedicata **108.1** (2004), 163–192.

[7] ———, *Presentations of higher dimensional Thompson groups*, Journal of Algebra **284.2** (2005), 520–558.

[8] ———, *On the baker's map and the simplicity of the higher dimensional Thompson groups nV*, Publicacions Matemàtiques **54.2** (2010), 433–439.

[9] James W. Cannon, William J. Floyd, and Walter R. Parry, *Introductory notes on Richard Thompson's groups*, Enseignement Mathmatique **42** (1996), 215–256.

[10] Max Dehn, *Über unendliche diskontinuerliche Gruppen*, Mathematische Annalen **69** (1911), 116–144.

[11] Whitfield Diffie and Martin E. Hellman, *New directions in cryptography*, Information Theory, IEEE Transactions on **22.6** (1976), 644–654.

[12] Ki Hyoung Ko, Sang Jin Lee, Jung Hee Cheon, Jae Woo Han, Ju-sung Kang, and Choonsik Park, *New public-key cryptosystem using braid group*, Advances in Cryptology - CRYPTO 2000 (2000), 166–183.

[13] Alex D. Myasnikov and Alexander Ushakov, *Length based attack and braid groups: cryptanalysis of Anshel-Anshel-Goldfeld key exchange protocol*, Public Key CryptographyPKC (2007), 76–88.