Heidelberg University
Faculty for Mathematics and Computer Science

Bachelor Thesis

Algorithmic Implementation of the Solution to the Word Problem in Right-Angled Artin Groups

Submission Date: 12 March 2021

Supervisor:
JProf. Dr. Maria Beatrice Pozzetti

Submitted by:
Jannis Leo Heising
jannis.heising@web.de

Preface

The goal of this thesis is to write a program which generates Cayley graphs of right-angled Artin groups. For this, the word problem must be solved. The particular method which we will use, namely pilings, was invented by J. Crisp, E. Godelle, and B. Wiest [1], however, I will first present my own variation of it which is more theoretical but, in my opinion, more intuitive in its functionality. It requires a statement about directed graphs for which I was unable to find a source, thus the proofs in Section 1.4 are my own.
The generated Cayley graphs were originally meant to be used in a machine learning project by F. López, B. Pozzetti, S. Trettel, and A. Wienhard. Since right-angled Artin groups naturally have both free abelian and free subgroups, their Cayley graphs exhibit both flat and hyperbolic features, which makes them quite similar to real-world datasets and thus useful for the testing of graph embedding techniques.
Chapters 2 and 3 were heavily influenced by W. Bell and M. Clay's chapter on right-angled Artin groups in [2]. All illustrations are my own.

Contents

Chapter 1

Preliminaries

1.1 Group Presentations

Group presentations are a versatile way of defining groups by their desired properties. As an introduction, observe that every group is isomorphic to a quotient of a free group[1] in the following way: For a fixed group G, define a homomorphism $\varphi : F(G) \to G, g \mapsto g$ that maps each element of the generating set of the free group $F(G)$, which is the set of elements in G, to itself in G.[2] This is clearly surjective and thus

$$F(G)/\ker(\varphi) \cong G.$$

In particular, this means that we can obtain G from $F(G)$ by declaring when the product of elements in G is the neutral element (in other words when the respective word is in $\ker(\varphi)$). The key insight now is that we can usually pick a much smaller generating set for the free group. It's time for some definitions.

Notation. For a set S, we call $S^{-1} = \{s^{-1} | s \in S\}$ the set of *formal inverses* of S, $L_S = (S \cup S^{-1})$ the set of *letters* over S, and L_S^* the set of *words* with letters in L_S.

Definition 1.1. Let S be an arbitrary set and $R \subset L_S^*$.

- We define $\langle\langle R \rangle\rangle$ to be the smallest normal subgroup in $F(S)$ that contains R.
- We call $\langle S|R \rangle := F(S)/\langle\langle R \rangle\rangle$ a *group presentation* of $G \cong \langle S|R \rangle$. S is the set of *generators* and R is the set of *relations* in $\langle S|R \rangle$.

[1]For a definition of free groups see [3, Sect. 2.2.2].

[2]This definition extends to a homomorphism via the universal property of free groups.

- $\langle S|R\rangle$ is *finitely generated* if S is finite, and it is *finitely presented* if both R and S are finite.

When dealing with specific sets S and R, it is common to omit their curly brackets:[3]

$$\langle\{s_1,\dots,s_n\}|\{r_1,\dots,r_m\}\rangle = \langle s_1,\dots,s_n|r_1,\dots,r_m\rangle.$$

Also, as explained above, the elements of R determine which words in L_S^* represent the neutral element in $\langle S|R\rangle$. To underline this function, we often write

$$\langle s_1,\dots,s_n|r_1,\dots,r_m\rangle = \langle s_1,\dots,s_n|r_1=1,\dots,r_m=1\rangle.$$

Finally, we can treat $r_i = 1$ as a formula: If, for example, $r_1 = s_1s_2s_1^{-1}s_2^{-1}$, we can write $s_1s_2 = s_2s_1$ instead of $s_1s_2s_1^{-1}s_2^{-1} = 1$.

Example 1.2.

- $F_n = \langle s_1,\dots,s_n|\varnothing\rangle$ is the free group with n generators.
- $\mathbb{Z}^n = \langle s_1,\dots,s_n|[s_i,s_j]\ \forall i,j \in [\![1,n]\!]\rangle = \langle s_1,\dots,s_n|s_is_j = s_js_i\ \forall i,j\rangle$ is the n-th free abelian group.
- ${}^{\mathbb{Z}}/_{n\mathbb{Z}} = \langle s|s^n = 1\rangle$ is the cyclic group of order n.

Group presentations satisfy a universal property which we will need later on. Its proof can be found in various places, for example in [3, Sect. 2.2.3].

Theorem 1.3 (Universal property of group presentations)**.** *Let S be an arbitrary set and $R \subset L_S^*$. Then for any group G and any map $\varphi : S \to G$ with the property that $\varphi(r) = e\ \forall r \in R$,[4] there exists a unique homomorphism $\overline{\varphi} : \langle S|R\rangle \to G$ that extends φ.*

1.2 Graphs

Definition 1.4.

- A *graph* Γ is a triple (V, E, ∂) consisting of the *vertex set* $V = V(\Gamma)$, the *edge set* $E = E(\Gamma)$ and the *edge map* $\partial = \partial_\Gamma : E \to \{\{u,v\}|\ u,v \in V\}$.
- Two vertices $u, v \in V$ are *adjacent* if $\{u,v\} \in \partial(E)$. An edge $e \in E$ is called a *loop* if $|\partial(e)| = 1$.
- If ∂ is injective, i.e. there are no double edges, we identify E with $\partial(E)$ and thus omit ∂ from the definition. A graph with injective edge map and without loops it called *simplicial.*

[3]The sets don't have to be finite, I just assume it here for convenience.

[4]Since φ is only defined on letters, we apply it to r letter-wise and multiply the images.

- In a simplicial graph, the *order* of a vertex $v \in V$ is the number of its neighbours, i.e. the number of vertices adjacent to v.

- A *path* is an n-tuple $(v_1, \dots, v_n) \in V^n$ where v_i and v_{i+1} are adjacent for all i and $v_i \neq v_j$ for all $i \neq j$.

- A *cycle* is a path of length $n \geq 3$ where v_1 and v_n are also adjacent.

We want to highlight a few examples of simplicial graphs. See Figure 1.1 for illustration.

Example 1.5.
- A simplicial graph is a *tree* if it does not contain a cycle. A simplicial graph is a tree if and only if for each $u \neq v \in V$ there is exactly one path from u to v, i.e. $v_1 = u$ and $v_n = v$.

- A simplicial graph is a *complete graph* if every pair of vertices is connected, i.e. $E = \{\{u, v\} |\ u \neq v \in V\}$.

- The graph $P_n := \{\{v_1, \dots, v_n\}, \{\{v_i, v_{i+1}\} \mid i \in [\![1, n-1]\!]\}\}$ is called the *n-th path graph.*

- The graph $C_n := \{\{v_1, \dots, v_n\}, \{\{v_i, v_{i+1}\} \mid i \in [\![1, n-1]\!]\} \cup \{v_1, v_n\}\}$ is called the *n-th cycle graph.*

Definition 1.6. Let Γ, Δ be graphs.

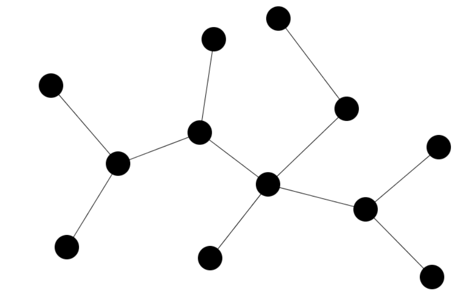
- Δ is called a *subgraph* of Γ if $V(\Delta) \subset V(\Gamma)$, $E(\Delta) \subset V(\Gamma)$ and $\partial_\Delta = \partial_\Gamma|_{E(\Delta)}$. (Again, if there are no double edges, the last condition can be omitted.)

- Δ is called an *induced subgraph* of Γ if it is a subgraph and all edges in Γ connecting vertices in Δ (with respect to ∂_Γ) are also edges in Δ.

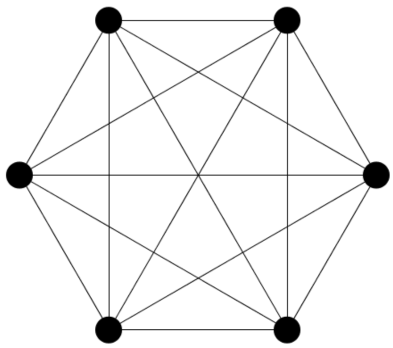
  If Γ is simplicial, this simplifies to the following:[5]

$$\forall v, w \in V(\Delta) : \{v, w\} \in E(\Gamma) \Leftrightarrow \{v, w\} \in E(\Delta).$$

[5] The "$\Leftarrow$" stems from the fact that Δ is a subgraph of Γ.

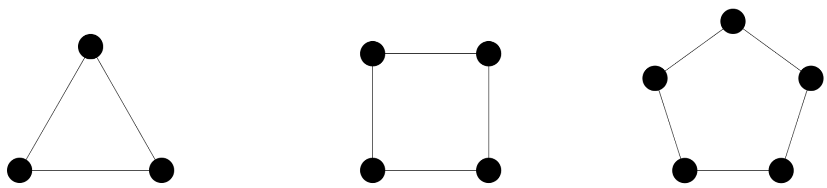


(a) A tree



(b) The complete graph with 6 vertices

(c) The path graphs P_2, P_3, and P_4



(d) The cycle graphs C_3, C_4, and C_5

Figure 1.1: A few examples of simplicial graphs.

1.3 Cayley Graphs

Cayley graphs are a useful way to visualize groups.They are an essential tool of geometric group theory because they make it possible to link abstract algebraic properties of groups to more or less intuitive geometric properties of graphs, or rather of their metric realization[6]. For example, if the Cayley graph of a given group is Gromov-hyperbolic, then the group itself must be finitely presented.

We won't be making much use of this link, but it should serve as a motivation to study Cayley graphs in the first place. For an excellent introduction to geometric group theory, see C. Löh's aptly titled book [3].

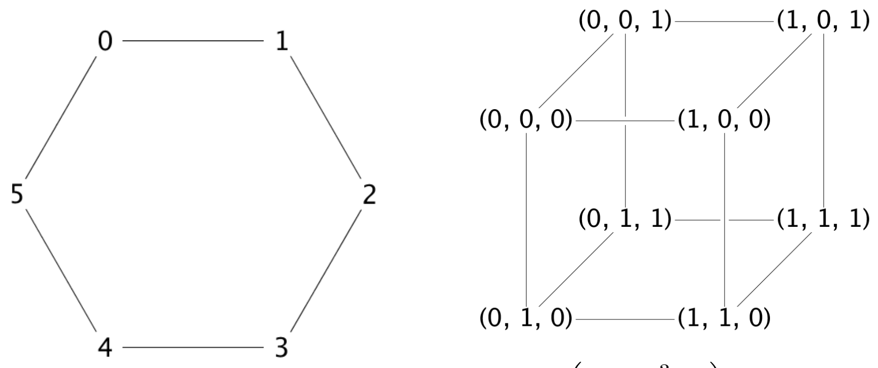
Definition 1.7. Let G be any group and $S \subset G$ be a generating set of G. The *Cayley graph* $\mathrm{Cay}(G, S)$ is defined as the graph $\Gamma = (V, E)$ whose vertex set consists of the elements of G, i.e. $V = G$, and whose edges are defined by

[6]The metric realization of a graph is a continuous metric space which "looks" like that graph, where each pair of adjacent vertices has distance 1.

which elements of G can be directly linked via the right-multiplication of an element of S, i.e.:

$$E = \{\{g, gs\} \mid g \in G, s \in S\}.$$

For examples of Cayley graphs see Figure 1.2.



(a) $\mathrm{Cay}\left(\mathbb{Z}/6\mathbb{Z}, \{1\}\right)$.

(b) $\mathrm{Cay}\left(\left(Z/2\mathbb{Z}\right)^3, S\right)$, where $S = \{(1,0,0), (0,1,0), (0,0,1)\}$.

Figure 1.2: Examples of Cayley graphs.

Corollary 1.8 (Basic Properties). *a) Every vertex in* $\mathrm{Cay}(G, S)$ *has the same order, namely* $|S|$.

b) $\mathrm{Cay}(G, S)$ *is a tree if and only if* G *is generated freely by* S.

1.4 Directed Graphs

We later make use of a theorem about directed graphs, which makes this section necessary. Directed graphs are very similar to graphs[7], the key difference being that, as the name suggests, their edges have a sense of direction. Mathematically, this means that these are not sets but (ordered) pairs. Unfortunately, the theorem requires an extensive amount of terminology[8].

[7]One might call them undirected graphs to avoid confusion, but since I rarely reference directed graphs, I choose to keep the shorter name.

[8]Most of the terminology can also be found in [4, Chap. 16]

Definition 1.9.

- A *directed graph*, or *digraph*, Γ is a tuple (V, E) consisting of the *vertex set* $V = V(\Gamma)$ and the *edge set* $E = E(\Gamma)$, where E consists of pairs of distinct vertices: $E \subset V^2 \backslash \{(v, v) | v \in V\}$.[9]
- The *underlying graph* Γ' of a digraph $\Gamma = (V, E)$ is its undirected counterpart:
$$\Gamma' = (V, \{\{u, v\} | (u, v) \in E\}).$$

We want to classify a handful of ways in which different vertices in a digraph can be related. See Figure 1.3 for illustration.

Definition 1.10. Let $\Gamma = (V, E)$ be a digraph and $u, v \in V$.

- We say that u and v are *adjacent* if they are so in the underlying graph.
- We say that u is *connected to* v if $(u, v) \in E$ (in this order).
- A *path* in Γ is an n-tuple $(v_1, \ldots, v_n) \in V^n$, $n \geq 1$, where v_i is connected to v_{i+1} for all i and $v_i \neq v_j$ for all $i \neq j$.
- A *cycle* is a path of length $n \geq 2$ where v_n is connected to v_1.
- We call v a *successor* of u if there is a path in Γ from u to v. In this case, u is a *predecessor* of v.[10]
- We call v a *direct successor* of u if it is a successor of u and they are adjacent. In this case, u is a *direct predecessor* of v.
- We call u a *source* if it has no direct predecessors.
- We call v a *sink* if it has no direct successors.

$$v_1 \longrightarrow v_2 \longrightarrow v_3$$

Figure 1.3: Es an example, v_1 is connected to v_2. In fact, v_1 is a direct predecessor of v_2, who in turn is a direct successor of v_1. In particular, v_1 and v_2 are adjacent. v_3 is a successor of v_1, though not a direct one. v_1 is a source and v_3 is a sink. There is a path from v_2 to v_3, but none from v_3 to v_2.

[9] Alternatively, this could be referred to as a *simplicial* directed graph because, of course, the definition can be broadened analogously to that of graphs in Section 1.2 to account for double edges and loops.

[10] Note that each vertex is a successor (and predecessor) of itself.

In addition to this basic terminology, we need to label certain classes of digraphs. Figure 1.4 presents an example.

Definition 1.11. Let $\Gamma = (V, E)$ be a directed graph.

- Γ is *finite* if V (and thus E) is finite.
- Γ is *acyclic* if it does not contain any cycles.
- Γ is *weakly connected* if its underlying graph Γ' is connected, i.e. if for every pair of vertices $u, v \in V$, there is a path in Γ' from u to v.[11]
- A *weakly connected component* of Γ is a non-empty induced subgraph[12] of Γ that is weakly connected and maximal (in terms of the subset-ordering) with this property.

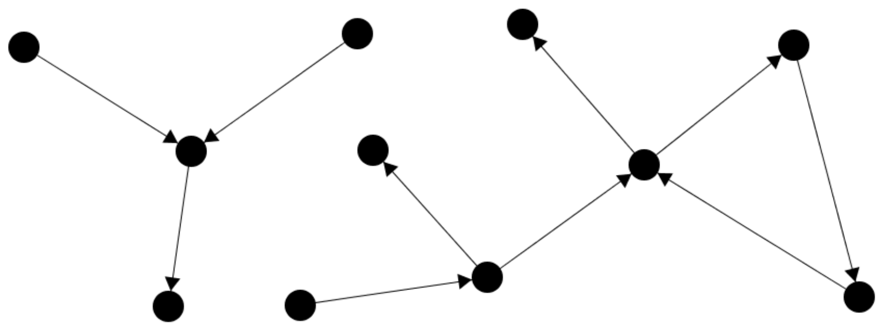


Figure 1.4: This graph is finite, but not acyclic. It has two weakly connected components and is thus not weakly connected.

Finally, we can state the theorem, preceded by a lemma.

Lemma 1.12. *Let Γ be a finite acyclic digraph where for each vertex $v \in V(\Gamma)$, any two direct successors of v have a successor in common*[13]*. Then any two (arbitrary) successors of v have a successor in common.*

[11] In contrast, Γ would be called *strongly connected* if such a path exists in Γ for all $u, v \in V$. Every strongly connected digraph is weakly connected, and no (non-trivial) strongly connected digraph is acyclic.

[12] See Definition 1.6 with minor abuse of notation.

[13] To clarify, this means that there exists a vertex which is a successor of both of these vertices. I will also refer to this as (one of) their *common successor(s)*.

Proof. The idea of the proof is to cast a sort of "net" of paths to common successors over Γ (see Figure 1.5). Let $v \in V(\Gamma)$ be fixed and let $a, b \in V(\Gamma)$ be two successors of v with paths $(v, a_1, \ldots, a_n = a)$ and $(v, b_1, \ldots, b_m = b)$ from v to a and b respectively. First we find some common successor c_1 of a_1 and b_1. The next step is to find a common successor c_2 of a_2 and c_1, which is automatically a common successor of a_2 and b_1. For this we repeatedly find common successors of vertices on the path from a_1 to c_1 and on the paths that arise from these (again, see Figure 1.5). This process stops because Γ is acyclic (so every vertex appears at most once) and finite. Similarly, we can now find a common successor c_i of a_i and c_{i-1} until we reach $i = n$, i.e. $a_i = a$. At this point we have found a common successor $d_1 := c_n$ of a and b_1. Now we can find $d_2, \ldots, d_m$, analogously defined. Finally, d_m is a common successor of a and b. $\square$

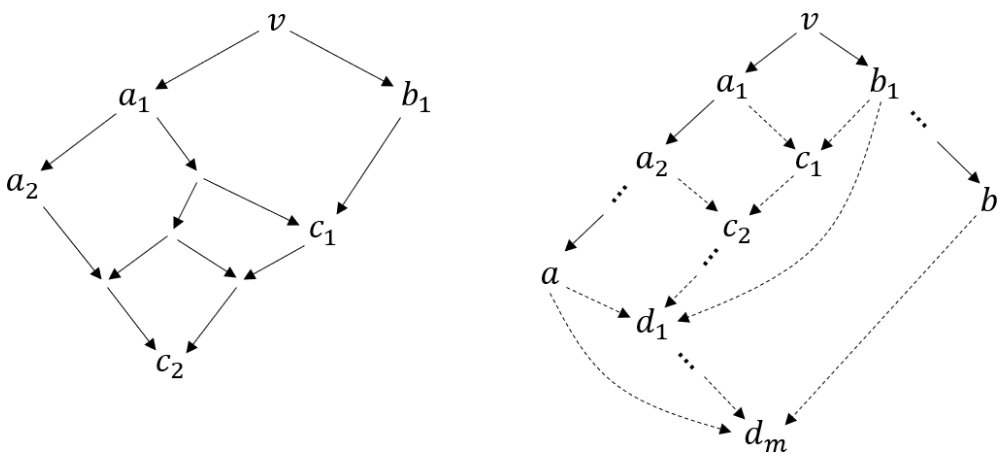


Figure 1.5: On the left we see an example of how to find c_2. On the right we see how this process yields a common successor of a and b. Solid lines indicate direct successors and dotted lines indicate successors of any sort.

Theorem 1.13. *Let Γ be a (non-empty) finite weakly connected acyclic digraph where for each vertex $v \in V(\Gamma)$, any two direct successors of v have a successor in common. Then Γ has exactly one sink.*

Proof. First we prove that each vertex in Γ has precisely one sink as a successor (in particular that there is at least one sink), using Lemma 1.12 to show that this is the maximum amount. Then we lead the assumption that there is more than one sink to a contradiction.

Let $v \in V$ be a vertex. Define a path starting at v by repeatedly choosing some direct successor of the last vertex of the path. Since Γ is acyclic, this is indeed a path, meaning that no vertex appears more than once, and because Γ is finite, this process has to stop. By definition, the last vertex of this path has no direct successor, in other words it is a sink. Since the path started at v, this sink is a successor of v.

Now assume v has two different sinks as successors. Lemma 1.12 tells us that these must have a common successor. But since they have no successors other than themselves, this cannot be $\lightning$.

Let $S \subset V$ be the set of all sinks in Γ. Define a function $f : V \to S$ that maps each vertex to the sink it precedes. Assume that there exist two distinct sinks $s_1, s_2 \in S$. Let $M_i := f^{-1}(\{s_i\})$, $i \in \{1, 2\}$. Observe that $s_i \in M_i \neq \varnothing$. We want to show that M_1 is not weakly connected to M_2 in Γ.[14] So suppose they are. Let $v_1 \in M_1$ and $v_2 \in M_2$ be weakly connected. Without loss of generality they are adjacent[15]. But this means that, again without loss of generality, v_1 is a direct predecessor of v_2, in particular that both s_1 and s_2 are successors of v_1 $\lightning$.

So Γ must have more than one weakly connected component, in contradiction to our premise. Thus there can only be one sink. □

To make some more use of all this jargon and to aid ourselves later on, here is a quick corollary.

Corollary 1.14. *Let Γ be a finite acyclic digraph. Then each weakly connected component of Γ contains a source. In particular, Γ has a source (assuming it is non-empty) and must be weakly connected if it has just one.*

Proof. Let Δ be a weakly connected component of Γ and let $v \in \Delta$. We define a path similar to the one in the proof of Theorem 1.13, except that it *ends* in v and we successively choose direct *predecessors*. As we've seen, this process stops and does yield a path, the starting point of which by definition has no direct predecessor, thus is a source. Because there is a path from this source to v (in Γ and thus in the underlying graph), they both lie in the same weakly connected component. □

[14] As a sidenote, M_i itself is weakly connected, so this shows that M_i is indeed a weakly connected component.

[15] Otherwise change s_2 so that its preimage under f contains the first vertex of the path from v_1 to v_2 that isn't contained in M_1 and change v_1 and v_2 accordingly.

Chapter 2

Right-Angled Artin Groups

2.1 Definition

Right-angled Artin groups, RAAGs for short, are a very interesting and large class of groups that contain free groups on one end and free abelian groups on the other. They are defined by a (usually finite) number of generators and a set of commuting relations between them. This can be nicely visualized by using a simplicial graph whose vertices represent the generators and whose edges indicate which generators commute.

Definition 2.1. Let $\Gamma = (V, E)$ be a simplicial graph. The *right-angled Artin group* $A(\Gamma)$ assoziated to Γ is defined as follows:

$$A(\Gamma) = \langle V \mid [v, w] = 1 \; \forall \{v, w\} \in E \rangle.$$ [1]

Example 2.2.

- The complete graph $\Gamma = (\{v_1, \ldots, v_n\}, \{\{v_i, v_j\} | i \neq j\})$ (all vertices are connected) generates the free abelian group $A(\Gamma) = \mathbb{Z}^n$.
- The disconnected graph $\Gamma = (\{v_1, \ldots, v_n\}, \varnothing)$ generates the free group $A(\Gamma) = F_n$.

2.2 Properties

Right-angled Artin groups have some nice properties that make it comparatively easy to work with them. More specifically, there are a number of properties that follow directly from the structure of the underlying graph.

[1] Of course, it is also possible to define RAAGs the opposite way, i.e. to only have *non*-adjacent vertices commute. Both definitions are used in the literature.

For instance, induced subgraphs[2] of Γ yield subgroups of $A(\Gamma)$. Let's see how this works.

Theorem 2.3. *Let Γ be a simplicial graph and Δ be an induced subgraph of Γ. Then $A(\Delta)$ is a subgroup of $A(\Gamma)$.*

Proof. Let $\widetilde{\varphi} : V(\Delta) \to A(\Gamma), v \mapsto v$. This extends to a homomorphism $\varphi : A(\Delta) \to A(\Gamma)$ via the universal property of group presentations described in Theorem 1.3. To prove that $A(\Delta)$ is a subgroup of $A(\Gamma)$, we need to show that φ is injective, which can be done by finding a left inverse to it: Let

$$\widetilde{\psi} : V(\Gamma) \to A(\Delta), v \mapsto \begin{cases} v & v \in V(\Delta) \\ 1 & v \notin V(\Delta) \end{cases}.$$

This, too, extends to a homomorphism $\psi : A(\Gamma) \to A(\Delta)$.[3] Observe that $(\psi \circ \varphi)(v) = v$ for all $v \in V(\Delta)$. Finally, let $\pi : V(\Delta) \to A(\Delta), v \mapsto v$. This clearly extends to the identity on $A(\Delta)$, but $\psi \circ \varphi$ is also an extension, and thus by Theorem 1.3 these have to be equal, meaning that ψ is a left inverse to φ. □

From this point onward let Γ be a simplicial graph.

Corollary 2.4. *a) Let $v, w \in V(\Gamma)$ not be adjacent in Γ. Then*

$$\langle v, w \rangle_{A(\Gamma)} \cong F_2.$$

b) $V(\Gamma)$ is abelian if and only if Γ is complete.

Proof. a) Let Δ be the induced subgraph of Γ containing only v and w, that is $V(\Delta) = \{v, w\}$ and $E(\Delta) = \varnothing$. Then $A(\Delta) = \langle v, w | \varnothing \rangle \cong F_2$ and $A(\Delta) \hookrightarrow A(\Gamma)$ as we've just seen. Thus $\langle v, w \rangle_{A(\Gamma)} \cong F_2$.

b) It is clear that $V(\Gamma)$ is abelian if Γ is complete. On the other hand, if there are $v, w \in V(\Gamma)$ that aren't adjacent in Γ, then we've just seen that these elements do not commute, thus $V(\Gamma)$ isn't abelian.

□

[2]See Definition 1.6.

[3]Note that this is only true because Δ is an *induced* subgraph. Otherwise the requirement "$\psi(r) = e$" in the universal property wouldn't be fulfilled.

Chapter 3

The Word Problem in RAAGs

3.1 What is the Word Problem?

Let $G = \langle S|R \rangle$ be any finitely presented group. The word problem states the following: Given any word $w \in L_S^*$ with letters in $L_S = (S \cup S^{-1})$, is there an algorithm to determine whether or not w represents the identity in G? We say that the word problem is solvable for a group if such an algorithm exists. This might seem like an easy problem at first glance because in many of the widely known types of groups the word problem is solvable in a straightforward manner (e.g. finite groups, free groups, free abelian groups). But in general such an algorithm is hard or even (provably) impossible to find.

3.2 Normal Forms

One possible way to solve the word problem in a given group is to find a so-called *normal* or *preferred form* for its elements. For now, this will be a subset of all words $\mathcal{N} \subset L_S^*$ such that every element in G is represented by precisely one word in $\mathcal{N}$. Of course there are many different options to choose such a normal form. The trick is to find one that can be generated by an algorithm, meaning that whenever two words represent the same element, the algorithm maps them to the same word in $\mathcal{N}$.

To illustrate this, we will find such a normal form, or rather the corresponding algorithm, for free abelian groups. The usual normal form for elements in $\mathbb{Z}^n$ is an n-tuple with entries in $\mathbb{Z}$ (the cartesian form). I would like to describe a different normal form which will seem more complicated but is essentially the same. This is simply a stepping stone towards RAAGs.

We have

$$\mathbb{Z}^n = \langle z_1, \dots, z_n |\ [z_i, z_j] = 1\ \forall i, j \in [\![1, n]\!]\rangle.$$

Given any word $w = a_1 \cdots a_m$ with letters $a_i \in L_S$, where $S = \{z_1, \dots, z_n\}$, we can iteratively apply a set of simplifying moves:

i) If $a_i = a_{i+1}^{-1}$ for some i, remove $a_i a_{i+1}$ from w.

ii) For some i let j, k be such that $a_i = z_j^{\epsilon_1}$, $a_{i+1} = z_k^{\epsilon_2}$, $\epsilon_1, \epsilon_2 \in \{1, -1\}$. If $j > k$, replace $a_i a_{i+1}$ by $a_{i+1} a_i$ in w.

The second move sorts the letters by their chosen order and the first move simply applies the definition of inverses in groups to formal inverses in words, eliminating redundancy.[1] It is intuitively clear that these moves don't change the element w represents, that iteratively applying them will at some (finite) point stagnate, that the order in which they are applied does not change the end product, which we will call the word's *reduced form*, and, most importantly, that every element in $\mathbb{Z}^n$ corresponds to precisely one such reduced form, in other words that this reduction process yields a normal form. By reducing (i.e. simplifying to the point of stagnation) every word in L_S^*, we get $\mathcal{N}$. It follows from the explanation above that

$$\mathcal{N} = \{z_1^{c_1} \cdots z_n^{c_n} | c_i \in \mathbb{Z}\},$$

where of course $z_i^{c_i}$ stands for $|c_i|$ copies of the letter $z_i^{\operatorname{sgn}(c_i)}$.

In general, once we have an algorithm that produces a normal form, the word problem is thusly solved: Given a word w, find its normal form and compare it to that of the identity. If these match, w must represent the identity, if they don't, w cannot do so.

Example 3.1. With the algorithm above, we get that the normal form of the identity is the empty word ε. As an example, we will determine whether the word $z_2 z_1^{-1} z_3 z_2^{-1} z_3^{-1} z_1$ represents the identity in $\mathbb{Z}^3$:

$$\begin{aligned}
z_2 z_1^{-1} z_3 z_2^{-1} z_3^{-1} z_1 &\overset{(ii)}{=} z_2 z_1^{-1} z_2^{-1} z_3 z_3^{-1} z_1 \\
&\overset{(i)}{=} z_2 z_1^{-1} z_2^{-1} z_1 \\
&\overset{(ii)}{=} z_2 z_1^{-1} z_1 z_2^{-1} \\
&\overset{(i)}{=} z_2 z_2^{-1} \\
&\overset{(i)}{=} \varepsilon.
\end{aligned}$$

[1] To get from here to the cartesian form, we simply count the number of letters corresponding to each generator z_i and choose the sign based on whether the letters are inverses or not (they are either all z_i or all z_i^{-1}). This will be the i-th entry.

So indeed it does.

Observe that by omitting (ii), we receive an algorithm to find normal forms in free groups. Indeed, by modifying (ii), we can generalize this method to work for any right-angled Artin group. This, however, we will do more rigorously.

3.3 Normal Forms in RAAGs

Let Γ be a simplicial graph with vertex set $V = \{z_1, \ldots, z_n\}$. Given a word $w \in L_V^*$, we define the following labelling function:

Definition 3.2. For $w = a_1 \cdots a_n$, $a_i \in L_V$, let $\sigma_w : [\![1, m]\!] \to [\![1, n]\!]$ be such that $a_i = z_{\sigma_w(i)}^{\epsilon_i}$, where $\epsilon_i \in \{1, -1\}$ is arbitrary.

To illustrate, if $w = z_1 z_1 z_2^{-1}$, we have $\sigma_w(1) = 1$, $\sigma_w(2) = 1$ and $\sigma_w(3) = 2$. Just as before, we describe a set of simplifying moves:

i) If $a_i = a_{i+1}^{-1}$ for some i, remove $a_i a_{i+1}$ from w.

ii) For some i, j with $i < j$, if $z_{\sigma_w(j)}$ is adjacent to all $z_{\sigma_w(i)}, \ldots, z_{\sigma_w(j-1)}$ in Γ and $\sigma_w(i) > \sigma_w(j)$, replace $a_i \cdots a_j$ with $a_j a_i \cdots a_{j-1}$ in w.

Refer by $r(w)$ to the reduced form[2] of any word w (in a second we will show that this is well-defined). What follows is an array of lemmata that describe certain robustness properties of the reduction process.

Lemma 3.3. *Applying (i) and (ii) iteratively will at some point yield a word to which neither (i) nor (ii) are applicable, i.e. the reduction process terminates.*

Proof. Both moves don't make the word longer, and since (i) actively reduces its size it can only be applied a finite number of times.

With this in mind we can prove the lemma by proving it purely for (ii); (i) might mess the proof up, but it can only do so finitely often. First look at all the letters in w that are z_1 or its inverse. These can never be moved to the right, only to the left, so it is clear that they can only move finitely often. Next look at all the letters that are z_2 or its inverse. These can only be moved to the right if there is a letter z_1 or z_1^{-1} to their right, which again can only happen finitely often, so they, too, are limited in their number of moves. Inductively we can see that this applies to all letters and thus to w as a whole. □

[2] To reiterate, by this we mean the result of repeatedly simplifying w until there is nothing left to simplify.

Lemma 3.4. *The order in which (i) and (ii) are applied does not matter, i.e. $r(w)$ is well-defined.*

Proof. Let w be any word with letters in L. The proof consists of two steps: First we show that whenever two different simplifying moves are applicable to w, we can apply a set of moves to each of the resulting words to reach the same word (see Figure 3.1 for illustration). Then we translate the problem into the language of directed graphs[3] and apply Theorem 1.13.

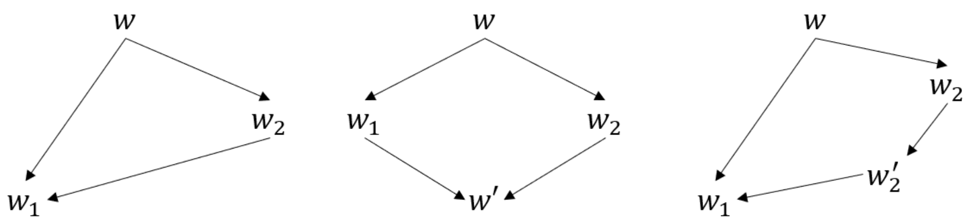


Figure 3.1: w_1 and w_2 can always be simplified to the same word in some way.

For the first step, we need to check three cases, namely when (i) and (ii) are applicable at the same time and when either of them is applicable at two positions. In each case we will look at specific examples of words, but the general case follows directly from these through index swapping, inversion and multiplication from the left and the right. We will also omit cases where the moves don't interfere with one another since their solution is obvious. For convenience, let v, v', v'' always be words in L whose letters all commute with z_1, and let z_2 commute with z_1.

- "(i), (ii)": Let $w = z_2 z_2^{-1} v z_1$. Then (i) and (ii) are applicable and we have
$$w_1 = vz_1, w_2 = z_2 z_1 z_2^{-1} v$$
or
$$w_1 = vz_1, w_2 = z_1 z_2 z_2^{-1} v.$$
In the first case, we can apply (ii) again to reach the second case. Here we apply (ii) to w_1 and (i) to w_2 to reach the word $z_1 v$ both times.

  Now let $w = vz_1 z_1^{-1}$. Then we have
$$w_1 = v, w_2 = z_1 v z_1^{-1}$$
or
$$w_1 = v, w_2 = z_1^{-1} v z_1.$$

[3] See Section 1.4.

In both cases we can commute the last letter in w_2 with v and then apply (i), resulting in $v = w_1$.

- "(i), (i)": Let $w = z_1 z_1^{-1} z_1$. Although (i) is applicable in two different places, the result is the same both times, namely z_1.
- "(ii), (ii)": Let $w = z_2 v z_2 v' z_1 v'' z_1$. Then no matter how (ii) is applied, we can always reach $z_1 z_1 z_2 v z_2 v' v''$ from there.

This concludes step one. Now we define a directed graph (whose vertices are words) in the following way: The first vertex is the word w. From here, we successively apply (i) and (ii) to each existing vertex, add the outcomes as new vertices and connect it to them (in this direction). The resulting digraph Γ has exactly one source, namely w, so by Corollary 1.14 it is weakly connected. Furthermore, Lemma 3.3 shows that Γ is both finite and acyclic (otherwise the reduction process wouldn't have to terminate). Lastly, we've just shown in step one that Theorem 1.13 is applicable, so we know that Γ has precisely one sink. But a sink, by definition, is an unsimplifiable, thus reduced, word. So translating this back, we see that w can only be reduced in one way. □

Lemma 3.5. $r(r(w)) = r(w)$.

Proof. By definition $r(w)$ is a word that cannot be simplified further. Thus applying $r(.)$ again has no effect. □

Lemma 3.6. *$r(w)$ and w represent the same element in $A(\Gamma)$.*

Proof. We only need to check that applying (i) or (ii) doesn't change the element which a word represents. Conveniently, they are designed for precisely this purpose: (i) is merely the definition of an inverse, and (ii) possesses this property because only letters whose counterparts in $A(\Gamma)$ commute may be swapped. □

We now want to show that there is a one-to-one correspondence between reduced words in $L^*_{V(\Gamma)}$ and elements in $A(\Gamma)$. For this, we define a group structure on $H := \{w \in L^*_V | w \text{ is reduced, i.e. } r(w) = w\}$ and show that there is an isomorphism from H to $A(\Gamma)$.

Lemma 3.7. *Define multiplication in H as concatenation plus reduction: $v \circ w := r(vw)$. Then $(H, \circ)$ is a group.*

Proof. The neutral element is the empty word and inverses are formal inverses. It remains to show that $\circ$ is associative.

Let $a, b, c \in H$. Observe that both $a \cdot r(bc)$ and $r(ab)c$ are merely simplifications of the word abc, so by Lemma 3.4 we know that they get reduced to the same word:

$$a \circ (b \circ c) = r(a \cdot r(bc)) = r(r(ab)c) = (a \circ b) \circ c.$$

□

Now let $\varphi : H \to A(\Gamma)$ map words to their respective elements. Lemma 3.6 tells us that this is a homomorphism.

Lemma 3.8. *The kernel of φ is trivial: $ker(\varphi) = \{\varepsilon\}$.*

Proof. Let $w = a_1 \cdots a_m \in ker(\varphi)$, $m \geq 0$.[4] In particular, w is reduced. We first want to show that for each $z \in V$ there is an equal amount of a_i's equal to z as there is of a_i's equal to z^{-1}. Fix a $z \in V$ and consider the induced subgraph Δ of Γ consisting only of z together with the homomorphism

$$\psi : A(\Gamma) \to A(\Delta), v \mapsto \begin{cases} z & v = z \\ 1 & v \neq z \end{cases}$$

from the proof of Theorem 2.3 which, when dealing with words, simply removes all the letters apart from z and z^{-1}. We know that w represents the neutral element in $A(\Gamma)$, so $\psi(w)$ must also be 1, meaning that all the z's and z^{-1}'s in w perfectly cancel.

Now assume $m > 0$. Let i be minimal such that $a_i = a_1^{-1}$. We know that $i > 2$ because otherwise w would start with $a_1 a_1^{-1}$, which could be further simplified. Assume that $z_{\sigma_w(1)}$ is adjacent to all $z_{\sigma_w(2)}, \ldots, z_{\sigma_w(i-1)}$ in Γ. For w to be reduced, it must hold that $\sigma_w(1) < \sigma_w(2)$ (otherwise they would have been swapped in the reduction process). But because of $\sigma_w(1) = \sigma_w(i)$, this means that $\sigma_w(2) > \sigma_w(i)$, and since $z_{\sigma_w(i)} = z_{\sigma_w(1)}$ is adjacent to all $z_{\sigma_w(2)}, \ldots, z_{\sigma_w(i-1)}$, (ii) can be applied to w, so w is not reduced $\lightning$.

Thus there is a $j \in [\![2, i-1]\!]$ such that $z_{\sigma_w(1)}$ and $z_{\sigma_w(j)}$ aren't adjacent. Once again, we look at an induced subgraph of Γ. Let $\Delta' = (\{z_{\sigma_w(1)}, z_{\sigma_w(j)}\}, \varnothing)$ and $\psi' : A(\Gamma) \to A(\Delta')$ just like in Theorem 2.3. We know thanks to Corollary 2.4 that $A(\Delta')$ is free. As mentioned earlier, an element of a free group is the neutral element precisely if any word representation of it is reducible to the empty word only by cancelling letters with their formal inverses. One representation of $\psi'(\varphi(w))$ is the word w' defined by removing all the letters from w apart from $z_{\sigma_w(1)}$ and $z_{\sigma_w(j)}$ and their formal inverses, which, by this logic, cannot represent the neutral element. But this means that $\psi'(\varphi(w))$ and, in particular, $\varphi(w)$ aren't neutral. Thus $w \notin \ker(\varphi)$ $\lightning$. □

[4] $m = 0$ would mean that w is the empty word.

Theorem 3.9. *For any two words $v, w \in L^*$, $r(v) = r(w)$ if and only if v and w represent the same word in $A(\Gamma)$.*

Proof. In Lemma 3.8 it was shown that φ is a bijection. The statement follows from this. $\square$

This means that H can be used as a normal form and, via the process described earlier, that we have solved the word problem in right-angled Artin groups.

3.4 Pilings

There is a nice way to visualize the normal form we've just found. As it happens, it lets us solve the word problem in RAAGs in linear time (in reference to word length)[5]. This method was introduced in [1].

Let Γ be a simplicial graph with vertex set $V = \{z_1, \ldots, z_n\}$, and let $i \in [\![1, n]\!]$. Imagine n ordered vertical strings (see Figure 3.2). Place a "$\oplus$" symbol on top of the i-th string and let it "slide" to the bottom. Now for every $j \in [\![1, n]\!]$ such that z_i and z_j are not adjacent in Γ, place a neutral "$\bigcirc$" symbol on the j-th string in the same manner as before. The resulting picture is called the *piling* $\pi_\Gamma(z_i)$ of z_i. The piling of z_i^{-1} is produced similarly, using "$\ominus$" instead of "$\oplus$".

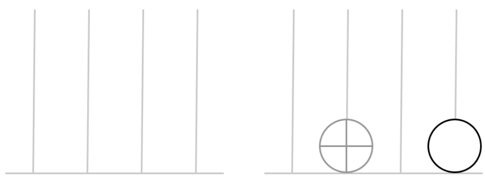


Figure 3.2: On the left we see $n = 4$ vertical strings. On the right we see the piling of z_2 in $A(P_4)$.[6]

More rigorously, the strings may be thought of as (initially empty) words, the symbols "$\oplus$" and "$\bigcirc$" may be distinct letters, and "$\ominus$" may be the formal inverse of "$\oplus$".

Now let $w = a_1 \cdots a_m \in L_V^*$. To find the piling of w, we start with $\pi_\Gamma(a_1)$. On top of this we place $\pi_\Gamma(a_2)$, which we let slide down just as before. Should a "$\oplus$" land on a "$\ominus$" or vice versa, the following cancellation occurs: Let i be the index of the string on which it happened. Now on

[5] I will not explicitly prove this, but it is fairly straightforward.

[6] For a definition of P_4 see Section 1.2.

every string whose corresponding vertex is not adjacent to z_i, remove the top two symbols, which must necessarily be "$\bigcirc$".[7] Finally, remove the initial "$\oplus$" and "$\ominus$" symbols. In our more rigorous setting, this corresponds to the following: First, exchange all "$\bigcirc$" symbols in the piling of a_2 with their formal inverse. Then concatenate its words to their respective counterparts in $\pi_\Gamma(a_1)$ with subsequent cancellation of formal inverses. We repeat this process with $a_2, \ldots, a_m$. The result is called the piling $\pi_\Gamma(w)$ of w (see Figure 3.3).

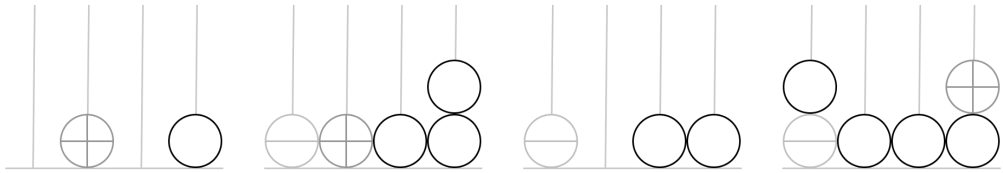


Figure 3.3: From left to right we see the pilings of z_2, $z_2 z_1^{-1}$, $z_2 z_1^{-1} z_2^{-1}$, and $z_2 z_1^{-1} z_2^{-1} z_4$ in $A(P_4)$.

We now want to show that the piling of any word in L_V^* is identical to that of its reduction and that no two reduced words generate the same piling. In other words, we want to show that $\pi_\Gamma(.)$ generates a normal form.

Theorem 3.10. *Let $w \in L_V^*$. Then $\pi_\Gamma(r(w)) = \pi_\Gamma(w)$.*

Proof. We need to show that applying the simplifying steps (i) or (ii) to w does not change its piling. Note that (i) has the same effect as the cancellation process included in the definition of a piling, so it obviously does not change it. As for (ii), observe that the order in which pilings of adjacent vertices are dropped does not matter. This is because neutral symbols are only placed on strings whose corresponding vertices are not adjacent (see Figure 3.4). This proves the theorem. □

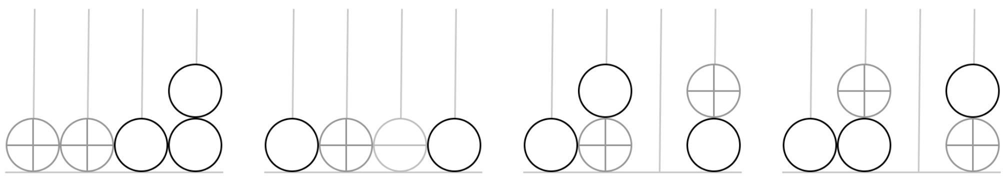


Figure 3.4: From left to right we see the pilings of $z_1 z_2$ (or $z_2 z_1$), $z_2 z_3^{-1}$ (or $z_3^{-1} z_2$), and finally $z_2 z_4$ and $z_4 z_2$ (which are not adjacent) in $A(P_4)$.

[7] I leave it as a challenge for the reader to explain why this is true.

Theorem 3.11. *Let $w_1, w_2 \in H$ be distinct. Then $\pi_\Gamma(w_1) \neq \pi_\Gamma(w_2)$.*

Proof. Assume that $\pi_\Gamma(w_1) = \pi_\Gamma(w_2)$. It is not hard to see that when generating a piling of a reduced word, no cancellation can occur. So we know that w_1 and w_2 must have the same length because otherwise, $\pi_\Gamma(w_1)$ and $\pi_\Gamma(w_2)$ would contain different amounts of "$\oplus$" and "$\ominus$". Let $w_1 = a_1 \cdots a_m$ and $w_2 = b_1 \cdots b_m$. Without loss of generality, $a_1 \neq b_1$. Let $i = \sigma_{w_1}(1)$ and $j = \sigma_{w_2}(1)$. So the i-th string of $\pi_\Gamma(w_1)$ and the j-th string of $\pi_\Gamma(w_2)$ must start (from the bottom) with either a "$\oplus$" or a "$\ominus$". Since they are the same, both strings must have this property in both pilings. But this means that z_i and z_j commute and are in a position to do so in both w_1 and w_2, which means that (ii) is applicable to either w_1 or w_2, depending on which of i and j is smaller. Thus one of them is not reduced $\lightning$. $\square$

To reiterate, not only are pilings a visually more appealing normal form compared to the one we found in the previous section, but they are also computationally superior. We will use this to our advantage when implementing an algorithm to solve the word problem in Chapter 4.

3.5 Side Note: Properties of Pilings

Before moving on, I would like to ponder pilings for a moment. As we've seen, they take the form of n words made up of three distinct letters. However, not all such constructs are pilings (for a given simplicial graph Γ) because some do not lie in the image of π_Γ. This raises the question of how to determine whether or not a given construct, which we shall call an *abstract piling*[8], is indeed a piling.[9]

As a start, let $\Gamma = P_4$. In Figure 3.5, we see two abstract pilings which are not "real" pilings (in relation to Γ)[10]. For example, the left piling suggests that the first generator has to commute with the last one. How can we see at a glance that the right piling also cannot stem from Γ? Suppose it did. Each "$\oplus$" and "$\ominus$" requires a certain number of "$\bigcirc$" to accompany it. The "$\ominus$" on the first (left-most) string, for instance, requires two "$\bigcirc$" since in Γ there are two vertices which aren't adjacent to z_1. The second "$\ominus$" only requires one.

[8] Terminology courtesy of [1].

[9] To clarify, there are abstract pilings which are not pilings in relation to *any* simplicial graph. Additionally, however, different graphs may not permit the same pilings. We will focus on the latter.

[10] However, they are pilings in relation to another graph. Can you figure out which one?

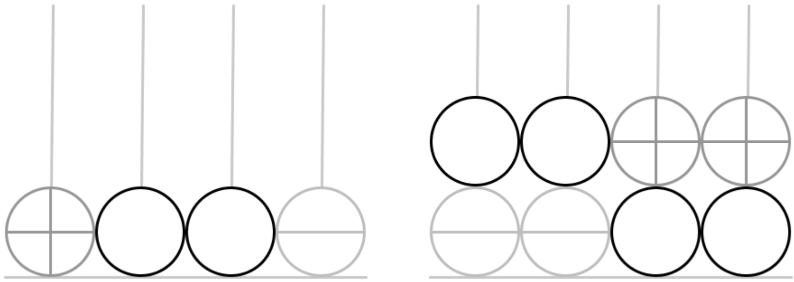


Figure 3.5: Examples of abstract pilings.

To generalize, let x_i be the number of "$\oplus$" and "$\ominus$" symbols on the i-th string and let y_i be the number of vertices in Γ which aren't adjacent to the i-th vertex. Then the piling should contain

$$m := \sum_{i=1}^{n} x_i \cdot y_i$$

"$\bigcirc$" symbols. In the case of the right piling, m amounts to six, however there are only four "$\bigcirc$" symbols present $\lightning$.

Satisfying this formula is necessary for being a piling. Sadly, it is not sufficient since it does not take the positions of the "$\bigcirc$" symbols into account. In order to be certain about the validity of a given abstract piling, we need to follow a simple algorithm: As long as any "$\oplus$" or "$\ominus$" is "exposed", i.e. at the top of a string (which is always the case except in the empty piling), remove it along with the top letter of every string whose corresponding vertex is not adjacent to that of the initial string. If any of those strings was empty or did not have a "$\bigcirc$" at the top, then the abstract piling cannot be "real" (in relation to Γ). On the other hand, if we reach the empty piling, then it must be "real".

Note that this algorithm can be modified to produce a word whose image under π_Γ is the given piling (assuming the algorithm reaches the empty piling): Each step is equivalent to a cancellation step in the piling-generating algorithm. So by recording the letters that would be needed for such a cancellation (in order), we must get a word which represents the inverse of the element being represented by the initial piling (because together they yield the empty piling). Thus, via inversion, we have found a word with the requested property. Indeed, by always choosing the right-most exposed "$\oplus$" or "$\ominus$", we end up with its reduction.

As a final note, take a look at the challenge posed in Footnote 10 (if you haven't already). It can be generalized in the following way: Given a set of abstract pilings, find the set of graphs for which they are "real"[11]. More precisely, we are interested in *isomorphism classes* of graphs (since isomorphic graphs generate isomorphic RAAGs). We can look at each piling individually and then take the intersection of the resulting sets of graphs, so the challenge can be rephrased to only allow for a single abstract piling instead of a set. From the width of the piling we can deduce the number of vertices needed. It is immediately clear that the vertices corresponding to exposed "$\oplus$" and "$\ominus$" must all be adjacent. As an example, the pilings in Figure 3.5 infer that, if we label the vertices canonically, z_1 and z_4 as well as z_3 and z_4 are adjacent. The same is true for the bottom-most such symbols, in this case z_1 and z_2. Alas, from here on out, as far as I can tell, finding the remaining adjacencies is generally a matter of trial-and-error. The difficulty lies in assigning the neutral symbols to the non-neutral ones[12]. It is not hard to see that the first piling in Figure 3.5 allows for four isomorphism classes of graphs whereas the second piling only allows for one (I urge the reader to find these). Generally, however, it seems to be very difficult to find all possible graphs and then check for isomorphisms.[13]

To give a rough upper bound for the difficulty of the first part (i.e. finding the graphs), suppose the given piling has n strings. If we label the vertices of the graphs, there are $2^{\binom{n}{2}} = \sqrt{2^{n(n-1)}}$ different graphs with n vertices. Now let m_i be the number of non-neutral symbols in the i-th string (thus the reduced word generating the given piling contains m_i instances of z_i or z_i^{-1}). Then the number of words with these amounts of certain letters and their inverses (in their given order) is

$$M := \prod_{i=1}^{n} \binom{\widetilde{m}_i}{m_i},$$

where $\widetilde{m}_i := \sum_{j=1}^{i} m_i$.[14] Thus, relying only on trial-and-error would require us to check (no more than)

$$N := \sqrt{2^{n(n-1)}} \cdot M$$

cases. For the second piling above, this amounts to $N = 64 \cdot 24 = 1536$.

[11] i.e. for each graph and each abstract piling there is a word whose piling in relation to that graph is that piling.

[12] Neutral symbols are only ever generated in conjunction with non-neutral ones.

[13] As a result, it may be possible to turn this concept into an entertaining logic puzzle.

[14] This formula is best understood by iterating backwards, i.e. from n to 1: The m_n letters $z_n^{\pm 1}$ can be placed freely, the next m_{n-1} letters $z_{n-1}^{\pm 1}$ have m_n fewer places to go etc.

Chapter 4

Implementation in Java

Our goal is to write a program that takes a simplicial graph Γ as an input and generates the Cayley graph of $A(\Gamma)$[1], using pilings as a way to solve the word problem. In fact, the final program will be easily adjustable to work for any class of groups for which a word-problem-solving algorithm is known. I will only outline the parts of the source code which are of mathematical interest; the complete code can be found in the appendix.

It ought to be mentioned that I am not a computer science but a mathematics undergraduate and, while my code runs quite well, one might not call it “clean”. On top of this, there are probably languages other than Java that are better-suited for the task at hand, alas Java is the language I know best. Specifically, I use a software called *processing* [5] which is designed to make visual programming much easier. It uses OpenGL for threedimensional rendering.

4.1 Overview

In order to output a Cayley graph, we require a *Graph* class, and since graphs consist of edges and vertices, it should prove useful to implement classes for both of these as well[2]. On the other hand, we need to generate the Cayley graph. We could hard-code the algorithm for RAAGs, but I found it to be of no significant increase in work to generalize the program, using an abstract *Algorithm* class which can be altered to work for a vast class of groups with solvable word problem.

[1]Of course I mean the Cayley graph obtained by the generating set $V(\Gamma)$.

[2]This is not strictly necessary. I touch on why that is and why we still do it later on.

Our specific algorithm uses pilings, so a *Piling* class seems appropriate, along with a *Word* class, since pilings consist of words.

We will now go through all of these classes in greater detail, using pseudocode to highlight a handful of things. The complete source code can be found in the appendix.

4.2 Vertices, Edges, and Graphs

We need graphs in two instances, once to define the RAAG $A(\Gamma)$ via the graph Γ and another time to store the Cayley graph of $A(\Gamma)$. In a slightly unorthodox manner, I will define a sort of hybrid *Graph* class that caters to both needs. This doesn't affect performance.

While it is possible to implement graphs without the need for a *Vertex* class - using integers instead - we would like to store some information alongside their mere existence. Specifically, we store the position of the vertex, the word it represents, and its distance from the vertex of the neutral element[3], all of which is only used in the Cayley graph case. Note that, because we rarely need this information, it is often computationally advantageous to still refer to them by integers, namely by their index.

Next up is the *Edge* class, which is only slightly more complex. We need to store which vertices are being connected and (in the Cayley graph case) which generator connects them[4], all of which can be done with integers. Since we are dealing exclusively with undirected graphs, the order in which we store the vertices does not matter. Thus, we can make our programming lives easier by sorting them. As a result, fewer checks are required when, for example, determining if two edges are equal. The following code should clarify this:

```
function SORT(vertex_1, vertex_2)
    if vertex_1 > vertex_2 then
        return (vertex_2, vertex_1)
    else
        return (vertex_1, vertex_2)
    end if
end function
```

[3]The latter is only used for some rendering options and may be omitted if one has no need for them. See Section 4.5.

[4]This is only needed for rendering purposes.

```
function CompareSorted((v1, v2), (v3, v4))
    if v1 == v3 and v2 == v4 then
        return true
    else
        return false
    end if
end function

function CompareUnsorted((v1, v2), (v3, v4))
    if (v1 == v3 and v2 == v4) or (v1 == v4 and v2 == v3) then
        return true
    else
        return false
    end if
end function
```

Finally, we get to the *Graph* class. Vertices are stored in an ArrayList. Edges could be treated the same way, but since we may be dealing with thousands of edges in a graph, it is smarter to store them in an ArrayList of ArrayLists such that the index of an edge in the outer ArrayList matches the smaller of the two indices of the vertices it connects. This way, any specific edge can be found much quicker.

The constructor is only used for the graph defining the RAAG. It takes the number of vertices (as an integer) and an ArrayList of edges and formats them appropriately. We add a handful of *hasEdge*, *getEdge*, *getEdges*, *getVertices*, *addEdge*, and *addVertex* methods along with an *adjustPos* method, the functionality and purpose of which is described in Section 4.5.

4.3 Letters, Words, and Pilings

Probably the most straightforward way to deal with letters when programming is to view them as integers: $z_1, \ldots, z_n$ correspond to $1, \ldots, n$, and their (formal) inverses correspond to the (additive) inverses, namely $-1, \ldots, -n$. Words, thus, should be lists of integers with a few special functions. Concretely, the class *Word* extends the class *ArrayList⟨Integer⟩*. To this we add standard *multLeft* and *multRight* functions[5] as well as a *copy*, a *getLastLetter*, and an *invert* function.

[5]For convenience, we define these such that formal inverses already behave like actual inverses in that they cancel, meaning that, for example, the word $z_1 z_2^{-1} z_2$ is immediately changed to z_1. This isn't technically correct, but it never matters.

Having defined words, we can move on to defining pilings. As described in Section 3.4, a (rigorous) piling consists of n words with letters $\{\oplus, \ominus, \bigcirc\}$ or rather, to keep in line with the convention described above, with letters $\{1, -1, 2\}$. When implementing the algorithm to produce pilings of arbitrary words, we should not actually generate a piling for each letter (since this would require more memory space than necessary and would probably be slower) but merely describe what effect such a piling would have on the piling of the word. The code should clarify what I mean:

```
function PILING(word, graph)
    size ← number of vertices in graph
    piling ← array of type Word of size size

    for all letters l in word do
        a ← ABSOLUTE(l)
        v ← a-th word in piling (starting at 1)

        if the last letter of v has the opposite sign of l then
            Remove the last letter of v.
            for all indices k such that the l-th and k-th vertices in graph are not adjacent do
                Remove the last letter of the k-th word in piling.
            end for

        else
            Add a letter equal to the sign of l to the right of v.
            for all indices k such that the l-th and k-th vertices in graph are not adjacent do
                Add a “neutral” letter to the right of the k-th word in piling.
            end for
        end if
    end for

    return piling
end function
```

To solve the word problem, we need to know when a piling is trivial, i.e. equal to the piling of the neutral element[6]:

```
function isTrivial(piling)
    for all words w in piling do
        if w is not empty then
            return false
        end if
    end for

    return true
end function
```

4.4 The *Algorithm* class

Next we define an *Algorithm* class, which is responsible for generating the Cayley graph. I chose to make it an abstract class, the only abstract method being one that determines when a word is trivial, which allows us to use the whole program for other types of groups as well without much additional work. As mentioned before, this approach is more general than needed, but it isn't much more complex than a more direct one. Also, for better user-experience, I chose to have the class be a thread.

The *Algorithm* class stores three things: The output graph, the number of generators, and an object used to determine the triviality of a word[7]. The main algorithm now works as follows: First we add a vertex to the graph g representing the neutral element. Then we start iterating over the vertices of g. For each vertex v, we sequentially concatenate all generators $1, \ldots, n$ and their inverses to the word associated with v and test whether the result is already represented by another vertex (this is where the abstract method *isTrivial* is used). If it is, we store the index of that vertex in the variable *connected* and add an edge from v to that vertex. If it is not, we add a vertex for it and connect it to v.

[6] It hasn't been stated explicitly, but clearly the the piling of the neutral element, or rather of the empty word, is the piling where every string/word is empty.

[7] In the case of the right-angled Artin group $A(\Gamma)$, this object is the graph Γ.

```
Let g be the output graph

function generateCayleyGraph()
    Add the vertex of the neutral element to g

    for each vertex v in g do
        word_v ← the word represented by v

        for each z in {1, −1, . . . , n, −n} do
            next ← word concatenated with z

            index ← −1
            for each vertex u in g do
                word_u ← the word represented by u

                if word_u · next^{−1} is trivial then
                    index ← the index of u
                end if
            end for

            if index == −1 then
                Add a vertex representing next to g and connect it to v
            else
                Connect v to the vertex at index
            end if
        end for
    end for
end function
```

To have this process not run indefinitely, we add a variable *MAX_RADIUS* and stop once a vertex reaches this distance from the initial vertex. Note that the vertices are naturally sorted by their distance from that vertex, so after one crosses the threshold, all subsequent ones follow suit.

The specifics of how to use the program for right-angled Artin groups as well as other types of groups can be found in the source code, specifically in lines 89 - 122 of the *setup* method.

4.5 Visuals

I will not go into much detail about the visual implementation since most of it is fairly straight-forward and mathematically not very interesting. However, there are a few design choices to be highlighted.

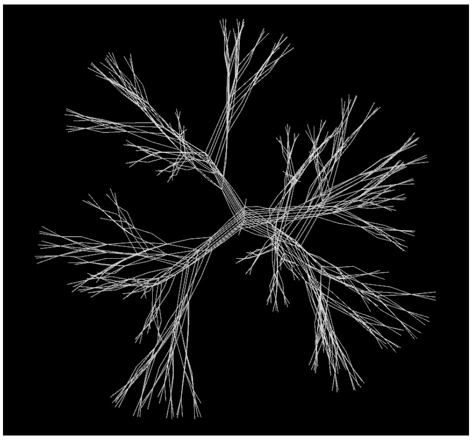
First and foremost, how do you get the Cayley graph into an ordered state? The reader might have noticed in the source code of the *Algorithm* class that vertices are spawned with random coordinates between −*SPAWN_SIZE* and *SPAWN_SIZE*. To bring order to this initial mess, I apply an intuitive, physical concept: adjacent vertices attract each other, non-adjacent ones repel one another. A function which handles this is run about a hundred times per second, making the vertices move slowly to their desired place[8]. Clearly, this process is far from deterministic and results may vary, though not too dramatically, as it turns out. Alternatively, one could try to write an algorithm which places the vertices in a fixed spot immediately, but I've found that approach to be less illuminating and certainly less flexible.

Much more illuminating – despite its name – is another feature that I added: "Shadowing", as I've called it, is the ability to make edges which stem from certain generators less attractive, thus splitting the Cayley graph into smaller, less strongly connected, subgraphs. This helps visually understand more complex Cayley graphs because it separates the effects of the shadowed generators from those of the non-shadowed ones.

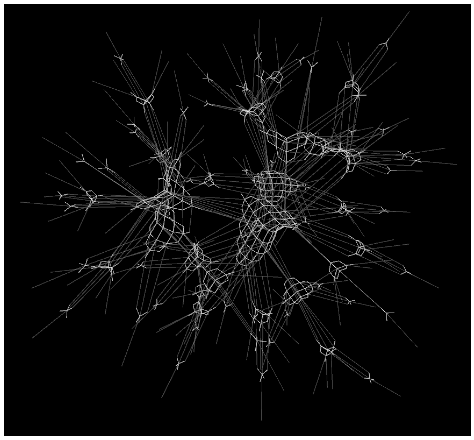
Furthermore, there is the option to only show the sphere of the given radius around the neutral element instead of the whole ball. In this case, all vertices with that exact distance from the origin are drawn as a small dot. For extra clarity, I chose to additionally draw the edges from these vertices to those one less step away.

Finally, I would like to show a few examples of what the program might produce. However, I urge the reader to try it out for themself because, in my opinion, most of the intricacy and three-dimensionality of some of these graphs gets lost in a still image. Figure 4.1 illustrates the usage of shadowing on the two groups $A(P_3)$ and $\mathbb{Z}_4$, while Figure 4.2 showcases the ball drawing feature.

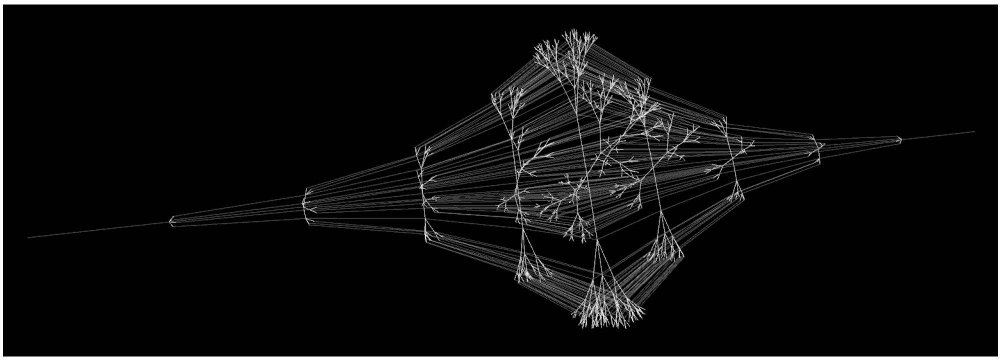
[8]See the source code: Method *adjustPos* in class *Graph*.



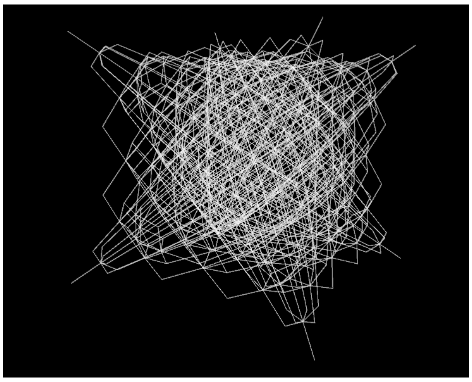
(a) $A(P_3)$



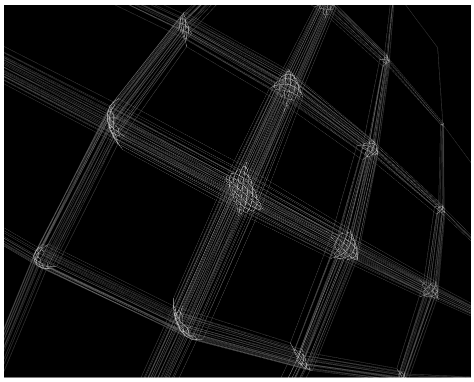
(b) $A(P_3)$ with z_1 (or z_3) shadowed



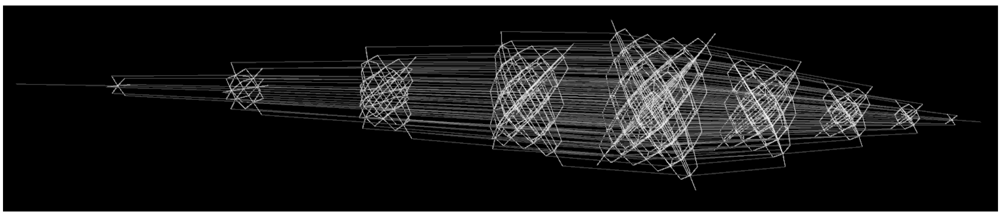
(c) $A(P_3)$ with z_2 shadowed; several copies of the free group F_2 in various sizes



(d) $\mathbb{Z}^4$; a convoluted mess

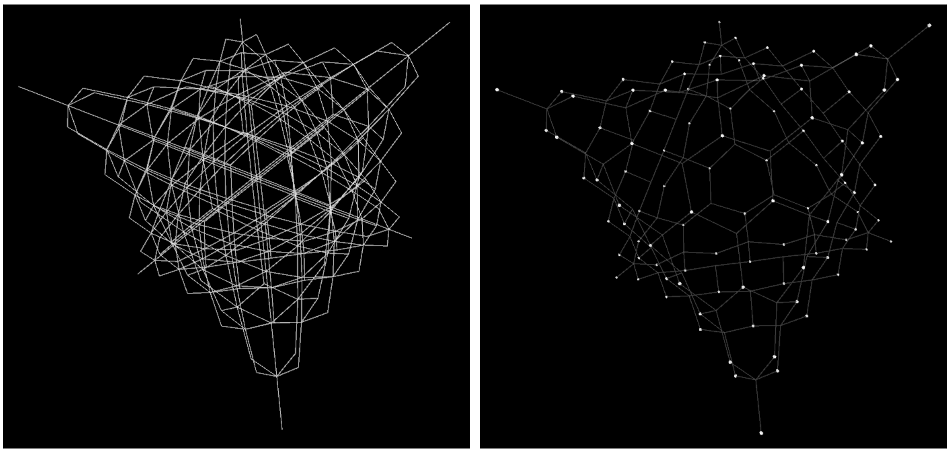


(e) $\mathbb{Z}^4$ with two generators shadowed

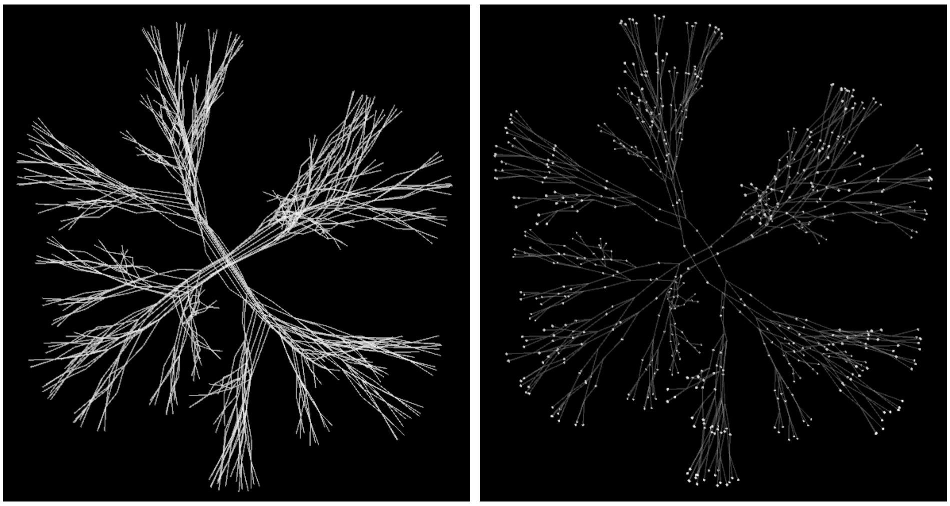


(f) $\mathbb{Z}^4$ with one generator shadowed; several copies of $\mathbb{Z}^3$ in various sizes

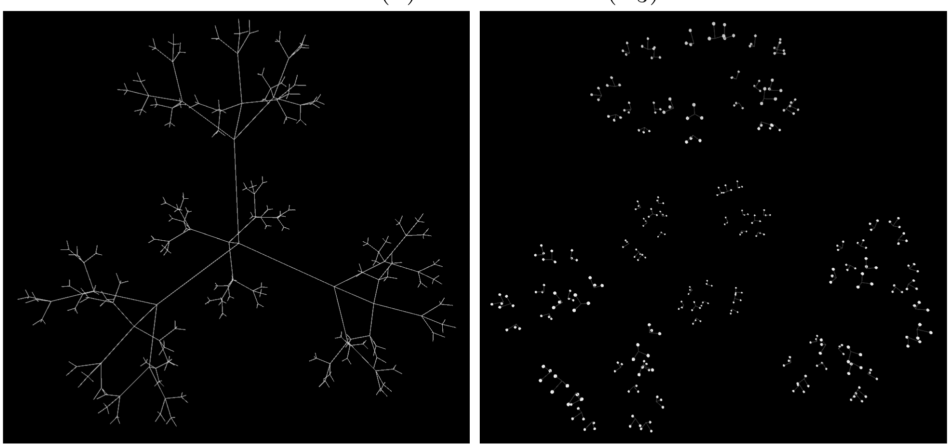
Figure 4.1: A few examples of Cayley graphs of right-angled Artin groups. Shadowed edges appear darker.



(a) The free abelian group $\mathbb{Z}_3$



(b) The RAAG $A(P_3)$



(c) The free group F_2

Figure 4.2: A few showcases of balls versus spheres.

Appendix A

Source Code

A.1 Global Variables

```
private Graph g;

private float zoom = -1000, r, s;
private PVector camPos;

// Non-final variables in all-caps can be modified with the
    settings file and may thus not be be declared "final",
    although they are to be treated as such
private boolean autoRotate = false;
private boolean showInterface = true;
private boolean showArrow = false;
private boolean holdingCtrl = false;
private boolean onlyDrawSphere = false;
private boolean stopThreads = false;
private boolean SAVE_WEIGHTS = false;

private final float SPAWN_SIZE = 1;
private final float SCALE = 200;
private int MAX_RADIUS = 5;
private final float PHYSICS_FRAMERATE_CAP = 100;
private float LAG_RELIEF = 0; // Probability that an
    unrendered vertex doesn't get computed

private int drawRadius; // Radius of the ball around the
    neutral element
private float physicsFramerate = 0;

// Initial slider values
private float R = .05;
private int orderAttract = 1;
private int orderRepel = -2;
```

```
private float repulsionRadius = 1;
private float shadowEffect = .01;

private Slider selectedSlider;
private ArrayList<Slider> sliders;

private IntList shadow;
```

A.2 The *setup* Method

```
void setup() {
  size(900, 600, P3D);
  surface.setResizable(true);
  frameRate(40);
  textSize(12);
  textAlign(LEFT, TOP);

  camPos = new PVector();

  String fileName = "Graph.json";
  try {
    // Load settings from "settings.json"
    JSONObject settings = loadJSONObject("settings.json");

    fileName = settings.getString("filename");
    MAX_RADIUS = settings.getInt("max_radius");
    LAG_RELIEF = settings.getFloat("lag_relief");
    SAVE_WEIGHTS = settings.getBoolean("save_weights");
  } catch(Exception ex) {
    // Save standard settings, defined above
    println("WARNING: Could not read \"settings.json\".");
    JSONObject settings = new JSONObject();

    settings.setString("filename", fileName);
    settings.setInt("max_radius", MAX_RADIUS);
    settings.setFloat("lag_relief", LAG_RELIEF);
    settings.setBoolean("save_weights", SAVE_WEIGHTS);

    saveJSONObject(settings, "settings.json");
  }

  drawRadius = MAX_RADIUS;

  int size;
  ArrayList<Edge> edges = new ArrayList<Edge>();

  try {
    // Load graph from file, "Graph.json" by default
```

```
    JSONObject jsonGraph = loadJSONObject(fileName);

    size = jsonGraph.getInt("#generators");

    JSONArray jsonEdges = jsonGraph.getJSONArray("edges");

    for(int i = 0; i < jsonEdges.size(); i++) {
      edges.add(new Edge(jsonEdges.getJSONObject(i).getInt("
   from") - 1, jsonEdges.getJSONObject(i).getInt("to") - 1));
    }
  } catch(NullPointerException ex) {
    // Resort to one vertex and no edges
    println("WARNING: Could not read \"" + fileName + "\".");
    size = 1;
  }

  // Algorithm for solving the word problem in RAAGs. Most of
    the work is done in the "Piling" class
  Algorithm raag = new Algorithm<Graph>(size, new Graph(size,
    edges)) {
    protected boolean isTrivial(Word w) {
      return new Piling(w, triv).isTrivial();
    }
  };

  g = raag.getGraph();

  // Sample algorithm for another class of groups: (Z/mZ)^n
   with input (n, m)
  // Remember to set the max. radius large enough (n*m/2) in
   the settings file.
  //Algorithm z_nm = new Algorithm<Integer>(2, 20) {
  //  protected boolean isTrivial(Word w) {
  //    int[] count = new int[nGens];

  //    for(int i : w) {
  //      if(i > 0) {
  //        count[i - 1]++;
  //      } else {
  //        count[-i - 1]--;
  //      }
  //    }

  //    for(int i : count) {
  //      if(i%triv != 0) {
  //        return false;
  //      }
  //    }

```

```
//     return true;
//   }
//};

//g = z_nm.getGraph();

// Slider initialization, nothing interesting
selectedSlider = null;
sliders = new ArrayList<Slider>();
sliders.add(new Slider(0, .5, false, "Step size") {
  protected void init() {
    setValue(R);
  }

  public void affect() {
    R = value;
  }
}
);
sliders.add(new Slider(0, 4, true, "Attraction exponent") {
  protected void init() {
    setValue(orderAttract);
  }

  public void affect() {
    orderAttract = int(value + .01);
  }
}
);
sliders.add(new Slider(-4, 0, true, "Repulsion exponent") {
  protected void init() {
    setValue(orderRepel);
  }

  public void affect() {
    orderRepel = int(value + .01);
  }
}
);
sliders.add(new Slider(-5, 5, false, "log(Repulsion radius)") {
  protected void init() {
    setValue(log(repulsionRadius));
  }

  public void affect() {
    repulsionRadius = exp(value);
  }
}
```

```
  );
  sliders.add(new Slider(0, .05, false, "Shadow effect") {
    protected void init() {
      setValue(shadowEffect);
    }

    public void affect() {
      shadowEffect = value;
    }
  }
  );

  shadow = new IntList();

  // Start the physics
  new Thread() {
    public void run() {
      int m = millis();

      while(!stopThreads) {
        while(millis() - m < 1000/PHYSICS_FRAMERATE_CAP) delay(1);
        physicsFramerate = 1000./(millis() - m);
        m = millis();

        g.adjustPos();
      }
    }
  }.start();
}
```

A.3 The *draw* Method

```
void draw() {
  if(mousePressed) {
    if(selectedSlider == null) {
      // Rotate the graph by dragging it around
      r += .005 * (mouseX - pmouseX);
      s -= .005 * (mouseY - pmouseY);
      s = max(-PI/2, min(PI/2, s));
    } else {
      // Adjust a slider
      float xAdj = min(max(mouseX - selectedSlider.pos.x, 0), selectedSlider.w);
      float v = map(xAdj, 0, selectedSlider.w, selectedSlider.min, selectedSlider.max);

      selectedSlider.setValue(v);
```

```
    }
  }

  if (autoRotate) {
    r += .003;
  }

  background(#000000);

  pushMatrix();
  translate(width/2, height/2, zoom);
  rotateX(s);
  rotateY(r);

  translate(-camPos.x, -camPos.y, -camPos.z);

  if(showArrow) {
    fill(#BBBBBB);
    drawArrow(0, 750, 0, 60, 20, 120);
  }

  ArrayList<Edge> edges = g.getEdges();
  for(Edge e : edges) {
    Vertex from = g.vertices.get(e.from);
    Vertex to = g.vertices.get(e.to);

    if (onlyDrawSphere) {
      if(to.dist != drawRadius) continue;
    } else {
      if(to.dist > drawRadius) continue;
    }

    if(onlyDrawSphere) {
      stroke(#ffffff, 70);
    } else if(shadow.hasValue(e.generator)) {
      stroke(#ffffff, 80);
    } else {
      stroke(#ffffff, 200);
    }

    line(from.pos.x * SCALE, from.pos.y * SCALE, from.pos.z *
     SCALE, to.pos.x * SCALE, to.pos.y * SCALE, to.pos.z *
    SCALE);
  }

  if(onlyDrawSphere) {
    ArrayList<Vertex> vertices = g.getVertices();

    noStroke();
```

```
    fill(#ffffff, 200);
    for(Vertex v : vertices) {
      if(v.dist != drawRadius) continue;
      pushMatrix();
      translate(v.pos.x * SCALE, v.pos.y * SCALE, v.pos.z *
   SCALE);
      sphere(4);
      popMatrix();
    }
  }
  popMatrix();

  if(showInterface) {
    textAlign(LEFT, TOP);
    fill(#ffffff);
    text("Framerate render/physics: " + int(frameRate) + ", "
    + int(physicsFramerate), 0, 0);
    text("#vertices: " + g.vertices.size() + ", #edges: " +
   edges.size(), 0, 15);
    text("Radius: " + drawRadius, 0, 30);
    text("Avg Speed: " + (int(g.avgSpeed * 100) / 100.), 0,
   45);

    if(shadow.size() > 0) {
      String str = shadow.get(0) + "";

      for(int i = 1; i < shadow.size(); i++) {
        str += ", " + shadow.get(i);
      }

      text("Shadowed: " + str, 0, 60);
    }


    for(int i = 0; i < sliders.size(); i++) {
      sliders.get(i).draw(width - sliders.get(i).w - 20, 10 +
    30*i);
    }
  }
}
```

A.4 Event Methods

```
void mousePressed() {
  if(mouseButton == LEFT) {
    for(Slider s : sliders) {
      if((s.pos.x <= mouseX && mouseX <= s.pos.x + s.w) && (s
   .pos.y <= mouseY && mouseY <= s.pos.y + s.h)) {
```

```
        selectedSlider = s;
        break;
      }
    }
  }
}

void mouseReleased() {
  selectedSlider = null;
}

void mouseWheel(MouseEvent evt) {
  if(holdingCtrl) {
    drawRadius = max(0, min(drawRadius - evt.getCount(),
   MAX_RADIUS));
  } else {
    zoom -= 50 * evt.getCount();
  }
}

void keyPressed() {
  if(key == 'r' || key == 'R') {
    g.resetPosition();
  } else if(key == 'a' || key == 'A') {
    autoRotate = !autoRotate;
  } else if(key == 's' || key == 's') {
    showArrow = !showArrow;
  } else if(key == 'd' || key == 'D') {
    onlyDrawSphere = !onlyDrawSphere;
  } else if(key == 'p' || key == 'P') {
    PrintWriter out = createWriter("edges.txt");
    out.print(g.toString());
    out.flush();
    out.close();
  } else if(key >= 49 && key <= 57) {
    // Number keys 1 - 9
    int n = key - 48;

    if (shadow.hasValue(n)) {
      shadow.removeValue(n);
    } else {
      shadow.appendUnique(n);
      shadow.sort();
    }
  } else if(keyCode == 17) {
    // Control key
    holdingCtrl = true;
  } else if(keyCode == 97) {
    // F1 key
```

```
    showInterface = !showInterface;
  }
}

void keyReleased() {
  if(keyCode == 17) {
    holdingCtrl = false;
  }
}
```

A.5 Additional Methods

```
void drawArrow(float x, float y, float z, float w, float h,
   float l) {
  beginShape();
  vertex(x - w/2, y, z);
  vertex(x + w/2, y, z);
  vertex(x, y, z + l);
  endShape(CLOSE);
  beginShape();
  vertex(x - w/2, y + h, z);
  vertex(x + w/2, y + h, z);
  vertex(x, y + h, z + l);
  endShape(CLOSE);
  beginShape();
  vertex(x - w/2, y, z);
  vertex(x + w/2, y, z);
  vertex(x + w/2, y + h, z);
  vertex(x - w/2, y + h, z);
  endShape(CLOSE);
  beginShape();
  vertex(x - w/2, y, z);
  vertex(x, y, z + l);
  vertex(x, y + h, z + l);
  vertex(x - w/2, y + h, z);
  endShape(CLOSE);
  beginShape();
  vertex(x + w/2, y, z);
  vertex(x, y, z + l);
  vertex(x, y + h, z + l);
  vertex(x + w/2, y + h, z);
  endShape(CLOSE);
}

void exit() {
  stopThreads = true;

  super.exit();
```

```java
}

private int sgn(int n) {
  if(n > 0) return 1;
  else if(n < 0) return -1;

  return 0;
}
```

A.6 The *Word* Class

```java
private class Word extends ArrayList<Integer> {
  public Word multLeft(int i) {
    if(i == 0) throw new ArithmeticException("generator count
    starts at 1, instead 0 given");

    Word out = this.copy();

    if(out.size() > 0 && out.get(0) == -i) {
      out.remove(0);
    } else {
      out.add(0, i);
    }

    return out;
  }

  public Word multLeft(Word w) {
    Word out = this.copy();
    for(int i = w.size() - 1; i >= 0; i--) {
      out = out.multLeft(w.get(i));
    }

    return out;
  }

  public Word multRight(int i) {
    if(i == 0) throw new ArithmeticException("generator count
    starts at 1, instead 0 given");

    Word out = this.copy();

    int s = out.size();
    if(s > 0 && out.getLastLetter() == -i) {
      out.remove(s - 1);
    } else {
      out.add(s, i);
    }
```

```

    return out;
  }

  public Word multRight(Word w) {
    Word out = this.copy();
    for(int v : w) {
      out = out.multRight(v);
    }

    return out;
  }

  public Word copy() {
    return (Word)super.clone();
  }

  public int getLastLetter() {
    if(this.size() == 0) return 0;
    return this.get(this.size() - 1);
  }

  public Word invert() {
    Word out = new Word();

    for(int v : this) {
      out = out.multLeft(-v);
    }

    return out;
  }

  public String toString() {
    if(this.size() == 0) return "e";

    String out = "";

    for(int i : this) {
      if(i > 0) {
        out += (char)(i + 96);
      } else {
        out += "(" + (char)(-i + 96) + "^-1)";
      }
    }

    return out;
  }
}
```

A.7 The *Piling* Class

```
private class Piling {
  public final Word[] p;

  public Piling(Word w, Graph g) {
    int size = g.vertices.size();
    p = new Word[size];

    for(int i = 0; i < size; i++) {
      p[i] = new Word();
    }

    try {
      for(int i = 0; i < w.size(); i++) {
        int z = w.get(i);
        int a = abs(z);

        if(a <= 0 || a > size) {
          throw new ArithmeticException();
        }

        boolean cancel = (p[a - 1].size() > 0 && p[a - 1].getLastLetter() == -sgn(z));

        for(int k = 0; k < size; k++) {
          if(k == a - 1) {

            p[k] = p[k].multRight(sgn(z));

          } else if(!g.hasEdge(k, a - 1)) {

            p[k] = p[k].multRight(2 * (cancel ? -1 : 1));

          }
        }
      }
    } catch(ArithmeticException ex) {
      println("WARNING: invalid word given");
    }
  }

  public boolean isTrivial() {
    for(int i = 0; i < p.length; i++) {
      if(p[i].size() > 0) {
        return false;
      }
    }

```

```
    return true;
  }
}
```

A.8 The *Vertex* Class

```
private class Vertex {
  public PVector pos;      // position coordinates (for
   rendering)
  public final Word word;  // the word this vertex represents
    ("null" if not applicable)
  public final int dist;   // distance to the neutral element
    ("-1" if not applicable)

  public Vertex(Word w, int d) {
    this(0, 0, 0, w, d);
  }

  public Vertex(float x, float y, float z, Word w, int d) {
    pos = new PVector(x, y, z);
    word = w;
    dist = d;
  }
}
```

A.9 The *Edge* Class

```
private static class Edge {
  public final int from, to, generator;

  public Edge(int cFrom, int cTo) {
    this(cFrom, cTo, 0);
  }

  public Edge(int cFrom, int cTo, int cGenerator) {
    // Mathematically, the order of "from" and "to" does not
   matter, so from a code standpoint, it is better to always
   store them so that "to" is larger than or equal to "from".
    For instance, the method below is a bit shorter because
   of this.
    if(cFrom <= cTo) {
      from = cFrom;
      to = cTo;
    } else {
      from = cTo;
      to = cFrom;
    }

```

```
    generator = cGenerator;
  }

  // This method helps keep some code in the "Graph" class a bit cleaner.
  public static boolean isValidEdge(int mFrom, int mTo, int upperLimit) {
    return 0 <= mFrom && mFrom <= mTo && mTo < upperLimit;
  }
}
```

A.10 The *Graph* Class

```
private class Graph {
  public final ArrayList<ArrayList<Edge>> edges;
  public final ArrayList<Vertex> vertices;
  public float avgSpeed = 0; // Information for rendering

  public Graph(int cSize, ArrayList<Edge> cEdges) {
    vertices = new ArrayList<Vertex>();

    edges = new ArrayList<ArrayList<Edge>>();

    for(int i = 0; i < cSize; i++) {
      vertices.add(new Vertex(null, -1));

      edges.add(new ArrayList<Edge>());
    }

    if(cEdges != null) {
      for(Edge e : cEdges) {
        addEdge(e);
      }
    }
  }

  public boolean hasEdge(Edge e) {
    return hasEdge(e.from, e.to);
  }

  public boolean hasEdge(int v, int w) {
    return getEdge(v, w) != null;
  }

  public Edge getEdge(int v, int w) {
    if(v > w) return getEdge(w, v);
    if(!Edge.isValidEdge(v, w, edges.size())) return null;

```

```
    ArrayList<Edge> copy = (ArrayList<Edge>)edges.get(v).
   clone();
    for(Edge e : copy) {
      try {
        if(w == e.to) return e;
      } catch(NullPointerException ex) {
        // Because the Cayley graph is generated in a thread,
    it sometimes happens that
        // "e.to" throws an exception here. This is nothing
   to worry about.
      }
    }

    return null;
  }

  public ArrayList<Vertex> getVertices() {
    return (ArrayList<Vertex>)vertices.clone();
  }

  public ArrayList<Edge> getEdges() {
    ArrayList<Edge> out = new ArrayList<Edge>();

    for(int i = 0; i < edges.size(); i++) {
      ArrayList<Edge> al = edges.get(i);

      for(int k = 0; k < al.size(); k++) {
        out.add(al.get(k));
      }
    }

    return out;
  }

  public void addVertex(Vertex v) {
    vertices.add(v);

    edges.add(new ArrayList<Edge>());
  }

  public void addEdge(int v, int w, int label) {
    addEdge(new Edge(v, w, label));
  }

  public void addEdge(Edge e) {
    if(this.hasEdge(e)) return;

    if(!Edge.isValidEdge(e.from, e.to, edges.size())) {
      println("WARNING: invalid edge given");
```

```java

      return;
    }

    edges.get(e.from).add(e);
  }

  public void resetPosition() {
    for(Vertex v : vertices) {
      v.pos = new PVector(random(-SPAWN_SIZE, SPAWN_SIZE),
                          random(-SPAWN_SIZE, SPAWN_SIZE),
                          random(-SPAWN_SIZE, SPAWN_SIZE));
    }
  }

  public void adjustPos() {
    if(R < EPSILON) return;

    int size = vertices.size();
    PVector[] velocity = new PVector[size];

    for(int i = 0; i < size; i++) {
      velocity[i] = new PVector(0, 0, 0);

      if(vertices.get(i).dist > drawRadius && random(1) <
    LAG_RELIEF) continue;

      for(int k = 0; k < i; k++) {
        if(vertices.get(k).dist > drawRadius && random(1) <
    LAG_RELIEF) continue;

        Vertex v1 = vertices.get(i);
        Vertex v2 = vertices.get(k);

        PVector f = PVector.sub(v2.pos, v1.pos);
        float m = f.mag();

        float r;
        Edge e = getEdge(k, i);
        if(e != null) {
          r = -pow(m, orderAttract);

          if(shadow.hasValue(e.generator)) r *= shadowEffect;
        } else if(m >= repulsionRadius) {
          continue;
        } else {
          r = 0.002 * pow(m, orderRepel);
        }

```

```
        f.setMag(min(r*R, .6));

        velocity[k] = PVector.add(velocity[k], f);
        velocity[i] = PVector.sub(velocity[i], f);
      }
    }

    PVector newCamPos = new PVector(0, 0, 0);
    float newAvgSpeed = 0;

    for(int i = 0; i < size; i++) {
      vertices.get(i).pos.add(velocity[i]);

      newCamPos.add(vertices.get(i).pos);
      newAvgSpeed += velocity[i].mag();
    }

    newCamPos.mult(SCALE/size);
    newAvgSpeed *= SCALE/size;

    camPos = newCamPos;
    avgSpeed = newAvgSpeed;
  }

  public String toString() {
    String out = "";

    for(Edge e : this.getEdges()) {
      out += e.from + " " + e.to + (SAVE_WEIGHTS ? " " +
    PVector.sub(vertices.get(e.from).pos, vertices.get(e.to).
    pos).mag() : "") + "\n";
    }

    out = out.substring(0, out.length() - 1);

    return out;
  }
}
```

A.11 The *Algorithm* Class

```
private abstract class Algorithm<T> extends Thread {
  private final Graph g;
  protected final int nGens;
  protected final T triv;

  public Algorithm(int n, T cTriv) {
    g = new Graph(0, null);
```

```
    nGens = n;
    triv = cTriv;

    this.start();
  }

  public Graph getGraph() {
    return g;
  }

  public void run() {
    g.addVertex(new Vertex(new Word(), 0));

    for(int i = 0; i < g.vertices.size(); i++) {
      Vertex v = g.vertices.get(i);

      if(v.dist >= MAX_RADIUS) break;

      for(int z = -nGens; z <= nGens; z++) {
        if(z == 0) continue;

        Word next = v.word.multRight(z);

        int connected = -1;
        for(int k = 0; k < g.vertices.size(); k++) {
          if(this.isTrivial(next.multRight(g.vertices.get(k).
 word.invert()))) {
            connected = k;
            break;
          }
        }

        if(connected == -1) {
          // New element
          g.addVertex(new Vertex(random(-SPAWN_SIZE,
 SPAWN_SIZE),
                                  random(-SPAWN_SIZE,
 SPAWN_SIZE),
                                  random(-SPAWN_SIZE,
 SPAWN_SIZE), next, v.dist + 1));

          g.addEdge(i, g.vertices.size() - 1, abs(z));
        } else {
          // New edge to old element
          g.addEdge(i, connected, abs(z));
        }
      }
    }
  }
```

```

  protected abstract boolean isTrivial(Word w);
}
```

A.12 The *Slider* Class

```
private abstract class Slider {
  public PVector pos;
  public final float w, h;
  public final float min, max;
  public final boolean discrete;
  public final String name;
  public float value;

  public Slider(float cMin, float cMax, boolean cDiscrete, String cName) {
    w = 200;
    h = 10;

    min = cMin;
    max = cMax;
    discrete = cDiscrete;
    name = cName;

    value = 0;
    init();
  }

  public void draw(float x, float y) {
    pos = new PVector(x, y);

    fill(#A5A5A5);
    stroke(#ffffff);
    rect(x, y, w, h);
    fill(#E0E0E0);
    ellipse(x + map(value, min, max, 0, w), y + h/2, 1.5 * h, 1.5 * h);

    textAlign(RIGHT, TOP);
    text(name + ": " + int(value * 100) * 1./100, x - 8, y - 3);
  }

  public void setValue(float v) {
    if(discrete) {
      value = floor(v + .5);
    } else {
      value = v;
```

```
    }
    affect();
  }

  protected abstract void init();

  public abstract void affect();
}
```

Bibliography

[1] J. Crisp, E. Godelle, and B. Wiest, "The conjugacy problem in subgroups of right-angled artin groups," *Journal of Topology*, vol. 2, no. 3, pp. 442–460, 2009.

[2] M. Clay and D. Margalit, *Office Hours with a Geometric Group Theorist.* Kassel: Princeton University Press, 2017.

[3] C. Löh, *Geometric Group Theory - An Introduction.* Berlin, Heidelberg: Springer, 2017.

[4] F. Harary, *Graph Theory.* New York: Avalon Publishing, 1969.

[5] `https://processing.org`.