Algorithmic Computation of Thickness in Right-Angled Coxeter Groups

Robbie Lyman

April 2, 2015

Abstract

The classification of right-angled Coxeter groups up to quasi-isometry has been a subject of recent inquiry, mostly via two related quasi-isometry invariants called thickness and divergence. A paper by Behrstock, Hagen and Sisto gave an algorithm to determine whether a right-angled Coxeter group was thick or not. A paper by Dani and Thomas gave explicit characterizations of thickness of orders 0 and 1 for triangle-free graphs and exhibited a family of groups \( \{W_k : k \in \mathbb{N}\} \) such that \( W_k \) has divergence of order \( r_k \) (and is thus thick of order \( k - 1 \)). We give a brief overview of Coxeter groups, review the previous results, and then give two new algorithms that not only determine whether a right-angled Coxeter group is thick or not but give upper bounds on the order of thickness. We then discuss applications of these algorithms to random graphs.

1 Coxeter Groups, Basic Notions

While many of the concepts introduced apply to the study of groups more generally, our main focus will be on Coxeter groups, a particular class of groups generated by reflections. (For more on Coxeter groups more generally, the interested reader should consult Davis’s book, [Dav07].) Consider a set \( S \), and to each pair \((s, t) \in S \times S\) associate a number \( m_{st} \in \mathbb{N} \cup \{\infty\} \) with the conditions that

1. \( m_{st} = 1 \iff s = t \), and
2. \( m_{st} = m_{ts} \).

Definition. A Coxeter group \( W \) with generating set \( S \) is a group defined by the following presentation:

\[
W = \langle s \in S \mid (st)^{m_{st}} \rangle
\]

with the convention that there is no relation between \( s \) and \( t \) if \( m_{st} = \infty \).
Thus every generator $s \in S$ has order 2, and if $s$ and $t$ are such that $m_{st} = 2$, then
\[ st = st(tsts) = s(t)sts = (ss)ts = ts, \]
i.e. $s$ and $t$ commute. We can encode the information in this presentation for $W$ in a graph $\Gamma$ as follows: let each $s \in S$ correspond to a vertex in $\Gamma$ labelled $s$. For each pair $(s,t)$ with $s \neq t$, let there be an unlabelled edge connecting $s$ and $t$ if $m_{st} = 2$, an edge labelled $m_{st}$ if $2 < m_{st} < \infty$, and no edge if $m_{st} = \infty$. Figure 1 gives an example of this process for $S_4$, the symmetric group on four elements.

**Definition.** We say that a Coxeter group $W$ is **right-angled** if for every pair $(s,t)$ of distinct generators in $S$, $m_{st}$ is either two or zero, i.e. every defining relation between generators is a commutator.

We will restrict ourselves to Coxeter groups that are right-angled and have finite generating sets. In view of the above discussion, given a finite simplicial graph $\Gamma$, we can construct a right-angled Coxeter group with generators the vertices of $\Gamma$ and relations the conditions that every generator has order 2 and two generators commute just when there is an edge between the corresponding vertices in $\Gamma$. We’ll denote this Coxeter group $W_\Gamma$.

Given a finite simplicial graph $\Gamma$ and its corresponding right-angled Coxeter group, $W_\Gamma$, if we choose a subgraph $\Lambda$ of $\Gamma$ by choosing a subset $V$ of the vertices of $\Gamma$ together with all the edges of $\Gamma$ that span vertices in $V$, $W_\Lambda$, the right-angled Coxeter group generated by $\Lambda$, will be a subgroup of $W_\Gamma$. We call such a subgroup $W_\Lambda$ a **special subgroup** of $W_\Gamma$, and such a subgraph $\Lambda$ a **full subgraph** of $\Gamma$, or the **induced subgraph** corresponding to the subset $V$ of the vertices of $\Gamma$.

To study the geometry of Coxeter groups, it will be useful to introduce geometric objects that the groups act on. The first of these is the Cayley graph. Further discussion of concepts introduced here can be found in [Hat02] and [Mei08].

**Definition.** Given a group $G$ and a generating set $S$, the **Cayley graph of $G$ with respect to $S$** is the graph with vertices the elements of $G$ and edges such that $g$ is connected to $h$ by an edge whenever there exists $s \in S$ such that $g = h \cdot s$. Between any two vertices there will be at most one edge, and no edges will have the same vertex as both endpoints.

The Cayley graph for $F_2$, the free group on two generators $a$ and $b$ with respect to the generating set $\{a, b\}$, for example, is a tree with four edges incident at each vertex, one each for $a, a^{-1}, b$ and $b^{-1}$, and the Cayley graph for $\mathbb{Z} \times \mathbb{Z}$ with respect to the generating set $\{(1, 0), (0, 1)\}$ is the 2-dimensional integer lattice with edges connecting $(m, n)$ to $(m + 1, n)$ and $(m, n + 1)$ for all integers $m$ and

\[
\begin{array}{c}
(1,2) \\
3 \\
(2,3) \\
3 \\
(1,4) \\
3 \\
(3,4) \\
3 \\
\end{array}
\]
If $C_G$ is the Cayley graph of $G$, then $G$ acts on $C_G$ by left-multiplication of vertex labels.

We’d also like to consider groups as metric spaces, for which we introduce the following notion of a word metric, which is closely related to the Cayley graph.

**Definition.** Given a finitely-generated group $G$ and a generating set $S$, the **word metric** on $G$ with respect to $S$ is the function $d_{G,S} : G \times G \rightarrow \mathbb{R}$, with

$$d_{G,S}(x, y) = \|x^{-1}y\|$$

where $\|x^{-1}y\|$ is the number of letters in $S$ needed to represent $x^{-1}y$ as a reduced word.

It is simple to check that $d_{G,S}$ is indeed a metric on $G$. In fact, $d_{G,S}(x, y)$ corresponds to the distance between $x$ and $y$ in the Cayley graph of $G$ with respect to $S$ if every edge is isometric to the unit interval. In the following discussion, we may suppress the subscript and write $d_G$ or $d$ if doing so will not cause confusion.

We might worry that because this metric depends on a choice of generating set, it might be possible to choose generating sets $S$ and $S'$ such that $d_{G,S}$ and $d_{G,S'}$ give very different values for the same input, and this is true. However, these word metrics are equivalent in a certain sense that we will now describe.

**Definition.** Given two metric spaces $(X, d_X)$ and $(Y, d_Y)$, a map $f : X \rightarrow Y$ is a **quasi-isometric embedding** if there exist constants $K \geq 1$ and $C \geq 0$ such that for all $x_1, x_2 \in X$,

$$\frac{1}{K} d_X(x_1, x_2) - C \leq d_Y(f(x_1), f(x_2)) \leq K d_X(x_1, x_2) + C$$

If in addition for all $y \in Y$, there exists $x \in X$ such that $d_Y(y, f(x)) < C$, then we say that $f$ is a **quasi-isometry**. Finally, given a quasi-isometry $f : X \rightarrow Y$, we say a quasi-isometry $g : Y \rightarrow X$ is a **quasi-inverse** of $f$ if

$$\sup_{x \in X} d_X(g \circ f(x), x) < \infty \quad \text{and} \quad \sup_{y \in Y} d_Y(f \circ g(y), y) < \infty$$

The main results that will be important to us are the following:

**Proposition 1.1.** Given a quasi-isometry $f : X \rightarrow Y$, there exists a quasi-isometry $g : Y \rightarrow X$ that is a quasi-inverse for $f$.

The above together with this proposition shows that quasi-isometry is an equivalence relation of metric spaces.

**Proposition 1.2.** If $S$ and $S'$ are two finite generating sets for a group $G$, then $(G, d_{G,S})$ is quasi-isometric to $(G, d_{G,S'})$. 


Thus classification up to quasi-isometry can be a useful tool in studying and classifying finitely-generated groups. Of course, if $G$ is a finite group, then $d_G(e, \cdot)$, the distance in $G$ from the identity to any other group element is a bounded function, so $G$ will be quasi-isometric to a single point. Therefore we will consider only infinite groups.

Finally, all the metric spaces we will consider have the property that between any two points $x, y$ in $(X,d)$, there is a path from $x$ to $y$ (i.e. a continuous function $f : [0,1] \to X$ with $f(0) = x, f(1) = y$) whose length, $\ell(f)$, satisfies $\ell(f) = d(x,y)$. Such a space is a geodesic metric space and such a path is a minimising geodesic. If $C_G$ is the Cayley graph of a group $G$, then an appropriate path that traces out the shortest path from $e$ to $g$ (by following the edges labelled with the generators in the reduced word for $g$) is a minimising geodesic.

2 Thickness, Divergence

Since we are interested in results on right-angled Coxeter groups, most of what follows, particularly in the definitions, will require some reformulation or extra assumptions to be valid for more general metric spaces or other classes of groups. Readers interested in a fuller exposition should consult [BHS13] and [DT15].

In order to motivate the definition of thickness, we begin by introducing a related notion of divergence. Roughly speaking, the divergence of a geodesic metric space measures how quickly the circumference of a metric ball grows as the radius of the ball does.

**Definition.** Let $(X,d)$ be a right-angled Coxeter group with a word metric, or the Cayley graph of a right-angled Coxeter group with each edge isometric to $[0,1]$. Fixing a basepoint $p \in X$, and letting $B(p,r)$ and $S(p, r)$ denote the open ball and sphere of radius $r$ at $p$, the $(p, r)$-avoidant distance between $x$ and $y$, $d_{p,r}^{av}(x,y)$ is defined for $x, y \in X - B(r,p)$

$$d_{p,r}^{av}(x,y) = \inf \{ \ell(f) : f \text{ is a path in } X - B(r,p) \text{ from } x \text{ to } y \}$$

The divergence of $X$, $\operatorname{div}_X : \mathbb{R} \to \mathbb{R}$ is defined as

$$\operatorname{div}_X(r) = \sup_{x, y \in S(r,p)} d_{p,r}^{av}(x,y)$$

If we define a partial order on functions $f : \mathbb{R} \to \mathbb{R}$ as

$$f \preceq g \text{ if } \exists C > 0 \text{ such that } f(r) \leq Cg(Cr + C) + Cr + C$$

and an equivalence relation $\simeq$ with $f \simeq g \iff f \preceq g$ and $g \preceq f$, then divergence becomes a quasi-isometry invariant, up to $\simeq$. We say $X$ has linear divergence if $\operatorname{div}_X \simeq r$, quadratic if $\operatorname{div}_X \simeq r^2$, and so forth. Since the circumference of a circle with radius $r$ in $\mathbb{R}^n$ is given by $2\pi r$, $\operatorname{div}_{\mathbb{R}^n}$ is linear. There exist spaces $X$ such that $r^k \simeq \operatorname{div}_X$ for all $k \in \mathbb{N}$; the Poincaré disk model of hyperbolic space is an example of one such space.
We're now ready to introduce the notion of thickness. Thickness, which is also a quasi-isometry invariant, guarantees that metric balls grow at most polynomially. Thus the Poincaré disk is not thick. The definition of thickness is inductive:

**Definition.** A finitely generated right-angled Coxeter group $W$ is **thick of order 0** if it can be written as a direct product, $W = G_1 \times G_2$ of infinite groups. For $n \geq 1$, $W$ is **thick of order at most** $n$ if there exists a finite collection $\mathcal{H}$ of subgroups such that

1. Each $H \in \mathcal{H}$ is quasi-isometrically embedded in $W$
2. $\langle \bigcup_{H \in \mathcal{H}} H \rangle$, the subgroup generated by the union of all the $H \in \mathcal{H}$ has finite index in $H$.
3. For all pairs $H, H' \in \mathcal{H}$, there is a finite sequence $H = H_1, \ldots, H_k = H'$ with each $H_i \in H$ such that $H_i \cap H_{i+1}$ is infinite for $1 \leq i < k$.
4. Each $H \in \mathcal{H}$ is thick of order at most $n - 1$.

$W$ is **thick of order** $n$ if $W$ is thick of order at most $n$ but is not thick of order at most $n - 1$.

Since the divergence characterises the upper limit of growth of geodesics in a space, we might want to interpret this definition with paths in mind: roughly any two elements of $\langle \bigcup_{H \in \mathcal{H}} H \rangle$ can be connected by a path that travels within subgroups that are all thick of order at most $n - 1$, so we might hope that an increment in the order of thickness will only increase the order of the divergence by one. Indeed, the following holds:

**Proposition 2.1.** (Corollary 4.17 of [BD]) If $W$ is thick of order $n$, then $\text{div}_W \preceq r^{n+1}$.

### 3 Combinatorial Approaches

Given our correspondence between finitely generated right-angled Coxeter groups and finite simplicial graphs, it is natural to ask how much information about the thickness of a group $W_{\Gamma}$ can be determined by examining its graph, $\Gamma$. This will require being able to make appropriate choices of subgroups based only on $\Gamma$—in fact, the special subgroups will do.

**Lemma 3.1.** If $W_{\Gamma}$ is a right-angled Coxeter group and $\Lambda$ is a full subgraph of $\Gamma$, then the special subgroup $W_{\Lambda}$ is quasi-isometrically embedded in $W_{\Gamma}$.

The proof relies on ideas found in [Dav07], and is beyond the scope of this paper.

**Lemma 3.2.** Two special subgroups $W_{\Lambda_1}$ and $W_{\Lambda_2}$ of $W_{\Gamma}$ have infinite intersection when $\Lambda_1 \cap \Lambda_2$ is not complete.
Proof. Note that if the intersection $\Lambda_1 \cap \Lambda_2$ is not a complete graph, then there exist $v_1$ and $v_2$ vertices in $\Lambda_1 \cap \Lambda_2$ that are not joined by an edge. If $\Lambda_3$ is the subgraph induced by $\{v_1, v_2\}$, then this means $W_{\Lambda_3} \cong \mathbb{Z}/2\mathbb{Z} * \mathbb{Z}/2\mathbb{Z}$, which is infinite and a (special) subgroup of both $W_{\Lambda_1}$ and $W_{\Lambda_2}$.

We also have the following result for the base case, when a group is thick of order zero, for which a little terminology will be useful.

**Definition.** A graph $\Gamma$ with vertex set $V$ is a join of two full subgraphs $\Lambda_1$ and $\Lambda_2$ (written $\Gamma = \Lambda_1 \star \Lambda_2$) if every vertex of $\Gamma$ belongs to exactly one of the two subgraphs, every vertex $v \in \Lambda_1$ is connected to every vertex of $\Lambda_2$, and vice versa.

**Definition.** A graph is said to be complete if every pair of (distinct) vertices is connected by an edge.

**Proposition 3.3.** If $\Gamma$ has no triangles, $W_\Gamma$ is thick of order zero $\iff$ $\Gamma$ is the join of two graphs that are not complete.

**Proof.** We'll show the only if direction first. Suppose $\Gamma = \Lambda_1 \star \Lambda_2$, where $\Lambda_1$ and $\Lambda_2$ are not complete. We want to show that $W_\Gamma$ can be written as a direct product of infinite groups. Note that $W_{\Lambda_1}$ and $W_{\Lambda_2}$ are infinite: because they are not complete as graphs, they contain at least one pair of generators that do not commute, so each admits an injective homomorphism from $\mathbb{Z}/2\mathbb{Z} * \mathbb{Z}/2\mathbb{Z}$, the free product of two copies of $\mathbb{Z}/2\mathbb{Z}$. As for the direct product, because every vertex of $\Lambda_1$ is connected to every vertex of $\Lambda_2$, each generator of $W_{\Lambda_1}$ commutes with every generator of $W_{\Lambda_2}$ and vice versa. But this is exactly what is required for us to write $W_\Gamma = W_{\Lambda_1} \times W_{\Lambda_2}$.

Now suppose $W_\Gamma = H_1 \times H_2$, where $H_1$ and $H_2$ are infinite subgroups of $W_\Gamma$. Thus every generator of $W_\Gamma$ is contained in either $H_1$ or $H_2$, and each subgroup must contain at least one pair that does not commute (otherwise the subgroup would be finite). Let $\Lambda_1$ be the subgraph induced from the generators contained in $H_1$, $\Lambda_2$ from $H_2$. Because $W_\Gamma = H_1 \times H_2$, each vertex in $\Lambda_1$ is connected to every vertex in $\Lambda_2$ and vice versa. No vertex can be in both $\Lambda_1$ and $\Lambda_2$, otherwise $\Gamma$ must contain a triangle. Thus $\Gamma = \Lambda_1 \star \Lambda_2$.

In the case where $\Gamma$ is not triangle free, we must allow the possibility that that $\Gamma = \Lambda_1 \star \Lambda_2 \star K$, where $K$ is a complete graph.

**Corollary 3.4.** $W_{K_{2,2}}$, where $K_{2,2}$ is the square graph, is thick of order zero.

Indeed, $K_{2,2}$ is the smallest graph whose right-angled Coxeter group is thick.

**Corollary 3.5.** $W_\Gamma$ is thick of order 1 $\iff$ $W_\Gamma$ is thick of order at most 1 and $\Gamma$ cannot be written as $\Lambda_1 \star \Lambda_2 \star K$, where $\Lambda_1$ and $\Lambda_2$ are not complete graphs and $K$ is a complete graph (possibly on zero vertices).
Since $K_{2,2}$ is the smallest graph whose right-angled Coxeter group is thick, and thickness is built up in stages, we might expect that the graph of a thick group can be built up out of squares somehow. There are two affirmative results, first for order 1 thickness from Dani and Thomas, and more generally for all right-angled Coxeter groups from Behrstock, Hagen and Sisto:

**Theorem 3.6. (Dani-Thomas)** If $\Gamma$ is triangle free, $W_\Gamma$ is thick of order 1 $\iff$ $\Gamma$ is not a join and is CFS. [DT15]

The condition CFS is on the square graph of $\Gamma$, denoted $\Gamma^4$, built from $\Gamma$ by taking as vertices all induced subgraphs of $\Gamma$ that are isomorphic to $K_{2,2}$, with an edge between two vertices if their corresponding squares share exactly three vertices in $\Gamma$. A graph $\Gamma$ is CFS if a connected component of $\Gamma^4$ has full support—that is, if every vertex in $\Gamma$ is contained in one of the squares corresponding to the vertices of this component. It is important to note that Dani and Thomas couch their arguments in the language of divergence, not thickness.

**Theorem 3.7. (Behrstock-Hagen-Sisto)** $W_\Gamma$ is thick $\iff$ $\Gamma \in T$, where $T$ is the smallest set of graphs satisfying the following conditions:

1. $K_{2,2} \in T$
2. If $\Gamma \in T$ and $\Lambda$ is an induced subgraph of $\Gamma$ that is not complete, then the graph $\Gamma'$ produced by adding a new vertex $v$ and edges between $v$ and every vertex in $\Lambda$ is in $T$. We say $\Gamma'$ is produced by coning off $\Lambda$ in $\Gamma$.
3. If $\Gamma_1, \Gamma_2 \in T$ and $\Lambda$ is a graph that is not complete and isomorphic to a full subgraph of both $\Gamma_1$ and $\Gamma_2$, then the graph $\Gamma$ produced by taking the disjoint union $\Gamma_1 \sqcup \Gamma_2$, identifying the copies of $\Lambda$, and possibly adding any number of edges, where each of the added edges is between a vertex in $\Gamma_1 \setminus \Lambda$ and one in $\Gamma_2 \setminus \Lambda$. We say that $\Gamma$ is a thick union of $\Gamma_1$ and $\Gamma_2$. [BHS13]

Although this characterisation allows us to tell whether a graph is thick or not, it says nothing about the graph’s order of thickness. Dani and Thomas also exhibit the family of graphs $\Gamma_k$ for $k \in \mathbb{N}$. The first few terms of $\Gamma_k$ are given in fig. 3.

**Figure 3:** $\Gamma_k$ for $k = 1, 2, 3, 4$ [DT15]
Theorem 3.8. (Dani-Thomas) $W_{\Gamma_k}$ has divergence $\text{div}_{W_{\Gamma_k}} \simeq r^k$.

i.e. $W_{\Gamma_k}$ is thick of order $k-1$ for each $k$, showing that right-angled Coxeter groups exhibit thickness of all orders.

4 Algorithmic Estimation of Thickness

Now we develop tools to algorithmically determine not only whether a graph corresponds to a thick right-angled Coxeter group or not, but also to give an upper bound on the order of thickness. To start, we will show that taking thick unions and coning to complete graphs increase the order of thickness by at most 1.

Lemma 4.1. (Thick Unions) Let $\Gamma$ be formed by taking a thick union of $\Gamma_1$ and $\Gamma_2$, and suppose $W_{\Gamma_1}$ and $W_{\Gamma_2}$ are thick of order at most $n-1$. Then $W_{\Gamma}$ is thick of order at most $n$.

Proof. I claim that $\mathcal{H} = \{W_{\Gamma_1},W_{\Gamma_2}\}$ is the collection of subgroups that demonstrates that $W_{\Gamma}$ is thick of order at most $n$. To show this, we need to establish that (i) Each $H \in \mathcal{H}$ is quasi-isometrically embedded in $W$, (ii) the subgroup $\langle \bigcup_{H \in \mathcal{H}} H \rangle$ has finite index in $W_{\Gamma}$, (iii) for each pair $H,H' \in \mathcal{H}$ there is a finite sequence $H = H_1, \ldots, H_k = H'$ with $H_i \cap H_{i+1}$ infinite for $1 \leq i < k$, and (iv) each $H$ is thick of order at most $n-1$.

(i) In the definition of the thick union, the only edges we add are between vertices not both in $\Gamma_i$ for $i = 1,2$, so the induced subgraph on the vertices of $\Gamma_1$ in $\Gamma$ will just be $\Gamma_1$, and similarly for $\Gamma_2$—i.e. $W_{\Gamma_1}$ and $W_{\Gamma_2}$ are special subgroups, and are thus quasi-isometrically embedded.

(ii) Because every vertex of $\Gamma$ is contained in some $\Gamma_i$, $\langle \bigcup_{H \in \mathcal{H}} H \rangle = W_{\Gamma}$.

(iii) Because $\Lambda = \Gamma_1 \cap \Gamma_2$ is assumed to be not complete, $W_{\Gamma_1} \cap W_{\Gamma_2}$ is infinite.

(iv) By assumption each $W_{\Gamma_i}$ is thick of order at most $n-1$. \qed

Lemma 4.2. (Coning to Complete Graphs) Suppose $W_{\Gamma}$ is thick of order $n-1$, $\Lambda$ is a full subgraph of $\Gamma$ that is not complete and let $C_k$ be the complete graph on $k$ vertices. Then if $\Gamma'$ is the graph obtained by coning to $C_k$ over $\Lambda$ (i.e. by connecting each vertex of $C_k$ to every vertex in $\Lambda$), then $W_{\Gamma'}$ is thick of order at most $n$.

Proof. We prove this by induction on $k$. First we’ll show it for $k = 1$, where if $a$ is the generator corresponding to the vertex of $C_1$, I claim that $\mathcal{H} = \{W_{\Gamma},aW_{\Gamma}a\}$ is the desired collection of subgroups.

$W_{\Gamma}$ is a special subgroup of $W_{\Gamma'}$, and $aW_{\Gamma}a$ is quasi-isometric to $W_{\Gamma}$ via conjugation by $a$, so it is also quasi-isometrically embedded in $W_{\Gamma'}$. Because thickness is invariant under quasi-isometry, $W_{\Gamma}$ and $aW_{\Gamma}a$ are thick of order $n-1$. Because $a$ commutes with every generator corresponding to a vertex in
For the inductive step, assume that coning \( \Gamma \) to \( C_k \) yields a graph \( \Gamma' \) such that \( W_{\Gamma'} \) is thick of order \( n \), and that a choice \( \mathcal{H} \) of subgroups of \( W_{\Gamma'} \) that demonstrate this is \( \mathcal{H} = \{ \omega W_{\Gamma'} : \omega \in W_{C_k} \} \), and that \( \langle \bigcup_{H \in \mathcal{H}} H \rangle = \text{Ker} \varphi \) for a homomorphism \( \varphi : W_{\Gamma'} \to \bigoplus_{i=1}^{k} \mathbb{Z}/2\mathbb{Z} \) defined by
\[
\varphi(W_{\Gamma'}) = 0, \quad \varphi(a) = 1.
\]
Clearly \( \text{Ker} \varphi \) is the set of all words \( \omega \in W_{\Gamma'} \) where \( a \) appears an even number of times in \( \omega \). Since this is true of each \( H \in \mathcal{H} \), we must have \( \langle \bigcup_{H \in \mathcal{H}} H \rangle \subset \text{Ker} \varphi \).

All that remains is the reverse inclusion. So suppose \( \omega \) is a word in \( W_{\Gamma'} \) where \( a \) appears an even number of times. We build \( \omega \) up as a product of words, each in some \( H \in \mathcal{H} \). If \( \omega \in W_{\Gamma} \), were done, so we may assume \( a \) occurs in \( \omega \). As an element of \( W_{\Gamma'} \), \( \omega \) is formed as a product of generators in \( W_{\Gamma} \) as well as \( a \), i.e.
\[
\omega = \gamma_0 \cdot a \cdot \gamma_1 \cdot a \cdots a \cdot \gamma_\ell
\]
where \( \gamma_i \in W_{\Gamma} \), (and with possibly \( \gamma_0 \) or \( \gamma_\ell = 1 \)). Since \( a \) occurs an even number of times, we can group these as
\[
\omega = \gamma_0 \cdot (a \gamma_1 a) \cdots (a \gamma_{\ell-1} a) \cdot \gamma_\ell
\]
thus \( \omega \in \langle \bigcup_{H \in \mathcal{H}} H \rangle \). Therefore \( \langle \bigcup_{H \in \mathcal{H}} H \rangle \) has finite index in \( W_{\Gamma'} \).

For the inductive step, assume that coning \( \Gamma \) to \( C_k \) yields a graph \( \Gamma' \) such that \( W_{\Gamma'} \) is thick of order \( n \), and that a choice \( \mathcal{H} \) of subgroups of \( W_{\Gamma'} \) that demonstrate this is \( \mathcal{H} = \{ \omega W_{\Gamma'} : \omega \in W_{C_k} \} \), and that \( \langle \bigcup_{H \in \mathcal{H}} H \rangle = \text{Ker} \varphi \) for a homomorphism \( \varphi : W_{\Gamma'} \to \bigoplus_{i=1}^{k} \mathbb{Z}/2\mathbb{Z} \) defined by
\[
\varphi(W_{\Gamma'}) = 0, \quad \varphi(x_i) = (0,\ldots,1,\ldots,0)
\]
where the 1 is in the \( i \)th coordinate, and \( x_i \) is the \( i \)th generator of \( W_{C_k} \). (Note that this is the case for \( k = 1 \).) Then we want to show that coning to \( C_{k+1} \) also yields a graph that is thick of order \( n \) via a collection of subgroups that satisfies these hypotheses.

Now we’ll write \( W_{\Gamma'} \) for the graph obtained by coning \( \Gamma \) to \( C_{k+1} \). Take \( \mathcal{H} \) to be a collection of subgroups that satisfy the assumptions of the previous paragraph for coning in \( k \) of the vertices in \( C_{k+1} \) and let \( j \) be the generator of \( W_{\Gamma'} \) corresponding to the remaining vertex. We’ll consider \( \mathcal{H}' = \mathcal{H} \cup \{ jH : H \in \mathcal{H} \} \). By assumption, each \( H \in \mathcal{H} \) is quasi-isometrically embedded in the full subgraph of \( W_{\Gamma'} \) obtained by deleting the vertex corresponding to \( j \), so since the composition of quasi-isometric embeddings is again one, the \( H \) are quasi-isometrically embedded in \( W_{\Gamma'} \), and once again conjugation by \( j \) shows that the remaining subgroups are as well, which also shows each \( H \in \mathcal{H}' \) is thick of order at most \( n - 1 \). Since \( W_{C_{k+1}} \) is abelian, word \( \omega \in W_{C_{k+1}} \) containing \( j \) can be built from those that do not by multiplying by \( j \) on the left—i.e. \( \mathcal{H}' \) is of the form of the previous paragraph.

Since by assumption we can find a finite sequences between any two elements of \( \mathcal{H} \) with infinite intersection, we only need to show that we can “connect” the
remaining subgroups to $\mathcal{H}$. In fact, given $jHj \in \mathcal{H}' \setminus \mathcal{H}$, we know that $W_\Lambda$ is contained in $H \cap jHj$, since $H$ and $jHj$ are both conjugates of $W_\Gamma$ by words in generators that all commute with elements of $W_\Lambda$, so the intersection is infinite since $W_\Lambda$ is. Since all the remaining subgroups are of this form, we can find always find a finite sequence connecting any two elements of $\mathcal{H}'$.

All that remains is to show $\langle \bigcup_{H \in \mathcal{H}} H \rangle = \text{Ker } \varphi$, for $\varphi : W_\Gamma \to \bigoplus_{i=1}^{k+1} \mathbb{Z}/2\mathbb{Z}$ as defined above. $\text{Ker } \varphi$ is clearly the set of words $\omega \in W_\Gamma$ such that each of the generators in $W_{C_{k+1}}$ appear an even number of times in $\omega$. Since this is true for each element of $H \in \mathcal{H}'$, $\langle \bigcup_{H \in \mathcal{H}} H \rangle \subset \text{Ker } \varphi$. So assume $\omega \in \text{Ker } \varphi$. As above, we write

$$\omega = \gamma_0 \cdot x_1 \cdot \gamma_1 \cdot x_2 \cdot \gamma_2 \cdots x_{\ell} \cdot \gamma_{\ell}$$

where $\gamma_i \in W_\Gamma$, and each $x_i$ is a generator of $W_{C_{k+1}}$. I claim that $\omega$ can be expressed as

$$\gamma_0 \cdot (x_1 \gamma_1 x_1) \cdot (x_1 x_2 \gamma_2 x_2 x_1) \cdots (x_1 x_2 \cdots x_{\ell-1} \gamma_{\ell-1} x_{\ell-1} \cdots x_2 x_1) \cdot \gamma_{\ell}$$

Each term in the product is (after reducing) clearly an element of $\omega W_\Gamma \omega$ for some $\omega \in W_\Gamma$, so it only remains to check that $x_{\ell-1} \cdots x_2 x_1 = x_\ell \iff x_1 \cdots x_\ell$ represents the identity. But since by assumption each generator of $C_{k+1}$ appears an even number of times in the product $x_1 \cdots x_\ell$, and the generators all commute and have order 2, this is indeed the case, so $\omega \in \langle \bigcup_{H \in \mathcal{H}} H \rangle$. \hfill \square

These two lemmas provide the justification for the following theorem which gives two algorithms for estimating the order of thickness of a right-angled Coxeter group. Note first that if $\Gamma$ is not connected, then $W_\Gamma$ is not thick, because $K_{2,2}$ is connected and both transformations for creating graphs in $T$ yield connected graphs.

**Theorem 4.3.** Let $\Gamma$ be a finite simplicial graph, and let $\mathcal{M}$ be the collection of maximal thick join subgraphs of $\Gamma$ (i.e. each such subgraph is thick of order zero). Let $t = 0$. Then if either of the following algorithms returns a number in $\{0\} \cup \mathbb{N}$, $W_\Gamma$ is thick of order at most $n$. If either algorithm fails, $W_\Gamma$ is not thick.

**Alternate unions and cones**

1. Check whether $\Gamma \in \mathcal{M}$. If so, return $t$.

2. Otherwise, take unions in $\mathcal{M}$ so that each $M \in \mathcal{M}$ is thick of order at most $t + 1$.

3. Check whether $\mathcal{M}$ changed; if so, increment: $t = t + 1$. Check again whether $\Gamma \in \mathcal{M}$ and return $t$ if so.

4. If not, cone in the biggest possible complete subgraph of $\Gamma$ to each $M \in \mathcal{M}$ so that each $M \in \mathcal{M}$ is thick of order at most $t + 1$. (If there are multiple choices for biggest complete subgraph, pick an arbitrary one.)
(5) Check whether $\mathcal{M}$ changed; if so, increment: $t = t + 1$ and repeat from
step (1). If $\mathcal{M}$ is the same as it was the last time we performed step (1),
report failure.

**Prefer unions**

(1) Check whether $\Gamma \in \mathcal{M}$. If so, return $t$.

(2) Take unions in $\mathcal{M}$ so that each $M \in \mathcal{M}$ is thick of order at most $t + 1$.

(3) Check whether $\mathcal{M}$ changed; if so, increment: $t = t + 1$ and repeat from
step (1).

(4) If not, cone the biggest possible complete subgraph of $\Gamma$ to each $M \in \mathcal{M}$
so that each $M \in \mathcal{M}$ is thick of order at most $t + 1$.

(5) Check whether $\mathcal{M}$ changed; if so, increment: $t = t + 1$ and repeat from
step (1). If not, report failure.

We see an immediate advantage of beginning by computing join subgraphs:

**Corollary 4.4.** If either algorithm returns 0 or 1, $W_{\Gamma}$ is thick of order 0 or 1,
respectively.

**Proof.** Since we begin each algorithm with the collection of maximal join sub-
graphs, if either algorithm returns 1, $W_{\Gamma}$ is thick of order at most 1 and $\Gamma$ is
not a thick join, so $W_{\Gamma}$ is thick of order 1. Likewise, if either algorithm returns
0, $W_{\Gamma}$ is a thick join, and thus thick of order zero. ⋄

However, for orders other than 0 and 1, both algorithms indeed only prove
that the right-angled Coxeter group corresponding to a given graph is thick of
order *at most* $n$—there are examples of graphs for which one or both algorithms
return 2, but the graph in question is $CFS$. Nevertheless, both algorithms give
the correct answer for each graph in the Dani-Thomas family, and otherwise
appear to be very accurate. Both algorithms were implemented in Haskell, and
the code with light comments is attached in an appendix. Because the inverse
graph (the graph with the same vertices and an edge between two vertices if and
only if there is none in the original graph) of a join is disconnected, computing
maximal join subgraphs is the same as computing separating sets of vertices on
the graph’s inverse, for which we use the algorithm discussed in [BBC99].

5 Applications of Algorithms to Random Graphs

A random graph in the Erdős-Rényi $G(n,p)$ model [ER59] has $n$ vertices, and
pair of distinct vertices are joined by an edge with probability $p$. By running
our algorithms on graphs generated in this way, we can obtain a sense of the
thickness of a random right-angled Coxeter group. In particular, this also allows
us to consider the *asymptotic* behavior of right-angled Coxeter groups.
**Definition.** An event $E_{n,p}$ defined on random graphs in the $G(n, p)$ model is said to hold **asymptotically almost surely (a.a.s.)** if the probability that $E$ occurs, $P(E_{n,p}) \to 1$ as $n \to \infty$.

As shown in [BHS13] with the algorithm introduced there, for any constant probability $0 < p < 1$, a.a.s. a random right-angled Coxeter group will be thick (of some order). Because the algorithm runs very quickly for sufficiently small graphs, producing a sample set of random graphs for a given probability and number of vertices and varying each can provide insight into the thickness of small random graphs with an eye to giving a bound on the order of thickness.

As it happens, even for graphs with fewer than 100 vertices, random graphs at certain densities exhibit a general trend: at first most graphs are not thick, with the algorithms returning large orders of thickness on some graphs. As the number of vertices increases, so does the proportion of thick graphs, and the order of thickness decreases until almost all graphs are thick of order 1. As this begins to happen, the computation time increases, roughly because the algorithm for enumerating maximal join subgraphs is linear in the total number of such joins, and a graph that is thick of order 1 will often have many such join subgraphs. This trend is illustrated in fig. 4, which graphs thresholds for order 1 thickness as a function of probability density.

![Figure 4: Thresholds for order 1 thickness as probability density varies](image)

Data from running the algorithm on a cluster also corroborates this evidence for graphs between 100 and 200 vertices (see fig. 6). In fact, for fixed density, the order of thickness appears to behave like $e^{-x}$ as a function of the number of vertices, as shown in fig. 5.

All of this leads us to conjecture that, at least for constant probability density $0 < p < 1$, a random right-angled Coxeter group is a.a.s. thick of order 1. This
conjecture has been proven by Behrstock, Hagen and Susse, but a discussion of
the proof is beyond the scope of this paper.

Figure 5: Thickness decreasing as $e^{-x}$ in the number of vertices

Figure 6: Mean thickness against probability density and number of vertices

6 Appendix: Algorithm Code

The code reproduced here attempts some optimization for faster computations,
including an option to only perform a few steps of join generation before at-
tempting to generate the graph from unions and cones. Using this option has
the obvious downside of potentially increasing the overestimation of the order of
thickness, and the non-obvious downside of slowing the computation of graphs
that are not thick of any order.
The code is split across several files: “ListOps” contains helper functions for dealing with Haskell lists, “Graph” contains the definitions for graph objects, “Joins” contains the functions for computing join subgraphs of a graph, “Thickness” contains the code to implement the algorithms, and “tea” is the main file, from which the executable “tea” (for “thickness estimation algorithm”) is created.

**ListOps.hs:**

```haskell
module ListOps (unique, uniquely, (+), (-), (=), (<)) where

-- (+) adds two lists
-- Elements from the right list will only be added if they're not already
-- contained in the left list. The function is defined recursively on the right list.
(+):: (Eq a) => [a] -> [a] -> [a]
(+)[ ] = [ ]
(+)(x:ys) = | y 'elem' xs = xs ++ y : ys
| otherwise = (x:xs) ++ y : ys

-- (-) subtracts two lists.
-- Elements will be removed from the left list if they're in the right list.
(-):: (Eq a) => [a] -> [a] -> [a]
(-)[ ] = [ ]
(-)xs ys = [x | x <- xs , not (x 'elem' ys)]

-- (=) tests whether lists are equal or equivalent.
-- Under this function, lists are the same if they each contain the same elements.
-- Note that a list with duplicates will be equivalent to a list without.
(=):: (Eq a) => [a] -> [a] -> Bool
(=)[ ] = True
(=)(x:xs) = | x 'elem' ys = zs 'elem' xs = zs 'elem' ys
| otherwise = False

-- Calls a helper function to build a list of unique elements
unique:: (Eq a) => [a] -> [a]
unique [ ] = []
unique (x:xs) = uniq xs 

-- Recursively builds a list without duplicates.
-- The argument on the right is used to build the new list
uniq:: (Eq a) => [a] -> [a] -> [a]
uniq[ ] = []
uniq(x:xs) = | any (x ==) ys = uniq xs ys
| otherwise = uniq x : ys

-- A version of the above function but for lists of lists
-- where list equivalence is used instead of equality
uniqL:: (Eq a) => [[a]] -> [[a]]
uniqL[ ] = []
uniqL(x:xs) = | any (x '==' [ ] ) = uniqL xs [ ]
| otherwise = uniqL x : uniqL (x:xs)

-- Calls the helper function for the list of lists
uniquely:: (Eq a) => [[a]] -> [[a]]
uniquely zs = uniqL zs [ ]
```

**Graph.hs**

```haskell
module Graph (Vertex, Edge, Graph, (~:), edge, edges, vertices, makeGraph, invert, WithGraph, ksa, toGraph) where

import Data.List (intersect, sort)
import ListOps
import Control.DeepSeq
import Control.Monad.Reader
import Control.Monad.Identity
import Control.Monad.State

-- Vertices
--
type Vertex = Int

-- Edges
newtype Edge = Edge (getEdge :: (Vertex,Vertex))

edge :: (Vertex,Vertex) -> Edge
```
edge \((a,b)\) = Edge \((a,b)\)

instance Eq Edge where
   Edge \((a,b)\) == Edge \((x,y)\)
   \| \((a,b)\) == \((x,y)\) = True
   \| \((b,a)\) == \((x,y)\) = True
   \| otherwise = False

instance Show Edge where
   show (Edge \((a,b)\))
   \| a < b = show \((a,b)\)
   \| otherwise = show \((b,a)\)

instance NFData Edge where
   rnf (Edge \((a,b)\)) = rnf \((a,b)\)

to = snd . getEdge
from = fst . getEdge

-- Graph definition --
newtype Graph = HiddenConstructor \((\text{\{Vertex\},\text{\{Edge\}}})\)
vertices = fst . getGraph
edges = snd . getGraph

instance Eq Graph where
   g1 == g2 = (vertices g1 \(\cong\) vertices g2) && (edges g1 \(\cong\) edges g2)

instance Show Graph where
   show graph = "graph with vertices: " ++ (show . sort $ vertices graph)
   ++ " and edges: " ++ (show $ edges graph)

instance NFData Graph where
   rnf = rnf . getGraph

makeGraph :: \([\text{Vertex}]\) \(\rightarrow\) \([\text{Edge}]\) \(\rightarrow\) Graph
makeGraph vs es = HiddenConstructor (vs ,es ')
where es ' = \[e \mid e \leftarrow \text{unique} \text{es} , \text{to e} \neq \text{from e} , \text{to e} \text{ 'elem ' vs} , \text{from e} \text{ 'elem ' vs\]}

-- Graph transformations --
-- inducing a graph from a subset of its vertices
(\(\text{\{\}}\)) :: Graph \(\rightarrow\) [\text{Vertex}] \(\rightarrow\) Graph
graph' :: Graph \(\rightarrow\) [\text{Vertex}] \(\rightarrow\) Graph
   \(\text{\{\}}\) = makeGraph \(\text{\{}\text{Vertex}\}\) \(\rightarrow\) Graph
   graph' = makeGraph vs as = HiddenConstructor (vs,as')
where as' = \[a \mid a \leftarrow \text{\text{unique}} \text{as , to } a \neq \text{from } a , \text{to } a \text{ 'elem ' vs} , \text{from } a \text{ 'elem ' vs\]}

invert :: Graph \(\rightarrow\) Graph
invert graph = let
   vs = vertices graph
   es = edges graph
   in makeGraph vs \[edge (v,w) \mid v \leftarrow vs , w \leftarrow vs , v < w , \text{not } edge (v,w) \text{ 'elem ' es}\]

-- WithGraph monad: allows functions to 'carry around' a graph to reference --
type WithGraph a = ReaderT Graph Identity a
-- ksa also allows functions in other files to 'carry around' the graph's inverse --
ksa :: \(\text{\{}\text{Monad m}\}\) \(\rightarrow\) ReaderT Graph m Graph
ksa = do
   graph <- ask
   return $ invert graph
-- toGraph takes a list of vertices and returns the induced subgraph on those vertices
-- note that it returns the subgraph with reference to the 'original' graph --
toGraph :: \(\text{\{}\text{Monad m}\}\) \(\rightarrow\) [\text{Vertex}] \(\rightarrow\) ReaderT Graph m Graph
toGraph vs = do
   graph <- ask
   return (graph \(\text{\{}\) vs\]}

-- module Joins (\{\text{start, next, allI, joinComponents, JoinState, isConnected, paths, cliques}\}) where

import Graph
import ListOps
import Control.Parallel.Strategies
import Control.Monad State
import Control.Monad Reader
import Control.Monad Identity

-- checks whether the graph in question is connected 'monadically'
-- i.e. the return value also contains the context of the graph in question --
isConnected :: WithGraph Bool
isConnected = do
   graph <- ask
   if vertices graph == []
      then return False
      else...
else return (-' vertices graph) >>= component >>= return . head . vertices >>= return . graph

-- return the neighbours of a vertex --
\n\n\n\n
\n
-- the non-monadic version of 'n' --
\n\n\n\n
-- given a vertex, returns a list of its path components, where each path component is a list of vertices
\n
-- given a vertex, returns the path component containing that vertex (with context) --
\n
-- recursively calls neighbours to build up the path component containing a set of vertices
\n
-- (rather slowly) returns the totally disconnected subgraphs of a graph recursively.
\n
-- the cliques of a graph is just the set of completely disconnected subgraphs of the inverse graph --
\n
-- allows the program to carry the 'context' of being partly finished computing the join subgraphs of a graph --
\n
-- given a set of vertices, returns the path components of the graph with these vertices removed.
\n
-- starts the join-generation algorithm, generating one join for every vertex in the graph
\n
-- runs the join algorithm until it's completed
\n
allJ = do
    helper (seps,seen) <- get
    graph <- ask
    mapM (return . withStrategy (parList rparWith rdeepseq)) seps
    return . foldl (+) [] (\n        do
            scriptC =\n            scriptC =\n    )

16
-- recursively generates more joins (a la 'next') until there aren't any more to be found
helper :: WithJoin ()
helper = do
  (seps,seen) <- get
  invgraph <- ksa
  let notChecked = seps ~-~ seen
  if notChecked == [] then return () else do
    let newSeps = foldl (~+~) seps . withStrategy (parList rpar) . map (
      g -> runReader ((return $ vertices g) >>=
        mapM (mapM (toGraph <+> neighbours) <+> scriptC . (vertices g ~<~) . ng)
        invgraph) $ notChecked
    ) invgraph
    put (newSeps,seps)
    helper

-- for every join that hasn't already 'been checked,' generates a new join by
-- swapping out each vertex of the join component for its neighbours, roughly.
next :: WithJoin [Graph]
next = do
  (seps,seen) <- get
  invgraph <- ksa
  let notChecked = seps ~-~ seen
  if notChecked == [] then return [] else do
    let newSeps = foldl (~+~) seps . withStrategy (parList rpar) . map (
      g -> runReader ((return $ vertices g) >>=
        mapM (mapM (toGraph <+> neighbours) <+> scriptC . (vertices g ~<~) . ng)
        invgraph) $ notChecked
    ) invgraph
    put (newSeps,seps)
    graph <- ask
    mapM (toGraph . (vertices graph ~<~) . vertices) (newSeps ~-~ seps)
    >>= return . withStrategy (parList $ rparWith rdeepseq)

Thickness.hs

module Thickness (execThickness, Flag (Short,Long,Extra)) where

import Graph
import Joins
import ListOps
import Control.Monad.State
import Control.Monad.Writer
import Control.Monad.Identity
import Control.Monad.Reader
import Control.Parallel.Strategies
import Control.Parallel (par,pseq)
import Control.DeepSeq

-- in the end our monads make the actual algorithm function really short, but also pretty difficult to read.
-- roughly we have a lot of extra contexts to our computations: a running log of computations
-- (which in the more verbose modes gives us real-time updates of progress when significant things happen),
-- a graph that we're processing, a flag that tells whether we want short results, long results, or extra
-- long,
-- a list of blocks as well as a variable that tells us whether we took unions or cones last
-- and we sometimes also have the join state from earlier.
-- after all is said and done, the execFunction is all about plugging in the initial values and then
-- just letting the result fall out.

type Results a = WriterT [String] (ReaderT (Graph,Flag) (StateT ([Graph],Maybe Bool) Identity)) a
runResults True flag & graph = runIdentity (runStateT (runReaderT $ runWriterT $ thickness algorithm n) (graph,flag)) ([]) Nothing
runResults False flag & graph = runIdentity (runStateT (runReaderT $ runWriterT $ thickness algorithm n) (graph,flag)) ([]) Nothing

execThickness :: Bool -> Flag -> Int -> Graph -> ([String]) Results ([Graph],JoinState)
execThickness b f n g = if log == [] then (maybe "n/a" show result) else log ++ ("Final result: " ++) (maybe "n/a" show result)
  where (result,log) = runIdentity $ runStateT (runReaderT (runWriterT (runStateT (runReaderT (runWriterT $ thickness algorithm n) (graph,flag)) ([] Nothing))))

data Flag = Short | Long | Extra
instance Eq Flag where
  Short == Short = True
  Long == Long = True
  Extra == Extra = True
  _ == _ = False

-- these functions more or less 'lift' the join generation functions into the contexts we're working in.
startR :: Results ([Graph],JoinState)
startR = do
  (graph,flag) <- ask
  case flag of
    Short -> return ()
    Long -> tell ["Starting new graph..."]
    Extra -> tell ["Starting new graph..."]
  return . runIdentity $ runStateT (runReaderT (startR graph)) ([],[])
-- remember that almost no extra computation is done if we call next more times than necessary
nextM :: JoinState -> Results ([Graph], JoinState)
nextM gen = do
  (graph, flag) <- ask
  if flag == Extra then tell ["Generating next batch of blocks..."] else return ()
  return . runIdentity . runStateT (runReaderT (next) graph) gen

allM :: Results ([Graph], JoinState)
allM = do
  (graph, flag) <- ask
  case flag of
    Short -> return ()
    Long -> tell ["Starting new graph...", "Generating all blocks..."]
    Extra -> tell ["Starting new graph...", "Generating all blocks...", "Generating next batch of blocks..."]
  return . runIdentity $ runStateT (runReaderT (allJ) graph) ([], [])

-- filters the results of join generation to be just those that are thick joins
orderZero :: ([Graph], JoinState) -> ([Graph], JoinState)
orderZero (blocks, j) = (withStrategy (parList rpar) . filter ((2 <=) . length . filter notComplete . joinComponents) $ blocks, j)

-- a little helper function to sequence calls of nextM
( >~ >) :: (JoinState -> Results ([Graph], JoinState)) -> (JoinState -> Results ([Graph], JoinState)) -> JoinState -> Results ([Graph], JoinState)
(a >~ > b) = 
  do
  (blocks2, j2) <- a j
  (blocks3, j3) <- b j2
  return (blocks2 ++ blocks3 'using' (parList rpar), j3)

-- given a number of steps of join generation and a flavour of algorithm, gives us the thickness of a graph by calling refine
thickness :: (Int -> Results (Maybe Int)) -> Int -> Results (Maybe Int)
thickness function n = do
  (graph, flag) <- ask
  if runReader (isConnected) graph == False
  then do
    if flag == Extra then tell ["Graph is disconnected."] else return ()
    return Nothing
  else do
    if n == -1
    then allM >>= return . orderZero >>= refine function
    else do
      let m = max (n - 1) 0
      if m == 0
      then startM >>= return . orderZero >>= refine function
      else return ([], []) >>= foldl ( >~ >) (
        (_ , flag) -> ask
        case flag of
          Short -> return ()
          Long -> tell ["Failed; trying again... "]
          Extra -> tell ["Trying again... "]
          refine function $ orderZero (blocks ~+~ nextblocks, nextgen)

-- if it's possible to generate more joins and try again after the algorithm fails to find thickness
-- refine is the function that will do that. It also reports final success or failure
-- (which is why we call it even if we already generated all joins)
refine :: (Int -> Results (Maybe Int)) -> ([Graph], JoinState) -> Results (Maybe Int)
refine function (blocks, gen) = do
  (graph, flag) <- ask
  case flag of
    Short -> return ()
    Long -> tell ["Failed; trying again... "]
    Extra -> tell ["Trying again... "]
      refine function (blocks ~+~ nextblocks, nextgen)

    if test /= Nothing
    then do
      if flag == Extra then tell ["Success!"] else return ()
      return test
    else do
      if flag == Extra then tell ["Failure!"] else return ()
      return Nothing
  else do
    case flag of
      Short -> return ()
      Long -> tell ["Failed; trying again... "]
      Extra -> tell ["Trying again... "]
      refine function $ orderZero (blocks ++ nextblocks, nextgen)

    if test /= Nothing
    then do
      if flag == Extra then tell ["Success!"] else return ()
      return test
    else do
      if flag == Extra then tell ["Failure!"] else return ()
      return Nothing
    else do
      case flag of
        Short -> return ()
        Long -> return ("Failed; trying again...")
        Extra -> return ("Trying again...")
        refine function $ orderZero (blocks ++ nextblocks, nextgen)

18
the 'alternate unions and cones' algorithm

algorithm :: Int -> Results (Maybe Int)
algorithm n = do
  (blocks,last) <- get
  if blocks == [] then return Nothing else do
    case last of
      Just True -> do
        m <- doUnions n
        if m == n then return Nothing else algorithm n
        else algorithm m
      _ -> do
        m <- doCones n
        if m == n then return Nothing else algorithm m
  else algorithm m

the 'prefer unions' algorithm

algorithm' :: Int -> Results (Maybe Int)
algorithm' n = do
  (blocks,_) <- get
  if blocks == [] then return Nothing else do
    (graph,_) <- ask
    if graph 'elem ' blocks then return $ Just n else do
      m <- doUnions n
      if m == n then do
        o <- doCones n
        if n == o then return Nothing else algorithm' o
        else algorithm m
      else algorithm m
taking unions, there are -
++ show (length newBlocks) ++ " blocks: " ++ show
newBlocks ++ ":[]

put (newBlocks,Just True)
return $ n+1
-- checks whether a graph is complete
notComplete :: Graph -> Bool
notComplete graph
| (length . vertices) graph <= 1 = False
| otherwise = any (not . ('elem ' ('elem ' edges graph) . edge) $ (vertices graph) \ v \ vertices graph, v \< v )
-- checks whether the intersection of two graphs is not a clique
overlap :: Graph -> Graph -> WithGraph Bool
overlap graph j graph k
return . notComplete . (graph \:) . (\ v \ vertices j, v \ 'elem ' (vertices k))
-- the 'm' function is useful here, but it doesn't make sense to have the Joins file export it.
\ a :: Graph -> Vertex -> (Vert
a graph vertex = (\ v \ vertices graph, edge(v,vertex) \ 'elem ' edges graph)
-- asks whether two lists share an element
intersects :: (Eq a) => [a] -> [a] -> Bool
x \ intersects ' y = any ('elem ' x) y
-- cones (an arbitrary choice from) the largest possible clique to a given block
cone block = do
\ graph <- ask
\ let cs = runReader (cliques) . (graph \:) . filter (notComplete . (graph \:) . (filter ('elem ' vertices block)) . (n graph)) $ (vertices graph) \- (vertices block)
if cs == []
then return block
else do
\ let m = maximum $ map length cs
\ toGraph . (vertices block \+) \ head . filter ((m ==) . length) $ cs
-- takes all possible thick unions that can be done 'at once'
unionhelper :: [Graph] -> WithGraph [Graph]
unionhelper bs = do
\ graph <- ask
\ return . withStrategy (parList $ rparWith rdeepseq) . foldl (~+~) [] . map vertices $ do
\ blocks
\ if any ((\=" vertices graph) . foldl ('\+~) [] . map vertices) blocks
then return blocks
else do
\ head <- return . withStrategy (parList rparWith rdeepseq) . foldl ('\+~) [block] . filter (any (\ g \ runIdentity $ runReaderT (overlap block g) graph)) $ blocks
\ tail <- return . withStrategy (parList rparWith rdeepseq) . filterM (return . not . any (\ g \ runIdentity $ runReaderT (overlap block g) graph)) blocks
\ return $ (force head 'par' force tail) 'pseq' (head : tail)

usage = "nSYNTAX:
Input one graph per line as
[v1,v2,...,vn] [(v1,v12),(v2,v22),...,(vm1,vm2)]
n"#| -h| u| l|x

- given a list of vertices and a list of edges, prints an upper bound on the order of thickness of the corresponding graph or "n/a" if the graph is not thick. The upper bound may not be sharp."
otherwise = Nothing $ makeGraph (read $ w !! 0) (map (edge) . read $ w !! 1)
where w = words input

-- boilerplate to process the command line arguments.
-- Translates each flag into the proper flavour of the algorithm.
process :: [String] -> String -> IO (Graph -> [String])
process args prog
  | any ('elem ' args) ['"-h","-help","-?"] = do
    putStrLn $ " USAGE:\n prog $ USAGE \n prog $ callstyle $ usage $ note $ exitFailure
  | otherwise = do
    (n, args) <- filter ('elem ' args) args == []
    then return (n, args)
    else return (read (\ "$") . filter ('elem ' args) . args)

  case args of
    ['"-ux"'] -> do
      putStrLn " TEA started."
      return $ execThickness True Extra m
    ['"-u","-x"'] -> do
      putStrLn " TEA started."
      return $ execThickness True Extra m
    ['"x","-u"'] -> do
      putStrLn " TEA started."
      return $ execThickness True Extra m
    ['"ul"'] -> do
      putStrLn " TEA started."
      return $ execThickness True Long m
    ['"u","-l"'] -> do
      putStrLn " TEA started."
      return $ execThickness True Long m
    ['"l","-u"'] -> do
      putStrLn " TEA started."
      return $ execThickness True Long m
    ['"x"'] -> do
      putStrLn " TEA started."
      return $ execThickness False Extra m
    ['"l"'] -> do
      putStrLn " TEA started."
      return $ execThickness False Long m
    ['"u"'] -> return $ execThickness False Short m
    [] -> return $ execThickness False Short m
  _ -> do
    putStrLn $ " USAGE:\n prog $ callstyle $ usage $ note $ exitFailure

  main = do
    args <- getArgs
    prog <- getProgName
    function <- process args prog
    contents <- getContents
    mapM_ (toTry function) (lines contents)
    catch handler

  handler :: IOError -> IO ()
  handler _ = putStrLn usage >> exitFailure

  toTry :: (Graph -> [String]) -> String -> IO ()
toTry function contents = do
  if readGraph contents == Nothing
  then putStr usage >> exitFailure
  else putStrLn $ function fromJust . readGraph $ contents

References


