Linearization of Genome Sequence Graphs using Integer Linear Programming

Christopher Schmitz

A thesis presented for the degree of
Bachelor of Science

Formal supervisor: Prof. Dr. G. Klau
Algorithmic Bioinformatics
Heinrich Heine University Düsseldorf
Germany
17th August, 2022

Acknowledgments

I would like to thank Sven and Gunnar for their advice, my parents for proofreading and Anna Lisiecka and Norbert Dojer for helping me obtain the test data and test the Flow Procedure.

Abstract

Genome sequence graphs can represent genetic variation within a species. A linearization of these graphs improves readability and further computational analysis. The three metrics weighted reversing joins, weighted feedback arcs and the average cut width are to be minimized in the linearization. We show that the problem can be solved using Integer Linear Programming in reasonable time and with satisfactory results.

Contents

1 Introduction

Due to the increasing amount of genome sequence data, new methods for storing, constructing, visualizing and computationally analysing multiple genome sequences of a population are needed ("Computational pan-genomics: status, promises and challenges" 2018).
Genome sequence graphs are data structures that address this problem ("Computational pan-genomics: status, promises and challenges" 2018). By considering several genomes instead of just a single reference genome, genomic variation in a population can be represented (Garrison et al. 2018). It was shown that the use of a single reference genome leads to reference biases (Brandt et al. 2015).

In this thesis we will work with bidirected genome sequence graphs, in which each node is labelled with a DNA sequence. Edges can be connected to either the right or the left side of a node. These two sides allow us to represent the double strandedness of DNA, because depending on which side an edge enters a node, the DNA sequence labelling the node can be interpreted as the reverse complement. A directed path on a genome sequence graph can represent a DNA sequence (Lisiecka and Dojer 2021).

A linearization of these graphs improves readability and makes further computational analysis of the graph easier (Lisiecka and Dojer 2021). Heuristic methods for this problem have already been developed (Haussler et al. 2018; Lisiecka and Dojer 2021).
In Figure 1, first a genome sequence graph is constructed from two DNA sequences and then the graph is linearized.

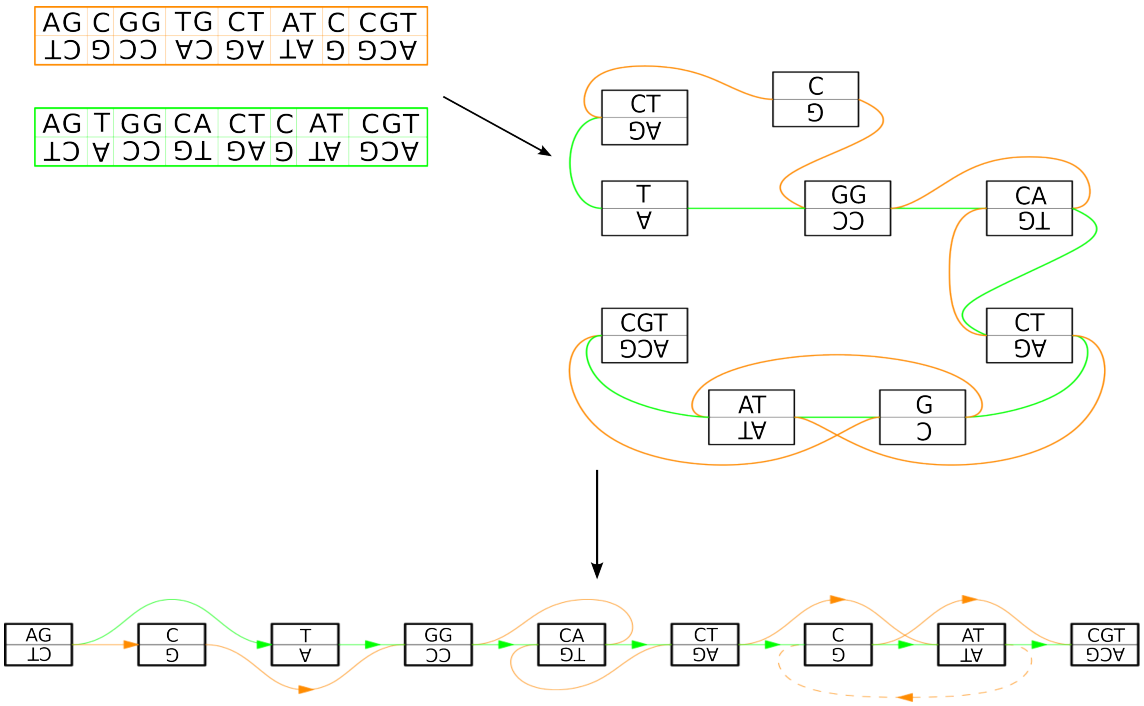


Figure 1: In the top left corner we can see two DNA sequences and their respective reverse complements. From these two DNA sequences, a genome sequence graph can be constructed. Both colored paths represent one of the DNA sequences. In this thesis, we will focus on the second step; the linearization of the genome sequence graph. We will order and orient the nodes: the nodes are ordered from left to right and each node is oriented by labelling one of its sides as an in-side and the other side as an out-side. In this Figure, nodes have been rotated in such a way that their in-side is always left and their out-side right.
Whenever a path enters a node from its out-side, the reverse complement of the DNA sequence in the node is read. For example, the orange path enters the fifth node on its out-side, so “TG” is read.
Feedback arcs do not conform to the ordering and are drawn as dashed edges and reversing joins connect two out-sides or two in-sides and are drawn without arrows, formal definitions are given in Section 2.
This Figure is strongly inspired by a Figure by Lisiecka and Dojer (Lisiecka and Dojer 2021).

2 Problem Formulation

Definition 1. Directed graph

A directed graph is a graph $D = (V,A)$ with nodes V and arcs A. Arcs $a \in A, a = (v,u)$ are directed from v to u. A path p in a directed graph is a sequence of arcs $a_1, \ldots, a_l$, such that for $a_i = (v_i, u_i)$ and $a_{i+1} = (v_{i+1}, u_{i+1})$ with $i \in \{1, \ldots, l-1\}$, $u_i = v_{i+1}$ holds. A path is simple if no node is traversed more than once on p. A circle c in a directed graph is a path p with $v_1 = u_l$. It is called a simple circle if p is simple. Two simple circles consisting of an identical set of edges are equivalent. A directed graph D is acyclic, if it does not contain any circles.

Definition 2. Bidirected graph

A bidirected graph $G = (V,E)$ is a graph with nodes V and edges E. Each node has two sides. An edge $e \in E, e = (v,s,v',s')$ with $v, v' \in V$ and $s, s' \in \{right, left\}$ connects side s of node v with side s' of node v' (Lisiecka and Dojer 2021). Let $n := |V|$ and $m := |E|$. We define the $\neg$-operator on sides as $\neg left = right$ and $\neg right = left$.

We also assign a numeric value to the sides left and right. Let $d(s) := \begin{cases} 1, & \text{if } s = right \\ -1, & \text{if } s = left \end{cases}$

During the process of linearizing a bidirected graph, edges are interpreted as undirected, so $e = (v,s,v',s')$ and $e' = (v',s',v,s)$ are equivalent.

Definition 3. Paths in bidirected graphs

A path p in a bidirected graph is a sequence of edges $e_1, \ldots, e_l$ with $e_i \in E$, $i \in 1, \ldots l$, for which the following holds: For two consecutive edges $e_i = (v_i, s_i, v'_i, s'_i)$ and $e_{i+1} = (v_{i+1}, s_{i+1}, v'_{i+1}, s'_{i+1})$, $v'_i = v_{i+1}$ for all $i \in \{1, \ldots, l-1\}$. Furthermore, p is directed, if $s'_i \neq s_{i+1}$ holds, meaning that the path enters each node on one side and exists it on its other side (Lisiecka and Dojer 2021). A path p is simple, if no node is traversed more than once.

We can say, there is a path from s_1 of v_1 to s'_l of v'_l.

Note that, although the terms are similar, there is a difference between directed paths in bidirected graphs and paths in directed graphs and they must not be confused.

Definition 4. Circles in bidirected graphs

A circle c in a bidirected graph $G = (V,E)$ is a path p from a node $v \in V$ to itself. c is a simple circle if p is simple. c is directed, if p is directed and p does not start on the same side of v as it ends (Paten et al. 2018). Otherwise, c is undirected. Two simple circles consisting of an identical set of edges are equivalent.
In the following, whenever we talk about circles, these are simple circles.

Definition 5. Linearization of a bidirected graph

Given a bidirected graph $G = (V, E)$ a linearization of the graph consists of an orienting function and ordering function.
The orienting function $a : V \to \{0, 1\}$ defines an in- and out-side for each node.

$$a(v) = \begin{cases} 0, & \text{if the right side of } v \text{ is labelled as an in-side and the left side as an out-side} \\ 1, & \text{if the left side of } v \text{ is labelled as an in-side and the right side as an out-side} \end{cases}$$

The ordering function is a bijection $ord : V \to \{1, \ldots, n\}$. If $ord(v) < ord(v')$, v precedes v' (Lisiecka and Dojer 2021).

Definition 6. Forward arcs

Forward arcs are edges $e \in E$ that join the out-side of node $v \in V$ with the in-side of node $v' \in V$ and $ord(v) < ord(v')$ (Lisiecka and Dojer 2021, see Figure 2).

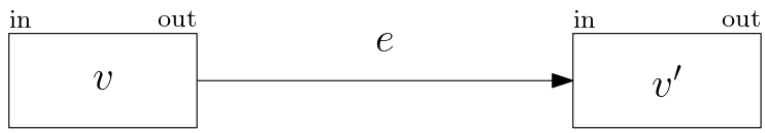


Figure 2: An edge e connecting the out-side of node v with the in-side of node v' and conforming to the ordering of nodes (here from left to right).

Definition 7. Reversing joins

Reversing joins are edges $e \in E$ that join two in-sides or two out-sides (Lisiecka and Dojer 2021).
For example, an edge $e = (v, right, v', right)$ with $a(v) = 1$ and $a(v') = 1$ is a reversing join, because due to $a(v) = a(v') = 1$ both the right side of v and the right side of v' are labelled as out-sides. Thus e joins two out-sides (see Figure 3).

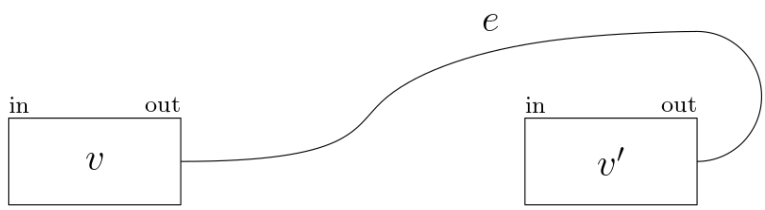


Figure 3: An edge e connecting the out-side of node v with the out-side of node v'.

Definition 8. Feedback arcs

Feedback arcs are edges $e \in E$ that join the out-side of node $v \in V$ with the in-side of node $v' \in V$ and $ord(v') < ord(v)$ (Lisiecka and Dojer 2021), meaning that e does not conform to the ordering of the nodes (see Figure 4).
An edge can not be a feedback arc if it is a reversing join, because if e joins two in-side or two out-sides, there is no sensible ordering of its nodes (Lisiecka and Dojer 2021).

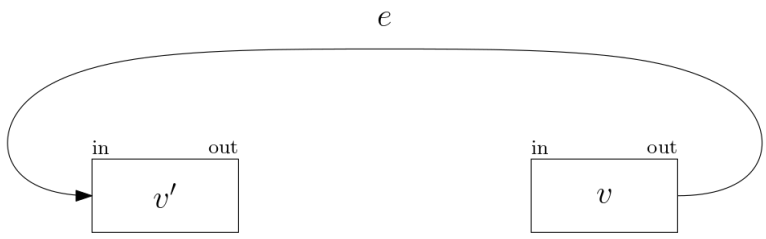


Figure 4: An edge e joining the out-side of node v with the in-side of node v' but not conforming to the ordering of nodes, meaning $ord(v') < ord(v)$.

Definition 9. Average cut width

Imagine a cut between each pair of consecutive nodes. The cut width is the number of edges crossing a cut (Haussler et al. 2018). Note that the average cut width can be calculated in two ways; either by adding the number of edges that cross each cut and dividing the sum by the number of cuts, which is equal to $n - 1$. Or by adding the number of cuts each edge crosses and again dividing this sum by the number of cuts (see Figure 5).

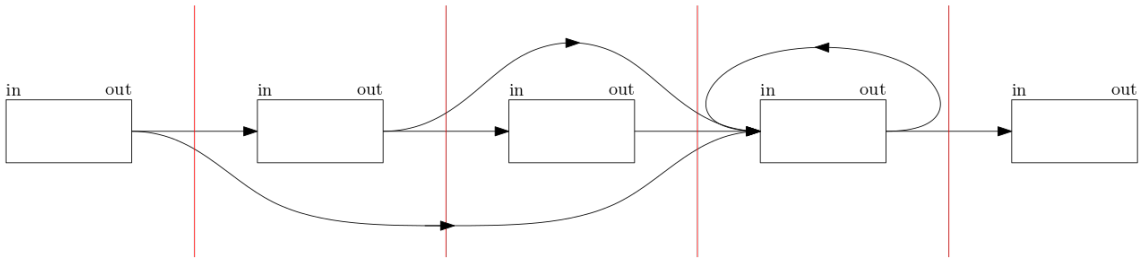


Figure 5: A linearized graph with red cuts placed between each pair of consecutive nodes. The respective cut widths are 2, 3, 3 and 1. The average cut width is $\frac{1}{4}(2+3+3+1) = \frac{9}{4}$. This is equal to the sum of the number of cuts each edge crosses, divided by the number of cuts. Here the number of cuts crossed by the seven edges respectively are 1, 1, 1, 1, 3, 2 and 0. The average cut width is indeed $\frac{1}{4}(1+1+1+1+3+2+0) = \frac{9}{4}$. Inspired by a Figure by Lisiecka and Dojer (Lisiecka and Dojer 2021).

Definition 10. Pareto optimal solution

A pareto optimal linearization (a, ord) of a bidirected graph is a linearization, so that there exists no linearization (a', ord') which decreases at least one of the metrics, weighted feedback arcs, weighted reversing joins or average cut width, without increasing another one of these metrics. The set of pareto optimal solutions forms the pareto front (see Figure 6 for a two-dimensional illustration).

Definition 11. Genome sequence graph

A genome sequence graph is a bidirected graph $G = (V, E)$ that represents a set of DNA sequences. Each node is labelled with a DNA fragment. For each DNA sequence that was used to construct the graph there is a directed path. A directed path on a genome sequence graph can form a DNA sequence by concatenating the DNA fragments labelling the nodes on the path. If a node v on this path is entered from its out-side, the reverse complement of the DNA fragment of v is used to construct the resulting DNA sequence of the path.

The weight function $w : E \to \mathbb{N}$ assigns a weight to each edge equal to the number of times it is traversed by the paths used to construct the graph (Lisiecka and Dojer 2021).

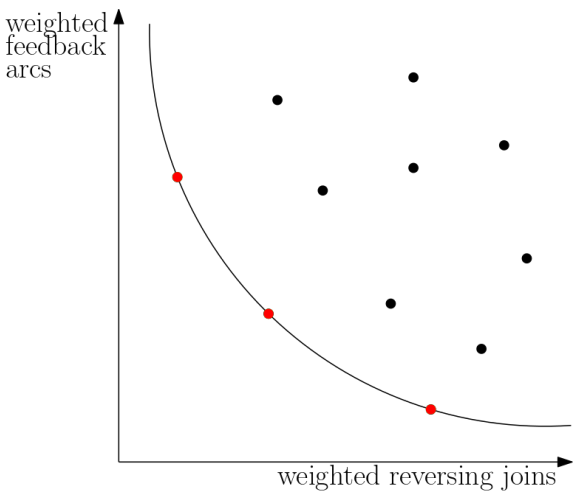


Figure 6: The dots represent an imaginary set of linearizations and their respective weighted feedback arcs and reversing joins. The red dots are pareto optimal solutions. The set of pareto optimal solutions forms the pareto front.

2.1 Problem Statement

Given a genome sequence graph $G = (V, E)$, find a linearization that minimizes a linear combination of weighted reversing joins, weighted feedback arcs and the average cut width.

3 ILP using ordering variables

Given a genome sequence graph $G = (V, E)$, $V = \{v_1, \ldots, v_n\}, E = \{e_1, \ldots, e_m\}$ and weights for the three described metrics $\alpha, \beta, \gamma \in \mathbb{R}_{\geqslant 0}$. Let the weight of an edge e be w_e.
We define the following Variables:

For all $v \in V$:

$$a_v = \begin{cases} 0, & \text{if the right side of } v \text{ is labelled as an in-side and the left as an out-side} \\ 1, & \text{if the left side of } v \text{ is labelled as an in-side and the right as an out-side} \end{cases}$$

For all $e \in E$:

$$\rho_e = \begin{cases} 0, & \text{if edge } e \text{ is not a reversing join} \\ 1, & \text{otherwise} \end{cases}$$

$$\varphi_e = \begin{cases} 0, & \text{if edge } e \text{ is not a feedback arc} \\ 1, & \text{otherwise} \end{cases}$$

κ_e = number of cuts crossed by e, $\kappa_e \in \{1, \ldots, n-1\}$

In order to represent an ordering of nodes we use the approach by Grötschel, Jünger, and Reinelt (Grötschel, Jünger, and Reinelt 1984).
For all $i, j \in \mathbb{N}$ such that $1 \leqslant i < j \leqslant n$:

$$y_{i,j} = \begin{cases} 0, & \text{if } v_i \text{ precedes } v_j \\ 1, & \text{if } v_j \text{ precedes } v_i \end{cases}$$

The ILP is:

$$min\big(\alpha \sum_{e \in E} w_e \rho_e + \beta \sum_{e \in E} w_e \varphi_e + \frac{\gamma}{n-1} \sum_{e \in E} \kappa_e\big)$$

s.t.

$$y_{i,j} + y_{j,k} - y_{i,k} \leqslant 1 \qquad \forall i, j, k \in \mathbb{N}, \text{ such that } 1 \leqslant i < j < k \leqslant n \tag{1}$$

$$-y_{i,j} - y_{j,k} + y_{i,k} \leqslant 0 \qquad \forall i, j, k \in \mathbb{N}, \text{ such that } 1 \leqslant i < j < k \leqslant n \tag{2}$$

$$a_v + a_w + \rho_e \geqslant 1 \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s = s' \tag{3}$$

$$a_v + a_w - \rho_e \leqslant 1 \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s = s' \tag{4}$$

$$a_v + \rho_e \geqslant a_w \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s \neq s' \tag{5}$$

$$a_v - \rho_e \leqslant a_w \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s \neq s' \tag{6}$$

$$d(s) \cdot (2a_{v_i} - 1) + y_{i,j} - d(s') \cdot (2a_{v_j} - 1) - \varphi_e \leqslant 2 \qquad \forall e \in E, \text{ with } e = (v_i, s, v_j, s') \tag{7}$$

$$d(s) \cdot (2a_{v_i} - 1) + y_{i,j} - d(s') \cdot (2a_{v_j} - 1) + \varphi_e \geqslant -1 \qquad \forall e \in E, \text{ with } e = (v_i, s, v_j, s') \tag{8}$$

$$\sum_{h=1, h \neq i}^{n} y_{h,i} - \sum_{h=1, h \neq j}^{n} y_{h,j} \leqslant \kappa_e \qquad \forall e \in E, \text{ with } e = (v_i, s, v_j, s') \tag{9}$$

$$\sum_{h=1, h \neq i}^{n} y_{h,j} - \sum_{h=1, h \neq j}^{n} y_{h,i} \leqslant \kappa_e \qquad \forall e \in E, \text{ with } e = (v_i, s, v_j, s') \tag{10}$$

3.1 Ordering

Inequalities (1) and (2) (Grötschel, Jünger, and Reinelt 1984; Baharev et al. 2021) are triangle inequalities which ensure that y represents a valid ordering by preventing circles $ord(v_i) > ord(v_j) > ord(v_k) > ord(v_i)$ (1) and $ord(v_i) < ord(v_j) < ord(v_k) < ord(v_i)$ (2) (see Figure 7, inspired by Mitchell and Borchers 2000).
Earlier, we defined $y_{i,j}$ only for $1 \leqslant i < j \leqslant n$. In the case of $i > j$ we can use the substitution $y_{i,j} = 1 - y_{j,i}$ (Grötschel, Jünger, and Reinelt 1984). $y_{i,i}$ is not defined and will not be needed.

3.2 Loops

Any edge $e = (v, s, v, s')$ is either, in the case of $s = s'$, a reversing join, or, in the case of $s \neq s'$, a feedback arc. So we can remove them from the graph before starting the process of linearization.

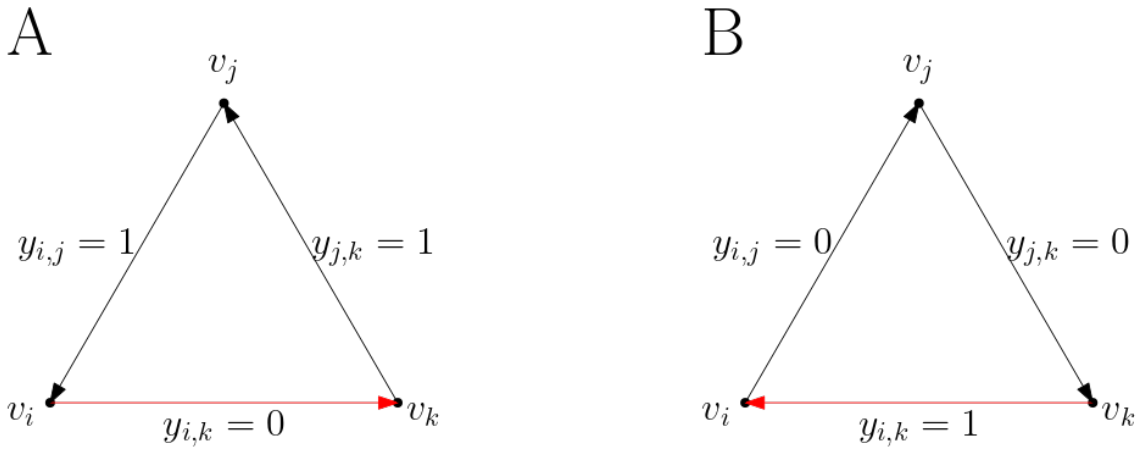


Figure 7: Directed edges from node v to node u symbolize that in the ordering v precedes u. Circles like the ones we see here would not be logical. The circle in A is prevented by $y_{i,j} + y_{j,k} - y_{i,k} \leqslant 1$; if v_j precedes v_i ($y_{i,j} = 1$) and v_k precedes v_j ($y_{j,k} = 1$), v_k has to precede v_i ($y_{i,k} \overset{!}{=} 1$), otherwise, if v_i would precede y_k ($y_{i,k} = 0$), the red arrow between v_i and v_k would lead to a circle. This is reflected by the inequality, which would not be satisfied, if $y_{i,k} = 0$.
The circle in B is prevented by $-y_{i,j} - y_{j,k} + y_{i,k} \leqslant 0$ similarly; if both v_i precedes v_j and v_j precedes v_k ($y_{i,j} = y_{j,k} = 0$) then v_i has to precede v_k ($v_{i,k} \overset{!}{=} 0$).

3.3 Reversing joins

Because of inequalities (3) to (6), $\rho_e = 0$ implies that the edge $e = (v, s, w, s')$ is not a reversing join.
If e joins the same sides of v and w (case $s = s'$), e is a reversing join, if $a_v = a_w$ and e is not a reversing join, if $a_v \neq a_w$. For example if $s = s' = right$, $a_v = 1$ and $a_w = 0$, e would not be a reversing join, because the right side of v would be labelled as an out-side and the right side of w would be labelled as an in-side.
So, in the case of $s = s'$, we want to add the constraint $a_v \neq a_w$, which is equal to $a_v + a_w = 1$, because of $a_v, a_w \in \{0, 1\}$ and can be written as two inequalities $a_v + a_w \geqslant 1$ and $a_v + a_w \leqslant 1$. In most cases, there will be at least some reversing joins and not all of these constraints can be satisfied. Therefore, we extend the constraints to $a_v + a_w + \rho_e \geqslant 1$ (3) and $a_v + a_w - \rho_e \leqslant 1$ (4). If $\rho_e = 1$, e is marked as a reversing join and these inequalities are always true.
In the other case of $s \neq s'$, e is not a reversing join, if and only if $a_v = a_w$ holds, which is equal to $a_v \geqslant a_w$ and $a_v \leqslant a_w$. As above, we add ρ_e to get $a_v + \rho_e \geqslant a_w$ (5) and $a_v \leqslant a_w + \rho_e$ (6).

3.4 Feedback arcs

Analogous to the last section, inequalities (7) and (8) ensure that $\varphi_e = 0$ implies that e is not a feedback arc.
An edge $e \in E$ can only be a feedback arc, if e is not a reversing join. For any edge, that connects an out-side with an in-side, the out-side node should precede the in-side node in the ordering.
We define the function $\mu : \{0, 1\} \times \{right, left\} \to \{-1, 1\}$, $\mu(a_v, s) := d(s) \cdot (2a_v - 1)$. In

label of side s	$\mu(a_v,s)$	a_v	s	$2a_v-1$	$d(s)$
in	-1	0	right	-1	1
		1	left	1	-1
out	1	0	left	-1	-1
		1	right	1	1

Table 1: Label of side s and μ for all combinations of a_v and s

Table 1, $\mu(a_v,s)$ and the label of side s of node v for all combinations of node orientations a_v and sides s, to which an edge e might be connected, are listed. We can see the following:

$$\mu(a_v,s) = \begin{cases} -1, & \text{if side } s \text{ of node } v \text{ is labelled as an in-side} \\ 1, & \text{if side } s \text{ of node } v \text{ is labelled as an out-side} \end{cases}$$

We also define the functions $\psi_1(a_v,s) := \mu(a_v,s)+1$ and $\psi_2(a_v,s) := \mu(a_v,s)-1$.

Given an edge $e=(v_i,s,v_j,s')$, let us first look at the case $\mu(a_{v_i},s)=1$ which means that e is connected to the out-side of v_i. We assume that e is not a reversing join. Then e is also connected to the in-side of v_j and v_i should precede v_j and therefore $y_{i,j}=0$ should hold. If $y_{i,j}=1$, e is a feedback arc. The inequality $\mu(a_{v_i},s)+y_{i,j} \leqslant 1$ is only true, if e is not a feedback arc.

If e is a reversing join, e can not be a feedback arc and the constraint should always be satisfied. In order to generalize the constraint, we defined ψ_1. From the definition of ψ_1 and μ we can see:

$$\psi_1(a_v,s) = \begin{cases} 0, & \text{if side } s \text{ of node } v \text{ is labelled as an in-side} \\ 2, & \text{if side } s \text{ of node } v \text{ is labelled as an out-side} \end{cases}$$

The final inequality is $\mu(a_{v_i},s)+y_{i,j}-\psi_1(a_{v_j},s')-\varphi_e \leqslant 1$

$\Leftrightarrow d(s)\cdot(2a_{v_i}-1)+y_{i,j}-d(s')\cdot(2a_{v_j}-1)-\varphi_e \leqslant 2$ (7). This inequality is also true, if e is connected to the out-side of v_j ($\psi_1(a_{v_j},s')=2$), meaning in the case of e also being connected to the out-side of v_i, that e is a reversing join. If $\varphi_e=1$ the constraint is always satisfied.

So far, we made sure, that $\varphi_e=0$ implies, that e is not a feedback arc only for the case, in which $e=(v_i,s,v_j,s')$ is connected to the out-side of v_i, meaning $\mu(a_{v_i},s)=1$. The above inequality is always true in the other case in which e is connected to the in-side of v_i indicated by $\mu(a_{v_i},s)=-1$. If in this case e is also connected to the out-side of v_j, v_j should precede v_i and $y_{i,j}=1$ should hold. This can be expressed as $\mu(a_{v_i},s)+y_{i,j} \geqslant 0$. As above, we must factor in the possibility of e being connected to the in-side of v_j and therefore a reversing join. This would be indicated by $\psi_2(a_{v_j},s')=-2$. We add the exceptions for this case and for the case of e being a feedback arc ($\varphi_e=1$): $\mu(a_{v_i},s)+y_{i,j}-\psi_2(a_{v_j},s')+\varphi_e \geqslant 0$

$\Leftrightarrow d(s)\cdot(2a_{v_i}-1)+y_{i,j}-d(s')\cdot(2a_{v_j}-1)+\varphi_e \geqslant -1$ (8).

3.5 Average cut width

The average cut width can be calculated by adding the number of cuts crossed by each edge and dividing this sum by the number of cuts, which is $n-1$.
This metric does not depend on the node orientations and sides.

We defined κ_e as the number of cuts crossed by e, constraints (9) and (10) ensure that the output of the ILP conforms to this definition. For an edge $e = (v_i, s, v_j, s')$, the number of cuts crossed by this edge in an ordering ord, is equal to the number of nodes between v_i and v_j plus one (see Figure 8). Therefore, we want to model $\kappa_e = |ord(v_i) - ord(v_j)|$ with our constraints.

Let us first look at the case of $y_{i,j} = 0$, meaning v_i precedes v_j in the ordering. Then κ_e would be equal to the number of nodes succeeding v_i minus the number of nodes succeeding v_j. For any node v_x, the number of nodes succeeding v_x is $post(x) := \sum_{h=1,h\neq x}^{n} y_{h,x}$. Since we minimize the sum of κ_e, the constraint $\kappa_e \geqslant post(i) - post(j)$ would lead to $\kappa_e = post(i) - post(j)$.

We were looking at the case of v_i preceding v_j ($y_{i,j} = 0$). If this is not the case, $post(i) - post(j)$ would be negative and thus the constraint would always be met. So, to ensure that $\kappa_e = post(i) - post(j)$, we add the constraint $\kappa_e \geqslant post(i) - post(j) \Leftrightarrow$
$\kappa_e \geqslant \sum_{h=1,h\neq i}^{n} y_{h,i} - \sum_{h=1,h\neq j}^{n} y_{h,j}$ (9).

Now let us turn to the case of $y_{i,j} = 1$. If v_j precedes v_i, κ is equal to $post(j) - post(i)$ and using the same ideas as above, this leads to the constraint $\kappa_e \geqslant post(j) - post(i) \Leftrightarrow$
$\kappa_e \geqslant \sum_{h=1,h\neq j}^{n} y_{h,j} - \sum_{h=1,h\neq i}^{n} y_{h,i}$ (10).

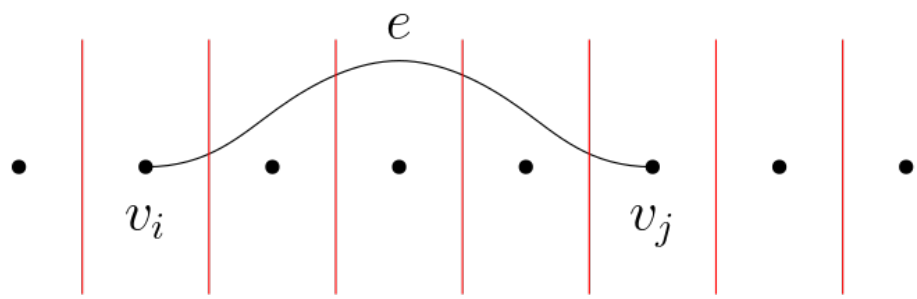


Figure 8: We see an edge $e = (v_i, s, v_j, s')$ in a possible ordering of nodes with $ord(v_i) < ord(v_j)$. The number of nodes between v_i and v_j is equal to the number of nodes succeeding v_i minus the number of nodes succeeding v_j minus one. In order to get the number of cuts e traverses, we add one. We can conclude that the number of cuts an e crosses are the number of nodes succeeding v_i minus the number of nodes succeeding v_j.

3.6 Reductions

The ILP with $O(n^2)$ variables and $O(n^3)$ variables is not feasible for large graphs.

In Appendix A a reduction technique is discussed which did not suffice to make the ILP feasible.

4 ILP using a circle matrix

The ILP using ordering variables proved to be infeasible for large graphs. One could instead use an ILP with a circle matrix to minimize feedback arcs similar to a method used for the feedback arc set problem. Each circle in the directed graph has to be covered by at least one feedback arc (Baharev et al. 2021). We can use a similar approach for our problem.
Note that this approach only minimizes weighted feedback arcs and weighted reversing joins. So for a pareto optimal solution we only consider reversing joins and feedback arcs.

Given a genome sequence graph $G = (V, E)$, $V = \{v_1, \dots, v_n\}, E = \{e_1, \dots, e_m\}$ and $\alpha, \beta \in \mathbb{R}_{\geqslant 0}$.
We define the following Variables:
For all $v \in V$:

$$a_v = \begin{cases} 0, & \text{if the right side of } v \text{ is labelled as an in-side and the left as an out-side} \\ 1, & \text{if the left side of } v \text{ is labelled as an in-side and the right as an out-side} \end{cases}$$

For all $e \in E$:

$$\rho_e = \begin{cases} 0, & \text{if edge } e \text{ is not a reversing join} \\ 1, & \text{otherwise} \end{cases}$$

$$\varphi_e = \begin{cases} 0, & \text{if edge } e \text{ is not a feedback arc} \\ 1, & \text{otherwise} \end{cases}$$

We also need the circle matrix $C \in \{0,1\}^{l \times m}$ with $l :=$ number of directed circles.

$$c_{ij} := \begin{cases} 0, & \text{if edge } e_j \text{ is not in directed circle } i \\ 1, & \text{if edge } e_j \text{ is in directed circle } i \end{cases}$$

The ILP is:

$$min\big(\alpha \sum_{e \in E} w_e \rho_e + \beta \sum_{e \in E} w_e \varphi_e\big)$$

s.t.

$$a_v + a_w + \rho_e \geqslant 1 \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s = s' \tag{11}$$

$$a_v + a_w - \rho_e \leqslant 1 \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s = s' \tag{12}$$

$$a_v - a_w - \rho_e \geqslant -1 \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s = s' \tag{13}$$

$$a_v - a_w + \rho_e \leqslant 1 \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s = s' \tag{14}$$

$$a_v + \rho_e \geqslant a_w \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s \neq s' \tag{15}$$

$$a_v - \rho_e \leqslant a_w \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s \neq s' \tag{16}$$

$$a_v + a_w - \rho_e \geqslant 0 \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s \neq s' \tag{17}$$

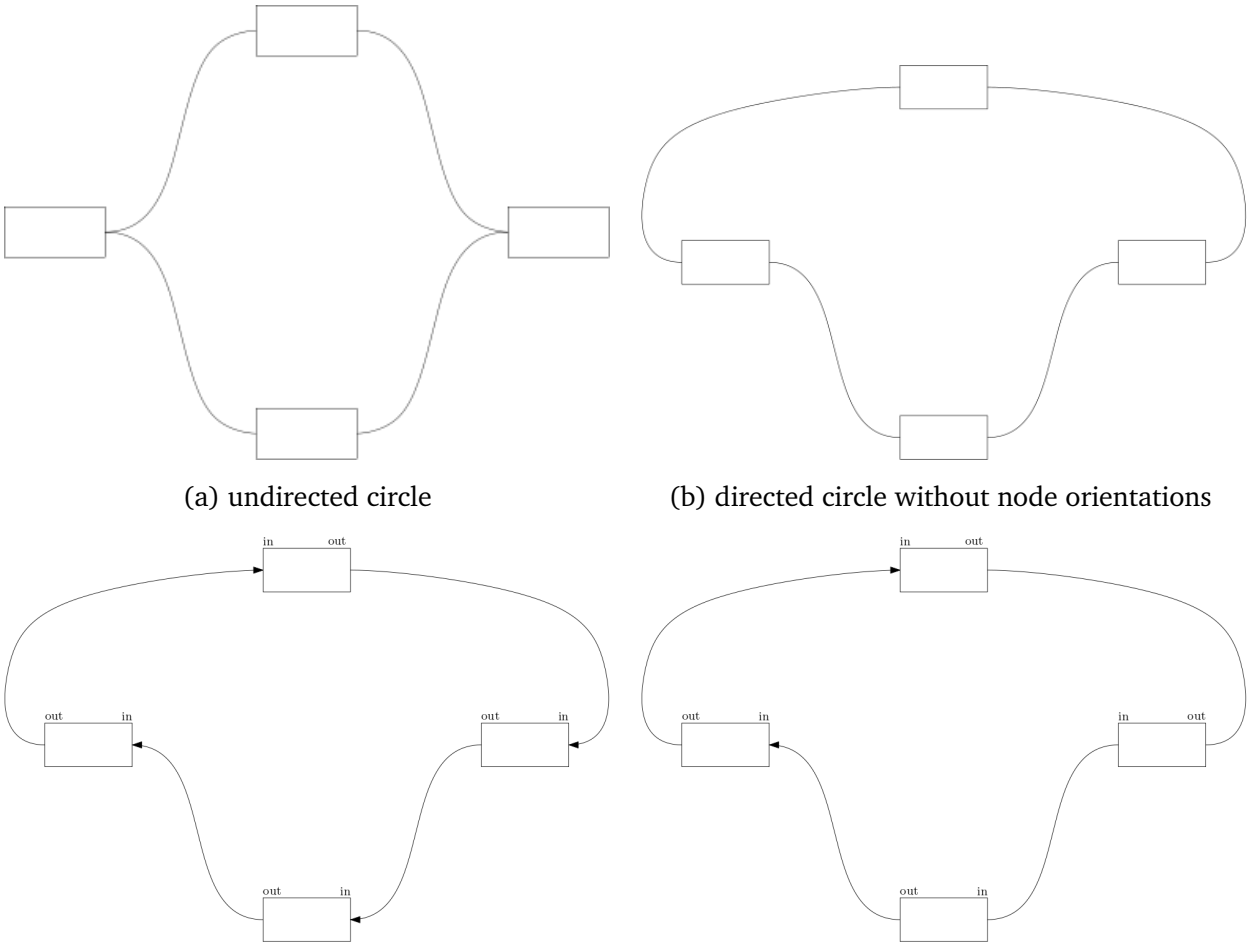
$$a_v + a_w + \rho_e \leqslant 2 \qquad \forall e \in E, \text{ with } e = (v, s, w, s'), \text{ if } s \neq s' \tag{18}$$

$$\sum_{j=1}^{m} c_{ij}(\varphi_{e_j} + \rho_{e_j}) \geqslant 1 \qquad \forall i \in \{1, \dots, l\} \tag{19}$$

4.1 Feedback arcs

Let us now have a closer look at how feedback arcs come to be. In the following, we will prove that all feedback arcs in pareto optimal solutions are edges in directed circles in the bidirected graph (Theorem 4.4, see Figure 9). We also show that in any linearization there is at least one reversing join or feedback arc in each directed circle of the bidirected graph (Lemma 4.3), and that one reversing join or feedback arc per directed circle suffices (Lemma 4.5). In our ILP constraint (19) ensures that at least one edge on each directed circle in G is a reversing join or a feedback arc.

When the ILP returns the sets of reversing joins and feedback arcs, we can, by ignoring these edges, construct an acyclic directed graph and obtain an ordering using a Kahn's topological sorting Algorithm (Kahn 1962). Because the Algorithm uses breadth-first search it can reduce the average cut width. Haussler et al. compared the Algorithm with the Flow Procedure



(a) undirected circle

(b) directed circle without node orientations

(c) directed circle with a feedback arc

(d) directed circle with two reversing joins

Figure 9: In (a) we see an undirected circle. We can ignore these circles, since they do not need to contain any feedback arcs. In (b) we can see a directed circle before the linearization. In (c) the same circle is depicted with node orientations. Every ordering would produce a feedback arc. In (d) the nodes have been oriented in such a way that there are two reversing joins.

We will show that in each directed circle, there is at least one reversing join or feedback arc.

described in their paper and Kahn's Algorithm proved to be decent but not as good as the Flow Procedure in reducing the average cut width (Haussler et al. 2018).

Construction of a directed graph using a bidirected graph and an orienting function

Given a bidirected graph $G = (V, E)$ and an orienting function a, we construct a directed graph $D = (V, A)$ by ignoring all reversing joins $\mathrm{P} \subset E$ (Haussler et al. 2018). All other edges $e \in E$ connect an out-side of node $v \in V$ with the in-side of node $u \in V$ and we can add the directed edge $r = (v, u)$ to A. This arc r is made from e and accordingly, we define the parent function $p : A \rightarrow E - \mathrm{P}$, which assigns a parent to each arc in D. In this case $p(r) = e$.

Lemma 4.1. All circles in a directed graph constructed from a bidirected graph G and orienting function a are directed circles in G

Proof: Given a directed graph $D = (V, A)$ constructed from a bidirected graph G and an orienting function a. We suggest that all circles $c_d = r_1, \ldots, r_l, r_i \in A$ in D are directed circles in G, meaning that $c_g = p(r_1), \ldots, p(r_l)$ is a directed circle in G.
This follows from the transformation of bidirected graphs into directed graphs. Let us look at the arcs in $c_d = r_1, \ldots r_l$. Since c_d is a circle in a directed graph, for each $r_i = (v_i, u_i)$ and $r_{i+1} = (v_{i+1}, u_{i+1})$, $i \in \{1, \ldots, l-1\}$, $u_i = v_{i+1}$ holds; r_i is directed towards u_i and r_{i+1} is directed away from u_i. This also holds for the two consecutive edges r_l and r_0.
For each edge in c_d there is a parent edge in G. Let $c_g = p(r_1), \ldots, p(r_l)$ be the sequence of these parent edges in E. For two consecutive edges $p(r_i) = (v_i, s_i, u_i, s'_i)$ and $p(r_{i+1}) = (v_{i+1}, s_{i+1}, u_{i+1}, s'_{i+1})$, $u_i = v_{i+1}$ holds.
Because r_i enters u_i and r_{i+1} exits u_i, as stated above, $p(r_i)$ has to be connected to the side s'_i of u_i that is labelled as the in-side and $p(r_{i+1})$ has to be connected to the side s_{i+1} of u_i that is labelled as the out-side. Therefore, $s'_i \neq s_{i+1}$ holds. This also holds for the two consecutive edges $p(r_l)$ and $p(r_0)$.
So c_g is a directed circle in G.

Note that the inverse does not hold; not every directed circle in G produces a circle in its directed child graph. We saw an example of this in Figure 9 (d).

Lemma 4.2. A topological ordering of an acyclic directed graph which was constructed from a bidirected graph G and an orienting function a, is an ordering for G that does not produce feedback arcs

Proof: Consider an acyclic directed graph $D = (V, A)$ with parent graph $G = (V, E)$ and orienting function a. Let us partition the set of edges E in the bidirected graph G into reversing joins P and non reversing joins F. Since we did not order the nodes in G, we can not yet determine which edges in F are feedback arcs and which edges are forward arcs. We know that reversing joins can not be feedback arcs and we also know that in the construction on the directed graph D, all reversing joins P were ignored.

Let ord be an ordering function that was obtained by a topological sorting of D. Since ord is a topological sorting of D, for each arc $a \in A, a = (v, u)$, v precedes u in the ordering (Cormen et al. 2001). We can use the same ordering for the bidirected graph G and in the following we will show that using this ordering, all edges in F are forward arcs and therefore G does not contain any feedback arcs.

There is a one-to-one relation between parent edges $e \in F$ and child arcs $r \in A$. So for each arc $r \in A$, there is a parent edge $p(r) \in F$ with $p(r) = (v, s_v, u, s_u)$. Because of the construction of D side s_v must be an out-side and s_u and in-side. So e connects the out-side of v with the in-side of u and since we use the same ordering we obtained from topological sort, v precedes u and thus $p(r)$ is a forward arc.

Lemma 4.3. In a linearization of a bidirected graph there is at least one reversing join or feedback arc in each directed circle of the bidirected graph

Proof by contradiction : Let us assume, that there are neither reversing joins nor feedback arcs in a directed circle of G. Let v be the vertex in the circle with the highest ordering. This vertex has at least one edge on each side which is connected to a vertex on the circle. Let e be the edge connected to the out-side of v and let v' be the other vertex on the circle connected to e. Then, since e is not a reversing join, e must be connected to the in-side of v'. But since $ord(v)$ was maximal in the circle, $ord(v) > ord(v')$ holds and thus e is a feedback arc, a contradiction.

Theorem 4.4. All feedback arcs in pareto optimal linearizations of a bidirected graph G are part of directed circles in G

Proof by contradiction : Given a bidirected graph $G = (V, E)$ and a pareto optimal linearization consisting of an orienting function a and an ordering function ord. Let $\Phi \subset E$ be the set of feedback arcs in the linearized graph. Let us now assume that there exist feedback arcs $\Phi_0 \subset \Phi, \Phi_0 \neq \emptyset$ which are not part of directed circles in G. Let $\Phi_c = \Phi - \Phi_0$ be the feedback arcs which are part of directed circles in G.

Every circle in a directed graph was constructed from a directed circle in its parent graph (Lemma 4.1). For every directed circle in G, there is at least one reversing join or feedback arc (Lemma 4.3). In the construction of a directed graph we do ignore all reversing joins. Let us also ignore all feedback arcs Φ_c that are part of directed circles in G for the construction of D. In the construction of D, for each directed circle in G, we ignored at least one edge. So D is acyclic.

We can now obtain an ordering ord' of the nodes in D using topological sort. Because of Lemma 4.2, this ordering is a valid ordering for G, in which the feedback arcs Φ_0, which were not part of directed circles in G, are now forward arcs. We can use the same orienting function a for the new linearization (a, ord'), which has the same amount of weighted reversing joins, since a did not change, and less weighted feedback arcs. This means that the original linearization (a, ord) was not a pareto optimal solution, a contradiction.

Corollary 4.5. There is an ordering for a bidirected graph $G = (V, E)$ where one edge in every directed circle is either a reversing join or a feedback arc

Proof: For an orienting function a, let $K = \mathrm{P} \cup \Phi$ be a set of reversing joins P and additional edges $\Phi \in E$ so that for each directed circle in G at least one edge on the circle is in K. The edges in K cover each directed circle in G. Let us construct a directed graph $D = (V, A)$ from the bidirected graph G and the orienting function a, while ignoring all edges in K. Then, by Lemma 4.1, D is acyclic and, by Lemma 4.2, the topological ordering of D produces no feedback arcs in G. Because of Lemma 4.3, if a circle is not covered by a reversing join but by an edge $e \in \Phi$, e must be a feedback arc.

Construction of the circle matrix

In order to enumerate all simple directed circles in G, we first find all biconnected components in G and then we find all simple directed circles in each of these components. For a detailed description, see Section 5.

4.2 Reversing joins

We use the same method as above to ensure that $\rho_e = 0$ implies that e is not a reversing join. Since we minimized reversing joins in the objective function, this effectively led to an equivalence.

In this approach we can see that inequality $\sum_{j=1}^{m} c_{ij}\varphi_{e_j} + \sum_{j=1}^{m} c_{ij}\rho_{e_j} \geqslant 1$ (19) will be satisfied if either a φ_e or a ρ_e on a circle is 1. For $\alpha < \beta$ (reversing joins are weight less than feedback arcs) the Algorithm could set ρ_e to 1 in order to satisfy (19), even if e is not a reversing join, but a feedback arc. To prevent this problem we add (13), (14), (17) and (18), which ensure that $\rho_e = 1$ implies that e is indeed a reversing join.

Let us start with the case of $s = s'$. As mentioned above, any edge $e = (v, s, w, s')$ is a reversing join, if and only if $a_v = a_w$. So if $\rho_e = 1$, a_v must be equal to a_w. This is ensured by $a_v - a_w - \rho_e \geqslant -1$ (13) and $a_v - a_w + \rho_e \leqslant 1$ (14). If $a_v = a_w$, $a_v - a_w = 0$ holds and both inequalities are satisfied regardless of ρ_e. If on the other hand $a_v \neq a_w$, there are two cases: In the case of $a_v = 0$ and $a_w = 1$, the first inequality is not satisfied, if $\rho_e = 1$. In the case of $a_v = 1$ and $a_w = 0$, the second inequality is not satisfied, if $\rho_e = 1$. So $\rho_e = 1$ implies $a_v = a_w$.

Now let us look at the case of $s \neq s'$. $a_v + a_w - \rho_e \geqslant 0$ (17) and $a_v + a_w + \rho_e \leqslant 2$ (18) ensure that $\rho_e = 1$ implies that e is a reversing join, meaning, in this case, $a_v \neq a_w$. Again, we notice, that if $a_v \neq a_w$, $a_v + a_w = 1$ and both inequalities are satisfied. If, on the other hand, $\rho_e = 1$ and $a_v = a_w$, one of the two inequalities will not be satisfied.

Instead of adding these additional constraints, one could also merge ρ_e and φ_e into a single variable which is equal to one if e is a feedback arc or a reversing join and which is equal to

zero if e is a forward arc. However, using this approach we would no longer be able to control which one of the metrics should be more important with α and β.

5 Algorithms on bidirected graphs

I was not able to find any research about the Algorithms on bidirected graphs we need to solve our problem.

5.1 Biconnected components in bidirected graphs

We can use the exact same definition of biconnected components Tarjan used for undirected graphs (Tarjan 1972).

Definition 12. Biconnected components

"Let $G = (V, E)$ be a [bidirected] graph. Suppose that for each triple of distinct nodes v, w, a in V, there is a path p [from v to w] such that a is not on the path p. Then G is biconnected. [...] If, on the other hand, there is a triple $v, w, a \in V$ such that a is on any path [from v to w], and there exists at least one such path, then a is called an articulation point of G." (Tarjan 1972). A subgraph of G is called a biconnected component if and only if it is biconnected and it is no proper subgraph of a biconnected subgraph in G (Tarjan 1972).

We can use Tarjan's Algorithm (Tarjan 1972) to find all maximal biconnected subgraphs by transforming the bidirected graph into an undirected graph ignoring the sides of the nodes.

5.2 Enumerating directed circles in bidirected graphs

In Appendix B an Algorithm which enumerates all simple directed circles in a bidirected graph is described.

A directed graph contains $O(2^n)$ simple circles (Johnson 1975). Bidirected graphs are generalizations of directed graphs, so they can contain at least as many simple circles as directed graphs, which made enumerating all simple directed circles intractable in most of the cases I encountered.

6 ILP with lazy constraint generation

We can not enumerate all directed circles in most cases. A method to solve such instances was developed by Baharev et al. for the feedback arc set problem on directed graphs. The circle matrix is build iteratively and the respective constraints are added lazily to the ILP (Baharev et al. 2021). We can use the same Algorithm with some minor changes for our problem, see Algorithm 1 (by Baharev et al. 2021, the modifications are discussed below).

Algorithm 1: Finding a linearization using Integer Linear Programming with lazy constraint generation

Function `find_linearization`$(G = (V, E), \alpha, \beta)$:

  Let $\hat{y}$, consisting of an orienting function a and a set of feedback arcs F, denote the best feasible solution found so far.

  Find a feasible solution with orientation function a and ordering function ord.

  Let $F^{(0)} \subset E$ be the feedback arcs in this solution. Set $\hat{y}$ to $(a, F^{(0)})$.

  Let $w_r, w_f \in \mathbb{N}$ be the amount of weighted reversing joins and feedback arcs in the solution.

  Set the lower bound $\underline{z}$ to 0 and the upper bound $\bar{z}$ to $\alpha w_r + \beta w_f$. The lower bound must not be associated with a feasible solution, the upper bound is associated with the best feasible solution $\hat{y}$.

  Let D be the directed graph constructed from a and G.

  Let $C^{(i)}$ denote the incomplete circle matrix, $C^{(0)}$ is an empty circle matrix.

  call $extend\_circle\_matrix(D, C^{(0)})$ to get the first circle matrix $C^{(1)}$.

  for $i \leftarrow 1, 2, \ldots$ **do**

    Solve the relaxed Problem $\tilde{P}^{(i)}$ with circle matrix $C^{(i)}$ and get the solution $y^{(i)} = (a, S)$, where a is an ordering function, S is a set of feedback arcs and R the set of reversing joins. Let w_r and w_f be the amount of weighted reversing joins and feedback arcs.

    The objective value, which does not have to be associated with a feasible solution, since we used the lazily generated circle matrix $C^{(i)}$, is $z^{(i)} = \alpha w_r + \beta w_f$.

    Set the lower bound $\underline{z}$ to $\max(\underline{z}, z^{(i)})$.

    if $\underline{z}$ *equals* $\bar{z}$ **then stop**, $\hat{y}$ is optimal.

    Let $G^{(i)}$ denote the bidirected graph obtained by removing all feedback arcs S and reversing joins R from G.

    Let $D^{(i)}$ denote the directed graph constructed from $G^{(i)}$ and a.

    if $D^{(i)}$ *can be topologically sorted* **then stop**, $y^{(i)}$ is optimal.

    For the directed graph $D^{(i)}$, compute a feedback arc set $A^{(i)}$ with total weight $\tilde{w}_f$, using a heuristic.

    Let $F^{(i)}$ be the parent edges in $G^{(i)}$ of the feedback arcs $A^{(i)}$.

    $\tilde{y}^{(i)} \leftarrow (a, F^{(i)} \cup S)$ `// ` $\tilde{y}^{(i)}$ `is now a feasible solution`

    $\hat{z} \leftarrow z^{(i)} + \beta \tilde{w}_f$

    if $\hat{z} < \bar{z}$ **then**

      $\bar{z} \leftarrow \hat{z}$

      $\hat{y} \leftarrow \tilde{y}^{(i)}$

    call $extend\_circle\_matrix(D^{(i)}, C^{(i)})$ to get the extended circle matrix $C^{(i+1)}$.

Algorithm 2: Extend the circle matrix

```
Function extend_circle_matrix(D, C):
    SCCS ← set of maximal strongly connected components in the directed graph D
    forall SCC ∈ SCCS do
        A ← heuristic set of feedback arcs in SCC
        forall a ∈ A do
            find the shortest circle c containing a in SCC using breadth-first search,
              starting at the node a is directed towards
            if such a circle c exists then
                Add a new row r to the circle matrix C corresponding to c, if r is not
                  already in the matrix.
```

6.1 Description

First, we need to heuristically find a feasible solution to the problem (Baharev et al. 2021). In our case, we start by determining a node orienting function a that minimizes reversing joins. For this task, we can use the part responsible for reversing joins of our first ILP from Section 3. From the orienting function and G, we can construct a directed graph D. We then obtain the initial circle matrix with Algorithm 2. Circles in directed graphs are part of strongly connected components (Johnson 1975). Algorithm 2 partitions the graph into strongly connected components, calculates a feedback arc set for each of these components and for each feedback arc uses breadth-first search to find a circle. Note that Baharev et al. also uses breadth-first search to find a circle for each feedback arc, but they do not split the graph into strongly connected components before calculating the feedback arc set (Baharev et al. 2021).

Other ways to extend the circle matrix or to initialize the circle matrix are possible.

Let P denote the problem with complete circle matrix and $\tilde{P}^{(i)}$ denote the relaxed problem in iteration i of the Algorithm with circle matrix $C^{(i)}$ (Baharev et al. 2021). In our case, the solution of the relaxed problem $\tilde{P}^{(i)}$ is an orienting function a and a set of feedback arcs S. Since $C^{(i)}$ most likely does not contain all directed circles of G, some directed circles in G might not be covered by the set of reversing joins and feedback arcs produced by the solution to the relaxed problem. Let $G^{(i)}$ denote G after removing all feedback arcs and reversing joins. In the case of the feedback arc set problem on directed graphs, one would check if the directed graph is acyclic after removing all feedback arcs found for the relaxed problem (Baharev et al. 2021). On bidirected graphs, in order to check whether $G^{(i)}$ does not contain any directed circles, we construct the directed graph $D^{(i)}$ using a and $G^{(i)}$. If $D^{(i)}$ is acyclic, $G^{(i)}$ does not contain directed circles (note that this statement is not true because of Lemma 4.1 ; it follows from the fact that $G^{(i)}$ does not contain reversing joins) and thus all directed circles in G were covered and the solution is feasible. Since the solution was optimal for the relaxed problem,

it must also be optimal for the problem P with complete circle matrix and the Algorithm is stopped (Baharev et al. 2021).
Otherwise, if $D^{(i)}$ contains circles, we use Algorithm 2 to extend the circle matrix (Baharev et al. 2021). $G^{(i)}$ contains only forward arcs, so all parent edges of the arcs in $D^{(i)}$ are forward arcs. This means that all parent edges in G of the circles found in Algorithm 2 are forward arcs. Since the solution to $\tilde{P}^{(i)}$ was feasible, all circles in $C^{(i)}$ contained at least one reversing join or feedback arc and therefore all new circles are not in $C^{(i)}$. In each iteration we add at least one new circle or stop the Algorithm and the number of simple directed circles in bidirected graphs is finite. So Algorithm 1 does halt.

In the implementation, whenever a feasible, but not necessarily optimal, solution to the relaxed problem $\tilde{P}^{(i)}$ was found, the graph G is checked for directed circles which were not covered. If there exist such circles, the solution is not feasible for P and new circles are added to the circle matrix. Otherwise, no circles are added and if the solution was optimal, an optimal solution to P has been found. The ILP solver keeps track of the bounds and the best solution.

7 A problem of the circle matrix ILP

A general problem about the ILP is that it does not care about the ordering. Weighted reversing joins and feedback arcs are reduced while covering each directed circle with at least one reversing join or feedback arc; the ordering is obtained after the ILP . We specifically choose Kahn's topological sort Algorithm (Kahn 1962) in hopes of reducing the average cut width. But the node orientations which determine the direction of the edges are already set and influence the average cut width. Figures 10 and 11 illustrate this problem.

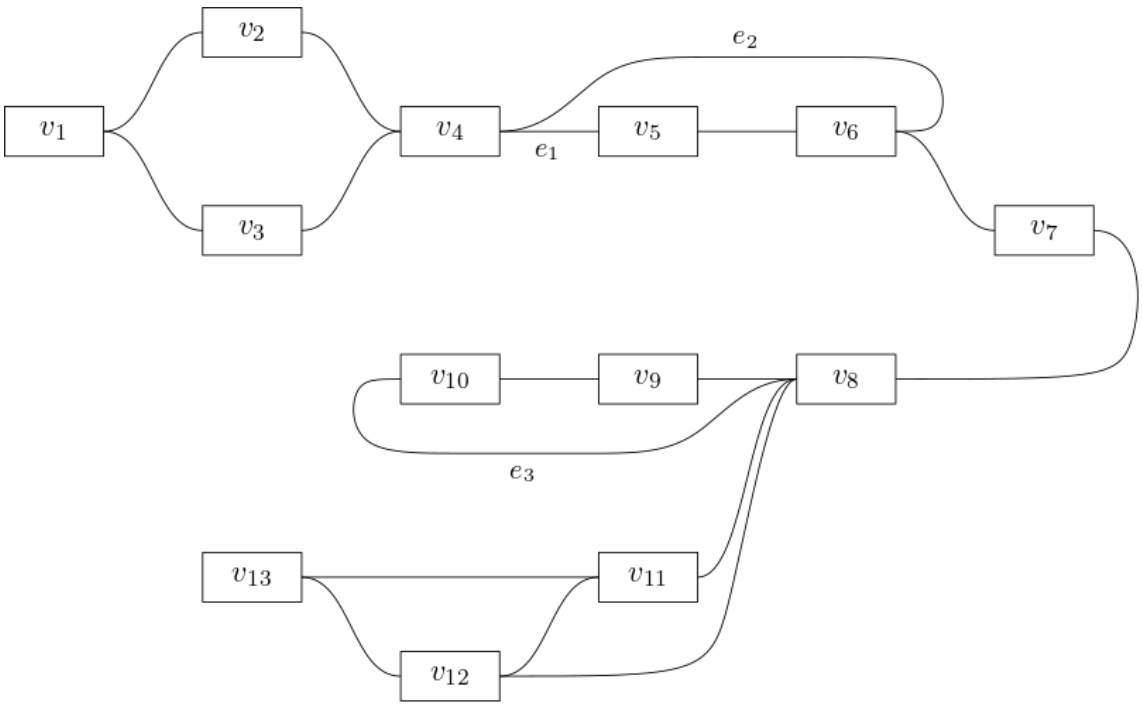
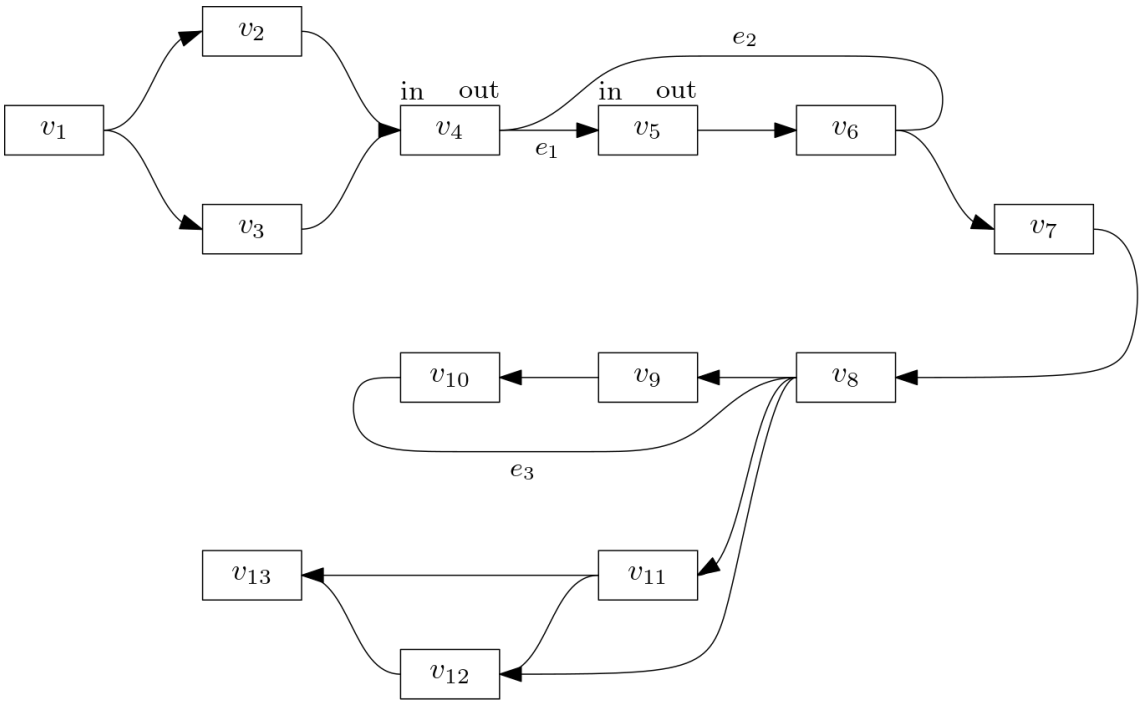
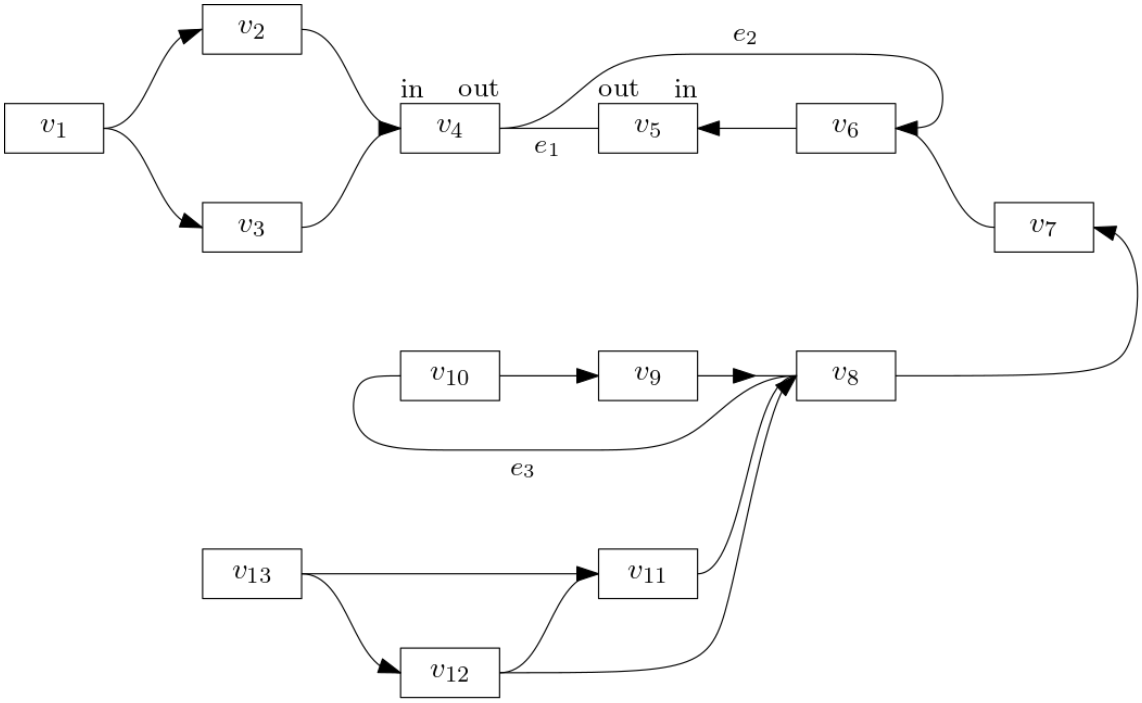


Figure 10: A bidirected graph with no directed circles.



(a) e_2 and e_3 are reversing joins



(b) e_1 and e_3 are reversing joins

Figure 11: The graph from Figure 10 with different node orientation functions in (a) and (b) which both produce two reversing joins. The arrows imply the node orientations and they are specifically shown for v_4 and v_5. To the ILP , both node orientation functions are equally good, but (b) would produce a higher average cut width when ordering the nodes.

7.1 Proposed solution

Let us explore a heuristic method which sets the node orientation for some nodes before the ILP . The purpose of this idea is to reduce the average cut width.
First, we split the bidirected graph into biconnected components. Each edge is part of exactly one biconnected component and articulation points are nodes that are part of more than one biconnected component (Tarjan 1972). We construct an undirected graph, the nodes of which consist of the articulation points and the biconnected components. There is an edge between an articulation point and a biconnected component if and only if the articulation point is part of the biconnected component. The resulting undirected graph is a tree, since if it contained a circle, the biconnected components on the circle could be combined into a larger biconnected component, so the original biconnected components would not have been maximal.

Starting at any biconnected component in the tree, we can traverse it using breadth-first search. For every articulation point n we find during this search, let b be the last biconnected component visited. If b has at least one edge connected to the right and one edge connected to the left side of a, all biconnected components containing a are merged into an extended biconnected component. Otherwise, let in be the side to which all edges in b adjacent to a are connected to. For any other biconnected component b_n which contains a, if b_n contains an edge connected to side in of a, again all biconnected components containing a are merged into an extended biconnected component.
If merging was not needed, the orientation of a is set such that in is labelled as the in-side of a. See Figure 12 and Figure 13 for an example.

Additionally, the graph is split and each component is linearized separately. This can, especially for large graphs, improve the runtime. In the results section we refer to this idea as the "ILP with tree" whereas the ILP with lazy constraint generation without setting some node orientations is refered to as "ILP".

After the ILPs determine the remaining node orientations and return sets of feedback arcs, the tree containing the biconnected and extended biconnected components is again traversed using breadth-first search and each component is ordered using Kahn's topological sort Algorithm (Kahn 1962).

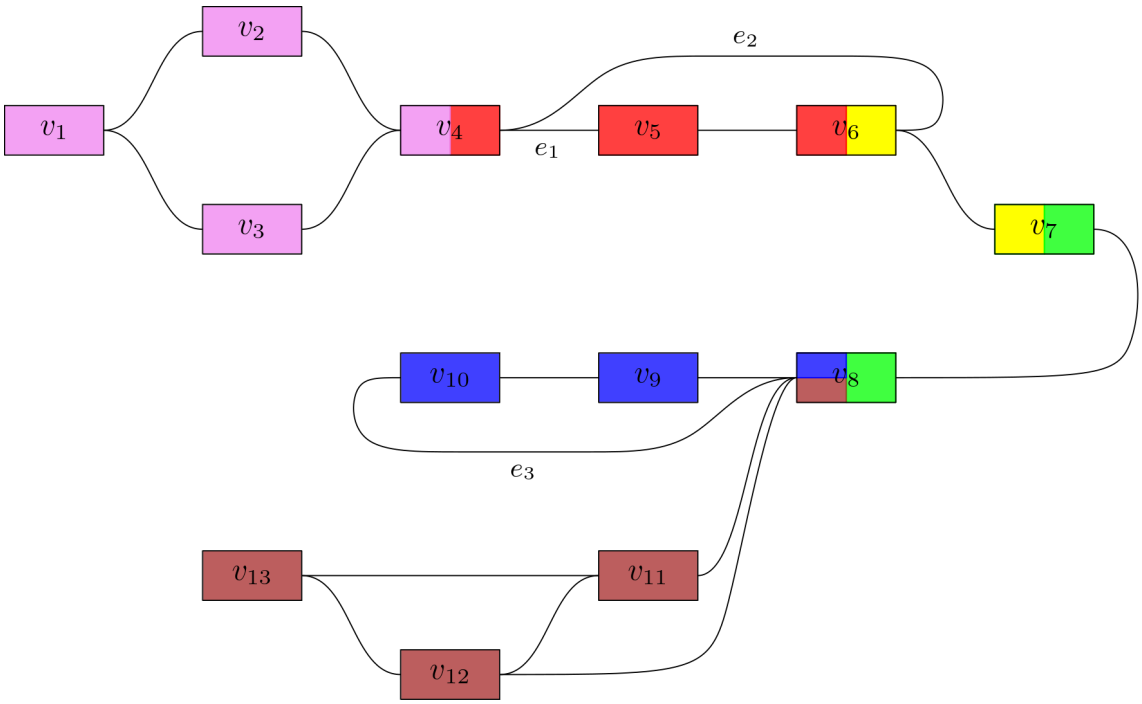


Figure 12: The graph is split into biconnected components. The articulation points which are part of multiple biconnected components are v_4, v_6, v_7 and v_8.

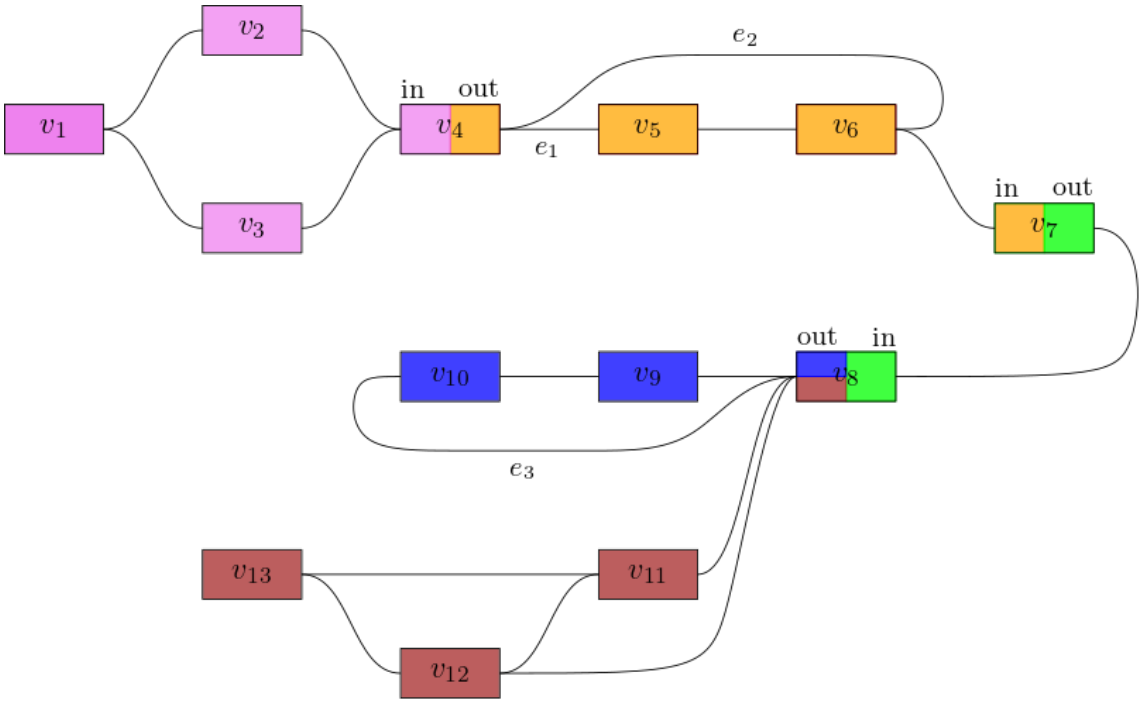


Figure 13: We start the breadth-first search at the component containing v_1. The red and yellow biconnected components are merged into the orange extended biconnected component because the red biconnected component had one edge connected to the left side of v_6 and one edge connected to the right side of v_6. Each articulation point that is left is given an orientation. This can prevent the problem described earlier.

8 NP-Hardness

Both the feedback arc set problem (FAS) (Karp 1972) and the average cut width problem (Gavril 1977; Haussler et al. 2018) are NP-hard on directed graphs.

8.1 The feedback arc set problem on bidirected graphs

Definition 13. The definition of the feedback arc problem by Karp

Does a set of less than k arcs exist, the deletion of which breaks all circles in a directed graph? (Karp 1972)

Definition 14. Definition 13 is equivalent to:

Does an ordering ord exist, such that for less than k arcs $a = (v,u)$, $ord(v) > ord(u)$ holds? (equivalent because a directed graph has a topological sorting if and only if it is acyclic)

Definition 15. The feedback arc set problem on bidirected graphs (FAS_B)

Given a bidirected graph $G = (V,E)$ and $k \in \mathbb{N}$ for which an orienting function a exists, such that there are less than $x \in \mathbb{N}$ reversing joins. Does an ordering function ord exist, such that the linearization (a, ord) produces less than k feedback arcs?

Theorem. $FAS \leqslant_p FAS_B$

Proof: Given an instance for the feedback arc set problem, a directed graph $D = (V,A)$ and $k \in \mathbb{N}$. Let us create a bidirected graph $G = (V,E)$ with $E = \{(v, right, u, left) \mid (v,u) \in A\}$. The orienting function a with $a(v) = 0, \forall v \in V$ produces 0 reversing joins.
A feedback arc in the linearization (a, ord) of G is a feedback arc in D. So there exists a feedback arc set for D with less than k arcs if and only if there exists and ordering ord for G which produces less than k feedback arcs in G with orienting function a.

8.2 Average cut width

Neither the sides of the nodes nor the direction of edges affect the average cut width. The average cut width is NP-hard on directed graphs (Gavril 1977; Haussler et al. 2018), so it is also NP-hard on bidirected graphs. The reduction would be similar to the one in the above proof.

9 Results

I used the same method to create simulated data as Haussler et al. and Lisiecka and Dojer did (Haussler et al. 2018; Lisiecka and Dojer 2021). A number of structural variations were applied to a fragment of the human genome using RSVSim from Bioconductor (*RSVSim* n.d.). The random variations include deletions, insertions, inversions and duplications of lengths 20, 20, 200 and 500, respectively. These variations were applied separately 10 times to the base DNA sequence to obtain 10 different random DNA sequences with a fixed number of variations in each sequence. Then, these 10 DNA sequences were used to create a genome sequence graph using the msga command of the vg tool (*vg tool* n.d.). The file format was changed from .vg to .gfa using the view command. This procedure was repeated 7 times with the number of variations ranging from 5 to 11 to obtain in total 7 different genome sequence graphs (Haussler et al. 2018). The graphs with fewer variations are in general less complex, thus have fewer nodes and edges.

9.1 Comparison

Four Algorithms were tested. ALIBI by Lisiecka and Dojer (Lisiecka and Dojer 2021), the Flow Procedure (FP) by Haussler et al. (Haussler et al. 2018) and the ILP with lazy constraint generation with and without setting the orientation of some nodes using the tree of extended biconnected components in both cases with parameters $\alpha = \beta = 1$ (ILP with tree and ILP).
The ILP minimizes the linear combination of weighted reversing joins and weighted feedback arcs. In Figures 14 and 15 one can see that concerning weighted reversing joins and feedback arcs the heuristic by Lisiecka and Dojer is close to the results of the ILP . Also, setting some node orientations in the ILP with tree apparently does lead to some additional reversing joins and feedback arcs.

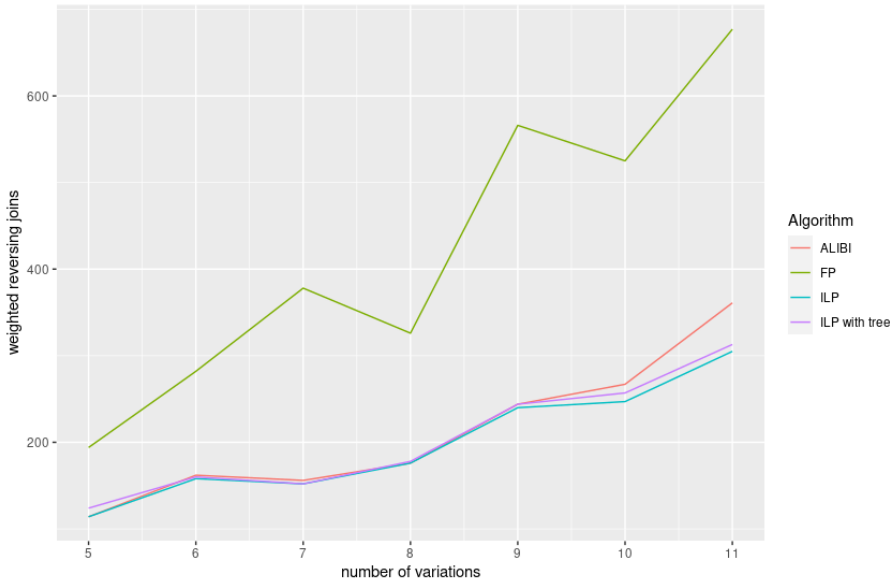


Figure 14: weighted reversing joins vs number of variations

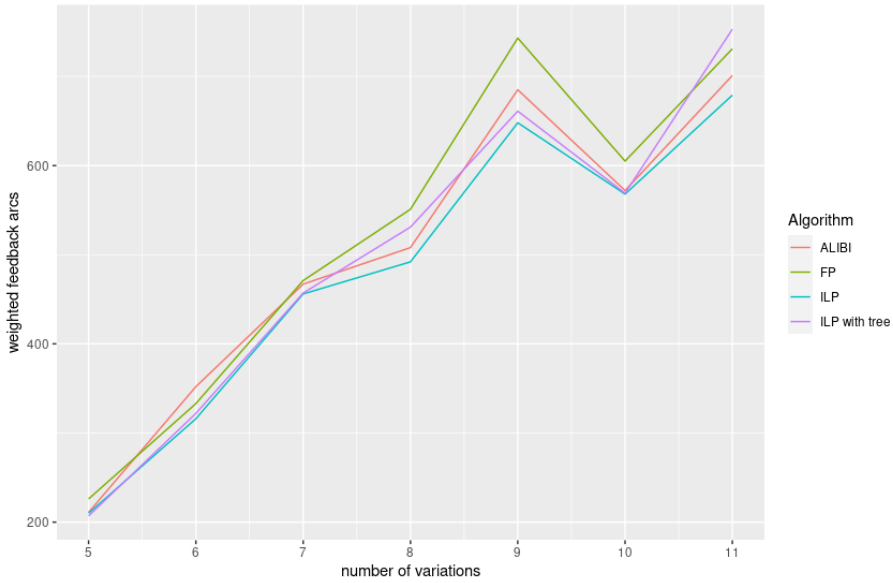


Figure 15: weighted feedback arcs vs number of variations

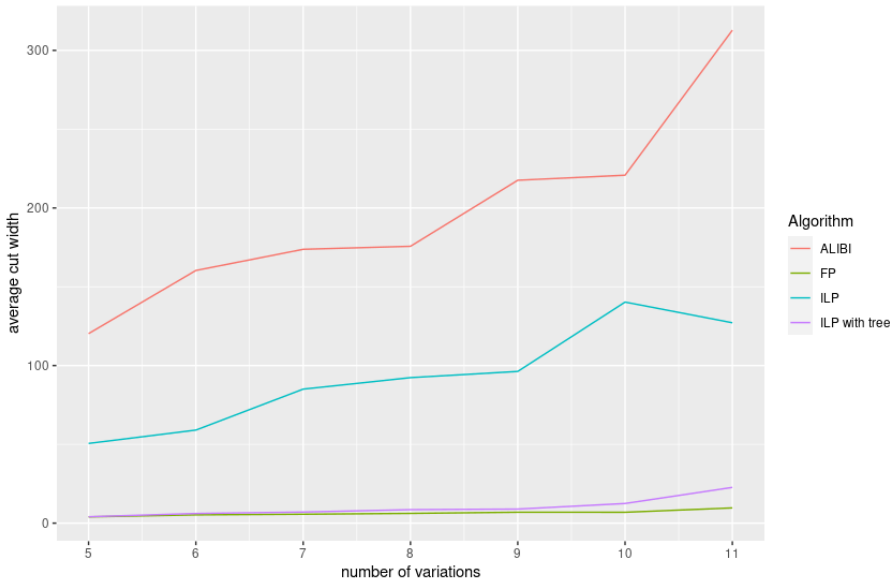


Figure 16: average cut width vs number of variations

The Flow Procedure produces the lowest average cut width, see Figure 16. We were indeed able to reduce the average cut width by setting some node orientations before the ILP . The focus of ALIBI does not lie on reducing the average cut width (Lisiecka and Dojer 2021).
As expected the ILP is significantly slower than ALIBI and the Flow Procedure, however the ILP with tree has acceptable runtime on the graphs we encountered. Note the logarithmic scale in Figure 17. The programs were started with the time command and the user time is shown. Since the ILP solver uses multithreading, on machines with several cores the real time is significantly lower. Refer to Table 3 in the Appendix for detailed results.

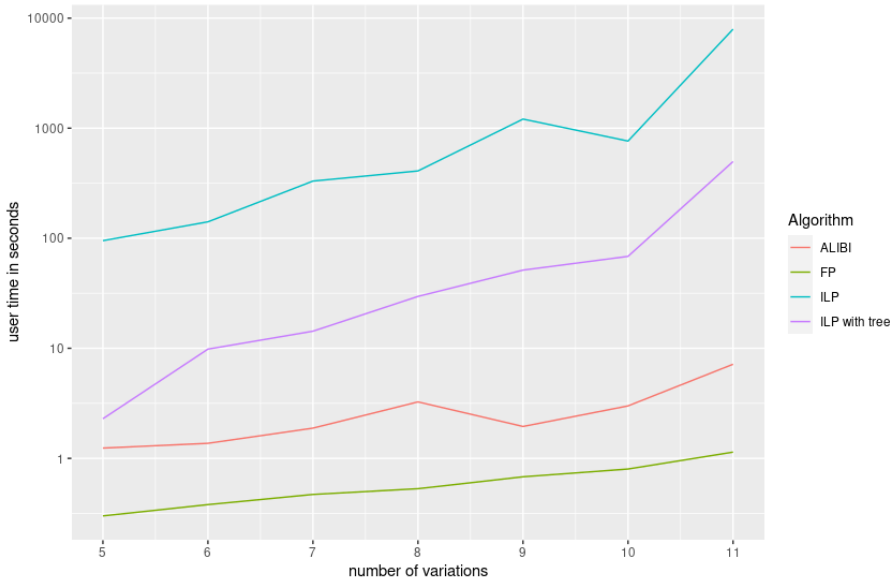


Figure 17: user time in seconds (logarithmic scale) vs number of variations

The Algorithms were implemented in Python 3 and tested on Ubuntu version 20.04.4 with an AMD Ryzen 9 5900X processor with base clock speed of 3.7GHz, 12 cores and 24 threads. The ILPs were solved with Gurobi version 9.5.1 (Gurobi Optimization, LLC 2022).

9.2 Pareto front

To get an idea of how the pareto front for the graph with 5 variations on each DNA sequence might look like, a set of solutions with parameters $\{(\alpha, \beta) | \beta = 10 - \alpha, \alpha \in \{0, 1, \ldots, 10\}\}$ was calculated. The graph had 2641 nodes, 3227 edges and 92597 directed circles. The circle matrix ILP without setting some node orientations using the tree of extended biconnected components returns pareto optimal solutions if $\alpha > 0$ and $\beta > 0$. In Figure 18 and Table 2 we can see the weighted reversing joins and feedback arcs for the solutions.
We see that both metrics can not be reduced below a certain threshold. The minimum of weighted reversing joins is reached even before the weight β for weighted feedback arcs is set to 0.

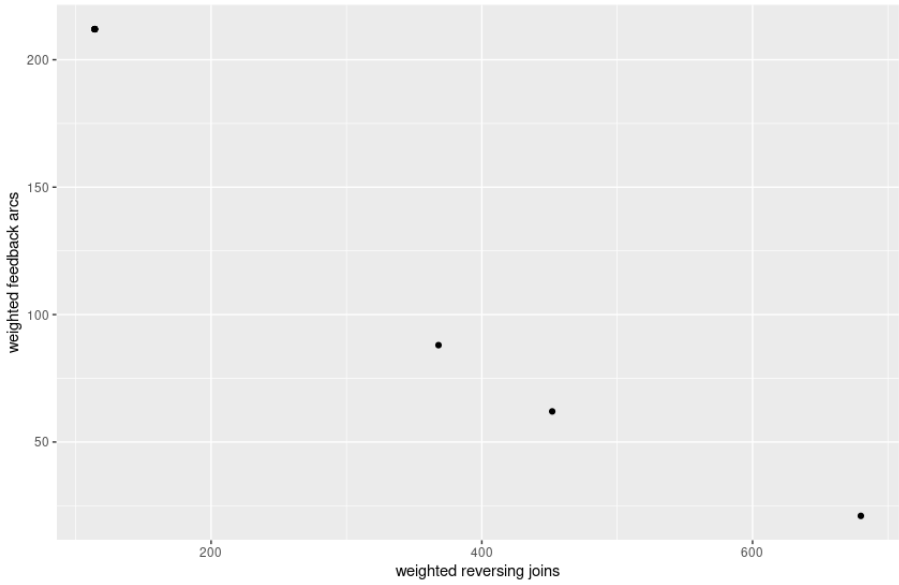


Figure 18: The pareto optimal solutions from Table 2 for the graph with 5 variations. The first and last row of the table are excluded. There is a trade-off between reversing joins and feedback arcs.

α	β	weighted reversing joins	weighted feedback arcs
0	10	1884	4
1	9	680	21
2	8	452	62
3	7	368	88
4	6	114	212
5	5	114	212
6	4	114	212
7	3	114	212
8	2	114	212
9	1	114	212
10	0	114	10809

Table 2: Weighted reversing joins and feedback arcs for different sets of parameters α and β for the graph with 5 variations. For $\alpha, \beta > 0$, the Algorithm returns pareto optimal solutions. Clearly, the solution for $(\alpha, \beta) = (10, 0)$ is not pareto optimal.

10 Conclusion

An Integer Linear Programming formulation which minimizes a linear combination of all three metrics was presented. Further reductions are needed to render the first ILP presented in this thesis feasible on large graphs.

The ILP using a circle matrix and lazy constraint generation minimized a linear combination of weighted reversing joins and feedback arcs. It was shown that the heuristic by Lisiecka and Dojer (Lisiecka and Dojer 2021) reduces these two metrics nearly as well as the ILP . Through α and β, a weight can be assigned to the two metrics. Varying these parameters gave us an idea of what the pareto front might look like; there is a trade-off between the two metrics but both metrics can be reduced simultaneously.

Additionally, a heuristic was explored which was able to significantly lower the average cut width and the running time.

The problem can be solved in reasonable time using Integer Linear Programming with a circle matrix and lazy constraint generation with satisfactory results.

References

[1] "Computational pan-genomics: status, promises and challenges". In: *Briefings in bioinformatics* 19.1 (2018), pp. 118–135.

[2] Erik Garrison et al. "Variation graph toolkit improves read mapping by representing genetic variation in the reference". In: *Nature biotechnology* 36.9 (2018), pp. 875–879.

[3] Débora YC Brandt et al. "Mapping bias overestimates reference allele frequencies at the HLA genes in the 1000 genomes project phase I data". In: *G3: Genes, Genomes, Genetics* 5.5 (2015), pp. 931–941.

[4] Anna Lisiecka and Norbert Dojer. "Linearization of genome sequence graphs revisited". In: *Iscience* 24.7 (2021).

[5] David Haussler et al. "A flow procedure for linearization of genome sequence graphs". In: *Journal of Computational Biology* 25.7 (2018), pp. 664–676.

[6] Benedict Paten et al. "Superbubbles, ultrabubbles, and cacti". In: *Journal of Computational Biology* 25.7 (2018), pp. 649–663.

[7] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. "A cutting plane algorithm for the linear ordering problem". In: *Operations research* 32.6 (1984), pp. 1195–1220.

[8] Ali Baharev et al. "An exact method for the minimum feedback arc set problem". In: *Journal of Experimental Algorithmics (JEA)* 26 (2021), pp. 1–28.

[9] John E Mitchell and Brian Borchers. "Solving linear ordering problems with a combined interior point/simplex cutting plane algorithm". In: *High performance optimization*. Springer, 2000, pp. 349–366.

[10] Arthur B Kahn. "Topological sorting of large networks". In: *Communications of the ACM* 5.11 (1962), pp. 558–562.

[11] Thomas H Cormen et al. "Introduction to algorithms second edition". In: *The Knuth-Morris-Pratt Algorithm* (2001).

[12] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[13] Donald B Johnson. "Finding all the elementary circuits of a directed graph". In: *SIAM Journal on Computing* 4.1 (1975), pp. 77–84.

[14] Richard M Karp. "Reducibility among combinatorial problems". In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[15] Fanica Gavril. "Some NP-complete problems on graphs". In: *Proceedings of the 11th conference on information sciences and systems* (1977), pp. 91–95.

[16] *RSVSim*. URL: https://www.bioconductor.org/packages/release/bioc/html/RSVSim.html.

[17] *vg tool*. Accessed: 2022-07-30. URL: https://github.com/vgteam/vg.

[18] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2022. URL: https://www.gurobi.com.

[19] Etienne Birmelé et al. "Optimal listing of cycles and st-paths in undirected graphs". In: *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. SIAM. 2013, pp. 1884–1896.

Appendix

A Reductions for the ILP using ordering variables

A possible heuristic could be splitting the graph into biconnected components and introducing set of ordering variables for each biconnected component. This works because there are no edges connecting two nodes from different biconnected components (Tarjan 1972). Also, it seems that in genome sequence graphs, biconnected components mostly only have two articulation points and therefore the ordering of biconnected components is clear.

The reductions proved to be insufficient. There were biconnected components with several hundred nodes.

Haussler mentions, that further decomposition of the graph before the linearization would be possible (Haussler et al. 2018). The author refers to the paper "Superbubbles, ultrabubbles and cacti" (Paten et al. 2018). A decomposition of the biconnected components into ultrabubbles might be possible.

B Enumerating directed circles in bidirected graphs

We will use directed depth-first search. In the context of directed paths we will talk about the insides and outsides of vertices on these paths. This must not be confused with the in-side and out-sides given to each vertex by the orienting function.

The Algorithm for enumerating all directed circles in a bidirected graph presented here is similar to an Algorithm enumerating all circles in a directed graph, discovered by Johnson and the structure of the proof will also be similar to Johnson's (Johnson 1975). However, the output of the Algorithm described here are the edges of each directed circle, since in bidirected graphs there might be several circles with identical vertices (see Figure 19) and because we need edges for the construction of a circle matrix.

Johnson splits the graph into maximal strongly connected components before searching for circles, since there are no circles that span over more than one maximal strongly connected component (Johnson 1975). When searching for circles in undirected graphs, one would split the graph into maximal biconnected components, since there are no circles that are part of more than one biconnected component (Birmelé et al. 2013). This is also true for directed circles in bidirected graphs, so $find\_circles$ can be applied to each biconnected component separately.

Theoretically, we could even split the graph into smaller components by considering directed biconnected components. Directed biconnected components are defined just as biconnected component, where in addition, the path p must be directed. There are no directed circles that span more than one directed biconnected component. An Algorithm for finding all directed

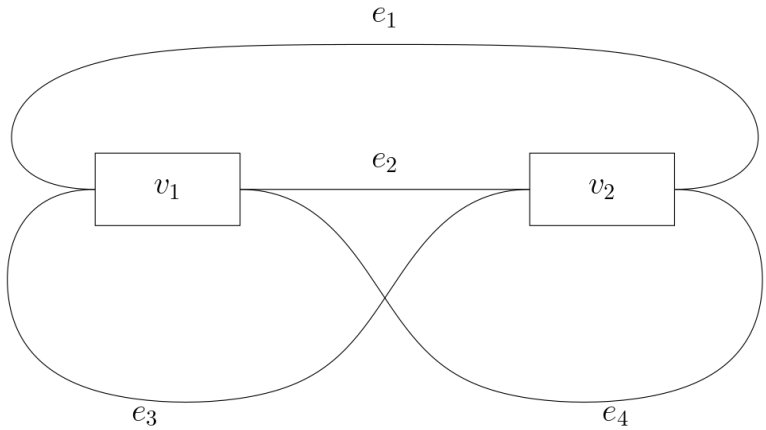


Figure 19: There are two directed circles consisting of vertices v_1, v_2, v_1. The first one is e_1, e_2 and the second one is e_3, e_4.

biconnected components in a bidirected graph is, to my knowledge, still to be discovered.

Also, a vertex v can not be part of a directed circle, if it does not have edges on both sides which are in the same biconnected component.

In the following we will prove the correctness of Algorithm 3.

The current path is always represented by the edges on the edge stack and the vertices on the vertex stack. If a part of a path did not lead to a circle, we block the insides of its vertices to prevent unnecessary searches in this part of the graph. The path might not have found a circle, because it encountered a blocked vertex. If this vertex is unblocked, this path should be considered again. The blocked set makes sure of it; if a side s of a vertex v is on the blocked set of the side s' of a vertex w, it means, that the Algorithm visited v from side s, found no circle and w with side s' was one of its neighbours. So, when side s' of w is unblocked, a new path from s of v to the start vertex might have opened up and thus, side s of v is unblocked. The vertex stack ensures that the path is simple. In an implementation, the order of its vertices is not relevant. It could be implemented as a set or as an array to provide access in constant time.

Lemma B.1. No edge occurs more than once on the stack

Proof: Without loss of generality we can assume that in line 23 in a call of $circuit(v, in_v)$, $e = (v, \neg in_v, w, in_w)$ is added to the edge stack. In line 13, in the call of $circuit(v, in_v)$, v was added to the vertex stack and after e is added to the edge stack, in the call of $circuit(w, in_w)$, w is added to the vertex stack, too. From line 22 we can see that an edge e' can only be added to the edge stack, if one of its vertices is not on the vertex stack. While e is on the edge stack, both v and w are on the vertex stack. So no edge can be added to the edge stack, when it is already on the edge stack.

Algorithm 3: Enumerate all simple directed circles in a bidirected Graph

```
Function find_circles(G = (V, E)):
    if V ≠ ∅ then
        start ← a vertex in V
        if start has at least one edge on both sides then
            blocked_right ← [false for v ∈ V], blocked_left ← [false for v ∈ V]
            blocked_set_right ← [{} for v ∈ V], blocked_set_left ← [{} for v ∈ V]
            in_start ← left
            circuit(start, in_start)
        G' ← G[V \ start]          // G' is the induced subgraph by V \ start
        forall biconnected components B̃ = (Ṽ, Ẽ) of G' do
            find_circles(B̃)

Function circuit(v, in):
    add v to the vertex stack
    found ← false
    blocked_right[v] = blocked_left[v] ← true
    out ← ¬in
    directed_edges ← {e ∈ E | e = (v, out, w, in_w), w ∈ V, in_w ∈ {left, right}}
    forall e = (v, out, w, in_w) ∈ directed_edges do
        if w == start and in_w == start_in then
            output new circle consisting of the edges on the edge stack and e
            found ← true
        else if ¬blocked_{in_w}[w] and w is not in vertex stack then
            add e to the edge stack
            found ← circuit(w, in_w) or found
            pop e from the edge stack
    if found then unblock(v, in)
    else
        forall e = (v, out, w, in_w) ∈ directed_edges do
            add (v, in) to blocked_set_{in_w}[w]
    unblock(v, out)
    remove v from the vertex stack
    return found

Function unblock(v, s):
    blocked_s[v] ← false
    forall (w, s_w) ∈ blocked_set_s[v] do
        unblock(w, s_w)
        remove (w, s_w) from blocked_set_s[v]
```

Lemma B.2. The Algorithm reports only directed circles

Proof: We use directed depth-first search; an edge can only be put on the stack if it is connected to the outside of the current vertex v. The Algorithm reports a circle only if a directed edge is connected to the inside of start.

Corollary B.3. The Algorithm reports only simple directed circles

Proof: Follows directly from Lemma B.1, Lemma B.2 and the fact that the circles that are found, are constructed from the edges on the stack. The vertex stack ensures that no vertex is traversed more than once.

Lemma B.4. In any call of $circuit$ the edge stack is identical at the start of the call, at the end of the call and in the beginning of each iteration of the for-loop in line 18

Proof: This Lemma holds for the calls associated with the leaf nodes of the depth-first search tree. Therefore, this Lemma holds for all calls of $circuit$.

Lemma B.5. The Algorithm calls $circuit(v_{k+1}, in_{k+1})$. Let there be a simple directed circle $c = e_1, \ldots, e_l$ with $e_i = (v_i, \neg in_i, v_{i+1}, in_{i+1}), i \in \{1, \ldots l-1\}$. The edge stack is $s_e = e_1, \ldots, e_k$ and the vertex stack is $s_v = v_1, \ldots, v_k$. Also $blocked_{in_i}[v_i]$ is $false$ for all $i \in \{k+2, \ldots, l\}$. Then, e_{k+1} will next be added to the stack and when it is added to the stack, $blocked_{in_i}[v_i] = false$ still holds for all unvisited vertices on the circle v_i, $i \in \{k+2, \ldots, l\}$.

Figure 20 is a complementary illustration for this proof.

Proof by contradiction: Let us assume that the Lemma did not fail so far.
We will show, that in the beginning of each iteration of the for loop in the call $circuit(v_{k+1}, in_{k+1})$, $blocked_{in_i}[v_i] = false$ holds for all $i \in \{k+2, \ldots, l\}$. Thus, when e_{k+1} is considered, it is added to the stack and, by Lemma B.4, then the edge stack is $s_e = e_1, \ldots, e_k, e_{k+1}$.

Let us now assume the contrary; there is a vertex v_t, $t \in \{k+2, \ldots, l\}$ on our path, the side in_t of which is blocked at the start of an iteration of the for loop. It can not be the first iteration, because it was not blocked when $circuit(v_{k+2}, in_{k+2})$ is called. So, for a directed neighbour n, in_n of v_{k+1} at some point during the call of $circuit(n, in_n)$, in_t of v_t must have been blocked and must not have been unblocked so far. That means there was a call of $circuit(v_t, s_1)$, $s_1 \in \{left, right\}$ in which in_t of v_t was blocked. The call did return, since v_t is not on the vertex stack. If $s_1 = \neg in_t$, in line 30 in_t would have been unblocked. So in order for v_t, in_t to have stayed blocked, it must have been a call of $circuit(v_t, in_t)$, in which no circle was found. But we know, that there is a directed path from $\neg in_t$ of v_t to the inside of v_1 and that the insides of this path were not blocked, before the call of $circuit(n, in_n)$. So, on the directed path from n to v_t, there must be a vertex v_r which is on the path from v_t to v_1, so $r \in \{t+1, \ldots, l\}$. Let v_r be the first vertex on the circle such that this condition is met. Let $w_1, \ldots, w_b$ with $w_1 = v_t$ and $w_b = v_r$ be the vertices on the path that was taken from v_t to v_r

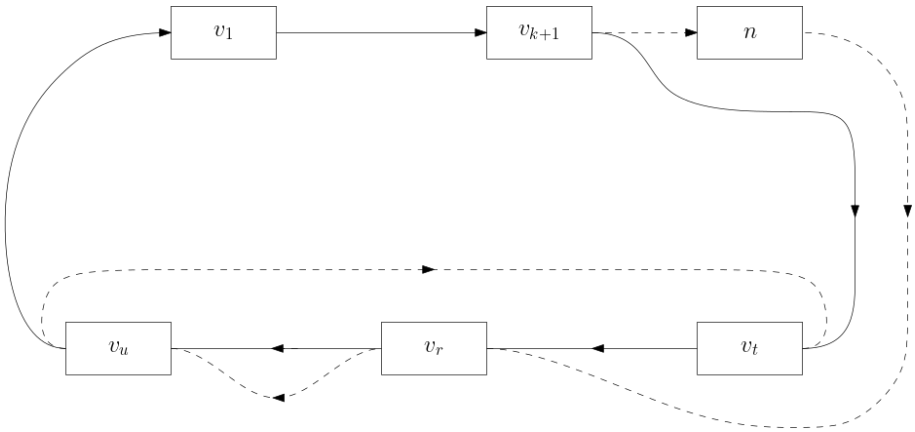


Figure 20: An illustration showing the idea behind the proof in Lemma B.5. The circle starting on the right side of v_1 is drawn solid. The path which was taken from v_{i+1} to v_t is dashed. Here we show the case of $v_r \neq v_u$ and the path from n to v_t uses the same insides as the circle. Also, here vertex n is not on the path from v_{i+1} to v_t and the path $w_1, \ldots, w_b$ from v_t to v_r is on the circle path.

with insides $w\_in_1, \ldots w\_in_b$. These vertices might be $v_t, \ldots, v_r$. Since no circle was found in the call of $circuit(v_t, in_t)$, no circle was found in the respective calls of the vertices of this path. Therefore, for $j \in \{2, \ldots, b\}$, (w_{j-1}, w_in_{j-1}) is added to $blocked\_set_{w\_in_j}[w_j]$, in particular (v_{b-1}, w_in_{b-1}) is added to $blocked\_set_{in_r}[v_r]$. A call of $unblock(v_r, in_r)$ would trigger a chain of calls $unblock(w_{b-1}, w\_in_{b-1}), unblock(w_{b-2}, w\_in_{b-2}), \ldots, unblock(v_t, in_t)$ which would unblock the inside of v_t and thus lead to a contradiction.
Let us look at the call $circuit(v_r, s_2)$. If $s_2 = \neg in_r$, in line 30, there would have been a call $unblock(v_r, in_r)$. So $s_2 = in_r$. Again there is a directed path from $\neg in_r$ to to inside of v_1. If a circle is found, there is a call $unblock(v_r, in_r)$ in line 26. So no circle is found and therefore the inside of at least one of the vertices on the path from v_r to s must be blocked. Among these, choose v_u with highest index u. Once again, due to the workings of the $blocked\_set$, when in_u of v_u is unblocked, a call of $unblock(v_r, in_r)$ follows, which in turn would unblock the inside of v_t. Since before the call of $circuit(n, in_n)$ the insides of the vertices from v_{i+1} to v_1 were all unblocked, in_u of v_u must have been blocked during the call of $circuit(n, in_n)$ and it is on the path from n to v_t which contains v_r. Finally, in the call of $circuit(v_u, s_3)$ we can not circumvent a contradiction. In the case of $s_3 = \neg in_u$, there is a call $unblock(v_u, in_u)$ in line 30. In the case of $s_3 = in_u$, there is a directed path from the outside of u to the inside of v_1, on which no insides are blocked because all the insides of all vertices v_j, $u < j \leqslant l$ are not blocked. So assuming that this Lemma did not fail before a circle is found, a contradiction to the asserting we made earlier, that at v_r no circle was found.

Theorem B.6. The Algorithm reports every simple directed circle exactly once

Proof: No circuit is output more than once since for any edge stack $s_e = e_1, \ldots e_i$ with e_i on top, once e_i is removed, s_e can not reoccur. Once every directed circle containing the start nodes is enumerated, the start node is removed from the graph.
Let $c = e_1, \ldots, e_l$ be a simple directed circle. c is part of a biconnected component. Let the

edges be $e_i = (v_i, \neg in_i, v_{i+1}, in_{i+1}), i \in \{1, \dots l-1\}$ and $e_l = (v_l, \neg in_l, v_1, in_1)$ such that on the directed path, v_i is entered through in_i. Without loss of generality, we can assume that there is a call $circuit(v_1, in_1)$, where $in_1 = left$.
In the following, we will prove by induction that the Algorithm builds a stack $s_e = e_1, \dots, e_l$, adding e_i to the stack in a call of $circuit(v_i, in_i)$ and afterwards in the case of $i < l$ calling $circuit(v_{i+1}, in_{i+1})$ and in the case of $i = l$, reporting the circle. In the end of each step, the insides of the remaining vertices on the path, in_j of v_j for $i + 1 < j \leqslant l$ are not blocked.

Base case ($i = 1$):
As mentioned above, there is a call of $circuit(v_1, in_1)$. At the start of this call, no edges are on the edge stack and no vertices are blocked. Then using Lemma B.5 with $k = 0$, we can conclude, that e_1 is put on the edge stack and no insides of vertices on the path are blocked. After this there is a call $circuit(v_2, in_2)$, in which v_2 is added to the vertex stack and in_2 of v_2 is blocked.

Induction step ($i \to i + 1$, $i < l$):
The edge stack is $e_1, \dots, e_i$, the vertex stack is $v_1, \dots, v_{i+1}$ and the insides of the remaining vertices on the path in_j of v_j, $i + 1 < j \leqslant l$ are not blocked. e_i was added to the edge stack in a call of $circuit(v_i, in_i)$ and after that $circuit(v_{i+1}, in_{i+1})$ is called. There are two cases:
If $i = l - 1$, there is an edge $e_l = (v_l, \neg in_l, v_1, in_1)$ which is considered in the for-loop and a circle including this edge is reported. Because of Lemma B.4, we know that the edge stack is identical for each iteration of the for loop, so the stack is $e_1, \dots, e_l$ when the circle is reported.
In the case of $i < l - 1$, following Lemma B.5 with $k = i$, e_{i+1} is added to the edge stack, and after the call of $circuit(v_{i+2}, in_{i+2})$, v_{i+2} is added to the vertex stack and all insides of vertices on the remaining path are not blocked.

We did show that any simple directed circle containing $start$ is reported exactly once in the call of $circuit(start, left)$. After this, we remove $start$ from G. For better performance we split the graph into biconnected components before searching for all the circles that contain the next vertex $start'$ but do not contain any of the previous start vertices.

C Results Table

variations	algorithm	# rj	wrj	# fba	wfba	acw	real time	user time	storage
5	ALIBI	102	114	130	211	120.26	0.96 s	1.24 s	37488 kb
	FP	109	194	128	226	3.96	0.30 s	0.30 s	44624 kb
	ILP	102	114	128	210	50.59	25.31 s	95.08 s	725276 kb
	ILP with tree	100	124	123	207	4.06	1.19 s	2.29 s	67768 kb
6	ALIBI	137	162	183	352	160.35	1.04 s	1.37 s	37528 kb
	FP	147	282	170	333	5.24	0.38 s	0.38 s	51632 kb
	ILP	125	158	169	316	59.06	48.76 s	140.95 s	1133216 kb
	ILP with tree	124	160	174	322	6.12	2.49 s	9.83 s	137920 kb
7	ALIBI	140	156	224	467	173.78	1.58 s	1.88 s	37484 kb
	FP	181	378	213	471	5.65	0.48 s	0.47 s	51768 kb
	ILP	136	152	214	456	85.05	105.39 s	330.68 s	1127864 kb
	ILP with tree	136	152	218	457	7.03	2.90 s	14.30 s	125540 kb
8	ALIBI	161	176	238	508	175.67	2.98 s	3.26 s	37744 kb
	FP	162	326	237	551	6.18	0.55 s	0.53 s	54116 kb
	ILP	154	176	225	492	92.34	139.67 s	408.80 s	1748736 kb
	ILP with tree	157	178	231	531	8.54	4.18 s	29.61 s	195660 kb
9	ALIBI	195	244	282	685	217.60	1.63 s	1.95 s	37644 kb
	FP	227	566	258	743	6.91	0.69 s	0.68 s	58456 kb
	ILP	183	240	256	648	96.28	266.99 s	1209.24 s	1888560 kb
	ILP with tree	177	244	257	661	8.93	6.65 s	51.32 s	226596 kb
10	ALIBI	217	267	284	572	220.80	2.73 s	3.00 s	37708 kb
	FP	242	525	263	605	6.90	0.82 s	0.80 s	61500 kb
	ILP	201	247	284	568	140.27	292.23 s	763.43 s	2047496 kb
	ILP with tree	203	257	289	569	12.48	11.75 s	68.44 s	324700 kb
11	ALIBI	278	361	411	701	312.83	6.82 s	7.17 s	37912 kb
	FP	326	677	367	731	9.67	1.14 s	1.14 s	67568 kb
	ILP	228	305	381	679	127.14	1152.01 s	7933.77 s	7170440 kb
	ILP with tree	241	313	403	753	22.69	79.96 s	497.43 s	1164428 kb

Table 3: The number of reversing joins (# rj), weighted reversing joins (wrj), number of feedback arcs (# fba), weighted feedback arcs (wfa), average cut width (acw) rounded to two decimal points, time and storage rounded to 4 decimal points for the Algorithms on the graphs with variations ranging from 5 to 11. The command /usr/bin/time -v was used to measure the time in seconds and maximum working memory (maximum resident set size).
Interestingly, in row 3 and 4 the amount of weighted feedback arcs in the ILP with tree which uses a heuristic is lower than the one of the ILP . This must not indicate a mistake. We can not ensure that both metrics reach their minimum, since the ILP is multi-objective.