Visualization of bipartite graphs in limited window size

Bachelor Thesis of

Kassian Köck

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science

In cooperation with the Departments of Computer Science

Reviewer:	Prof. Dr. Ignaz Rutter, University of Passau
Advisors:	Prof. Stephen Kobourov, University of Arizona
	Prof. Will Evans, The University of British Columbia

Time Period: November 2022 – January 2023

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Vancouver, January 29, 2023

Abstract

Visualization of bipartite graphs in easy-to-read drawings is a challenging task. Consider a network graph of international companies and all the countries they operate in. A large number of nodes affects legibility, so a good-looking drawing is required. We consider drawings in which the nodes of the two partitions are drawn on two separate curves; either two horizontal lines or two concentric circles. There are several factors that affect readability. One such factor is the length of the edges in the drawing.

Another factor is the window size, where the window of a node is the smallest interval, either x-interval or angular interval, whose image on the two curves contains both the position of the node and the position of its neighbors.

In this thesis, we use two techniques to improve these factors: From a given drawing of a bipartite graph, we either move the nodes of one partition to the positions where the window size or edge length is minimal while keeping the other partition fixed or move the nodes in both partitions. The former technique is, in most cases, feasible, while the latter is $\mathcal{NP}$-complete.

After that, we introduce an unconventional way of visualizing bipartite graphs by placing the partitions on the boundaries of an annulus. We present an algorithm that computes the best inner-to-outer-radius ratio of the two circles to draw straight edges without intersecting the inner circle.

Deutsche Zusammenfassung

Die Darstellung bipartiter Graphen in leicht lesbarer Form ist nicht immer einfach. Betrachtet man beispielsweise einen Netzwerkgrafen, der die Verbindung darstellt von internationalen Unternehmen zu den Ländern, in denen sie operieren. Eine hohe Zahl an Knoten beeinträchtigt die Lesbarkeit, sodass eine möglichst übersichtliche Visualisierung vonnöten ist. Wir betrachten Zeichnungen, in denen die zwei disjunkten Mengen auf zwei separaten Kurven dargestellt wird; entweder auf parallelen, horizontalen Linien oder auf konzentrischen Kreisen. Es gibt verschiedene Faktoren, um gute Lesbarkeit zu gewährleisten und diverse Wege, um zu diesem Ziel zu gelangen. Einer dieser Faktoren ist die Länge der Kanten in dieser Zeichnung.

Einen weiteren Faktor stellt die Fenstergröße dar. Ein Fenster eines Knotens ist das kleinstmögliche, entweder horizontale oder gekrümmte, Intervall dar, dessen Projektion auf die Darstellungskurven sowohl die den Knoten selbst, als auch seine benachbarten Knoten beinhaltet.

In dieser Arbeit verwenden wir zwei Methoden, um diese Faktoren zu verbessern. Entweder nehmen wir die Zeichnung eines bipartiten Graphen und suchen für Knoten aus einer der Partitionen die bestmöglichen Positionen während die Knoten der anderen Partition auf ihren Platzierungen verbleiben oder wir suchen bewegen alle Knoten. Die erste Methode ist in den meisten Fällen lösbar, die zweite $\mathcal{NP}$-vollständig.

Anschließend führen wir eine unkonventionelle Methode der Visualisierung bipartite Graphen ein. Dafür platzieren wir die beiden disjunkten Mengen auf zwei konzentrischen Kreisen. Wir präsentieren einen Algorithmus, um das beste Verhältnis der beiden Radien zu berechnen. Dabei soll es noch möglich sein, die Kanten der Knoten geradlinig und ohne den inneren Kreis zu schneiden, zu zeichnen.

Contents

1. Introduction

Bipartite graphs are used in many ways to display relationships between two disjoint sets. Computer scientists use them to connect users' access to items on a computer, journalists display actors and the movies they appear in, and biologists model proteins and genes [HGKW16, PKP+18, WS98].

Most bipartite graphs are visualized on two parallel lines, one for each partition, and the edges are drawn with straight lines between their adjacent nodes like in Figure 1.1 [PFH+18]. The visualization should be easy to read for extracting information as quickly as possible. However, quantifying readability criteria is non-trivial: What makes Figure 1.1(a) easier to read than Figure 1.1(b). Is it the lower number of crossings or the shorter edges? Of course, it is a mixture of both, and there are additional factors, too, like the angles in which the edges cross each other or enter and leave nodes or even the graph symmetry [HEH14, Pur02].

Battista et al. [DBETT94] call the main aesthetics:

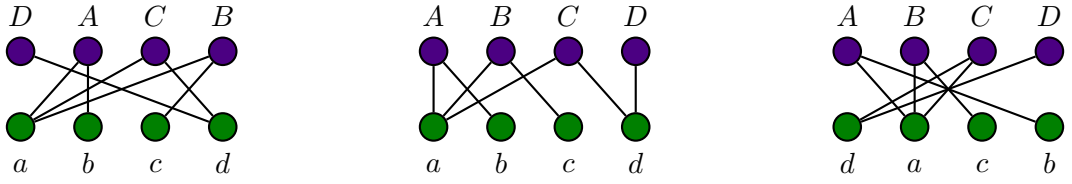
- avoid edge crossing
- avoid bends in edges
- keep edge lengths uniform
- distribute vertices uniformly

Improvements in one metric likely have a positive effect on the other. In Figure 1.1(b), the graph results from reducing the number of the crossing of Figure 1.1(a) or Figure 1.1(c), yet at the same time, the average edge length improved, which means, decreased, too. Sometimes, the optimum for one solution is also an optimum for the other, but that is not always true.

A graph with fewer edges (and nodes) is also easier to read, but we want to draw the same graph in a different way with improved readability without losing information. For that reason, in this thesis, we focus on rearranging the nodes of a graph to improve its window sizes and edge lengths. The window is defined by the maximum distance between a parent and one of its adjacent children or between two children, whether larger. We calculate edge length by projecting one layer to the other and measuring the horizontal distance of the edge's adjacent nodes.

Especially in digital reading, the graphs do not have to fit on a sheet of paper; they can have many nodes and edges [BFK+23]. One example is the *ASCT+B Reporter*[1]. It uses

[1] `https://hubmapconsortium.github.io/ccf-asct-reporter/`



(a) (b) (c)

Figure 1.1: The same bipartite graph in three drawings where the number of crossings, the edge length, and the window size differ

a particular way to improve the readability of this complex graph: Clicking on a specific node highlights it and all adjacent nodes, while the other nodes and edges become greyed out. This way, it is easy to follow a path from one node to another or look at all of a node's connections.

However, this does not help much if it is impossible to display a node with all its connections simultaneously. There may be a screen or paper size that is too small. Hence one target may be the minimization of the window size of a node, i.e., the part of the graph necessary to display the node and its adjacent vertices. In a bipartite graph with two parallel lines, we call one partition *parent* and the other *children*. The smaller a window is, the larger the graph can be visualized while still showing the parent and all their children at once. In the ASCT+B Reporter, small windows allow zooming in, thereby improving readability.

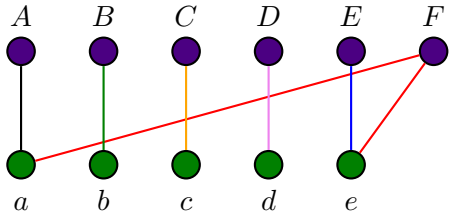
1.1 Related work

One well-examined legibility factor is the number of crossing edges in a drawing of a graph. A common task for minimizing this number for bipartite graphs is finding a drawing on parallel lines for both partitions of nodes with the smallest number of crossings [EW94]. A related method is to fix the positions of one set of nodes in a given drawing and rearrange only the other with the same goal [ÇEKS09]. Unfortunately, both have been proven to be $\mathcal{NP}$-hard [EW94, GJ83]. In this thesis, we look at several problems in minimizing window sizes and edge lengths using the same technique.

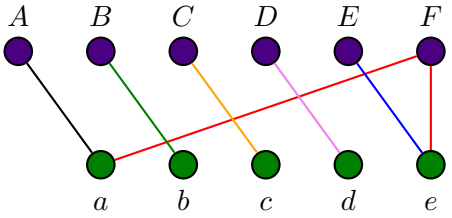
Avoiding large windows and improving the window sizes of a graph can be done in two ways: The first method is achieved by making the broadest window as thin as possible. The second method is computing the minimal sum of window sizes, i.e., the minimal average window size. Though closely related, both methods can lead to different optimal solutions. An example where the drawings minimizing the window sum and the maximum window differ is shown in Figure 1.2.

Trying to minimize window sizes has been examined by Bekos et al. [BFK$^+$23]. They wanted to calculate the smallest upper bound for windows, so they focused on minimizing the largest window. This improvement helps if one wants to guarantee that any node with its neighbors fits on a graph clip with a fixed size. They discovered that it makes a difference whether we fix the children and move the parents to the best possible positions or keep them in their positions while changing the children's locations.

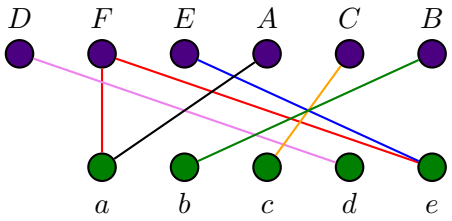
The former is solvable in polynomial time, while the latter is $\mathcal{NP}$-complete. So for minimizing the sum of the window sizes, it is reasonable to conjecture that the same relationship holds. Two slightly different problems support this conjecture, the `BandwidthProblem` and the `SimpleLinearArrangement` where the same relationship holds. These problems, described in Section 2.2, ask for a placement of nodes of a graph on a straight line at integer coordinates. In both problems, it is desired to reduce the edge lengths, i.e., the difference in



(a) A graph in a drawing with a good window size sum of $0+0+0+0+0+5$, but a larger maximum window size of $w_F = 5$



(b) A graph in a drawing with a worse window size sum of $1+1+1+1+1+4$, but smaller maximum windows size of $w_F = 4$



(c) A graph in a drawing with minimum largest window of $w_F = 4$ but much larger window size sum of $4+4+3+2+1+3 = 17$

Figure 1.2: A graph in three different drawings, the first has a minimal sum of window sizes, the second and the third have a minimal largest window. The way these values get calculated is described closer in Chapter 2. A window is the horizontal distance needed to see a node of the top layer and all its neighbors in the bottom layer.

the coordinates, to a minimum. In the `BandwidthProblem`, the longest edge is tried to be as small as possible, while the minimal sum is minimized in the `SimpleLinearArrangement`. Both are $\mathcal{NP}$-complete [GJS76, Pap76].

However, there can be reasons for not focusing on the maximum window but searching for the minimal sum of the window sizes, i.e., the minimal average window size. Consider a graph as shown in Figure 1.2, which contains many parents (nodes in the top layer) with a low number of children (nodes in the bottom layer). This graph is easy to display and read when placing the parents opposite the children, as shown in Figure 1.2(a). Nevertheless, this graph has parent F with a large window which gets smaller by shifting the parents to the left, as shown in Figure 1.2(b). The more of the latter kind of parents exist, the more the former kind will be shifted away from their children. Hence though it would be easy to display these nodes in a simple format, minimizing the maximum window can sometimes make a graph less legible. So if one puts up with a few large windows, the overall window size can be decreased, and the average parent is close to its neighbors. Additionally, the graph has a parent with a large window, so the parents with smaller window sizes can be placed without optimization. An example is illustrated in Figure 1.2(c): For algorithms that compute the minimal largest window, this drawing with the largest window size of 4 is equally minimized as the drawing in Figure 1.2(b).

So if finding the smallest maximal window size in this problem is hard, why should it be different from minimizing the sum of all the window sizes? They reduced the problem of finding a minimum maximal window size from `BandwidthProblem` as the longest-edge-minimizing problem. Thus it is plausible to reduce the sum-of-the-window-size minimizing from `SimpleLinearArrangement` as the sum-of-the-edge-length minimizing problem. However, both `BandwidthProblem` and `SimpleLinearArrangement` are $\mathcal{NP}$-complete when taking arbitrary graphs as input. When restricting the input to trees, the

longest-edge minimization problem stays hard, but the sum-of-edge-length minimization problem becomes feasible [Shi79, HM97]. Trees can be separated into two partitions: the layers with even and odd depths. So they are a particular kind of bipartite graph. It is plausible that the same difference in complexity applies to minimizing the largest window vs. minimizing the sum of the window size [ADH98].

Though drawing bipartite graphs on two parallel lines may be the most common way, it is not the only way. A generalization of parallel lines is the drawing of bipartite graphs on convex curves. Giacomo et al. [DGGL07, DGGL08] stated "that the class of bipartite graphs that admit a planar straight line drawing on two horizontal lines is entirely restricted", which motivates this visualization method. Using radial visualization to display information is, for instance, used to display intrusion detection by the approach of VisAlert [FAL$^+$06, DLR09]. Foresti et al. used radial visualization to simultaneously show the location, time, and nature of many detected attacks. In AlertWheel, a similar system, Dumas et al. [DMRW12, DRM12] used the same technique and came up with different ways to improve this way of displaying the mass of information by clustering edges.

Before we go deeper into this topic, let us look at how we want to accomplish easiness-to-read in this thesis.

1.2 Problem statement

We have to answer four questions to define a problem considering the easiness-to-read of a bipartite graph.

1. Is the graph drawn on parallel lines or concentric circles?
2. Do we focus on window sizes or edge lengths?
3. Is our goal the minimum sum of windows/edge lengths, or do we want to achieve the minimal longest edge/largest window?
4. Do we move the parents with fixed children? Or move the children and fix the parents? Or do we move both sets?

Each combination of how to answer these questions creates a problem. However, not each of the $2 \cdot 2 \cdot 2 \cdot 3 = 24$ combinations leads to a different problem, and not all variations relate to real issues. The difference between parents and children only exists in window minimizing problems. So for improving edge lengths, question five can be changed to "Do we fix one side or move both sides?".

So the actual number of distinct questions is 20. In the end, we know the answer to 13 combinations, eleven newly answered in this thesis. Eight problems are in $\mathcal{P}$, and five are $\mathcal{NP}$-hard. A quick overview gives Table 7.1 at the end of this thesis.

Additionally, some questions do not fit in this question pattern. We look at similar problems not generated by these questions, which give us a more detailed impression of the modifiability of bipartite graphs.

1.3 Outline

In Chapter 2, we define terms used in this thesis. Furthermore, we give an overview of some solved $\mathcal{NP}$-complete problems and known algorithms, which we use in the proofs later.

Chapter 3 introduces the topic of rearranging nodes to improve easiness-to-read. We begin by looking only at children first. In that regard, we show that even when neglecting the

parents, finding a drawing for a graph with a minimal span, i.e., the distribution of the children, is already complex.

Chapter 4 is the core of the thesis. It focuses on minimizing the windows of bipartite graphs in several ways. First, in Section 4.1, we provide a greedy algorithm to determine whether it is possible to move the parents so that all of them are opposite the children. If this is possible, the window sum and maximum window get as small as possible. Otherwise, we can use the algorithm in Section 4.2 to find the arrangement for the parents with minimal window sizes in both window sum and maximum window. The result in Section 4.3 is $\mathcal{NP}$-hardness for minimizing the window sum when we allow both parents and children to move.

In Chapter 5, we ask almost the same questions again, but we do not minimize windows but the edge lengths this time. The answers to the questions are nearly identical, but reusing the algorithms and proofs from the last chapter one by one is not possible. However, in Section 5.1, we modify our algorithm to find the best arrangement for edge lengths when moving one side before we adjust our hardness proof in Section 5.2 for the version where both sides move.

After that, in Chapter 6, we draw a bipartite graph not on parallel lines but on concentric circles. In Section 6.1, we adapt the algorithms again to the new circumstances in this kind of drawing. The circular version of graph drawing leaves the minimizing problem $\mathcal{NP}$-complete when permuting both sides. However, the proof needs to be adapted, given in Section 6.2. Furthermore, the concentric drawing asks a different question; "What is the minimum ratio between the inner circle and outer circle with keeping edges straight?" An algorithm for this calculation is provided in Section 6.3.

Chapter 7 recapitulates our results and highlights open problems and unproved presumptions.

2. Preliminaries

Before we dive deeper into the problems and introduce new concepts, we have to define several terms and techniques used in this thesis. To prove hardness, we need some already known $\mathcal{NP}$-complete problems that we can reduce to ours. These will be the `SimpleLinearArrangement` and the `BandwidthProblem`. Additionally, we give an overview of three algorithms that will prove helpful in devising our ones.

2.1 General Terms and Techniques

Let $G = (P \dot\cup C, E)$ be a bipartite graph with the nodes partitioned into two disjoint sets P and C. We call P the *parents* and C the *children*. Every edge in $E \subseteq P \times C$ connects a node in P with a node in C. Hence G has no multi-edges. For $p \in P$ let $C_p = \{c \mid (p, c) \in E\}$, called the *set of its (adjacent) children*.

We consider drawings of G with parents placed at distinct integer coordinates on a horizontal line and children placed the same way on a second parallel line below. We draw G by placing each parent p at $(x(p), 1)$ and each child c at $(x(c), 0)$ where $x : P \dot\cup C \to \mathbb{Z}$. So we call x a drawing of G.

Definition 2.1 (semi-injective)**.** *A drawing $x : P \dot\cup C \to \mathbb{Z}$ is called* semi-injective*, if x maps P and C injectively to $\mathbb{Z}$, respectively. Then the following holds:*

$$\forall a, b \in P \dot\cup C : x(a) = x(b) \Rightarrow (a \in P \wedge b \in C) \vee (a \in C \wedge b \in P) \vee a = b.$$

In this thesis, each drawing is semi-injective, meaning there are not multiple parents or children in a particular position. However, it is allowed for a parent and a child to have the same position.

Definition 2.2 (Span and window)**.** *For a given drawing x and parent p, let*

- $f_p = \arg\min_{c \in C_p} x(c)$ *be the* first child *of p,*
- $l_p = \arg\max_{c \in C_p} x(c)$ *be the* last child *of p,*
- $S_p = \{x(f_p), \ldots, x(l_p)\}$ *be the* span of p *with* span size $s_p = x(l_p) - x(f_p) = |S_p| - 1$,
- $W_p = \{\min\{x(f_p), x(p)\}, \ldots, \max\{x(l_p), x(p)\}\}$ *be the* window of p *with* window size $w_p = \max\{s_p, x(l_p) - x(p), x(p) - x(f_p)\} = |W_p| - 1$.

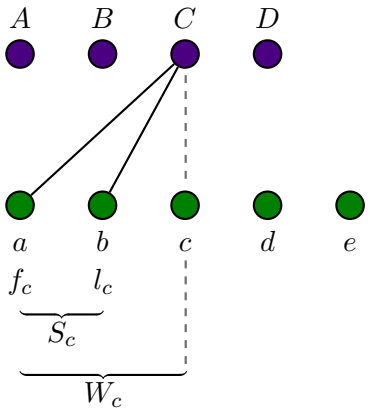


Figure 2.1: A drawing x of a graph G with parents $P = \{A, B, C, D\}$ and children $C = \{a, b, c, d, e\}$, where the span size s_C is 1 and the window size w_C is 2.

For an illustration, see Figure 2.1. So we get to the following proposition:

Proposition 2.3. *For a parent p in a drawing x the following holds:*

$$w_p = \begin{cases} s_p, & \text{if } x(p) \in S_p. \\ s_p + \min\{x(p) - x(f_p), x(l_p) - x(p)\} & \text{if } x(p) \notin S_p. \end{cases}$$

So the window size takes minimal values when the parent is placed in its span, i.e., $x(f_p) \leq x(p) \leq x(l_p)$.

In words, the window size equals the span and potentially the distance to it if placed outside.

To allow easier reading, from now on, we write for a node a in a drawing x its coordinate a instead of $x(a)$ whenever we do not have to distinguish between multiple drawings. If we want to compare two nodes a, b, then $a < b :\Leftrightarrow x(a) < x(b)$. So f_p describes not only the child itself but also its coordinate in a drawing x. We call f_p and l_p, *marginal children* and f_p and its position *beginning of the span* as well as l_p and its position *end of the span.*

Definition 2.4 (Partial and completing drawing)**.** *Suppose we are given a bipartite graph $G = (A \cup B, E)$. For a subset $A^* \subseteq A$ we call $x : B^* \to \mathbb{Z}$ a* partial drawing*. We call $y : B \to \mathbb{Z}$ a* completing drawing*.*

It is impossible to draw the whole graph with a given partial drawing, so it may be considered a function to fix the position of nodes of the graph. In short, a partial drawing of a bipartite graph only applies to a subset of one partition while a completing drawing applies to a whole partition of a bipartite graph.

In this thesis, we want to show several algorithms to improve the legibility of a given drawing x of a bipartite graph. For that reason, we search for a different y drawing of the same graph. Drawing y may inherit some properties of the given drawing x. For instance, we look for the best positions for partition B of the nodes in the graph while we inherit the positions A^* of the other partition from x. So we do not need to know the positions of partition B. It is enough to have information about A^* in a partial drawing $x : A^* \to \mathbb{Z}$.

That is why we only need partial drawings for some of the problems in the following chapters. The algorithms to solve these problems are working with these partial drawings

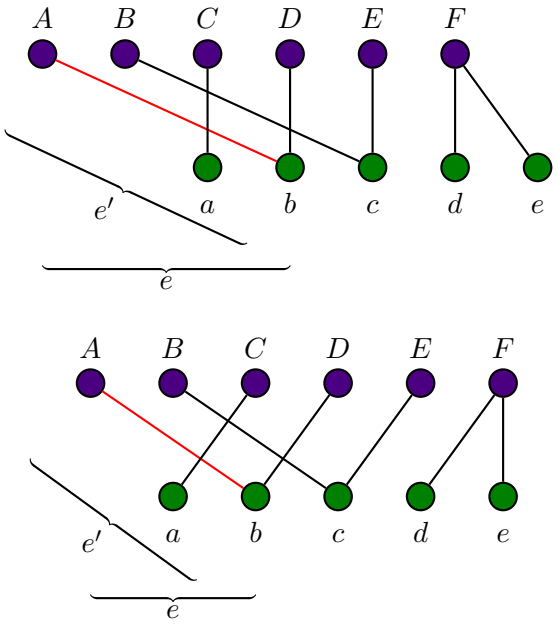


Figure 2.2: Two drawings of the same graph, where the projected edge length sum is smaller in the drawing above and the euclidean length sum is smaller in the drawing below.

and computing the completing drawings. However, since it is feasible to transform a non-partial drawing into a partial drawing, all algorithms can also be modified to work with non-partial drawings. Depending on the implementation of the drawing extracting the node on a given position or the position of a given node may not be accessible in constant time. Thus these modified algorithms may have a slower running time.

The *image* $x(B)$ of a set B in a drawing x is defined conventionally: $x(B) \coloneqq \bigcup_{b \in B} x(b)$. Additionally, we call a drawing x *gap-free*, if the following condition holds:

$$\forall a, b \in x(P \cup C) : \{a, \ldots, b\} \setminus x(P \cup C) = \emptyset.$$

Gaps are positions within the graph with neither a parent nor a child node.

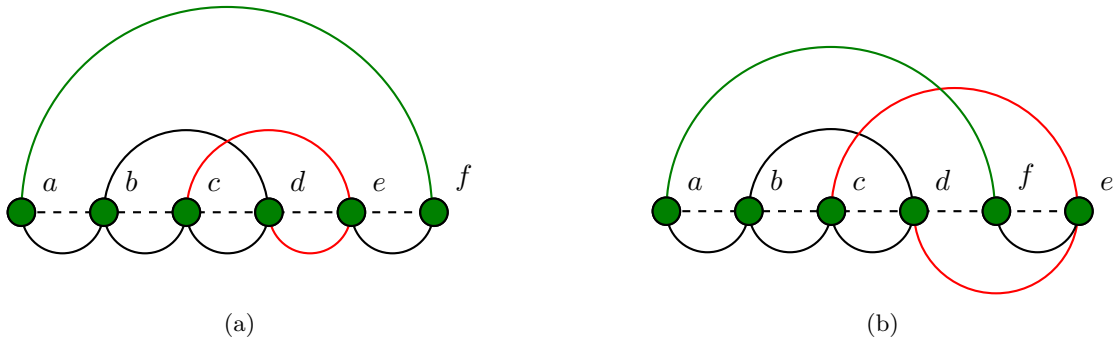
In this thesis, besides minimizing windows, we also focus on improving the lengths of edges.

Definition 2.5 (Projected edge length)**.** *Suppose we are given a drawing x of a bipartite graph $G = (P \cup C, E)$. Ror an edge $(p, c) \in E$ its* projected edge length λ *is defined as*

$$\lambda((p, c)) = |x(p) - x(c)|$$

This definition of edge length can be considered as the distance of the projected point of p onto the layer of c. In Figure 2.1 the edges have projected edge lengths of $\lambda((C, a)) = |c - a| = 2$ and $\lambda((C, b)) = |c - b| = 1$. Just like window sizes, edge lengths only assume natural numbers. This definition of edge length is related to the edge lengths as euclidean distance since converting can be done with the distance between the two parallel lines.

However, these metrics are unequal, and minimizing the edge length in the different definitions does not necessarily lead to the same result. In Figure 2.2, we present the same graph in two drawings. The projected edge length sum is smaller in the drawing in the figure above. The red edge (A, b) has a projected edge length of $e = 3$ while the euclidean



(a) (b)

Figure 2.3: A graph in two drawings, one with smaller edge lengths sum, the other with a smaller longest edge.

distance is $e' = \sqrt{3^2 + 1^2} = \sqrt{10} \approx 3.16$ (With a distance of the parallel lines of 1). The total projected edge length is $2 \cdot 3 + 4 \cdot 0 + 1 = 7$ while the euclidean edge length sum is about $2 \cdot \sqrt{10} + 4 \cdot 1 + \sqrt{2} \approx 11.74$. Meanwhile, the euclidean sum is smaller in the figure below. The projected edge sum is $2 \cdot 2 + 4 \cdot 1 + 0 = 8$ while the euclidean edge lengths are $2 \cdot \sqrt{5} + 4 \cdot \sqrt{2} + 1 = 11.29$.

However, this definition of edge length is independent of the distance of the parallel lines. Additionally, it helps us convert parallel drawings into radial drawings in Chapter 6. This is why we use this metric to compute edge lengths in this thesis.

2.2 Known $\mathcal{NP}$-complete problems

In this thesis, we use two $\mathcal{NP}$-complete problems to reduce to our problems. The problems are similar and take the same input. The goal to be achieved is, however, different.

Definition 2.6 (`SimpleLinearArrangement` (`LAP`))**.** *Suppose we are given a graph $G = (V, E)$ and an integer $W \in \mathbb{N}$. Is there an injective function $f : V \to \mathbb{Z}$ such that the following holds?*

$$\sum_{(u,v)\in E} |f(u) - f(v)| \leq W.$$

Definition 2.7 (`BandwidthProblem` (`Bandwidth`))**.** *Suppose we are given a graph $G = (V, E)$ and an integer $W \in \mathbb{N}$. Is there an injective function $f : V \to \mathbb{Z}$ such that the following holds?*

$$\forall (u, v) \in E : |f(u) - f(v)| \leq W.$$

Both `LAP` and `Bandwidth` are $\mathcal{NP}$-complete [GJS76, Pap76]. A different description of the problems is: Suppose we are given a graph. Can we place its nodes on a line with integer coordinates, such that the sum of the edge length (`LAP`) or the maximum edge length (`Bandwidth`) is upper-bounded by a particular value? At first view, they may seem the same, and though they are correlated, there are instances where the optimum of one is not the optimum of the other. In Figure 2.3, the same graph is shown in two drawings. The drawing in Figure 2.3(a) has a smaller edge length sum of $5 + 2 \cdot 2 + 5 \cdot 1 = 14$ versus $4 + 3 + 2 \cdot 2 + 4 \cdot 1 = 15$ in Figure 2.3(b). However, the longest edge in Figure 2.3(a) has a length of 5, while the longest edge in Figure 2.3(b) has a size of 4.

2.3 Known algorithms

Let $G = (A \cup B, E)$ be a bipartite graph with $n := |A| \leq |B|$. A subset $M \subseteq E$ is called *matching* if every node $v \in A \cup B$ is adjacent to at most one edge $e \in M$ [Tas12]. A Matching M is called *perfect* if every node $v' \in A$ is adjacent to exactly one edge, i.e., $|M| = n$.

Definition 2.8 (`PerfectBipartiteMatching` (`Match`))**.** *Suppose we are given a graph $G = (A \cup B, E)$, calculate $M \subseteq E$ with M as a perfect matching.*

This problem is also known under the names *assignment problem* and *marriage problem* [SSW05]. A useful observation is made by Hall [Hal98]:

Theorem 2.9 (Hall's marriage theorem)**.** *Suppose we are given a bipartite graph $G = (A \cup B, E)$. For a subset $U \subseteq A$ let $B_U = \bigcup_{u \in U} \{b \mid (u, b) \in E\}$ the neighborhood in G of U. Then exists a perfect matching if and only if for every subset $U \subseteq A$ the following is valid:*

$$|U| \leq |B_U|.$$

In particular, it follows the weaker corollary:

Corollary 2.10. *Suppose we are given a bipartite graph $G = (A \cup B, E)$. A perfect matching exists if each node $a \in A$ has at least $|A|$ adjacent nodes in B.*

For a matching M of a weighted bipartite graph, we call the sum of the weights of its edges, i.e., $\xi(M) := \sum_{e \in M} \xi(e)$ the *weight of a matching*. A *matching bottleneck* is defined by the biggest edge $\xi_{\max}(M) := \max_{e \in M} \xi(e)$.

Furthermore, given multiple perfect matchings, we add selection criteria to choose the best perfect matching. Suppose we are given a bipartite graph $G = (A \cup B, E)$ and a weight function $\xi : E \to \mathbb{N}_0$. We call $H = (A \cup B, E, \xi)$ a *weighted bipartite graph* [DP14]. So we look for the best perfect matching in two ways:

Definition 2.11 (`MinimumBipartiteMatching` (`MinMatch`))**.** *Suppose we are given a weighted bipartite graph $H = (A \cup B, E, \xi)$. Return $M \subseteq E$ with M as perfect matching with the lowest weight of the matching.*

Definition 2.12 (`BottleneckBipartiteMatching` (`BottMatch`))**.** *Given a weighted bipartite graph $H = (A \cup B, E, \xi)$. Return $M \subseteq E$ with M as perfect matching with the smallest bottleneck.*

While the unweighted version is comparably easy to solve, it gets complicated when we add weights. Solving `Match` is possible in $\mathcal{O}(\sqrt{n}m)$ by the algorithm of Peterson et al. [PL88] where $m := |E|$ is the number of edges. It is also solvable in linear time $\mathcal{O}(n)$ with the algorithm of Uno [Uno97]. However, several algorithms for `MinMatch` with different running times on different types of graphs have been published. Due to its many applications in modern times, `MinMatch` is even called "one of the fundamental problems in combinatorial optimization" [San09]. One of the first discovered algorithms is called the "Hungarian Method" [Kuh55]. The running time of this algorithm can be improved to $\mathcal{O}(n^3)$ with the help of Edmonds and Karp [EK72]. A different algorithm originates from Kao et al. [KLST99]. It uses the parameter of the total weight length $W = \sum_{e \in E} \xi(e)$ and has

Description	**Symbol**	**Definition**	**Extra description or correlation**
Bipartite graph	G	$G = (P \dot\cup C, E)$	
Weighted		$G = (A \dot\cup B, E, \xi)$	
Weight function	ξ	$\xi : E \to \mathbb{N}_0$	
Drawing	x, y	$x : V \to \mathbb{Z}$	
		$x : P \dot\cup C \to \mathbb{Z}$	For bipartite graphs
Partial		$x : V_1 \to \mathbb{Z}$	$V_1 \subseteq V$ of nodes V
Completing		$y : V_2 \to \mathbb{Z}$	$V_2 \coloneqq V \setminus V_1$

For a parent $p \in P$ of a bipartite graph with drawing x

Children of p	C_p	$\{c \in C \mid (p, c) \in E\}$	
First child / Beginning of span	f_p	$\arg\min_{c \in C_p} x(c)$	Marginal children
Last child / End of span	l_p	$\arg\max_{c \in C_p} x(c)$	
Span	S_p	$\{x(f_p), \dots, x(l_p)\}$	
Span size	s_p	$s_p = x(l_p) - x(f_p)$	$s_p = \lvert S_p \rvert - 1$
Window	W_p	$\{\min\{x(f_p), x(p)\}, \dots, \max\{x(l_p), x(p)\}\}$	$w_p = \lvert W_p \rvert - 1$
Window size	w_p	$\max\{s_p, x(l_p) - x(p), x(p) - x(f_p)\}$	$w_p = s_p +$ distance to span, if not within it
Edge length	λ	$\lambda((p, c)) = \lvert x(p) - x(c) \rvert$	projected, not euclidean

For a bipartite graph $G = (A \dot\cup B, E)$ with $|A| \leq |B|$

Matching	M	$M \subseteq E$	$v \in A \dot\cup B$ is adjacent to any $e \in M$
perfect		$\lvert M \rvert = \lvert A \rvert$	
Weight of a match.	$\xi(M)$	$\sum_{e \in M} \xi(e)$	
Bottleneck of a m.	$\xi_{\max}(M)$	$\max_{e \in M} \xi(e)$	

Table 2.1: An overview of the terms defined in this section.

a running time of $\mathcal{O}(\sqrt{n}W)$. The weight sum can grow too fast for dense graphs with high weights to more than $\mathcal{O}(n^3)$. However, one of the most recent discoveries shows the solvability of the problem in almost linear time $\mathcal{O}(m^{1+o(1)})$ [CKL+22]. Since in simple graphs, $m \leq n^2$, this leads to a maximum running time of $\mathcal{O}(n^{2+o(1)})$.

Finding the best algorithm to solve the `BottMatch` for our application is more manageable. There are fewer published algorithms since the focus lies on the other variant. Nevertheless, an algorithm has been discovered that succeeds in finding an optimal solution for this problem in $\mathcal{O}(n \cdot \sqrt{mn})$ [PN94].

Table 2.1 provides an overview of all definitions in this section for a fast lookup.

3. Minimizing the span size

Before we think about how to improve the windows of the parents, let us look at the span, i.e., the window without the parent node itself. Proposition 2.3 shows that the span marks a lower bound for the window. So minimizing the span size is the first step to improving window sizes. Unfortunately, we prove in this section that finding a drawing with minimal span sizes is $\mathcal{NP}$-hard.

For calculating the span, the parent's position does not play any role. In the later Chapters 4, 5 and 6, there are several methods for rearranging a drawing to improve windows or edge lengths. However, for span minimization, only one of them is applicable: So, given a drawing of a graph, how can we rearrange the children with fixed parents' locations for a placement with minimal spans? The other variant, fixing the children and moving the parents for the same goal, does not change the span. Combining both and allowing the parents and the children to move is the same as freezing the parents.

First, let us define the problem:

Definition 3.1 (`MinimumSpanSum` (`SpanSum`))**.** *Suppose we are given a bipartite graph $G = (P \dot\cup C, E)$ and an integer $W \in \mathbb{N}$. Is there a partial drawing $x : C \to \mathbb{Z}$, such that the following holds?*

$$\sum_{p \in P} s_p = \sum_{p \in P} (l_p - f_p) \leq W.$$

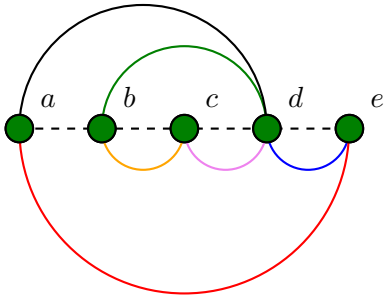
We prove that this problem is $\mathcal{NP}$-complete by reducing `LAP` (Definition 2.6) to it. For that reason, we transform an instance of this problem into our problem in the following way:

Definition 3.2 (Edge graph)**.** *Suppose we are given a graph $G = (V, E)$. Consider*

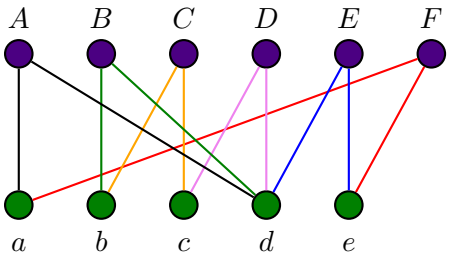
$$P \coloneqq \{p_{uv} \mid (u, v) \in E\} \text{ and}$$
$$E' \coloneqq \{(p_{uv}, u), (p_{uv}, v) \mid (u, v) \in E\}.$$

We call $H = (P \dot\cup V, E')$ edge graph *of G.*

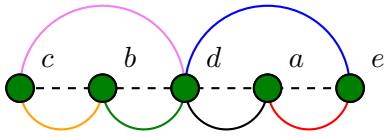
Each edge of an origin graph has a representative parent in its edge graph. This parent is connected to both adjacent nodes of the edge. Hence, the length of an edge in graph G



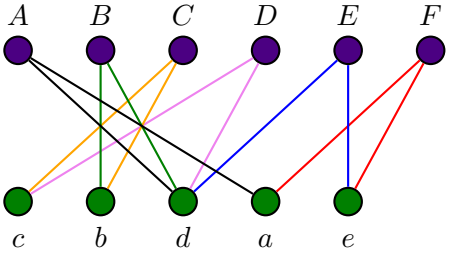
(a) A graph G in an arbitrary drawing, and the integer $W = 10$ as an instance of the LAP. In this drawing, G has an edge length sum of $12 > 10$.



(b) The construction of H, the edge graph of G. H, W with $W = 10$ is an instance of SpanSum. This ordering is the same as the drawing of G in the left graph, so the span sum $12 > 10$.



(c) The edge length sum minimizing drawing G. This ordering has a total edge length of $8 < 10$, so G, W is a solvable instance of LAP.



(d) The span sum minimizing drawing of H with a span sum of $8 < 10$. So $H, 10$ as an instance of SpanSum is solvable, too.

Figure 3.1: An instance of the LAP and its edge graph in random ordering (a) and (b) and in optimal ordering (c) and (d). The colors indicate the correlation of the edges to the parents.

equals the span size of its representative in the edge graph. So edge-length-sum minimizing in the original graph is the same problem as span-sum-minimizing for the edge graph.

Example: Suppose we are given an arbitrary graph G as an instance of LAP as shown in Figure 3.1(a). If we transform the graph as described into an edge graph, we get an instance of the SpanSum problem. Figure 3.1(b) shows the edge graph H of G. It is easy to see that the lengths of the colored edges equal the size of the span in H. These particular drawings of G and H have edge length or else span sum of $4+3+2+1+1+1 = 12$. The optimal ordering of G, as shown in Figure 3.1(c), has edge length sum $2+2+1+1+1+1 = 8$, which is the same as the span size in the optimal ordering of the children in H in Figure 3.1(d). This observation leads us to our first result:

Theorem 3.3. MinimumSpanSum *is $\mathcal{NP}$-complete.*

Proof. We first show that SpanSum is in $\mathcal{NP}$. Suppose we are given an instance H, W of SpanSum and a drawing x; we can check that the sum of the spans in the drawing x is at most W in time polynomial in the number of edges of H.

We show that we can reduce LAP to SpanSum in polynomial time. With a given instance $G = (V, E), W$ of LAP, we construct an instance H, W of SpanSum where $H = (P \uplus V, E')$ is the edge graph of G (Definition 3.2). Any drawing y of H corresponds to a drawing x of G where $x(u) = y(u)$ for all $u \in V$ and where the length $|x(u) - x(v)|$ of the edge $(u, v) \in G$ is the same as the length $|y(u) - y(v)|$ of the span of parent $p_{uv} \in P$. Thus there exists a drawing x of G with the sum of its edge lengths less than W if and only if there exists a drawing y of H with the sum of its parent-span sizes less than W.

Creating an edge graph runs in time polynomial to the number of nodes and edges. In conclusion, `SpanSum` is $\mathcal{NP}$-hard and with that $\mathcal{NP}$-complete. □

Variant: It is challenging to minimize the sum of the span sizes. However, minimizing the largest span is $\mathcal{NP}$-complete, too.

Definition 3.4 (`MinimumMaxSum` (`MaxSpan`))**.** *Suppose we are given a bipartite graph $G = (P \cup C, E)$ and an integer $W \in \mathbb{N}$. Is there a partial drawing $x : C \to \mathbb{Z}$, such that the following holds?*

$$\forall p \in P : s_p = l_p - f_p \leq W.$$

In Definition 2.7, we introduced `Bandwidth`, a problem similar to `LAP`. Both problems try finding an alignment for a graph on a line, with the edges as small as possible. While `LAP` searches for a sum-of-edge minimization, `Bandwidth` minimizes the longest edge. So we use the same steps to create with a given instance G, W of `Bandwidth` a corresponding edge graph H, that forms together with W an instance of `MaxSpan`. We already know that the span size of a parent in H equals the edge length of the edge in G it represents. Hence, a longest-edge-minimizing drawing for G corresponds to a largest-span-minimizing drawing in H.

Corollary 3.5. `MinimumMaxSum` *is $\mathcal{NP}$-complete.*

4. Minimizing the window size

Let us proceed with the core of this thesis. Unfortunately, in Chapter 3, the span minimization problem turned out to be hard in both variants, with minimizing span sum and maximum span. In this chapter, we examine if it is also hard to find drawings with minimum window sum or minimum largest window.

This target now has different approach methods we will talk about: Nevertheless, first, we provide a fast algorithm for one method. If we take an existing drawing for the children, this algorithm gives us the locations for the parents, such that their window sum is minimal. The second method, taking a bipartite graph and finding a drawing for both sets of nodes with minimal window sum, is $\mathcal{NP}$-hard.

In this chapter, we assume degree one for each node. Nodes without adjacent edges can be placed arbitrarily, and if they are parents and thereby the window is defined, its size is zero.

4.1 Placing parents within their span

One first observation follows from Proposition 2.3: When the children are fixed, so is the span, too. So the span marks a lower bound of a window. We ask if finding a drawing for the parents where they are placed in distinct positions, each within its span, is feasible. If there is such a drawing, this has the minimum window sum we can get with frozen children and moving parents. We provide a fast algorithm to find a drawing where all parents are within their span, if there is one.

Definition 4.1 (`ParentsInSpan` (`PinS`))**.** *Suppose we are given a set of parents P and the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ in a partial drawing $x : S \to \mathbb{Z}$., is there a completing drawing $x : P \to \mathbb{Z}$ in which every parent $p \in P$ is placed inside its span?*

In each step, the algorithm greedily chooses the next parent and tries to place it. For that purpose, the algorithm starts iterating over the positions $F, F+1, \ldots, L-1, L$, while $F = \min\{f_p \mid p \in P\}$ and $L = \max\{l_p \mid p \in P\}$ are the positions of the smallest and largest child node (Table 2.1). In the iteration, at position t, the algorithm adds all parents $p \in P$ to a priority queue ordered by the start of their span $f_p = t$. Thereby the parent gets marked as active, i.e., we can place this parent from this position onward as long as we

do not step outside its span. So after adding all parents, we look at the queue of active parents and decide which to place at the current positions. We choose the one where the risk of exceeding is most likely; that is the one with the closest end of the span. If it is already too late and the span is exceeded, it is impossible to place all parents in their span.

There is no drawing because, in the previous positions, placing a parent with an even nearer end was more urgent. However, if the algorithm successfully places all parents, it returns a correct drawing.

Furthermore, since the parents and the children can be distributed sparsely on a wide range of integers, the number of iterations can be very high and increase our running time considerably. For that reason, we add a component that reduces the number of iterations to at most $|P|$: If, after one iteration, the queue of parents to place is empty in the following iterations, as long as we do not mark a parent as active, we will not be able to place a parent. So we must wait until the next parent gets marked as active again. So if no parent is active, we do not continue with the next iteration on the following position but jump to the next start of a span instead. This way, we guarantee that we place exactly one parent in each iteration.

Algorithm: For an algorithm in pseudo-code, see Algorithm 4.1. Take two priority queues, Q and R, storing the parents p ordered by f_p and l_p, respectively. The first queue stores not active and not yet placed parents (with the start of their span to the right of the current position), while the second one stores the active parents. So initially, add all parents to Q.

Start with counter i at the smallest child $\min\{c \mid c \in C\}$. Repeat the following two steps until one of the two termination cases occurs: As long as i marks the start of the span f_p for the next parent p in Q, remove the parent p from Q and move it to R to mark it as active. When the next parent on Q has the start of its span right of i, we remove the next active parent p from R and place p at i, if $i \in S_p$, otherwise return `false` and stop the algorithm.

Increment the counter i in one of the following ways: If there are parents on R left, increment i by one and continue with the next step. Otherwise, look at the next parent p on Q and set i to f_p. If both queues are empty, stop the algorithm and return the drawing.

Example: Suppose we are given an arbitrary bipartite graph G and a drawing x (Figure 4.1(a)). Since it is not possible to visualize a partial drawing, we are given, in this example, a non-partial drawing.

We determine the spans and add all parents to Q. Therefore we order them by the start of the span and get A, B, D, E, C.

So we start on position 1, where we mark A and B as active by adding them to R since both have their start of span at this point (Figure 4.1(b)). On R, B gets prioritized before A because of the earlier span end. So we place B at the first position and remove it from Q (Figure 4.1(c)).

With R not empty, we continue on the following location 2 (Figure 4.1(d)). We have no span start, so there is nothing to push on R. However, A gets pulled and placed in this position.

There is no parent on R left, so we can jump over position three and get immediately to the start of the next span, i.e., position four. With both starts of the span f_E and f_D at 4, we push them on R (Figure 4.1(e)) before we place E at the current pointer position (Figure 4.1(f)).

Again, we go to the next position, 5, and push the last parent C on R (Figure 4.2(g)) and place it immediately (Figure 4.2(h)). When we continue, we see that by trying to place D

Algorithm 4.1: PLACING PARENTS IN THEIR SPAN

Input: A set of parents P and the set of their spans $\mathsf{S} = \{(\mathsf{f_p}, \mathsf{l_p}) \mid \mathsf{p} \in \mathsf{P}\}$ in a partial drawing $\mathsf{x} : \mathsf{S} \to \mathbb{Z}$

Data: Priority Queues Q, R of parents, ordered by their first child's and last child's coordinate, respectively

Output: Completing drawing of parents $\mathsf{p} \in \mathsf{P}$ in their spans, if possible, `false` otherwise

```
// Initialization of S, Q
1  Q.ADD(P);
2  i = f_{Q.EXTRACT()};
3  while true do
       // Pushing all active parents on R
4      while f_{Q.EXTRACT()} = i do
5          R.ADD(Q.REMOVE());
       // Placing the next parent
6      p = R.REMOVE();
7      if l_p < i then
8          RETURN false;
9      else
10         place p at i;
       // Incrementing i either by one or set it to the start of the
          next span
11     if R.NOTEMPTY() then
12         i ← i + 1;
13     else if Q.NOTEMPTY() then
14         i = f_{Q.EXTRACT()};
15     else
16         RETURN placement;
```

on position 6 (Figure 4.2(i)), we are already outside of the span S_D, and with that, we stop the algorithm and return `false`. It is easy to see that there is no way to place A, B, and C in their spans, i.e., the positions 4 and 5.

This algorithm leads to our next result.

Theorem 4.2. `ParentsInSpan` *is solvable in* $\mathcal{O}(|P| \log |P|)$ *time.*

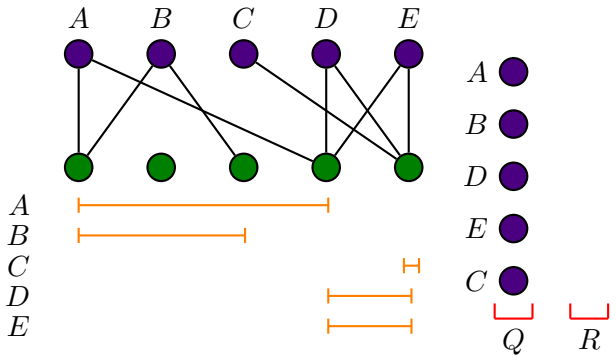
Proof. To prove correctness, we show these two statements:

(S1) A drawing returned by the algorithm is a correct solution of `PinS`.

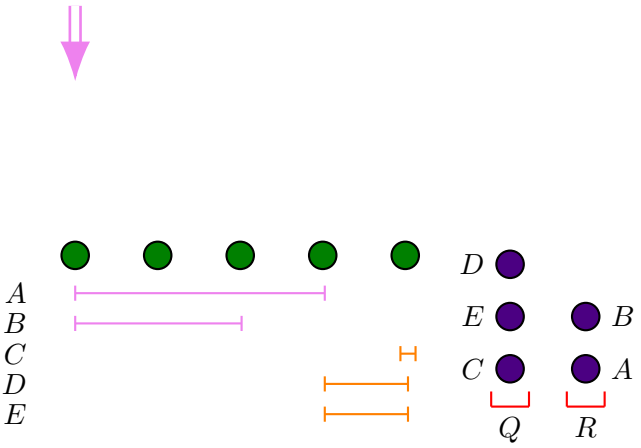
(S2) If there is a solution, the algorithm will find one.

Statement: (S1): The algorithm places each parent one by one, and as it does this, it checks if this parent is set in its span. So each parent is placed in its span. Additionally, the algorithm continues until all parents are placed, so the returned drawing has positions for all parents, which makes the solution correct.

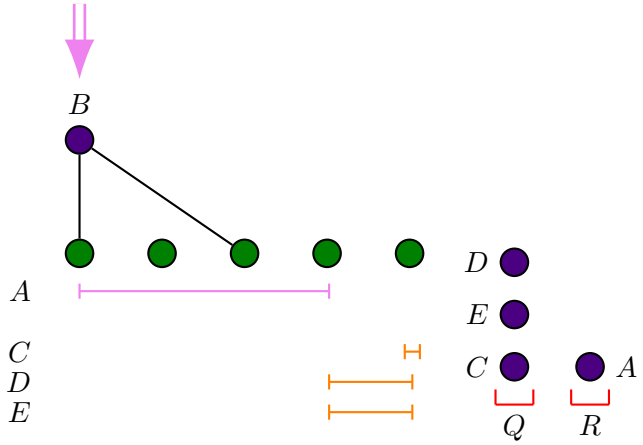
Statement: (S2): Suppose we are given a solution x for an instance of `PinS`. We want to prove that our algorithm returns a drawing, too. For that reason, we show that there



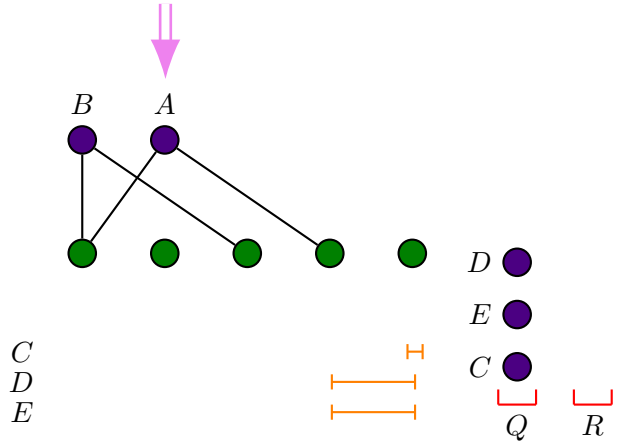
(a) A bipartite graph G in an arbitrary drawing with the span of the parents (orange) and the queues Q and R



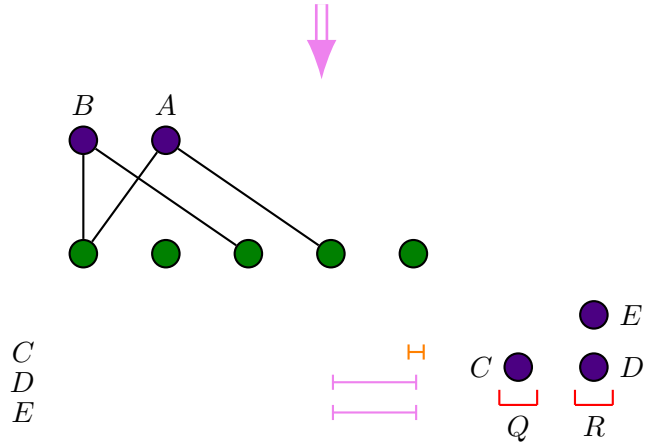
(b) A, B have their start at this point and so they become marked as active by moving it onto R.



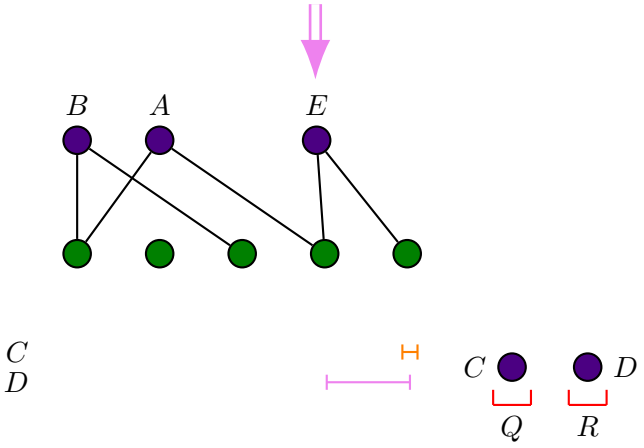
(c) The algorithm places now the active parent with the closer end of span, B, at this position.



(d) At the next position, there is no new active parent, so A gets placed next.

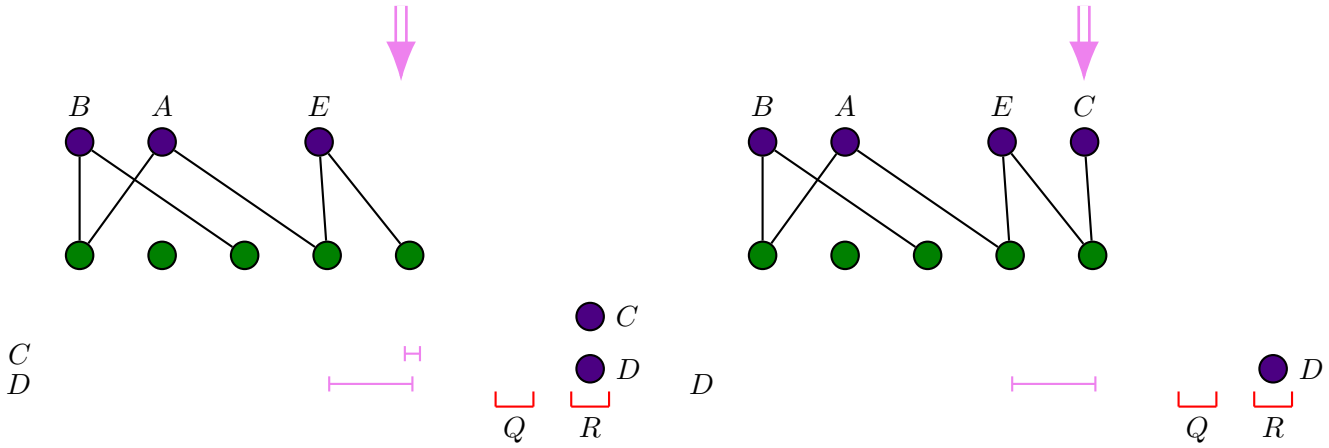


(e) The list of active parents is empty, so the counter jumps to the next start of a span, 4.



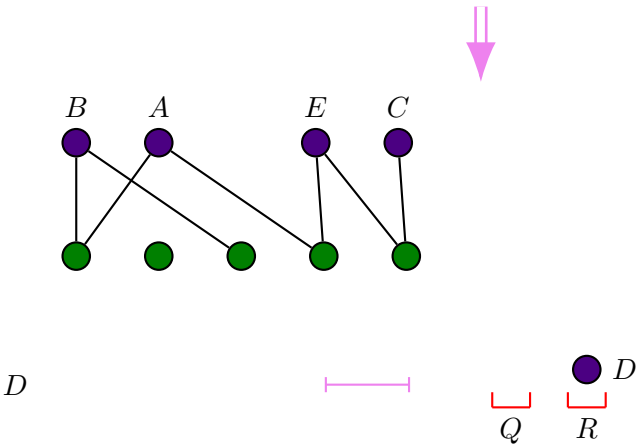
(f) Since both D and E have the same end of span, the order of placement does not matter, so E gets placed at position 4.

Figure 4.1: An example for the algorithm to place all parents inside their span with graph G. The successive steps are shown in Figure 4.2.



(g) C is marked as active by adding it to the queue.

(h) C is placed at position 5.



(i) The algorithm fails to place D at position 6 since this is already outside of S_D and thus returns `false`.

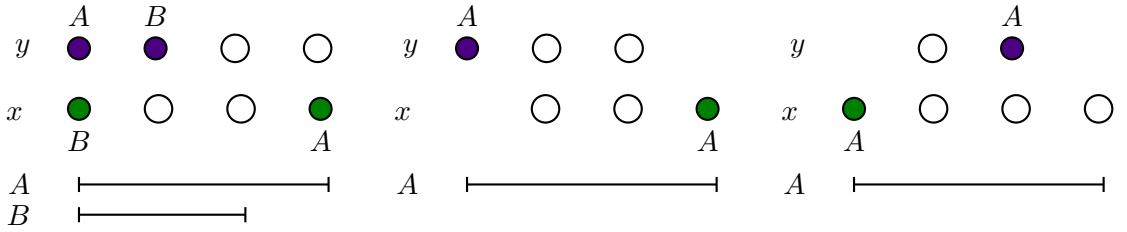
Figure 4.2: The second part of the example from Figure 4.1.

exists a greedy solution. Our algorithm only fails to provide a solution if stopped at any point when it is not possible to place the next parent. Let y be the drawing our algorithm creates up to a particular position i during the algorithm's run time. However, we show that if the optimal solution deviates from x at i, the greedy choice is valid.

We write $x^{-1}(a)$, to get the parent at position $a \in x(P)$ in drawing x. Let us take the first deviation at position $i = \min\{i \in x(P) \mid y^{-1}(i) \neq x^{-1}(i)\}$. By choosing the first deviation, we make sure that for $j < i$ the drawings x and y are identical, i.e., $y^{-1}(j) = x^{-1}(j)$ holds. The following deviations are possible:

Greedy algorithm chooses different parent Drawing x has parent $B \coloneqq x^{-1}(i)$. Meanwhile, the greedy choice of the algorithm is $A \coloneqq y^{-1}(i)$ as shown in Figure 4.3(a). B is placed at i on the solution x and A is chosen by our algorithm. So both A and B are active at i, so $i \in S_A$ and $i \in S_B$. With x and y being identical, $x(A) > i$ and $y(B) > i$. Our algorithm chose A while A and B are active, thus $l_A \leq l_B$. So $x(A) \in S_B$ and thereby $x(A)$ is a valid solution for B, too.

Greedy algorithm places parent, where x does not At i, the greedy algorithm would have placed $A \coloneqq y^{-1}(i)$, but x has a gap at i as shown in Figure 4.3(b). So A is active on i. Moving A to i in the drawing x leads to a more similar version to the greedy solution.



(a) The greedy algorithm chooses a different parent than the optimal solution.

(b) The greedy algorithm chooses a parent where x does not.

(c) The greedy algorithm chooses no parent where x has.

Figure 4.3: The three possible deviations of the greedy choice y to solution x

Greedy algorithm places no parent, where x does The optimal solution x has parent $B \coloneqq x^{-1}(i)$, while the algorithm would not place any node at i as shown in Figure 4.3(c). Since x and y equal at position $j < i$, $y(B) > i$. However, $i \in S_B$, and since B is active at i, the algorithm would not have chosen to keep i unoccupied.

No matter where the first position is, where x deviates from the greedy solution, it is always possible to swap parents appropriately to resolve the deviation. Each swap assimilates the solution up to position i. Thus after a limited number of swaps, it is possible to remove them. Hence if there is a solution for an instance, there is always a greedy one, too. So the algorithm finds a solution if there is one.

Suppose we are given a parent set P and a span set S as instance of `PinS`, then

The graph `ParentsInSpan` is solvable. $\Leftrightarrow$ The algorithm finds a valid solution.

The implication "$\Rightarrow$" is shown by Statement (S2), while "$\Leftarrow$" follows from Statement (S1).

Running time: To complete the proof, we need to calculate the running time for this algorithm. Notably, this algorithm does not require the input graph to be gap-free. The way the counter i is incremented guarantees jumps over too large gaps between the children.

The running time is computed in the following way: Let $n \coloneqq |P|$. Pushing n parents onto our priority queue needs $\mathcal{O}(\log n)$ time each so in total $\mathcal{O}(n \log n)$ We place a parent from the queue R in each loop iteration. Each of the parents will once be added to and removed from R. So the loop will be repeated precisely n times if not aborted earlier due to unsatisfiability. In each iteration, the most time intense operation is adding the nodes to R, i.e., the `push` operation, which runs in $\mathcal{O}(\log n)$. Hence we have a total running time of $\mathcal{O}(n \log n) = \mathcal{O}(|P| \log |P|)$, which completes our proof. □

4.2 Moving parents with fixed children

The span marks a lower bound for a window. It cannot get smaller. In this section, we provide an algorithm that calculates the sum-of-the-window-minimizing drawing even if it is not possible to place all parents in their span.

At first glance, the variant where we move the parents with frozen children looks the same as the variant with moving children and fixed parents. However, a significant difference makes one easily solvable while the question for the other variant stays unanswered. This difference is the number of parameters necessary to compute the minimum window size. If we search for an optimal location for a parent $p \in P$, we can calculate the window size w_p at a specific point upfront by looking at the frozen span and computing its distance to

the furthest child. Contrary to this, it seems impossible to determine the cost of placing a child in a particular position by knowing only the parents' positions. It is necessary to have the positions of other children, too.

Like in the last problem, the frozen interior children do not influence the window size of the parents, so we need only the parents and their marginal children alias span once again.

However, we can compute the window size at every location candidate for each parent when the children are fixed, so we only need to create a weighted matching from the parents to these locations. Like the `LAP`, we define this problem as a decision question. Suppose we are given a bipartite graph. Can we find a drawing with a window sum smaller than a particular number? However, the algorithm we provide calculates the minimal window size and simultaneously answers the related optimization question: What is the drawing with a minimum sum of window sizes?

Definition 4.3 (`MinimumWindowSumParents` (`WinSumP`))**.** *Suppose we are given a set of parents P, the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ (Table 2.1) in a partial drawing $x : S \to \mathbb{Z}$, and an integer $W \in \mathbb{N}$. Is there a completing drawing $x : P \to \mathbb{Z}$ such that the following holds?*

$$\sum_{p \in P} w_p \leq W.$$

The possibility of determining the cost of placing a parent at a particular point allows us to create a `MinMatch` (Definition 2.11) of the parents to their possible positions. For that reason, we create a weighted bipartite graph with the parents as one partition and all possible positions for the parents as the second. We connect all parents with the positions, and the weight of the edge equals the window sum this parent has at this position. When we then calculate the `MinMatch` with one of the known algorithms, we have a matching where every parent has exactly one matched position. At the same time, the total weight of these matched positions, i.e., the total window size, is minimal. So if this matching is perfect, we get one position for each parent with a minimum sum of the window sizes.

We need a preprocessing mechanism to avoid connecting the parents with too many positions and increasing the running time. With Corollary 2.10, we know that if we have $n := |P|$ edges for every parent, we make sure that there is a perfect matching. We only need to know where these positions are. Proposition 2.3 shows that if a parent is placed within its span, the window size equals the span size, while outside, it grows linearly with the distance to it. Since the window size takes minimal values in the span, we take the first positions from there. If the span is too small and we do not have enough positions, we continue to the left and the right of the span until we have n positions. Thus we need $\frac{n-|S_p|}{2} = \frac{n-(s_p+1)}{2}$ additional positions from both sides.

With these positions, we create the weighted bipartite graph and use a known algorithm for `MinMatch` to solve it. The returned perfect matching connects each parent with one position. Meanwhile, this matching has minimal weight, so the window size sum of the original graph is minimal.

Algorithm: Take an instance of `WinSumP` with graph parent set P, span set S, and an integer $W \in \mathbb{Z}$. Create a weighted bipartite graph $H = (P \cup T, E', \xi)$ where T is the set of possible positions and E' are the edges of the parents to the positions with weight ξ.

For each parent $p \in P$, execute the following steps: If the span size is larger or equal to $n - 1$, begin at the span's start $k = f_p$. Otherwise begin $\left\lceil \frac{n-(s_p+1)}{2} \right\rceil$ earlier. Starting at k, add the n successive positions to the list T with an edge to p. If the position is inside the

Algorithm 4.2: MINIMIZING WINDOW BY MOVING PARENTS.

Input: A set of parents P of a graph, the span set $\mathsf{S} = \{(\mathsf{f_p}, \mathsf{l_p}) \mid \mathsf{p} \in \mathsf{P}\}$ in a partial drawing $\mathsf{x} : \mathsf{S} \to \mathbb{Z}$

Data: Weighted bipartite graph $\mathsf{H} = (\mathsf{P} \uplus \mathsf{T}, \mathsf{E'}, \xi)$

Output: Completing drawing $\mathsf{x} : \mathsf{P} \to \mathbb{Z}$ of parents $\mathsf{p} \in \mathsf{P}$ with minimal window sum

```
// Initialization
T ← ∅
E' ← ∅
n ← |P|
// Adding the weighted edges
for p ∈ P do
    if s_p < n − 1 then
        k ← f_p − ⌈(n−(s_p+1))/2⌉
    else
        k ← f_p
    for t ← k to k + n − 1 do
        if t ∉ T then
            T.ADD(t)
        e ← (p, t)
        E'.ADD(e)
        if t < f_p then
            ξ(e) ← l_p − t
        else if t ≤ l_p then
            ξ(e) ← s_p
        else
            ξ(e) ← t − f_p
// Calculating the minimal weight perfect matching and creating the placing
H ← (P ∪ T, E', ξ)
M ← MINMATCH {H}
for (p, t) ∈ M do
    x(p) ← t
RETURN x
```

parent's span S_p, then the weight of the edge is s_p. Otherwise, it equals the distance to the closer marginal children plus s_p.

After adding all positions and the edges for the parents, create a weighted bipartite graph H to calculate a `MinMatch`. The result of the matching is a set where each parent is adjacent to exactly one position. Place the parents in these positions in a completing drawing $x : P \to \mathbb{Z}$.

Optionally, compute the window size W_P to determine if $W_P \leq W$. An algorithm in pseudo-code is shown in Algorithm 4.2.

Example: We are given an instance graph G in a drawing x as shown in 4.4(a). Once again, we take a non-partial instead of a partial drawing for visualization purposes while the positions for the first and last child are already enough.

Since we have seven parents, we also want to match seven locations for each of them. For a parent $p \in P$, we do the following, which we show exemplarily for node C. The span size is $s_C = 3$. Thus the first node $k = f_C - \left\lceil \frac{n-(s_C+1)}{2} \right\rceil = b - 2$. We match seven consecutive positions to C, beginning two before b (Figure 4.4(b)). For each edge, we have weight $s_C = 3$ plus the distance to the span. The figures above the positions indicate the window size at this position.

The red edges in Figure 4.4(c) illustrate a minimum weight bipartite matching, where each parent is connected to the optimal position. Since D, G, and F have c as shared only-child, only one can be placed optimally per se, but the sum is nevertheless optimal. The minimizing drawing is shown in Figure 4.4(d).

Theorem 4.4. `MinimumWindowSumParents` *is solvable in* $\mathcal{O}(|P|^{2+o(1)})$.

Proof. Suppose we are given a parent set $P, |P| =: n$ with span set S and $W \in \mathbb{N}$ as an instance of `WinSumP`.

We have to show that the algorithm returns a drawing, and this drawing has a minimal window sum. For that reason, we show that the following statements are true.

(K1) The matching part finds a perfect matching.

(K2) The sum of the weights in the matching is equal to the window sum with parents at the matched positions.

(K3) For each parent, the graph matches the n positions where the window is as small as possible.

(K4) The provided drawing has minimal window sum.

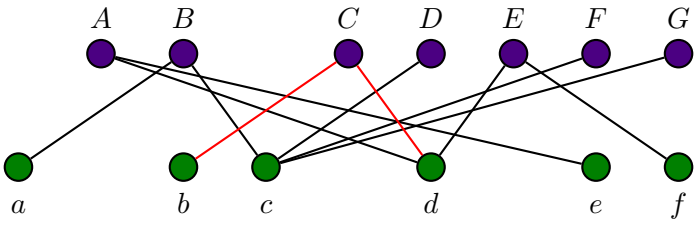
Statement (K1): The algorithm matches each of the n parents with n possible positions. So no matter where the other $n-1$ parents are located, there is at least one position unoccupied. With Corollary 2.10 of Hall's marriage theorem follows that there exists a perfect matching. Hence, it is possible to place every parent.

Statement (K2): Suppose we are given a perfect matching M. We define the drawing x_M such that for $(p,t) \in M$, $x_M(p) = t$. Let $e_p \in M$ be the edge to which $p \in P$ is adjacent. For the window sum function $\omega(x_M)$, the following yields:

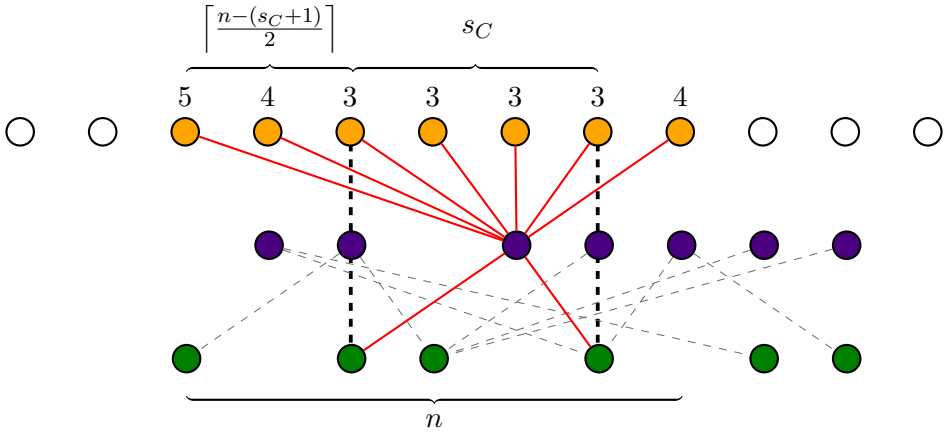
$$\omega(x_M) = \sum_{p \in P} w_p \overset{G \leftrightarrow H}{=} \sum_{p \in P} \xi(e_p) = \sum_{e \in M} \xi(e) = \xi(M).$$

Hence, the weighted sum equals the window sum.

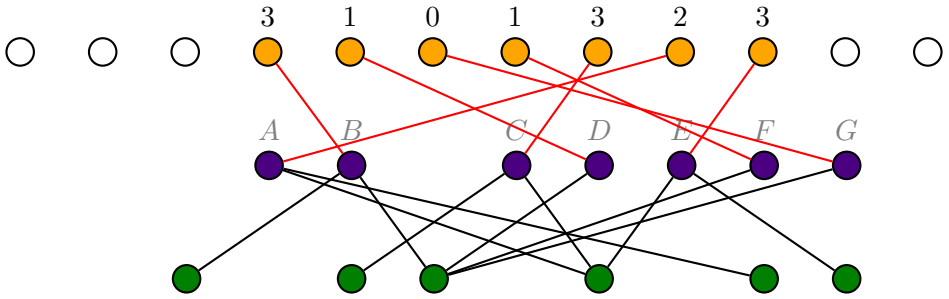
Statement (K3): Consider x as a drawing provided by the algorithm with matching M and window sum Ω_x. Assume there is a drawing y matching M' with a smaller window sum $\Omega_y < \Omega_x$. The matching algorithm always returns the minimum matching, so $M' \not\subseteq E'$. So there is a parent $p \in P$ that is matched with a lower window in M', so $(y(p), p)) \in M' \setminus E'$ with $\omega'_p := \xi(p, (y(p))) < \xi(p, (x(p))) =: \omega_p$. Consider $d_p := \omega_p - s_p$ and $d'_p := \omega'_p - s_p > d_p$ as distances to the span. With $d_p > 0$, the algorithm takes not only span locations and hence $s_p < n - 1$.



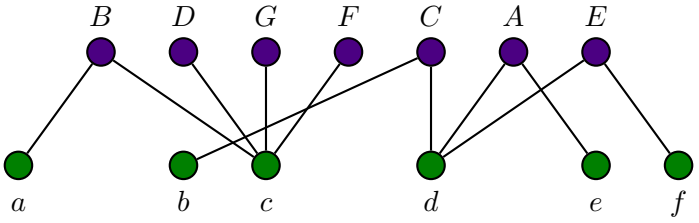
(a) A drawing of a graph G. For visualization purposes we focus on parent C.



(b) The matching of C to seven possible positions. The figures above the positions indicate the window size if we place node C on this position. This matching happens for all the other nodes, too.



(c) The red edges define a solution of the `MinMatch`. The figures above show the weight, i.e., the window size of the parent connected to it.



(d) The final solution of `WinSumP` as window sum minimizing drawing of the parents.

Figure 4.4: A graph G as example of the algorithm to find a sum-of-the-window-minimizing drawing.

Thus $\left\lceil \frac{n-(s_p+1)}{2} \right\rceil \geq d_p < d'_p$. This algorithms adds n locations from $f_p - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil$. So the last step takes place at

$$\begin{aligned}
& f_p && - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil + n - 1 \\
&= l_p - s_p && - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil + n - 1 \\
&= l_p + (n - (s_p+1)) && - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil \\
&= l_p + \left\lfloor \frac{n-(s_p+1)}{2} \right\rfloor + \left\lceil \frac{n-(s_p+1)}{2} \right\rceil - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil \\
&= l_p + \left\lfloor \frac{n-(s_p+1)}{2} \right\rfloor
\end{aligned}$$

Hence, the algorithm iterates over the whole span, so if $y(p) \in S_p$, $(p, y(p)) \in E'$. Otherwise, $d'_p \leq \left\lfloor \frac{n-(s_p+1)}{2} \right\rfloor$, so $y(p) \in \left[f_p - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil, l_p + \left\lfloor \frac{n-(s_p+1)}{2} \right\rfloor \right]$, and $(p, y(p)) \in E'$, too.

Statement (K4): This statement follows from the others. We know that the algorithm connects a parent with the positions where the window is as small as possible. The matching algorithm returns the locations for these parents with minimal weight sum. Since the weight sum equals the window sum, this is minimal too. So the algorithm is correct.

Running Time: The algorithm consists of two major parts: creating the weighted bipartite graph and solving the `MinMatch`. The former part connects each of the n parents to n positions, which leads to a running time of $\mathcal{O}(n^2)$. The more complicated part is the matching algorithm in the latter part. As mentioned after Definition 2.11, several attempts exist to create a fast running time. Since we have a minimum weight matching problem, we cannot use the simple linear time algorithm. With the Hungarian Method, we can at least guarantee an upper bound of $\mathcal{O}(n^3)$ [Kuh55].

We consider the algorithm by Kao et al. [KLST99] to improve the bound. It uses the parameter $W = \sum_{e \in E} \xi(e)$, which describes the sum over all weights. With Proposition 2.3, we showed that the span sizes are costs we always have to pay. Outside the span, the window size equals the span size plus the distance to the span. So we can subtract the span costs and start with 0 within the span and with $1, 2, \ldots$ to the left and right. Indeed, this provides us with the lowest possible weights. Nevertheless, in the worst case, i.e., a one-degree parent, the weights are $\frac{n}{2}, \ldots 1, 0, 1, \ldots, \frac{n}{2} \in \mathcal{O}(n^2)$. Thus, at this point, the sum equals:

$$W = \sum_{e \in E} \xi(e) = \sum_{p \in P} \frac{n}{2} + \cdots + 1 + 0 + 1 + \cdots + \frac{n}{2} = n \cdot (n, n-2, \ldots, 0) \in \mathcal{O}(n^3).$$

Hence the running time by using Kao's algorithm would be $\mathcal{O}(\sqrt{n}W) = \mathcal{O}(n^{3.5})$, which is even slower than the Hungarian Method.

The only way to find a matching faster is by the recently published algorithm by Chen et al. [CKL+22]. Since our weighted graph has exactly $m = n^2$ edges, n for each of the n parents, that gives us a running time of $\mathcal{O}(m^{1+o(1)}) = \mathcal{O}(n^{2+o(1)})$ for the matching part. Moreover, since the former part runs faster, this also marks the running time for the entire algorithm. □

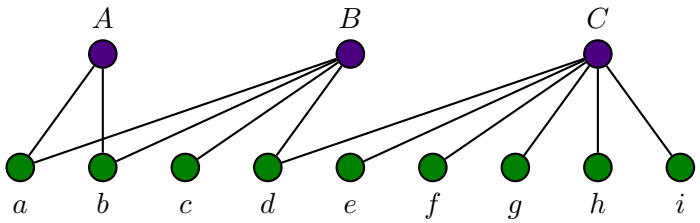


Figure 4.5: This drawing is not sum-of-the-window minimizing, but here the parents are placed opposite a triple of children.

Variant 1: The algorithm provided in this section places the parents in any position. However, is it still feasible if we only want to permute the parents' positions from a given drawing? Or consider a graph $G = (P \cup C, E)$ with $3|P| = |C|$: For creating a good-looking drawing, with children on positions $1, \ldots, |C|$, we may want to place parents on $2, 5, 8, \ldots, |C| - 1$. An example is shown in Figure 4.5 where the drawing places each parent opposite of three children. This solution may not be optimal, but better looking in this case. So we modify our algorithm to make this work.

So let us define a variation of the problem:

Definition 4.5 (`MinWinSumParentsWithLocations` (`WinSumLoc`))**.** *Suppose we are given a set of parents P, the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ in a partial drawing $x : S \to \mathbb{Z}$, an integer $W \in \mathbb{N}$ and for each parent $p \in P$ a set of possible locations L_p. Consider $L = \bigcup_{p \in P} L_p$. Is there a completing drawing $x : P \to L$ that the following holds?*

$$\sum_{p \in P} w_p \leq W$$
$$\wedge \qquad \forall p \in P : x(p) \in L_p.$$

The modification takes only place in the creation of the weighted bipartite graph. Instead of adding the positions in and around the span, we add for each parent $p \in P$ their locations L_p. In this variant, we do not have n^2, but $|L|$ edges, so the running time depends on the sum of possible locations of all parents.

If $|L_p| < n$ for a $p \in P$, finding a solution is not guaranteed anymore. Furthermore, this algorithm can determine *if* there is a placing for the parents in these positions while neglecting the window sum. These observations lead to the following conclusion:

Corollary 4.6. `MinWinSumParentsWithLocations` *is solvable in $\mathcal{O}(|L|^{1+o(1)})$.*

Variant 2: In this algorithm, we used the `MinMatch` (Definition 2.11). However, we get a different result if we use a `BottMatch` (Definition 2.12) instead of the matching part. This problem is specified to find the lowest maximum weight instead of the minimum weight sum. We showed that the weights equal the window sizes, which minimizes the maximum window. Thus lets us define another variant:

Definition 4.7 (`MimimumMaxWindowParents` (`MaxWinP`))**.** *Suppose we are given a set of parents P, the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ (Table 2.1) in a partial drawing $x : S \to \mathbb{Z}$, and an integer $W \in \mathbb{N}$. Is there a completing drawing $x : P \to \mathbb{Z}$ such that the following holds:*

$$\forall p \in P : w_p \leq W.$$

The answer to this problem is a known result discovered by Beckos et al. [BFK+23]. With a greedy algorithm, they have a faster running time than our matching, but it is still worth mentioning. It may be a goal to simultaneously find drawings for `WinSumP` (Definition 4.3) and `MaxWinP`, i.e., with minimum sum-of-the-window-sizes and minimum longest edge, respectively, to compare the solutions and choose by hand the better drawing. Since the weighted bipartite graph is the same for both problems, we only have to apply `MinMatch` and `BottMatch` and get the two drawings.

The current fastest algorithm for this problem runs in $\mathcal{O}(n \cdot \sqrt{mn})$ and with $m = |E'| = n^2$, this equals $\mathcal{O}(n^{2.5})$ [PL88]. In summary, we achieve this result:

Corollary 4.8. `MimimumMaxWindowParents` *is solvable (with our matching technique) in* $\mathcal{O}(|P|^{2.5})$.

4.3 Moving both parents and children

The last result showed that minimizing the window sum is feasible if we rearrange the parents with fixed children. However, we do not necessarily make any restrictions on what to move. The next question is if we can compute a sum-of-the-window minimizing drawing with moving both partitions. We show $\mathcal{NP}$-completeness for this problem via reduction from `LAP` (Definition 2.6).

Definition 4.9 (`MinimumWindowSumBoth` (`WinSumB`))**.** *Suppose we are given a bipartite graph* $G = (P \dot\cup C, E)$ *and an integer* $W \in \mathbb{N}$*. Is there a drawing* $x : P \dot\cup C \to \mathbb{Z}$ *(Table 2.1) that the following holds?*

$$\sum_{p \in P} w_p \leq W.$$

For the reduction, we need to transform an instance of `LAP` in polynomial time into an input graph for `WinSumB`. This transformation is achieved by the following method:

Definition 4.10 (Block graph)**.** *Suppose we are given a graph* $G = (V, E)$*. For a fixed* $k \in N$*, consider*

$$\begin{aligned} B_v &= \{v_1, \dots v_k\} \text{ for } v \in V, \\ B &= \bigcup_{v \in V} B_v, \\ P &= \bigcup_{(u,v) \in E} p_{uv}, \text{ and} \\ E' &= \bigcup_{v \in V} \bigcup_{v_i \in B_v} \{(v_i, v)\} \cup \bigcup_{(u,v) \in E} \{(p_{uv}, u), (p_{uv}, v)\}. \end{aligned}$$

We call $v_1, \dots v_k$ *the* block-parents *of child* $v \in V$*, and we call* B_v *the* block of the child v*. We call* p_{uv} edge-parent *of edge* $(u, v) \in E$*.*

Each block-parent, $v_i \in B$*, has* v *as its only child. Each edge-parent* $p_{uv} \in P$ *is adjacent to both children* $u, v \in V$*. We call* $H = ((P \cup B) \dot\cup V, E')$ *a* block graph*.*

Examples of block graph drawings are shown in Figures 4.6(c) and 4.6(d).

We will show that the drawing of a block graph, where the window sum is minimal, has a special form:

Definition 4.11 (Tight realization)**.** *A drawing x of a block graph $H = ((P \cup B) \uplus V, E')$ is a* tight realization*, if for a child $v \in V$ and an edge-parent $p_{uv} \in P$*

(P1) *edge-parent p_{uv} is placed in its span, i.e., $u \leq p_{uv} \leq v$,*

(P2) *the block-parents of block B_v are ordered consecutively without other nodes in between, i.e., $|x(B_v)| = k$, and*

(P3) *child v is placed strictly within its outermost block-parents, i.e., there are block-parents $v_i, v_j \in B_v$ with $v_i < v < v_j$.*

(P4) *If there exists a gap in the parents, i.e., a position $g \notin x(P \cup B)$ with $p_1, p_2 \in P \cup B$, where $p_1 < g < p_2$, then $L(g) \coloneqq \{v \in V \mid v < g\}$ and $R(g) \coloneqq \{v \in V \mid v > g\}$ as children left and right of g form two disconnected subgraphs of V.*

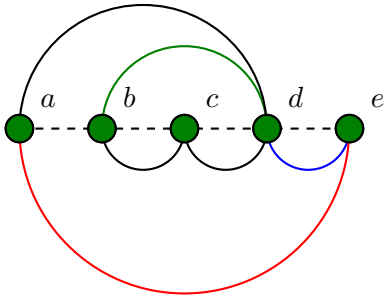
Suppose we are given a graph as an instance of `LAP`. We construct a block graph that consists of two parts. The nodes are placed as children; opposite them, there are k parent nodes adjacent to the block. The edges, which we have to minimize, are represented like in the edge graph from Definition 3.2 by two-degree edge-parents.

The sum-of-the-window-size minimizing drawing is a tight realization. In this ordering, the blocks are placed consecutively opposite their shared child. Between the blocks are the edge-parents. If we want to minimize the window sum of a block, we put the block-parents close together and their shared child in the middle. In a tight realization, this is quasi-achieved, so their window sum is close to the minimum. However, the edge-parents have large spans.

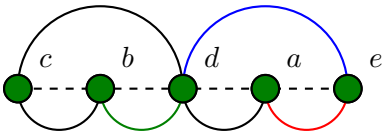
We assume we have placed the blocks strung together in this way. If we place one of the edge-parents in their best position, preferably inside their span, but between the blocks, it pushes the left and right, one away from each other. This position does not change the windows of the block-parents but the windows of the other edge-parents, with adjacent edges in both halves. So we could position the parent outside its span to avoid this effect. However, to the left and the right outside the span, there is about a half block of their adjacent children. They block the edge-parent from being placed there and the first position outside at least $\frac{k}{2}$ distant to the span, which extends the window by the same value. So we make k large enough to prevent this from happening. If that is achieved, the only way to reduce the window sum significantly is by finding the proper order of the children.

Example: Figure 4.6(a) presents an arbitrary instance of the `LAP`. The gadget ordering is constructed by connecting the vertices with k block-parents each, represented by the gray nodes. Additionally, the edges in the monoline graph are transformed into the edge-parents between the blocks with connections to their adjacent vertices. Three edges are colored for visualization in the tight realization of this ordering shown in Figure 4.6(c).

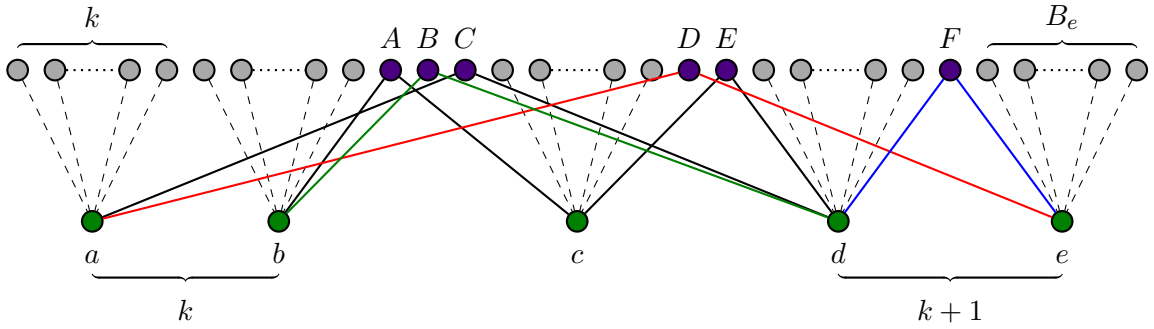
Consider k being large, e.g., 103. We compute the window size in this format, where the children are opposite their middle block-parent. The gray block-parents are already as close as possible to their shared child, and the windows are not improvable. However, let us calculate the window sum of the edge-parents. With no edge-parent between two blocks, like between B_a and B_b, the distance between their children is k. With a child in between, the window size w_F is $k + 1$. Nonetheless, $w_E = k + 2$, since between c and d are two edge-parents. In the same way $w_A = k + 3$, $w_B = 2k + 5, w_C = 3k + 5$ and $w_D = 4k + 6$. So the total window sum $\omega_P = 12k + 21 = 12 \cdot 103 + 21$. Hence, it is easy to see that with a large k, the number and position of edge-parents between the parents play a minor role.



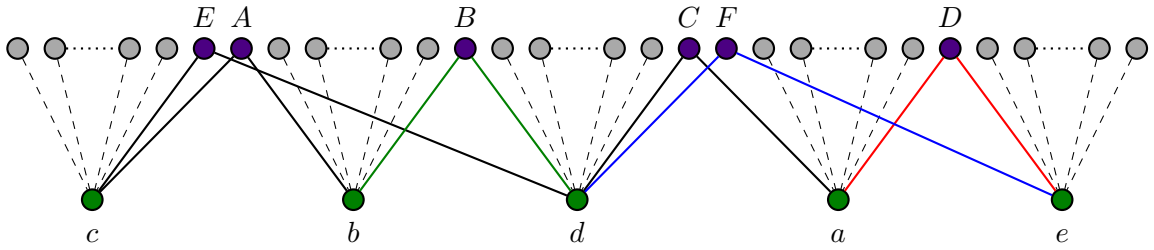
(a) A graph G in an arbitrary drawing which forms with $W = 10$ an instance of the LAP



(b) The optimal ordering of G in the LAP



(c) The tight realization of the block graph originating from G with the same order of child nodes as the drawing in (a)



(d) A tight realization with smaller and quasi-minimal window sum, originating from the optimal ordering of G in (b)

Figure 4.6: A graph G in two different drawings and their corresponding block graphs

Suppose we are given an optimal drawing in LAP as displayed in Figure 4.6(b). The related tight realization in Figure 4.6(d) gets significantly smaller:

$$\begin{aligned}\omega_P &= w_A + w_B + w_C + w_D + w_E + w_F \\ &= (k+2) + (k+1) + (k+2) + (k+1) + (2k+3) + (2k+3) \\ &= 8k + 12.\end{aligned}$$

While moving c one step rightwards would improve the window sum; the difference stays small compared to k. With improvements in moving the children or the edge-parents, the costs for the blocks stay similar so that we can omit this in our calculation. So for an instance with $k = 1000$, if we know that the tight realization has a minimal window sum (of the edge-parents) of 22032, there may be a realization of LAP with edge sum 22.

At first, we will exhibit that with large k, every minimizing solution is a tight realization. After that, we will prove that we can construct a block graph and an integer W', such that the window sum is smaller than W' if there is a solution for LAP with W.

Lemma 4.12. *For every block graph with $k > \max\{|P|, |E| + 2|P|\}$ a realization that minimizes the sum of the window sizes is a tight realization.*

Proof. Suppose we are given a drawing x of a gadget graph $H = ((P \cup B) \cup V, E')$ with minimum window sum. Let $n := |P|$. We show the validity of all four properties.

Property (P3): Assume w.l.o.g. all block-parents of a child $v \in V$ are located on positions greater or equal to v. Hence, v has at least $k - 1$ adjacent parents to the right and, at most, $|E|$ to the left. Moving v one step right would tighten at least $k - 1$ windows and broaden $|E|$ edge-parents' windows left and possibly the one of the block-parent opposite v. With $k - 1 > |E| + 1$, this leads to a smaller window sum, contradicting minimality. If this position is occupied by another child, moving v two steps right would reduce at least $k - 2$ windows and broaden at most $|E| + 2$ windows. Since $|C| = n$, after at most n steps v finds a place and even $k - n > |E| + n$ holds. So no matter where the other parents are located, there is a guaranteed better place unoccupied for v.

Property (P1): Assuming without any loss of generality, we have parent $p_{uv} \in P$ with $u < v < p_{uv}$. Using Property (P3) which we have already proven, there exists $v_1 \in V$ with $v_1 < v$. Consider the new drawing y we get by swapping v_1 and p_{uv}. With this, consider the new windows w'_{v_1} and $w'_{p_{uv}}$ in y are as visualized in Figure 4.7.

Case $x(v_1) \leq x(u)$ (Figure 4.7(a)):

$$\begin{aligned} w_{p_{uv}} + w_{v_1} &= (x(p_{uv}) - u) + (v - x(v_1)) \\ &= (x(p_{uv}) - v) + (v - u) + (v - x(v_1)) \\ &> (y(v_1) - v) + (v - y(p_{uv})) \\ &= w'_{v_1} + w'_{p_{uv}} \end{aligned}$$

Case $x(v_1) > x(u)$ (Figure 4.7(b)):

$$\begin{aligned} w_{p_{uv}} + w_{v_1} &= (x(p_{uv}) - u) + (v - x(v_1)) \\ &= (x(p_{uv}) - v) + (v - u) + (v - x(v_1)) \\ &> (y(v_1) - v) + (v - u)) \\ &= w'_{v_1} + w'_{p_{uv}} \end{aligned}$$

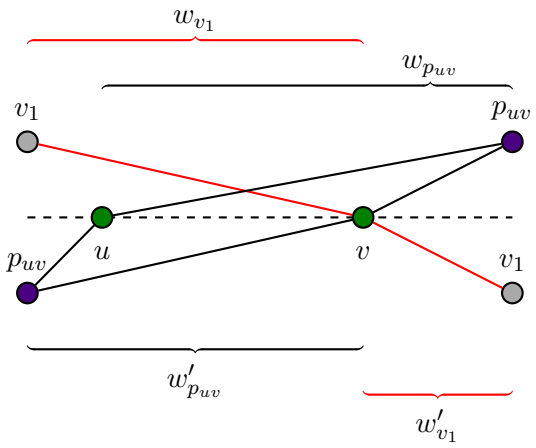
Property (P2): Consider $p_{ab} \in P$ and $u, v, a, b \in V$ with $a < b$. We aim to prove that two blocks do not intersect and that there is no edge-parent inside a block.

Suppose without loss of generality that $u < v$. Let $v_1 = \min B_v$ be the smallest block-parent of v and $u_1 = \max B_u$ be the largest of u. The proven Property (P3) gives us $v_1 < v$ and $u_1 > u$. If $u < v_1 < u_1 < v$, swapping u_1 with v_1 leads to an improvement in both window sizes w_{v_1}, w_{u_1}, contradicting minimality, so $v_1 > u_1$. With that, for all $v_i \in B_v, u_j \in B_u$, $v_i > u_j$. So no two blocks intersect.

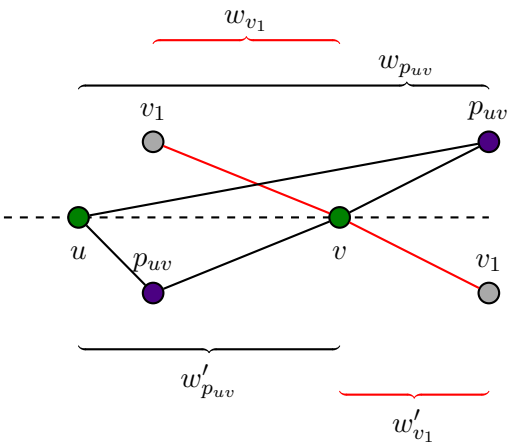
If exists a parent $p_{ab} \in x(B_v)$ in block $v_1 < p_{ab} < v_2$: We proved that blocks a, b, v do not intersect, so $a < v_1 < v_2 < b$. So swapping p_{ab} with v_1 or v_2 would leave p_{ab} in its span. If $v \leq p_{ab}$, swapping with v_2 decreases v_2's window. Otherwise, the same happens with v_1, and both ways contradict minimality.

Property (P4): Let $t \notin x(P \cup B)$ be a gap in the drawing where no parent is located. Assume a node $v \in V$ with $x(v) = t$. However, moving any $v_1 \in B_v$ to t would set its window to zero. So there is no child at t, either. If $L(t), R(t)$ are not disconnected subgraphs, there exist $a \in L(t), b \in R(t)$, $(a, b) \in E$. Thus $p_{ab} \in P$. So $x \in S_{p_{ab}}$ and with that, moving all parents and children right of x one down improves $s_{p_{ab}} = w_{p_{ab}}$ but broadens no window.

Hence we proved all four properties and, thereby, the lemma. □



(a) Swapping p_{uv} with a block-parent left of u before the swap (above) and after the swap (below)



(b) Swapping p_{uv} with a block-parent between u and v before the swap (above) and after the swap (below)

Figure 4.7: These are two swaps that would improve the window sum to show that all parents are placed in their span.

Using this lemma, we can now prove the main theorem of this section:

Theorem 4.13. `MinimumWindowSumBoth` *is $\mathcal{NP}$-complete.*

Proof. For $\mathcal{NP}$-completeness, we reduce `LAP` to `WinSumB`, show that this reduction is polynomial and prove membership in $\mathcal{NP}$.

Suppose we are given an instance of the `LAP` with $G = (V, E)$ and an integer $W \in \mathbb{N}$. Let $k > \max\{n, |E| + 2n, 3|E|^2 - |E|\}$ be odd. We create a block graph $H = ((P \cup B) \uplus V, E')$ and calculate the minimal window sum W' to show the equality of the following statements.

(S1) G, W as instance of `LAP` is solvable.

(S2) H, W' as instance of `WinSumB` is solvable.

We achieve this by proving $(S1) \Rightarrow (S2)$ and $\neg\,(S1) \Rightarrow \neg\,(S2)$. However, first, we need to calculate W':

Since k meets the conditions of Lemma 4.12, the minimal solution is a tight realization. We compute the minimal window sum function ω under the assumption that all children are placed opposite of their middle child before we calculate the improvement we will get from moving them to other positions.

Let us begin with the block gadgets. Consider E'_B as the edges of E' adjacent to block-parents and E'_P as the edges to edge-parents, so $E' = E'_B \uplus E'_P$. The n block gadgets with k nodes each have a total window sum of

$$\omega_B = n \cdot (t + t - 1 + \cdots + 1, 0, 1 + \cdots + t - 1 + t) = n \cdot 2 \sum_{i=0}^{t} i = nt(t+1),$$

while $t := \frac{k-1}{2}$.

For nodes $v \in V$, let $y(v)^* = |\{u \in V \mid u \leq v\}|$ denote the position among the children in a drawing y. Let $N(u, v)$ be the number of edge-parents between u and v. The distance

between two neighbored children $u, v \in V$ equals k plus the number of edge-parents between their blocks. Thus for a parent p in drawing y the window size

$$w_p = \sum_{f,l \in C_p} |y(l) - y(f)| = \sum_{f,l \in C_p} |y(l)^* + y(f)^*| \cdot k + N(f,l).$$

So if drawing y has the same order of the blocks of graph H as drawing x has for G, the window sum for the edge-parents

$$\begin{aligned}\omega_P &= \sum_{p \in P} w_p \\ &= \sum_{p \in P} \sum_{f,l \in C_p} |y(l) - y(f)| \\ &= \sum_{p_{uv} \in P} |y(u)^* - y(v)^*| \cdot k + N(u,v)\end{aligned}$$

Let us look at the dimensions of $N(u,v)$ for an edge-parent $p_{uv} \in P$. The upper bound is the number of edge-parents $|E|$, while the lower bound is 1, the parent itself. So $N(u,v) = 1 + N^*(u,v)$, with $N^*(u,v) \in \{0, \ldots, |E| - 1\}$. Thus,

$$\sum_{p_{uv} \in P} N(u,v) = \sum_{(u,v) \in E} 1 + N^*(u,v) = |E| + \sum_{(u,v) \in E} N^*(u,v) \coloneqq N^*$$

with $N^* \in \{|E|, \ldots, |E|^2\}$.

Next, we compare this drawing to one where the children are not strictly placed in the middle of their block gadgets. We know the children are placed in their block, but they can move left and right within it. Consider a child node $v \in V$ and d as the difference between the number of adjacent parents placed right to it, and the number of parents left. To improve the window sum by moving a child one position to the right, d must be at least 2. This is so because the right parents' windows shrink with this move, and the left parents' windows, including the parent on position v, increase. Since this realization is tight, there is always a parent at position v, too. Now we have a block-parent more left and one block-parent fewer right, and for an additional move, the difference between the number of parents to the right and the left again has to be at least two. Thus for two steps, d has to be four or more. Hence with difference d, the child steps $\left\lfloor \frac{d}{2} \right\rfloor$ times. The first step improves $d-1$ windows, the second $d-2$, so with $d < |E|$ the improvement made by moving one child is at most d^2 However, the total difference of all children is $2|E|$, with every edge-parent adding a difference of at most two to distinct children. So moving the children improves the total window sum by $\mu^* \in \{0, \ldots, 2|E|^2\}$.

This information makes it possible to compute the total window size sum Ω. Consider $W_{min} \coloneqq nt(t+1) - 2|E|^2 + |E|$ and $W^* \coloneqq N^* - \mu^* + 2|E|^2 - |E|$, so $W^* \in \{0, \ldots, 3|E|^2 - |E|\}$. So

$$\begin{aligned}\Omega &= \omega_B + \omega_P - \mu^* \\ &= nt(t+1) + N^* - \mu^* + k \cdot \sum_{p_{uv} \in P} |y(u)^* - y(v)^*| \\ &= W_{min} + W^* + k \cdot \sum_{p_{uv} \in P} |y(u)^* - y(v)^*|.\end{aligned}$$

Finally, we can prove the equivalence of the statements (S1) and (S2). Suppose we are given $G = (V, E), W$ as an instance of `LAP`. We create a block graph $H = ((P \cup B) \uplus V, E')$. Consider $W' \coloneqq W_{min} + 3|E|^2 - |E| + k \cdot W$.

$(S1) \Rightarrow (S2)$ So there is a drawing x of the nodes of G, where $\sum_{(u,v)\in E} |x(u) - x(v)| \leq W$. We create a tight realization y of H with the blocks in the same order, such that $\sum_{p_{uv}\in P} |y(u)^* - y(v)^*| \leq W$ yields. So for H in drawing y the window sum Ω is

$$\begin{aligned}\Omega &= W_{min} + W^* \qquad\qquad + k \cdot \sum_{p_{uv}\in P} |y(u)^* - y(v)^*| \\ &\leq W_{min} + 3|E|^2 - |E| + k \cdot W \\ &= W'.\end{aligned}$$

$\neg(S1) \Rightarrow \neg(S2)$ There is no realization x, where $\sum_{(u,v)\in E} |x(u) - x(v)| \leq W$. Thus for every drawing y of H, $\sum_{p_{uv}\in P} |y(u)^* - y(v)^*| > W$. Since $k > 3|E|^2 - |E|$, the following is valid for the window minimizing solution:

$$\begin{aligned}\Omega &= W_{min} + W^* + k \cdot \sum_{p_{uv}\in P} |y(u)^* - y(v)^*| \\ &\geq W_{min} + W^* + k \cdot (W + 1) \\ &\geq W_{min} \qquad + k \cdot W + k \\ &> W_{min} \qquad + k \cdot W + 3|E|^2 - |E| \\ &= W'.\end{aligned}$$

Hence, our block graph has a drawing where the sum of the window sizes $\Omega \leq W'$, if and only if the monoline version has an arrangement with an edge sum smaller or equal to W. The block graph has $vk + |E|$ parents, v children and $vk + 2|E|$ edges. With $k \leq \max\{n, |E| + 4, 3|E|^2 - |E|\} + 2$, the reduction is polynomial and `WinSumB` $\mathcal{NP}$-hard.

Nothing else remains to be done except to prove membership in $\mathcal{NP}$. Suppose we are given an instance of `WinSumB` with graph $G = (P \cup C, E)$, calculating the window size sum can be done in $\mathcal{O}(|P| \cdot |C|)$. □

5. Minimizing the edge length

Minimizing window sizes on drawings of bipartite graphs is one thing, but sometimes considering the number of edges displayed in the window makes sense, too. Thus let us talk about reducing the edge lengths where similar to the window size minimization, sum-of-the-edge-length, and longest-edge minimization is possible.

The problem can also be described as ink minimization since the amount of ink used to print a bipartite graph depends on three things. We cannot reduce the number of vertices and edges without losing information, so the only modifiable factor is the lengths of the edges (To be exact, that is true for the euclidean edge length, but as mentioned in Definition 2.5 we use projected edge lengths). Graphical factors like the size, shape, and distance of vertices and the stroke width of edges shall be neglected and considered separately.

However, window and edge length minimizing is correlated. A window of a parent is always lower-bounded by the longest edge, so minimizing window sizes has a similar effect on edge lengths, too. Nonetheless, a window depends only on the marginal children, so sum-of-the-edge-length minimization is not the same as sum-of-the-window minimization.

If we minimize the edge lengths, the difference between parents and children is lost, but we keep the naming for consistency. That gives us two different methods for pursuing our goal: Either we fix one side and allow the other to move freely (w.l.o.g. we freeze the children and change the parents' places), or we change the positions of both sets. We will first provide an algorithm for the half-frozen variant before we continue with proving $\mathcal{NP}$-completeness for the second problem. In this chapter, we also assume at least degree one for parents and children. Nodes without an adjacent edge can be placed arbitrarily.

5.1 Moving one side with fixed other side

The reason why the problem is not $\mathcal{NP}$-hard if we allow only one side to move is made by a slight difference that helped us at the `WinSumP` problem, too: When placing a parent in a particular position, the length of the adjacent edges only depends on the connected children. Since they are fixed, this is independent of the other parents. So we can again create a cost function and match the parents to their possible positions as we did in `WinSumP`.

However, it is not possible to use the algorithm of `WinSumP` without any changes. There are differences in the weight function. When looking at windows inside the span, the window size attains minimal values, i.e., the span size. Outside the span, the window equals

the span size plus the distance to the span. The weight calculation is more challenging when looking at edge length sums instead of window sizes. For instance, for a parent with connected children in positions $2, 3, 4, 5, 100$, it would make more sense to put the parent in 1 outside its span than in position 100 inside. So without good preprocessing, we need to iterate for each parent over its whole span and compute the edge length to every adjacent child.

See Table 2.1 for the used terms. So let us define our problem:

Definition 5.1 (`MinimumEdgeSumOneSide` (`EdgeSumOne`))**.** *Suppose we are given a bipartite graph $G = (P \dot\cup C, E)$, a partial drawing $x : C \to \mathbb{Z}$, and an integer $W \in \mathbb{N}$. Let $S_P \coloneqq \max_{p \in P} s_p$. Is there a radial drawing $x : P \to \mathbb{Z}$ (Table 2.1), such that the following holds?*

$$\sum_{e \in E} \lambda(e) \leq W.$$

We have to examine, which is the best position for a parent, so we introduce new terms:

Definition 5.2 (Edge minimum, minimizing position, left and right children, relative edge length)**.** *Suppose we are given a bipartite graph $G = (P \dot\cup C, E)$ and a drawing $x : C \to \mathbb{Z}$. Let $p \in P$ be a parent and i a position.*

- $\omega_p(i) \coloneqq \sum_{c \in C_p} \lambda((i, c))$ *is called the* edge length sum of p at position i.
- $\mu_p \coloneqq \min_{j \in \mathbb{Z}} \omega_p(j)$ *is called the* edge minimum of p.
- $X_p \coloneqq \{j \in \mathbb{Z} | \omega_p(j) = \mu_p\}$ *are the* minimizing positions of p.
- $\hat\omega_p(i) \coloneqq \omega_p(i) - \mu_p$ *is the* relative edge length sum.
- *To achieve a position-based partitioning of the children $C_p = L_p(i) \dot\cup M_p(i) \dot\cup R_p(i)$ we define*

$$\begin{aligned} L_p(j) &\coloneqq \{c \in C_p \mid c < j\} \\ M_p(j) &\coloneqq \{c \in C_p \mid c = j\} \\ R_p(j) &\coloneqq \{c \in C_p \mid c > j\} \end{aligned}$$

  as left, opposite and right child(ren) of p, *respectively.*

Note that a parent's relative edge length sum is always 0 at its minimizing position(s) and positive on the other positions. Additionally, $M_p(i)$ contains a single child at position i or is empty.

We need $n \coloneqq |P|$ possible locations for the matching again, so we want to find their locations. Let us take a parent p with degree n_p. We first have to be aware of the extraordinary form of our weight function. Outside of its span, at the position, $f_p - k$ or $l_p + k$ with a large-scale k, this position has considerable weight. However, with every step towards the span, the weight sum gets exactly n_p lower, one for each edge. So the best n positions are somewhere in and around the span. We demonstrate that this cost function has a minimum, an interval with not necessarily different bounds. From this minimum, the function monotonically increases to the left and the right.

So we can search for the minimum in the span and keep adding the positions with the lower cost from left and right up to n. Then we have our `MinMatch` and can compute the matching like in the window variant.

The cost function has this particular form for the following reason: Let us assume we have a parent $p \in P$ and a position j with costs $\omega_p(j) = \sum_{c \in C_p} \lambda((p,c))$. We calculate the cost change if we increment j. The lengths of the edges from p to children right to j, i.e., $R_p(j)$, decrease by one, while the other edges $L_p(j) \cup M_p(j)$ will get longer. So for j far to the left of the span, the edge length is considerable. With each step we get closer to the first child, the weight gets reduced by $|R_p| = |C_p|$ and increased by $|L_p| = 0$.

When we enter the span, nodes shift from R_p to L_p, and the cardinality difference between these two sets decreases. At some point, when $|R_p| = |L_p|$, this function will attain its minimum. The function will rise again as soon as more children are in L_p than are in R_p. So we start at the position where $|R_p| = |L_p|$ and add as many locations as needed to the matching.

One observation we make is that there is no use in calculating the actual edge sum. The relative edge sum is the only thing that matters for the matching part. The edge minimum is the cost we always pay, so we can use the algorithm by continuing with $\hat{\omega}_p(i)$. With that, we start at the one or two middle children with cost zero without calculating the actual value. Then we can compute step by step the weight increase left and right.

Algorithm: Create a weighted bipartite graph $H = (P \uplus T, E', \xi)$ following these steps for each parent $p \in P$. Initiate a Boolean array B for the positions of the children. Iterate over all children $c \in C_p$ and set $B(c)$ to `true`. Meanwhile, link the lowest child and count the number of children n_p. From the first child, get to the middle child t by iterating over the array and counting the number of passed children.

If the number of children is odd, add t to the positions and edge (p,t) with weight zero to the weighted graph. Otherwise, iterate with t from the left middle child to the right middle child and do the same each step. Count the number of added positions and if at any point this reaches n, stop this iteration and continue with the next parent. Add the minimal positions with costs of 0. Start with pointers i and j at the one below the left and one above the right middle children, respectively. Initiate counters l, r, ω_i, and ω_j with the number of middle children, 1 or 2. The variable l stores the number of children right of i minus the number of children left of and in i. The variable r stores the number of children left of j minus the number of children right of and in j. The relative edge length sum $\hat{\omega}_p(i)$ and $\hat{\omega}_p(j)$ are saved in ω_i and ω_j.

Until reaching the required amount of positions to match, choose the next with this algorithm: If $\omega_i \leq \omega_j$: Add i to the positions and (p,i) to the edges with weight ω_i. If a child is at position i marked in $B(i)$, increment l by two. Add l to ω_i and decrement i by one.

Otherwise, if $\omega_i > \omega_j$, add position j to T and edge (p,j) to E' with weight $\xi((p,j)) = \omega_j$. If $B(j)$ increment r by two. Add r to ω_j and increment j by one.

When this is done for all parents, solve a `MinMatch` with this graph to receive matching M with minimal weight, i.e., minimal window sum. Like in `WinSumP`, extract the positions in this matching and place the parents on these positions in a new drawing $x : P \to \mathbb{Z}$.

Optionally, compute the edge sum W_P to determine if $W_P \leq W$. A pseudo-code algorithm is shown in Algorithm 5.1.

Algorithm 5.1: Minimizing edge length with one side fixed. Part 1/2 5.2

Input: A bipartite graph $\mathsf{G} = (\mathsf{P} \cup \mathsf{C}, \mathsf{E})$, a partial drawing $x : C \to \mathbb{Z}$
Data: Boolean array B, weighted bipartite graph $\mathsf{H} = (\mathsf{P} \cup \mathsf{T}, \mathsf{E}', \xi)$
Output: Completing drawing $\mathsf{x} : \mathsf{P} \to \mathbb{Z}$ with minimal edge length sum

```
// Initialization
T ← ∅
E' ← ∅
n ← |P|
For p ∈ P
    n_p ← 0
    i
    for c ∈ C_p do
        // Calculating the number and the positions of the children
           together with the smallest child i
        n_p ← n_p + 1
        B(c) ←true
        if c < i then
            i ← c
    // Getting the middle children
    k ← ⌈n_p/2 − 1⌉
    while true do
        if B(i) then
            k ← k − 1
            if k == 0 then
                break
        i ← i + 1
    j ← i
    if n_p even then
        do
            j ← j + 1
        while ¬B(j)
    // Adding the minimal positions to the graph
    N ← n
    for t ← i to min{j, i + n − 1} do
        if t ∉ T then
            T.ADD(t)
        e ← (p, t)
        E'.ADD(e)
        ξ(e) ← 0
        N ← N − 1
    ...
```

Algorithm 5.2: MINIMIZING EDGE LENGTH WITH ONE SIDE FIXED. Part 2/2 5.1

```
    ...

    // Adding the remaining positions
31  i ← i + 1
32  i ← j − 1
33  if n_p even then
34      l, r ← 2
35  else
36      l, r ← 1
37  ω_i, ω_j ← l
38  for k ← 1 to N do
39      if ω_i ≤ ω_j then
40          t ← i
41          ω_t ← ω_i
42          if B(i) then
43              l ← l + 2
44          i ← i − 1
45          ω_i ← ω_i + l
46      else
47          t ← j
48          ω_t ← ω_j
49          if B(j) then
50              r ← r + 2
51          j ← j + 1
52          ω_j ← ω_j + r
53      if t ∉ T then
54          T.ADD(t)
55      e ← (p, t)
56      E'.ADD(e)
57      ξ(e) ← ω_t

    // Calculating the minimal weight perfect matching and creating the
       drawing
58  H ← (P ∪ T, E', ξ)
59  M ← MINMATCH{H}
60  for (p, t) ∈ M do
61      x(p) ← t
62  return x
```

Example: Suppose we are given a graph with an integer W as shown in Figure 5.1(a) as an instance of `EdgeSumOne`. We have $n = 7$ parents, so we need seven position candidates for the matching. The following steps are shown by C as an example but repeated for all parents.

We begin the initialization process to determine the degree n_C, the lowest child i, and the located children in B. The order in which our children are stored, e.g., in an adjacency list, is d, e, c, f, b, and in this order, we iterate over this list as shown in Figure 5.1. In each step, we take the next child, increment the degree counter n_C, and set the Boolean array B to `true` on this position. Additionally, we keep track of the first child.

After adding all nodes, we have b as the lowest child and want to get to d as the middle child. For that reason, we have to jump $\lceil \frac{n_C}{2} - 1 \rceil = 2$ times to get to the middle node d. So we increment i and look at B where the positions of the children are until we get to the third node.

We calculate the minor weight positions as shown in Figure 5.2. Since we have an odd number of adjacent children, we have only one minimizing position at the middle child. Thus, we add this with relative edge sum 0 to the new graph, in this figure visualized by Ω before we start the counter one left and right of this middle child.

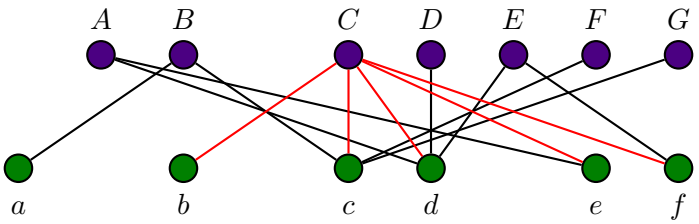
We initialize the variables l, r, and ω_i, ω_j with 1. So the difference between the number of nodes left of i minus the nodes right of i and in i equals $l = 1$. The same yields for j, where $|L(j)| - (n - |L(j)|) = r$. Let us double-check the values of ω_i and ω_j. We compare i and j with the middle position. At i, the edges to the first and second child shrink by one, while the other three children move one further, so placing the parent at i costs one more than the middle position. The same is valid symmetrically for j.

We check this with the actual edge lengths. At the middle position, we have edge distances from left to right of $3+1+0+2+3 = 9$. The positions i and j have costs $2+0+1+3+4 = 10 = 9+1$ and $4+2+1+1+2 = 10 = 9+1$. So in this position, the counters are correct.

Since we have one out of seven positions, we repeat the adding procedure six times: We compare the weight at both counter positions stored in ω_i and ω_j, respectively. They are now equal, so we add position i with the weight 1. Before shifting i one to the left, we look for a child in the old position. If there is one, we increment the counter l by two because the number of positions to the right of i increases while the number of nodes left decreases, so the difference grows by 2. Furthermore, ω_i grows by the new l. Let us calculate the edge sum: We have at the new position i edge lengths $1 + 1 + 2 + 4 + 5 = 13 = 9 + 4$.

At the next step, the left pointer indicates a location with a smaller weight, so this time, we choose j. Starting by adding an edge to this new position, we do not increment r, since $B(j) =$ `false`. We add r to ω_j and move j. The next position comes again from j before we switch back to i. We take another location from i and j and have our seven best positions. Let us recheck the last two weights to see if the weight difference is still correct. The left and right locations have edge sum $0 + 2 + 3 + 5 + 6 = 16 = 9 + 7$ and $6 + 4 + 3 + 1 + 0 = 14 = 9 + 5$.

Figure 5.3(a) shows C with its seven best positions as part of the weighted graph. With the matching algorithms as described in Definition 2.11, we get a solution with minimal weight sum (see Figure 5.3(b)). In this way, we achieve the positions with the minimal edge length sum and our solution to `EdgeSumOne`, see Figure 5.3(c).



(a) A graph in an arbitrary drawing. It forms with an integer W an instance of `EdgeSumOne`.

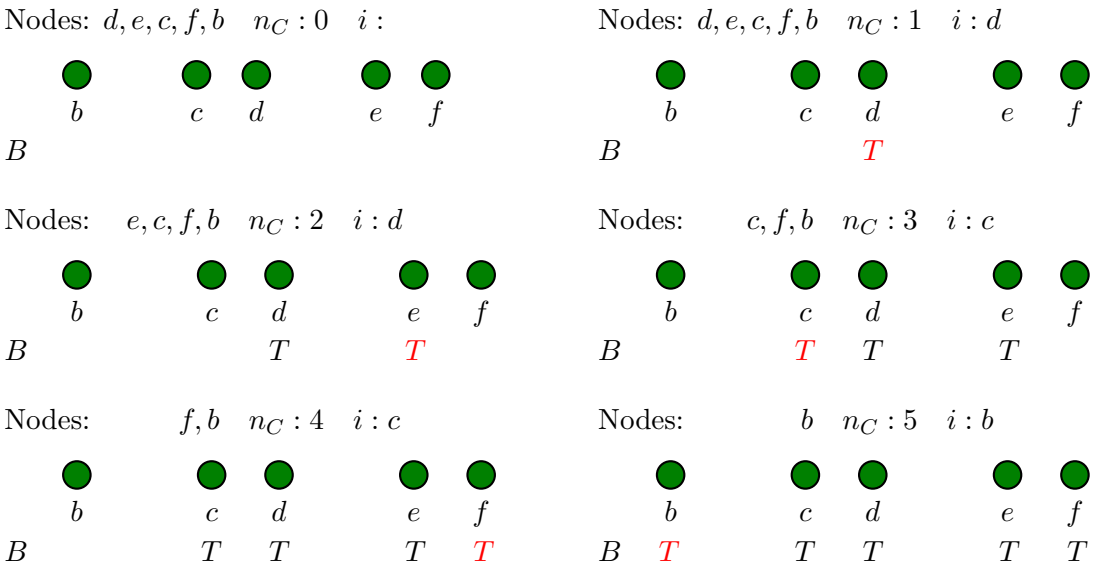


Figure 5.1: The initialization process of parent C for the child number n_C, the first child i and the Boolean array B. This is the first step of the algorithm to compute a minimizing drawing for an instance of `EdgeSumOne`.

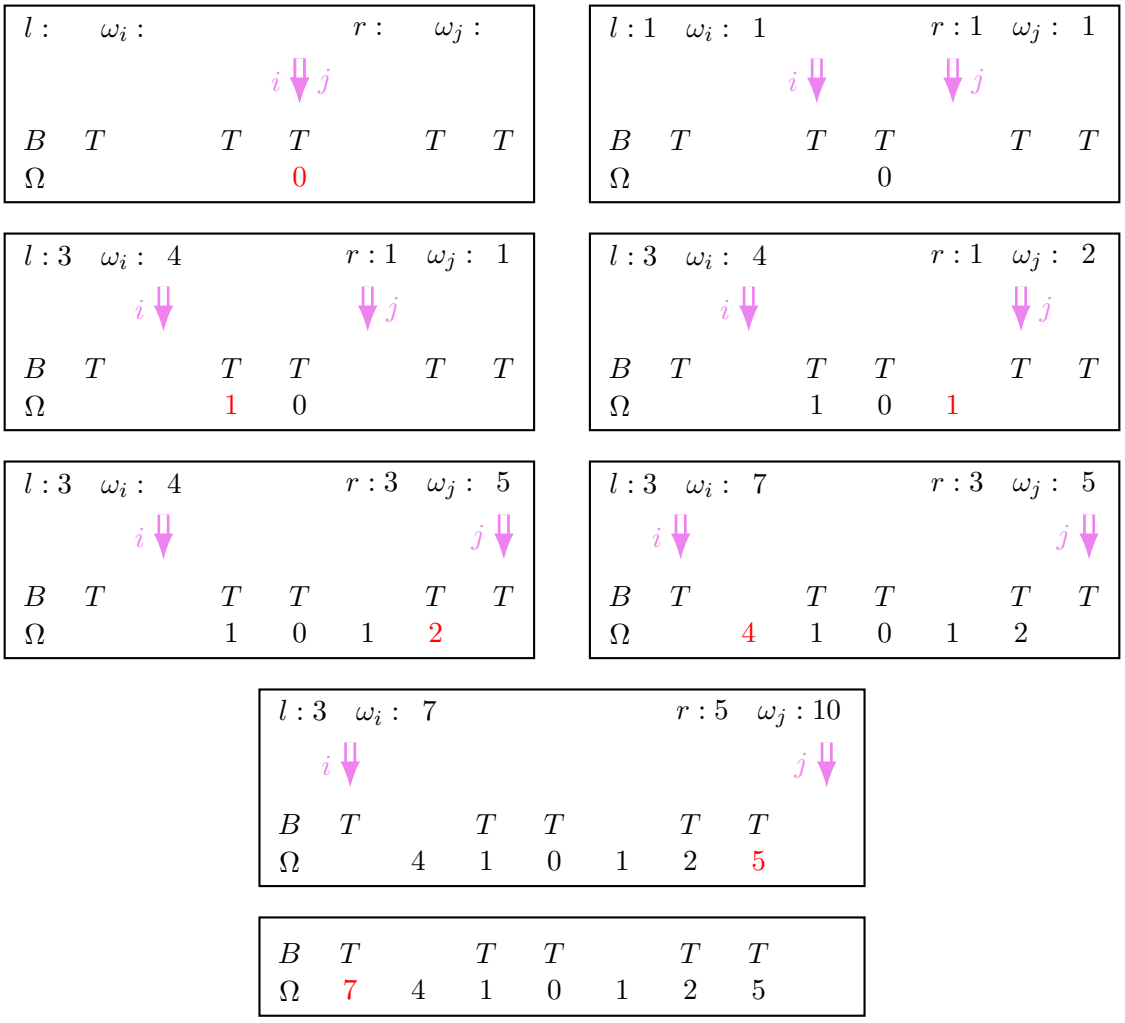
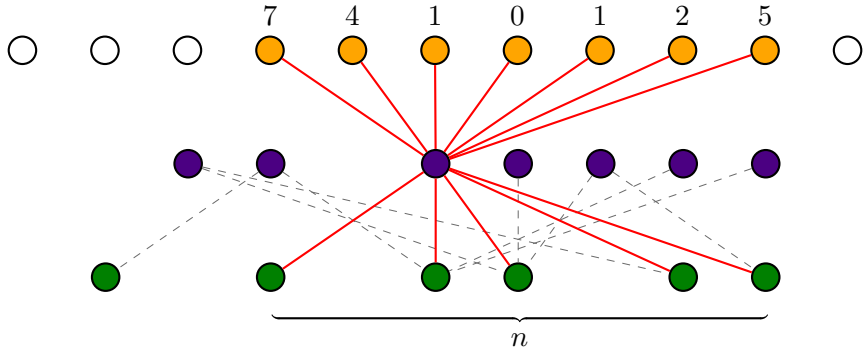
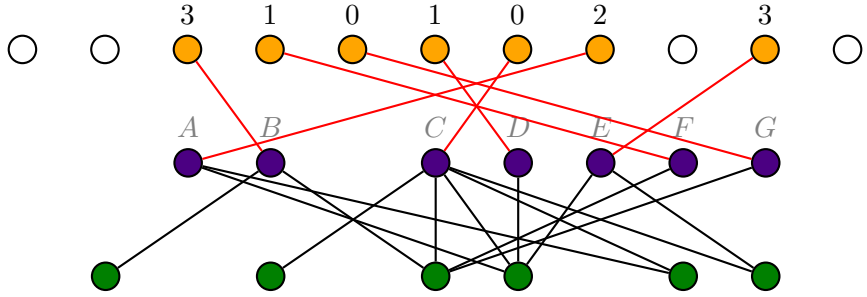


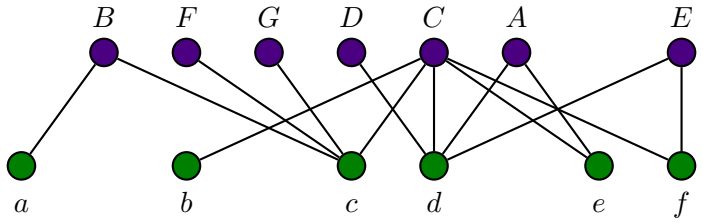
Figure 5.2: Starting at the middle child, the algorithm computes the relative edge lengths sum indicated by Ω.



(a) The matching of C to the seven best positions. The figures over the positions indicate the relative edge sum if we place node C on this position. This calculation happens for all the other nodes, too.



(b) The red edges define a solution of the `EdgeSumOne`. The figures show the weight, i.e., the relative edge sum of the adjacent parent.



(c) The window sum minimizing drawing of G, W as `EdgeSumOne` instance

Figure 5.3: The matching part of this algorithm to find a solution of `EdgeSumOne`

Lemma 5.3. *Suppose we are given a bipartite graph $G = (P \cup C, E)$, for each parent $p \in P$ the following statements hold:*

(S1) *For a position $i \in \mathbb{N}$,*

$$\omega_p(i-1) = \omega_p(i) + n_p - 2|L_p(i)|, \text{ and}$$
$$\omega_p(i+1) = \omega_p(i) + n_p - 2|R_p(i)|.$$

(S2) *For a minimizing position $m \in X_p$, the weight function ω_p is monotonously decreasing on the (discrete) interval $]-\infty, m]$ and increasing on $[m, \infty[$.*

(S3) *If $|C_p|$ is odd, the location of the middle child c, i.e., where $|L_p(c)| = |R_p(c)|$, is the only minimizing position $X_p = \{c\}$.*

(S4) *If $|C_p|$ is even, and $c < d$ are the middle children, i.e., where $|R_p(c)| = \frac{n_p}{2}$ and $|L_p(d)| = \frac{n_p}{2}$, the minimizing positions are $X_p = \{c, \dots, d\}$.*

Proof. We show the validity of each of the statements. Assume we are given a parent $p \in P$. For easier readability, we omit the index p where not necessary.

Statement (S1): Suppose we are given a position i with weight $\omega(i)$. We prove the second equation, the proof for the first is analog.

$$\begin{aligned}
&\omega(i+1)\\
=&\sum_{c\in C_p} |c-(i+1)|\\
=&\sum_{l\in L(i+1)} ((i+1)-l) + \mathbb{1}_{i+1\in M(i+1)}|(i+1)-(i+1)| + \sum_{r\in R(i+1)} (r-(i+1))\\
=&\sum_{l\in L(i)\cup M(i)} ((i+1)-l) + \sum_{r\in M(i+1)} (r-(i+1)) + \sum_{r\in R(i)\setminus M(i+1)} (r-(i+1))\\
=&\sum_{l\in L(i)} ((i+1)-l) + \sum_{l\in M(i)} ((i+1)-l) + \sum_{r\in R(i)} (r-(i+1))\\
=&\sum_{l\in L(i)} (i-l) + |L(i)| + \sum_{l\in M(i)} (l-i) - |M(i)| + \sum_{r\in R(i)} (r-i) - |R(i)|\\
=&\sum_{l\in L(i)} (i-l) + \mathbb{1}_{i\in M(i)}|i-i| + \sum_{r\in R(i)} (r-i) - |R(i)| + (|L(i)| - |M(i)|)\\
=&\sum_{c\in C_p} |c-i| - |R(i)| + (|C_p| - |R(i)|)\\
=&\omega(i) + n_p - 2|R(i)|
\end{aligned}$$

Statement (S2): We only have to show that the derivative $\Delta(i) = \omega(i+1) - \omega(i)$ is monotonously increasing and has a negative and positive function value. This property will prove that $\omega(i)$ is a function that starts high and gets lower before it increases again.

$$\begin{aligned}
\Delta(i) =&\omega(i+1) - \omega(i)\\
=&\omega(i) + n_p - 2|R(i)| - \omega(i)\\
=&n_p - 2|R(i)|.
\end{aligned}$$

So with $i < f_p$, $R(i) = C_p$ and $\Delta(i) = n - 2|R(i)| = n - 2n = -n$ and with $i > l_p$, $R(i) = \emptyset$ and $\Delta(i) = n$. Furthermore, Δ is monotonously decreasing with $|R(i)|$ increasing. Bearing in mind that $R(i+1) \subset R(i)$, $|R(i+1)| \leq |R(i)|$, this is also proven.

Statement (S3): Let m be the position of the middle child, where $|L(m)| = |R(m)|$. Then $M(m) = \{m\}$ and

$$n_p = |C_p| = |L(m) \cup M(m) \cup R(m)| > 2|R(m)| = 2|L(m)|.$$

With *(S1)* we know:

$$\omega(m-1) = \omega(m) - n_p - 2|L(m)| > \omega(m).$$
$$\omega(m+1) = \omega(m) - n_p - 2|R(m)| > \omega(m).$$

Following from (S2), there is only one global minimum, so $X_p = \{m\}$.

Statement (S4): Let $c < d$ be the middle children. For $i \in \{c, \dots, d-1\}, j \in \{c+1, \dots, d\}$, $2|R(i)| = n_p = 2|L(j)|$, while $2|L(c)| < n_p > 2|R(d)|$, so with (S1) we know:

$$\begin{aligned}
\omega(i) &= \omega(i) + n_p - 2|R(i)| &&= \omega(i+1).\\
\omega(j) &= \omega(j) + n_p - 2|L(j)| &&= \omega(j-1).\\
\omega(c-1) &= \omega(c) + n_p - 2|L(c)| &&> \omega(j).\\
\omega(d+1) &= \omega(d) + n_p - 2|R(d)| &&> \omega(j).
\end{aligned}$$

Putting this together, we get:

$$\omega(c-1) > \omega(c) = \omega(c+1) = \cdots = \omega(d-1) = \omega(d) < \omega(d+1).$$

Following from (S2), that is enough to show that $X_p = \{c, \dots, d\}$. □

With the help of this lemma, it is now possible to prove our next result:

Theorem 5.4. `MinimumEdgeSumOneSide` *is solvable in* $\mathcal{O}(|P|^{2+o(1)} + |P| \cdot S_P)$ *time.*

Proof. We show the following statements, similar to the proof of Theorem 4.4 of `WinSumP`.

(P1) The matching part finds a perfect matching.

(P2) The sum of the weights in the matching is equal to the relative edge sum with parents at the matched positions.

(P3) For each parent, the graph matches the n positions with minimal edge length sum.

(P4) The provided drawing is sum-of-the-edge-lengths minimizing.

Suppose we are given a bipartite graph $G = (P \cup C, E)$, a drawing $x : C \to \mathbb{Z}$, and an integer $W \in \mathbb{N}$ as an instance of `EdgeSumOne`.

Statement (P1): In this algorithm, we match $n := |P|$ parents again to n locations, so a perfect matching exists regarding Corollary 2.10 of Hall's marriage theorem. So this statement is similar to (K1) of Theorem 4.4.

Statement (P2): Consider $p \in P$. We prove the correctness of the inner loop using these invariants for iterations with counter i, j:

$$\begin{aligned}
r &= |L(j-1)| + |M(j-1)| - |R(j-1)|\\
l &= |R(i+1)| + |M(i+1)| - |L(i+1)|\\
\omega_j &= \hat{\omega}_p(j)\\
\omega_i &= \hat{\omega}_p(i)
\end{aligned}$$

We prove only the first and third invariants, numbers two and four are analogous.

If n_p is odd, let c be the middle child, and we start at $j = c + 1$. Per definition is $|L(c)| = |R(c)|$ and $M(c) = \{c\}$. So r is initialized correctly with

$$r = |L(j-1)| + |M(j-1)| - |R(j-1)| = |L(c)| + |M(c)| - |R(c)| = 1.$$

If n_p is even, let $c < d$ be the middle children, so $|L(d)| = \frac{n_p}{2}$ and we start with $j = d + 1$.

$$\begin{aligned} r &= 2 \\ &= n_p + 2 - n_p \\ &= 2|L(d)| + 2|M(d)| - n_p \\ &= |L(d)| + |M(d)| - |R(d)| \\ &= |L(j-1)| + |M(j-1)| - |R(j-1)| \end{aligned}$$

Both ways, with (S3) or (S4) we know $\hat{\omega}_p(j-1) = 0$. So with use of (S1),

$$\begin{aligned} \omega_j &= r \\ &= |L(j-1)| + |M(j-1)| - |R(j-1)| \\ &= \hat{\omega}_p(j-1) + n_p - 2|R_p(j-1)| \\ &= \hat{\omega}_p(j) \end{aligned}$$

is initialized correctly, too.

So both variables hold the equation at the beginning of the first iteration. In the case $\omega_i \leq \omega_j$, the variables j, r, and ω_j do not change, so consider the other case. The variable r is set to $r + \mathbb{1}_{j \in M(j)} \cdot 2$. Here, the updated variables are r' and ω'_j.

$$\begin{aligned} r' &= r + \mathbb{1}_{j \in M(j)} \cdot 2 \\ &= (|L(j-1)| + |M(j-1)|) - |R(j-1)| + 2|M(j)| \\ &= |L(j)| - (|R(j)| + |M(j)|) + 2|M(j)| \\ &= |L(j)| + |M(j)| - |R(j)| \\ &= |L((j+1)-1)| + |M((j+1)-1)| - |R((j+1)-1)|. \end{aligned}$$

$$\begin{aligned} \omega'_j &= \omega_j + r' \\ &= \hat{\omega}_p(j) + |L(j)| + |M(j)| - |R(j)| \\ &= \hat{\omega}_p(j) + n_p - 2|R_p(j)| \\ &= \hat{\omega}_p(j+1). \end{aligned}$$

So both variables get updated correctly.

Together with the invariant, we proved that the algorithm calculates the correct relative edge sum for all values of i and j. However, some values neither i nor j reach. These values are all minimizing positions, but with (S2) and (S3), we prove that these are at the middle child or from the first to the second middle child, respectively.

This proves (P2).

Statement (P3): The lemma of this section shows us with (S2) the particular form of ω_p. So we can start at the calculated minimum and choose from left and right greedily the successive positions. This proof is analogous to (K3).

Statement (P4): This statement follows from the others. We know that the algorithm matches the parents to the positions with the smallest relative weights. The matching algorithm returns the best locations for these parents with minimal weight sum. Since the weight sum equals the relative edge length sum, this is minimal, too, and the relative edge length sum is only a constant away from the actual weight sum. So the provided drawing has a minimal edge length sum.

Running Time: Once again, the algorithm has two parts: creating the weighted graph and calculating the `MinMatch`. The latter is not different from the one from `WinSumP`, so it still runs in $\mathcal{O}(|P|^{2+o(1)})$ just as the running time of Proof 4.4. The former part has an all-embracing loop with $|P|$ iterations. Each iteration calculates the children's locations with $|C_p| \leq S_P + 1$ iterations. Additionally, the next part jumps from the start of the span to the middle child, which needs less than S_p time. The following two loops together add in each iteration exactly one edge to the graph up to $|P|$ in total. So the former part runs in $\mathcal{O}(|P| \cdot (|P| + S_P + 1 + S_P + |P|)) = \mathcal{O}(|P| \cdot (|P| + S_P))$ time, which shows our total running time of $\mathcal{O}(|P|^{2+o(1)} + |P| \cdot S_P)$. □

Variant 1: We adopted the sum-of-the-window minimizing algorithm of Section 4.2 in Variant 1 to the problem `WinSumLoc` (Definition 4.6). This modified algorithm takes a set of possible positions as input for each parent. The same modification is possible with the problem in this section, too.

Definition 5.5 (`MinimumEdgeSumOneSideLocations` (`EdgeSumOneLoc`))**.** *Suppose we are given a bipartite graph $G = (P \cup C, E)$, a partial drawing $x : C \to \mathbb{Z}$, an integer $W \in \mathbb{N}$ and for each parent $p \in P$ a set of possible locations L_p. Consider $L = \bigcup_{p \in P} L_p$ and $d_{\max} = \max_{p \in P} |C_p|$ as the maximum degree of any parent. Is there a completing drawing $x : P \to L$ such that the following holds?*

$$\sum_{(p,c) \in E} \lambda((p,c)) \leq W$$
$$\wedge \qquad \forall p \in P : x(p) \in L_p.$$

The modification for this problem is similar to the one for the window sum variant. However, if the locations are dispersed, using the same method to calculate the relative edge length would require iterating at least from the first to the last possible position. After n steps with this algorithm, we only calculated n best positions around the minimum. Thus for each position, we have to calculate the edge sum manually with the distance to all children. This part takes $\mathcal{O}(d_{\max}|L|)$ time.

Nevertheless, the matching part is again solvable in $\mathcal{O}(|L|^{1+o(1)})$ time, so we get the result of the first variant:

Corollary 5.6. `MinimumEdgeSumOneSideLocations` *is solvable in $\mathcal{O}(|L|^{1+o(1)} + d_{\max}|L|)$.*

Variant 2: Now we examine if we can calculate a drawing for the parents such that not the edge length sum but the longest edge gets minimal. The longest edge is always to one of two marginal children, so we only need these children as input, like in the window minimizing variant.

Definition 5.7 (`MinimumMaxEdgeOneSide` (`MaxEdgeOne`))**.** *Suppose we are given a set of parents P, the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ in a partial drawing $x : S \to \mathbb{Z}$, and an integer $W \in \mathbb{N}$. Is there a completing drawing $x : P \to \mathbb{Z}$, such that the following holds?*

$$\forall (p,c) \in E : \lambda((p,c)) \leq W.$$

Calculating the longest edge is more straightforward than calculating the sum. The minimal value is the center of the span. Hence, our possible positions are at this point and equally distributed to the remaining locations left and right.

Furthermore, we have to adapt our matching algorithm, in this case, because otherwise, it would find the best solution for the sum of all the longest edges of the parents, which is not what we want to achieve. Using again `BottMatch` (Definition 2.12) finds the solution with the longest edge of all parents as small as possible.

For each parent, calculating the span's center runs in constant time and adding the nodes to the left and the right runs in linear time. Thus the running time of the first part of our algorithm would stay the same, just the different matching algorithm provided by Peterson et al. [PL88] has a running time of $\mathcal{O}(n\sqrt{mn}) = \mathcal{O}(n^{2.5})$. So we can conclusively get the following result:

Corollary 5.8. `MinimumMaxEdgeOneSide` *is solvable in* $\mathcal{O}(|P|^{2.5})$.

5.2 Moving both parents and children

Our last result indicates that finding a drawing with a minimum edge length sum with one side fixed is doable in polynomial time. This result follows from the similar problem when minimizing the total window size. Since minimizing window sum gets $\mathcal{NP}$-complete with no fixed size, we likewise suspect that this relationship holds for the edge length sum, too. Even the proof is similar and can be reused.

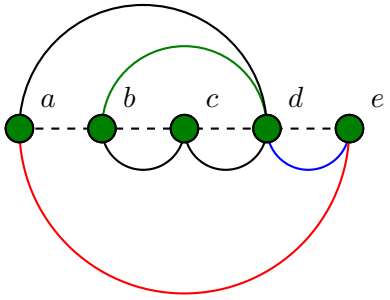
Definition 5.9 (`MinimumEdgeSumBoth` (`EdgeSumBoth`))**.** *Suppose we are given a bipartite graph* $G = (P \cup C, E)$ *and an integer* $W \in \mathbb{N}$*, is there a drawing* $x : C \cup P \to \mathbb{Z}$ *(Table 2.1) such that the following holds?*

$$\sum_{e \in E} \lambda(e) = \sum_{(p,c) \in E} |x(p) - x(c)| \leq W.$$

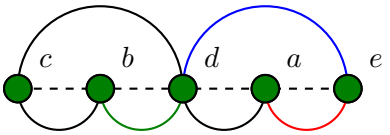
The concept behind this proof is the same as in Section 4.3. We reduce `LAP` (Definition 2.6) to this problem. For that reason, we reuse the block graph (Definition 4.10) and the tight realization (Definition 4.11). Each block-parent in a block graph is of degree one, so its window size equals its edge length. Furthermore, the total edge length of an edge-parent placed within its span equals its window size, too. For d being the distance of an edge-parent $p \in P$ to the span, the sum of the two edge lengths is equal to the span size plus $2d$ while its window size is its span size plus d. In Section 4.3, we showed that with a large k, all edge-parents in a sum-of-the-window minimizing drawing are placed within their span. With the cost difference for edge lengths of an edge-parent being even larger when placed outside, it is easy to prove that edge-parents in sum-of-the-edge-length minimizing drawings are placed within their span, too.

Example: We use the same example as in Section 4.3 shown in Figure 5.4. Consider a graph G as shown in a drawing in Figure 5.4(a). Let G with $W = 10$ be an instance of `LAP`. In this drawing, the edge length sum is $4 + 3 + 2 + 3 \cdot 1 = 12 > W$. In Figure 5.4(c), this drawing is transformed to a block graph H and visualized in a tight realization. The edge length sum of the edge-parents is $k + 3$ for A, $2k + 5$ for B, $3k + 5$ for C, $4k + 6$ for D, $k + 2$ for E and k for F. So the total edge length (for edge-parents) is $12k + 21 > W \cdot k$. Since all edge-parents are within their span, this is the same value as the sum of the window sizes.

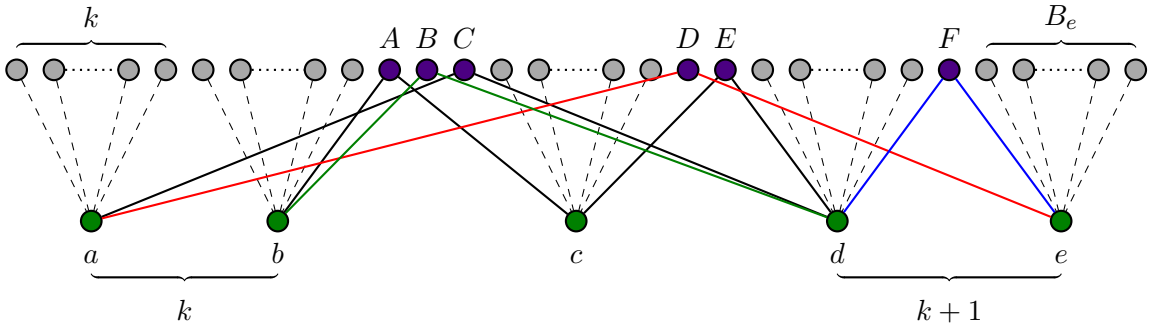
Moving C between the blocks B_a and B_b would decrease the edge length sums of A and B while the edge length sums of C and D stay the same. So there is a drawing with an



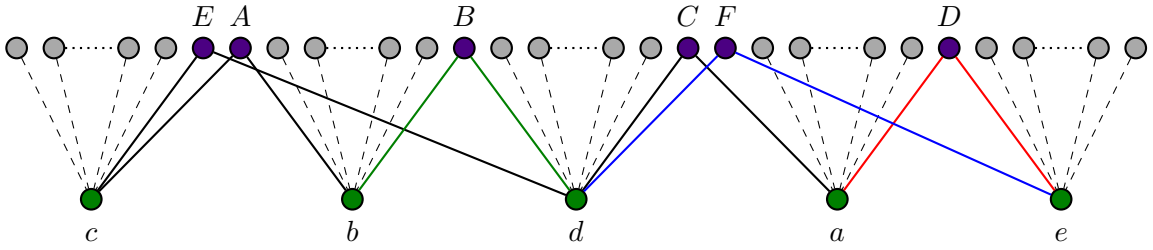
(a) An arbitrary graph G as instance of the LAP



(b) The optimal ordering of G in the LAP



(c) The tight realization of the block graph originating from G.



(d) A tight realization with smaller and quasi-minimal edge length sum, originating from the optimal ordering of G in (b)

Figure 5.4: A tight realization of a sample of LAP two versions with different window sum.

edge length sum of at least two lower than the one in this drawing. However, moving B between B_a and B_b instead would increase the length of both edges of B by $\frac{k}{2}$. Hence it is better to keep all edge-parents in their span.

Figure 5.4(b) shows the optimal ordering of G with an edge length sum of $2 \cdot 2 + 4 \cdot 1 = 8 < W$. So this instance of LAP is solvable. In Figure 5.4(d), the block graph is still a tight realization, but the order of the children is the same as the nodes in the optimal ordering in the monoline problem. The edge length sum in this drawing is $8k + 12 < W \cdot k$ (for large k). This value equals the window sum again.

Before we can reduce LAP to EdgeSumBoth, we show that if k is large enough, the edge length sum equals the window size sum, and with that, the minimizing drawing by one metric is a minimizing drawing of the other metric, too.

Lemma 5.10. *If $k > \max\{n, |E| + 4\}$, for every block graph $H = ((P \cup B) \cup V, E')$ and realization x the two statements are equivalent:*

(T1) *H has minimum edge lengths sum W' in realization x.*

(T2) *H has minimum window size sum W' in realization x.*

Proof. Suppose we are given a drawing x of a block graph $H = ((P \cup B) \cup V, E')$ with minimum edge length sum. We prove the validity of these statements:

(K1) Child v is placed strictly within its outermost block parents, i.e., there are block parents $v_i, v_j \in B_v$ with $v_i < v < v_j$.

(K2) Each edge-parent is placed in its span.

(K3) (T1) $\iff$ (T2).

Statement (K1): This proof is an adaption of the proof of the first property in Lemma 4.12, where we proved this third property of a tight realization (Definition 4.11) for the window version, too. Assume w.l.o.g. all block parents of a child $v \in V$ are located on positions greater or equal to v. Hence, v has at least $k-1$ adjacent parents to the right and, at most, $|E|$ to the left. Moving v one step right would tighten at least $k-1$ edges and broaden $|E|$ edges to the edge-parents' left and possibly the one from the block parent at position v. With $k-1 > |E|+1$, this leads to a smaller edge length sum, contradicting minimality. If this position is occupied by another child, moving v two steps right would reduce at least $k-2$ edges and broaden at most $|E|+2$ edges. Since $|C| = n$, after at most n steps v finds a place and even $k - n > |E| + n$ holds. So no matter where the other parents are located, there is a guaranteed better place free for v. Thus we know there is always an adjacent block-parent left and right of each child.

Statement (K2): This proof is also an adaption of the proof in Lemma 4.12 of Property (P1). Assuming without loss of generality, we have parent $p_{uv} \in P$ with $u < v < p_{uv}$. Using Statement (K1) which we have already proven, there exists $v_1 \in B_v$ with $v_1 < v$. Consider the new drawing y' we get by swapping v_1 and p_{uv}. Let us call the edge lengths e_{pu}, e_{pv} for the edges of p_{uv} and e_v for the edge of v_1 in drawing x and e'_{pu}, e'_{pv}, e'_v for the edges in drawing y'. So the following is valid, if $u \leq v_1 < v$ (Figure 5.5(a)):

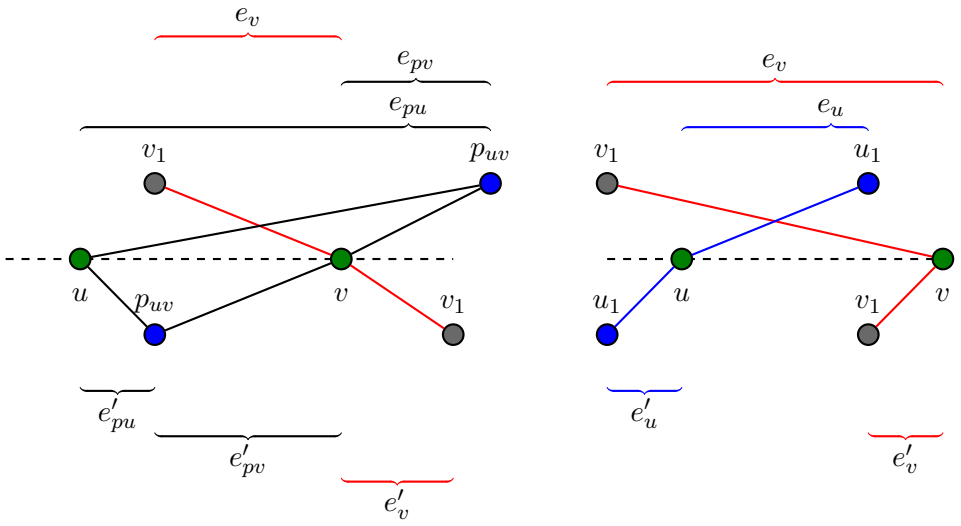
$$\begin{aligned} e_{pu} + e_{pv} + e_v &= (x(p_{uv}) - u) + (x(p_{uv}) - v) + (v - x(v_1)) \\ &= (x(p_{uv}) - x(v_1)) + (x(v_1) - u) + (x(p_{uv}) - v) + (v - x(v_1)) \\ &= (x(p_{uv}) - x(v_1)) + (y'(p_{uv}) - u) + (y'(v_1) - v) + (v - y'(p_{uv})) \\ &= (x(p_{uv}) - x(v_1)) + e'_{pu} + e'_v + e'_{pv} \\ &> e'_{pu} + e'_v + e'_{pv} \end{aligned}$$

So this swap leads to an improvement, contradicting minimality, so $v_1 < u$. With Property (K1), we know there is a $u_1 \in B_u$ with $u < u_1$, too. Let us call the edge length e_u for the edge of u_1 in drawing x and e'_u for the edge in drawing y'. If $u_1 \leq v$ (Figure 5.5(b)):

$$\begin{aligned} e_v + e_u &= (v - x(v_1)) + (x(u_1) - u) \\ &= (v - x(u_1)) + (x(u_1) - u) + (u - x(v_1)) + (x(v_1) - u) \\ &= (v - y'(v_1)) + 2(x(u_1) - u) + (u - y'(u_1)) \\ &= e'_v + 2(x(u_1) - u) + e_u \\ &> e'_v + e'_u \end{aligned}$$

The second case, if $u_1 > v$, is analogous. The swap would be an improvement since v_1 is closer to u than v, and u_1 is closer to v than u. So this contradicts minimality, too, which proves all edge-parents are placed inside their span.

Statement (K3): Consider an edge-parent $p_{uv} \in P$. W.l.o.g. $u \leq p_{uv} \leq v$. So the window of p_{uv} is $v - u$. Additionally, the edge sum of p_{uv} is $v - p_{uv} + p_{uv} - u = v - u$. Thus for the edge-parent, the window and the edge sum are the same. The block-parents have one edge each, so the window equals its length, too.



(a) The changes made by swapping p_{uv} with a block-parent between u and v before the swap (above) and after the swap (below)

(b) The changes made by swapping v_1 with u_1 before the swap (above) and after the swap (below)

Figure 5.5: These are two swaps that would improve the edge sum to show that all parents are placed in their span.

Suppose we are given a drawing with minimum edge lengths W. We know that the window sum and the edge length sum are the same, so the window sum is minimal, too. Given a drawing with minimum window sum W, we know from Lemma 4.12 that this is a tight realization. So in this drawing, all parents are placed inside their span. Hence the window sum equals the edge sum. Thus, this minimizes the edge length sum, too.

□

Using this lemma, we can prove our next result:

Theorem 5.11. `MinimumEdgeSumBoth` *is $\mathcal{NP}$-complete.*

Proof. For proving $\mathcal{NP}$-completeness, we have to show membership in $\mathcal{NP}$ and reduce in polynomial time an instance of `LAP` to `EdgeSumBoth`.

We start with the first property. Suppose we are given with $G = (P \dot\cup C, E)$ and $W \in \mathbb{N}$ an instance of `EdgeSumBoth`. Calculating the edge sum is possible in a time polynomial to the number of edges, which shows `EdgeSumBoth` $\in \mathcal{NP}$.

This proof uses the results of the proof of Theorem 4.13. Suppose we are given with $G = (V, E)$ and $W \in \mathbb{N}$ as an instance of the `LAP`. Let $k > \max\{n, |E| + 2n, 3|E|^2 - |E|\}$ be odd. We create a block graph $H = ((P \dot\cup B) \dot\cup V, E')$. Thus we use W' as we calculated it in the proof of Theorem 4.13. We show the equality of the following statements.

(J1) G, W as an instance of `LAP` is solvable.

(J2) H, W' as an instance of `WinSumB` is solvable.

(J3) H, W' as an instance of `EdgeSumBoth` is solvable.

(J1) $\iff$ (J2): This equality is proven in Theorem 4.13.

(J2) $\implies$ (J3): Suppose H, W' as an instance of `WinSumB` is solvable, so there exists a window minimizing realization with a window sum of at most W'. With Lemma 5.10, in

this realization, the edge length is at most W', too. So H, W' is also solvable as an instance of `EdgeSumBoth`.

(J3) $\implies$ (J2): Suppose H, W' as an instance of `EdgeSumBoth` is solvable, so there exists an edge minimizing realization with edge length sum of at most W'. With Lemma 5.10, in this realization, the window sum is at most W', too. So H, W' is also solvable as an instance of `WinSumB`.

The polynomiality of the reduction is derived from the `WinSumB` variant. There we proved that creating a block graph runs in time polynomial to the number of edges and nodes since the number of nodes and edges are $v(k+1) + |E|$ and $vk + 2|E|$ with k polynomial, too.

Hence `EdgeSumOne` is in $\mathcal{NP}$ and $\mathcal{NP}$-hard, so $\mathcal{NP}$-complete. □

6. Radial drawings

Most drawings of bipartite graphs order one partition on one side and the other opposite. However, in this type of drawing, the distance from the first to the last node is considerable. So in a different way of drawing, on two concentric circles with different radii, every node has two neighbors.

Additionally, displaying information in a radial visualization on two concentric circles brings the resolving figure in a more square-shaped form. Without changing the distance between the parallel lines, this kind of visualization grows in one dimension when adding nodes. Though adding nodes on concentric circles makes their radii larger, they grow equally in two dimensions.

In this chapter, we discretize the two circles into τ equidistant points for a $\tau \in \mathbb{N}^+$. Parallel drawings map the nodes to $\mathbb{Z}$, but by placing the partitions on the boundaries of an annulus, the mapped values are these finite sets of points. So we define $\mathbb{Z}_\tau = \mathbb{Z}/\tau\mathbb{Z} = \{0, \dots, \tau - 1\}$ as integer ring modulo τ.

Definition 6.1 (Radial drawing)**.** *Let $G = (P \cup C, E)$ be a bipartite graph and $\tau \in \mathbb{N}^+$. We call a semi-injective drawing (Definition 2.1) $\mathring{x} : P \cup C \to \mathbb{Z}_\tau$ a* radial drawing *and τ* fraction rate *of the radial drawing.*

We draw $\mathring{x} : P \cup C \to \mathbb{Z}_\tau$ on two concentric circles with radius $0 < r_p < r_c$, respectively. Here, we assign $p \in P, c \in C$ the coordinates $(r_p, \mathring{x}(p) \cdot \frac{2\pi}{\tau})$ and $(r_c, \mathring{x}(c) \cdot \frac{2\pi}{\tau})$.

This way of drawing induces a new definition of spans and windows.

Definition 6.2 (Spans and windows in radial drawings)**.** *Consider $\mathring{x}$ as a radial drawing of graph $G = (P \cup C, E)$ with fraction τ. For $p \in P$ let*

- $\{a, \dots, b\} = \begin{cases} \{a, \dots, b\}, & \text{if } a \leq b \\ \{a, \dots, \tau - 1, 0, \dots b\}, & \text{if } b > a \end{cases}$ *be an interval in $\mathbb{Z}_\tau$.*
- $\mathring{S}_p = \arg\min_{\{f_p, \dots, l_p\} \subseteq \mathbb{Z}_\tau} \{|\{f_p, \dots, l_p\}| \mid c \in \{f_p, \dots, l_p\}, \forall c \in C_p\}$ *the* span *of p with* marginal children *f_p, l_p and* span size *$\mathring{s}_p = |\mathring{S}_p| + 1$.*
- $\mathring{W}_p = \arg\min_{\{\tilde{f}_p, \tilde{l}_p\} \subseteq \mathbb{Z}_\tau} \{\{|\tilde{f}_p, \dots, \tilde{l}_p\}| \mid c, p \in \{\tilde{f}_p, \dots, \tilde{l}_p\}, \forall c \in C_p\}$ *the* window *of p with* marginal nodes *$\tilde{f}_p, \tilde{l}_p$ and* window size *$\mathring{w}_p = |\mathring{W}_p| + 1$.*

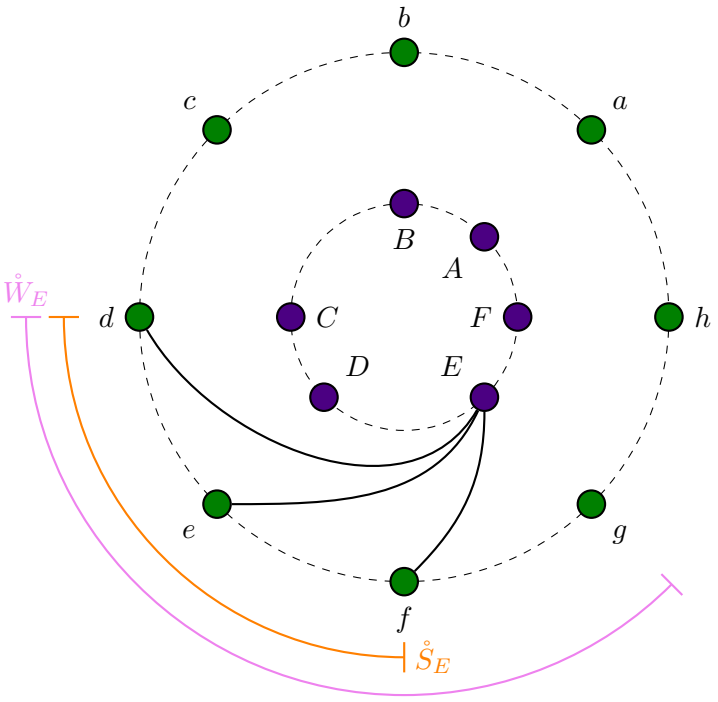


Figure 6.1: A simple graph in a radial drawing.

A visualization is shown in Figure 6.1. In this chapter, when talking about a node's window size, span size, or edge length, we will stay in $\mathbb{Z}_\tau$. Only for the sum of window sizes and edge lengths, we use $\mathbb{N}$ instead.

However, in this kind of visualization, there are disadvantages, too. As shown in the drawing of Figure 6.1, straight edges without intersections to the inner circle are not guaranteed anymore. So let us introduce a new concept:

Definition 6.3 (Frame placement)**.** *Suppose we are given a bipartite graph $G = (P \dot\cup C, E)$, a radial drawing $\mathring{x}$ yields a frame placement of size $\varphi \in \mathbb{N}$ with $0 < \varphi \leq \frac{\tau}{4}$ if the distance from each parent to its adjacent children is at most φ, i.e., the following holds:*

$$\forall p \in P : \forall c \in C_p : |\mathring{x}(p) - \mathring{x}(c)| \leq \varphi.$$

This frame placement can be considered in the following way: First, take a parent on the inner circle and its line to the center. Then, the two sectors of the circle of size $\frac{\varphi}{\tau}$ on both sides of the line shape the frame. So the frame size corresponds to the angle of the sector, and a drawing yields this frame size if all children are inside these sectors. A visualization is given in Figure 6.2.

Since in this thesis, we use projected edge lengths (Definition 2.5), a radial drawing yields a frame placement of size φ if and only if all edge lengths are smaller than or equal to φ.

6.1 Moving parents with fixed children and frame size

In a radial drawing, there are two differences from a parallel drawing; We limit the distance from parent to child via frames and place the first and last nodes next to each other. We want to examine if our algorithm for sum-of-the-window-size minimizing also works in this drawing method.

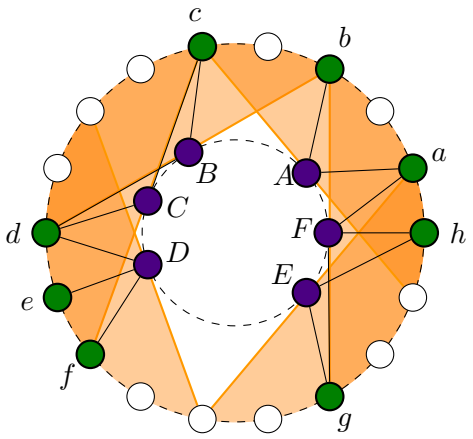


Figure 6.2: A graph in a radial drawing with frame size 3

Definition 6.4 (`MinimumWindowSumParentsRadial` (`WinSumPR`))**.** *Suppose we are given a set of parents P, the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ in a partial radial drawing $\mathring{x} : S \to \mathbb{Z}_\tau$, an integer $W \in \mathbb{N}$ and a frame size $\varphi \leq \frac{\tau}{4}$. Is there a completing drawing $\mathring{x} : P \to \mathbb{Z}_\tau$ holding the frame size φ such that the following holds?*

$$\sum_{p \in P} \mathring{w}_p \leq W.$$

We can reuse the matching algorithm. We only have to find a different solution to find the n possible positions for the matching graph and calculate their weight. For that reason, we only need to know where the positions are. Consider a parent $p \in P$. We have to find the interval, where each child has at most distance φ, i.e., the intersection $\bigcap_{c \in C_p} \{c - \varphi, \dots, c + \varphi\}$. The lowest position, which is in the range of the end of the span, is $l_p - \varphi$. At the same time, the highest position in the range of the start of the span is $f_p + \varphi$. If a position is larger than the $l_p - \varphi$, for each child $c \in C_p$, it is also higher than $c - \varphi$. Symmetrically the same is valid for f_p, which shows the next proposition:

Proposition 6.5. *For each parent $p \in P$, $I_p \coloneqq \{l_p - \varphi, \dots, f_p + \varphi\}$ contains the possible positions to place p without breaking the frame placement. There is no position for this parent with frame size φ, if $I_p = \emptyset$, i.e., the span size $\mathring{s}_p > 2\varphi$.*

If there is a possible position for each parent, we can reuse our technique to add the span to the number of possible positions and then take positions left and right until we have enough. In addition, we have to check if these positions are in this interval. Otherwise, we start not at f_p but at $f_p - (\varphi - \mathring{s}_p)$.

Algorithm: Given an instance of `WinSumPR` with graph $G = (P \cup C, E)$, partial drawing $\mathring{x} : S \to \mathbb{Z}_\tau$, frame size φ and $W \in \mathbb{N}$, create a weighted bipartite graph $H = (P \cup T, E', \xi)$:

For each parent $p \in P$ execute the following steps: If the span size is larger than 2φ, stop and return `false`. The start point k is either if the span is smaller than $n \coloneqq |P|$, $\left\lceil \frac{n - (\mathring{s}_p + 1)}{2} \right\rceil$ before, or otherwise right at the start of the span. Both ways, if this start point has more than distance φ to l_p, set the start k at $l_p - \varphi$. We continue to add points at positions $t \leftarrow k$ and repeat this either n times or until we reach $f_p + \varphi + 1$.

With the created graph, calculate a `MinMatch`. If that is not possible, return `false`, else place each parent on the matched position in the result set M.

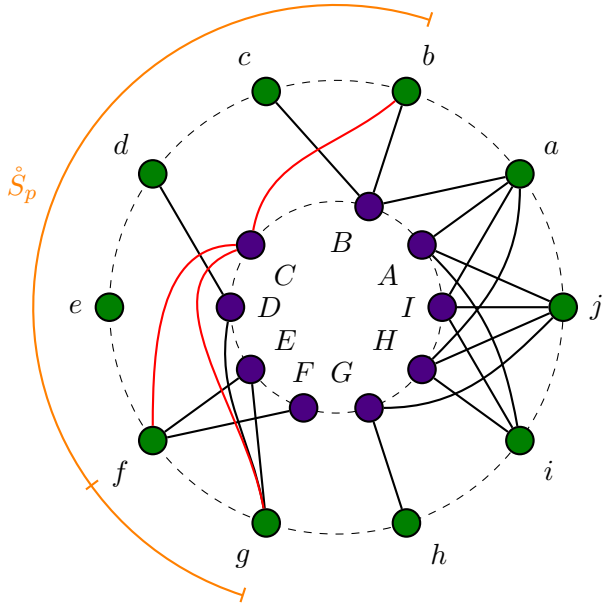
Algorithm 6.1: MINIMIZING WINDOW BY MOVING PARENTS WITH FRAME SIZE

Input: A set of parents P of a graph, the span set $\mathsf{S} = \{(\mathsf{f_p}, \mathsf{l_p}) \mid \mathsf{p} \in \mathsf{P}\}$ in a partial radial drawing $\mathring{\mathsf{x}} : \mathsf{S} \to \mathbb{Z}_\tau$
Data: Weighted bipartite graph $\mathsf{H} = (\mathsf{P} \cup \mathsf{T}, \mathsf{E'}, \xi)$
Output: Drawing $\mathring{\mathsf{P}} \to \mathbb{Z}_\tau \mathsf{x}$ of parents $\mathsf{p} \in \mathsf{P}$ with minimal window sum, or `false`, if it is not placeable

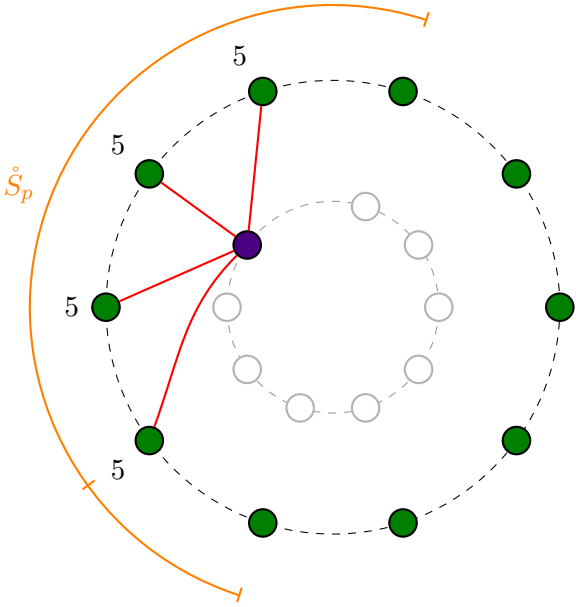
```
// Initialization
1  E' ← ∅
2  n ← |P|
// Adding the weighted edges
3  for p ∈ P do
4      if ŝ_p > 2φ then
5          RETURN false
6      else if ŝ_p < n − 1 then
7          k ← max{f_p − ⌈(n−(s_p+1))/2⌉, l_p − φ} mod τ
8      else
9          k ← max{f_p, l_p − φ}  mod τ
10     for t ← k to min{k + n − 1, f_p + φ} do
11         e ← (p, t  mod τ)
12         E'.ADD(e)
13         if t  mod τ < f_p then
14             ξ(e) ← l_p − t  mod τ
15         else if t ≤ l_p then
16             ξ(e) ← ŝ_p
17         else
18             ξ(e) ← t − f_p  mod τ
// Calculating the minimal weight perfect matching and creating the placing
19 H ← (P ∪̇ Z_τ, E', ξ)
20 M ← MINMATCH{H}
21 if |M| ≠ n then
22     RETURN false
23 else
24     for (p, t) ∈ M do
25         x̊(p) ← t
26 RETURN x̊
```

An algorithm in pseudo-code is presented in Algorithm 6.1. All necessary "mod τ" operations are listed in the algorithm.

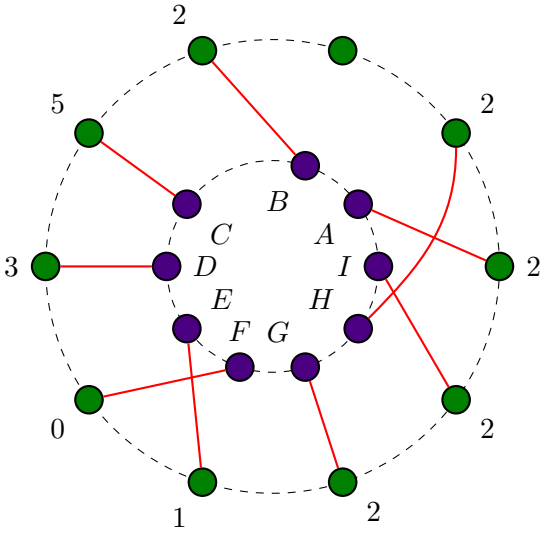
Example: Suppose we are given an arbitrary graph $G = (P \dot\cup C, E)$ as shown in Figure 6.3. Here, $n = |P| = 9$ and $\tau = 10$. Consider $\varphi = 4$. Like always, we focus on node C, which we can see with its children in Figure 6.3(a). The span $\mathring{S}_C = \{b, \ldots, g\}$, so $\mathring{s}_C = 5$. We start at $\max\{f_p - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil, l_p - \varphi\} = \max\{b - 2, g - 4\} = c$. Since $c \in \mathring{S}_C$, the edge (C, c) has weight $\mathring{s}_C$. We continue with positions d, e, and f as shown in Figure 6.3(b). Since $g - b > 4 = \varphi$, this position gets discarded, and we stop.



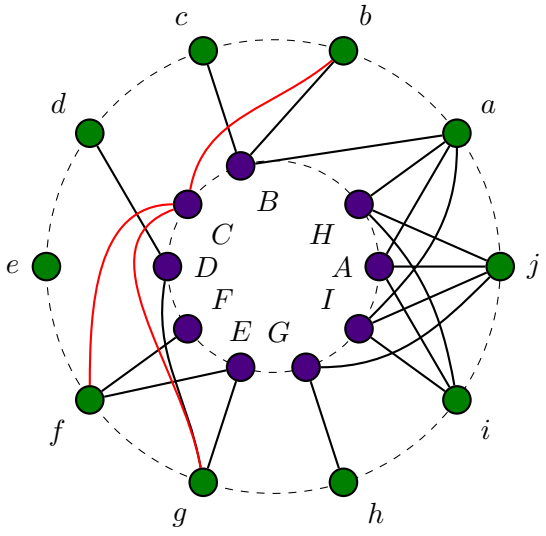
(a) A graph G with an integer W as instance of `WinSumPR`



(b) The matching of C to the four possible positions, the figures indicate the window size, if we place node C on this position.



(c) The minimal weight matching for all parents



(d) The final solution as window minimizing arrangement of the parents

Figure 6.3: An example of the algorithm of `WinSumPR`.

Similarly, we create a matching for all parents, which we solve with a `MinMatch`. The minimal matching is displayed in Figure 6.3(c), while Figure 6.3(d) shows the realization with minimal window sum.

These observations lead to the successive result:

Theorem 6.6. `MinimumWindowSumParentsRadial` *is solvable in* $\mathcal{O}(|P|^{2+o(1)})$.

Proof. Suppose, we are given a parent set $P, |P| =: n$ with span set S in a radial drawing $\mathring{x} : S \to \mathbb{Z}_\tau$, $W \in \mathbb{N}$ and $\varphi < \frac{\tau}{2}$ as an instance of `WinSumPR`. Unlike in the proof of Theorem 4.4, where the algorithm always returned a drawing, we have to show now that the algorithm returns a drawing with minimal window sum if there is one, and otherwise, it returns `false`. Similar to the last algorithms, this returns a window size minimizing drawing instead of the answer to the question if P, S, W, φ, τ is solvable.

We start proving that the algorithm returns a minimal solution, so suppose there is a solution. Let us prove the following statements.

(E1) The algorithm adds all correct positions to the matching graph but, at most, the n where this parent has the smallest window.

(E2) The algorithm detects if there is no placement with this frame size.

(E3) The provided drawing has minimal window sum.

Statement (E1): Let $p \in P$ be a parent. We know from Proposition 6.5 that the interval with the possible position is $I_p = \{l_p - \varphi, \dots, f_p + \varphi\}$. If the span is at least $n - 1$, the algorithm adds the first n positions in the span, but only positions in I_p, to the list. Since the window of p equals the span, if placed within it, these are minimal positions. Otherwise, if the span is smaller, the algorithm starts at $k = f_p - \left\lceil \frac{n-(\mathring{s}_p+1)}{2} \right\rceil$ and adds the conclusive n positions to the graph, but still only if they are inside I_p.

The correctness of this start point follows (K3) from Theorem 4.4 in the parallel drawing. This way, it adds the same number of points left and right to the list.

Statement (E2): Two issues can be the cause for there being no placement. With Proposition 6.5 we know, that if for a parent $p \in P$, $I_p = \{l_p - \varphi, \dots, f_p + \varphi\} = \emptyset$, there is no valid position for p. This situation occurs when $\mathring{s}_p > 2\varphi$, and in this case, the algorithm stops and returns `false`.

If each parent has position candidates, but it is not possible to place all parents in one of them, there is no proper drawing, either. In this case, the matching algorithm could not assign each parent a distinct position, so the matching is not perfect, and the algorithm would also stop.

Statement (E3): With the use of (E1) and (E2), we know that if there is no possible drawing, the algorithm detects it. Otherwise, it finds a perfect matching where the weight of the edges, i.e., the window sizes of the parents, are minimal. Hence the resulting drawing has a minimal window sum.

Running time: The running time is similar to the familiar algorithm of Section 4. The outer loop runs $n \coloneqq |P|$ times. In each iteration, the maximum running time is from k to $k + n - 1$, so there are n iterations, too. Thus this part runs in $\mathcal{O}(n^2)$.

The weighted input graph is also similar. The number of edges in the weighted graph is at most n^2, so we achieve again a running time of $\mathcal{O}(|P|^{2+o(1)})$ with the algorithm by Chen et al. [CKL+22]. □

Variant 1: Once again, we ask the related question, what happens, if we do not minimize the window but the edge sum? Like we altered the algorithm of `WinSumP` (Section 4.2) to an algorithm of `EdgeSumOne` in Section 5.1, we can also modify this version for the radial drawings.

Definition 6.7 (`MinimumEdgeSumOneSideRadial` (`EdgeSumOneR`))**.** *Suppose we are given a bipartite graph $G = (P \cup C, E)$, a partial radial drawing $\mathring{x} : C \to \mathbb{Z}_\tau$, an integer $W \in \mathbb{N}$, and a frame size $\varphi \leq \frac{\tau}{4}$. Is there a completing radial drawing $\mathring{x} : P \to \mathbb{Z}_\tau$, holding the frame size φ with valid statement:*

$$\sum_{e \in E} \mathring{\lambda}(e) \leq W.$$

Putting together the way the costs for the parents get calculated in `EdgeSumOne` and the limitations to the possible positions described in this section, we can quickly see how the algorithm works. As with parallel drawing, we start at the middle child and continue going left and right. The slight difference is that possible positions are limited to $\{l_p - \varphi, \ldots, f_p + \varphi\}$, so we stop comparing the costs as soon as one pointer reaches an end of the interval. In this case, we add positions from the other pointer until this reaches the other end, or we have n positions.

A possible scenario is that the middle child, where we put our start, is too far away from the center of the span and, therefore, not in this interval. For instance, with children at positions $1, 2, 3, 4, 10$ and frame size 6, the algorithm would start at 3. With $\{l_p - \varphi, \ldots, f_p + \varphi\} = \{4, \ldots, 7\}$, we keep incrementing the appropriate pointer until it reaches the interval. Without comparing, we add positions up to the other end. So we get to the following result:

Corollary 6.8. `MinimumEdgeSumOneSideRadial` *is also solvable in $\mathcal{O}(|P|^{2+o(1)} + |P| \cdot S_P)$.*

Variant 2: In both Sections 4.2 and 5.1 for parallel drawing, we transformed the problems from sum-of-the-window-sizes/sum-of-the-edge-lengths minimizing to largest window/longest edge minimizing and got the problems `MaxWinP` (Definition 4.7) and `MaxEdgeOne` 5.7. In the radial drawing, these modifications are also possible.

Definition 6.9 (`MimimumMaxWindowParentsRadial` (`MaxWinPR`))**.** *Suppose we are given a set of parents P, the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ in a partial radial drawing $\mathring{x} : S \to \mathbb{Z}_\tau$, an integer $W \in \mathbb{N}$ and a frame size $\varphi \leq \frac{\tau}{4}$. Is there a completing radial drawing $\mathring{x} : P \to \mathbb{Z}_\tau$ holding the frame size φ such that the following holds?*

$$\forall p \in P : \mathring{w}_p \leq W.$$

Definition 6.10 (`MinimumMaxEdgeOneSideRadial` (`MaxEdgeOneR`))**.** *Suppose we are given a set of parents P, the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ (Table 2.1) in a partial radial drawing $\mathring{x} : S \to \mathbb{Z}_\tau$, an integer $W \in \mathbb{N}$ and a frame size $\varphi \leq \frac{\tau}{4}$. Is there a completing radial drawing $\mathring{x} : P \to \mathbb{Z}_\tau$ holding the frame size φ such that the following holds?*

$$\forall (p, c) \in E : \mathring{\lambda}((p, c)) \leq W.$$

We can apply the same algorithm modification on both the window and the edge variant of this section to solve the longest-edge minimization (Definition 5.7) and largest-window minimization (Definition 4.7).

Corollary 6.11. `MimimumMaxWindowParentsRadial` *is solvable in $\mathcal{O}(|P|^{2.5})$ and* `MinimumMaxEdgeOneSideRadial`*, too.*

6.2 Moving both parents and children with frame size

In the parallel drawing, the results looked the same when we tried minimizing the window or edge sum. When moving the parents, the problem is easily feasible using matching, while it has been proven to be $\mathcal{NP}$-complete as soon as both layers are movable. The last section showed that the polynomial problems stay solvable in a radial drawing. Now we show that if we move parents and children, the problem is $\mathcal{NP}$-complete for radial drawings, too.

Definition 6.12 (`MinimumWindowSumBothRadial` (`WinSumBR`))**.** *Suppose we are given a bipartite graph $G = (P \dot\cup C, E)$, an integer $W \in \mathbb{N}$ and a frame size $\varphi \leq \frac{\tau}{4}$ for a fraction τ. Is there a radial drawing $\mathring{x} : P \dot\cup C \to \mathbb{Z}_\tau$ holding the frame size φ such that the following holds?*

$$\sum_{p \in P} \mathring{w}_p \leq W.$$

In this case, we do not need to adapt the proof of Section 4.3 and reduce `LAP` to it. To prove the $\mathcal{NP}$-completeness for the radial drawing, we can reduce the problem in a parallel drawing to this problem. The difference is made by wrapping around, which makes the edges from the first to the last node shorter. Suppose we are given an instance $G = (P \dot\cup C, |E|), W$ of `WinSumB` (Definition 4.3). Let $\vartheta = |P| + |C|$. We know that we can transform any drawing x of G into a drawing y where the distance between the first and last node is at most ϑ without increasing the size of any window. If the maximal distance is larger, there has to be a gap in the graph, i.e., a position without a parent or a child. When we move both parts of the graph to the left and the right of the gap closer together to close this gap, window sizes do not increase.

So with $\varphi \geq \vartheta$, we guarantee that every sum-of-the-window-size minimizing drawing of G, W as an instance `WinSumB` is a sum-of-the-window-size minimizing drawing of $G, W, \varphi, \tau = 4\varphi$ as an instance of `WinSumBR` and vice versa.

Example: Figure 6.4(a) shows a graph G in an arbitrary drawing x. Together with $W = 10$ is G, W an instance of `WinSumB` (Definition 4.9). With $\vartheta = |P| + |C| = 9$, let $\varphi = 9 \geq \vartheta$ and $\tau = 4\varphi = 36$. In Figure 6.4(b), we see the same graph in a radial drawing $\mathring{x}$ where $\mathring{x}(v) = x(v)$ for all nodes $v \in P \dot\cup C$. The frame of parent D is marked orange. It is easy to see that all gap-free drawings yield the frame size, i.e., all neighbored children are placed in this orange area. The drawing of G in Figure 6.4(c) has a minimal sum of the window sizes, which equals the same in the radial drawing in Figure 6.4(d).

With this information, we can reduce `WinSumB` to `WinSumBR` to show $\mathcal{NP}$-hardness.

Theorem 6.13. `MinimumWindowSumBothRadial` *is $\mathcal{NP}$-complete.*

Proof. Suppose we are given a bipartite graph $G = (P \dot\cup C, E)$ and a number $W \in \mathbb{N}$ as an instance of `WinSumB`. Consider $\vartheta := |P| + |C|$, $\varphi := \vartheta$ and $\tau := 4\varphi$.

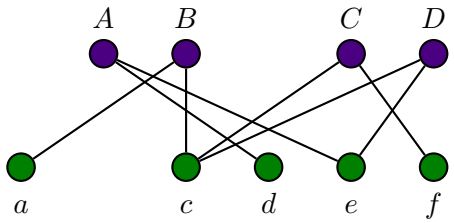
For the reduction, we show the equivalence of the following statements:

(X1) G, W as instance of `WinSumB` is solvable.

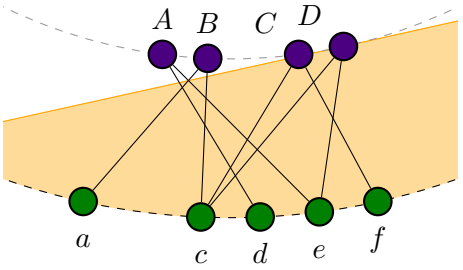
(X2) G, W, φ, τ as instance of `WinSumBR` is solvable.

We achieve this by showing (X1) $\implies$ (X2) and (X2) $\implies$ (X1).

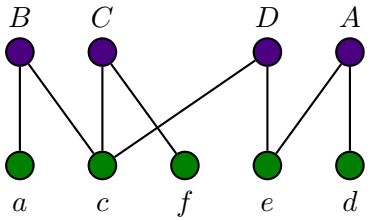
(X1) $\implies$ (X2) There is a parallel drawing x of G with a window sum at most W. We assume w.l.o.g., that x maps to $0, \ldots, \vartheta - 1$. Otherwise, with $\varphi = |P| + |C|$, the drawing is not gap-free and can be pushed into a closer drawing without increasing the window sum.



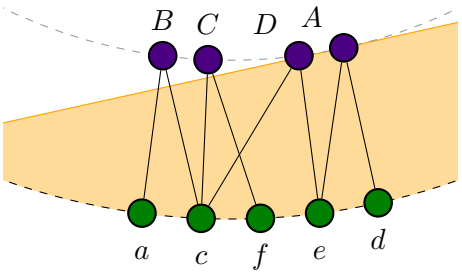
(a) A graph G in an arbitrary parallel drawing x



(b) The graph G in a radial ordering $\mathring{x}$ with large φ with the same positions for the nodes



(c) G in a sum-of-the-window-sizes optimizing parallel drawing



(d) G in a radial drawing with the same positions for the nodes

Consider the radial drawing $\mathring{x} : P \cup C \to \mathbb{Z}_\tau, v \mapsto x(v)$. So for each parent $p \in P$, window sizes w_p in x and $\mathring{w}_p$ in $\mathring{x}$ equal. So the window sizes in the drawings are the same. Hence the window size sum in $\mathring{x}$ is at most W. The maximum distance between two nodes is $\vartheta - 1 < \varphi$, so $\mathring{x}$ holds frame size φ.

(X2) $\implies$ (X1) There is a radial drawing $\mathring{x}$ of G with a window sum of at most W. Consider the parallel drawing $x : P \cup C \to \mathbb{Z}, v \mapsto \mathring{x}(v)$. Again, for each parent $p \in P$, window sizes w_p in x and $\mathring{w}_p$ in $\mathring{x}$ equal. So the window sizes in the drawings are the same, too. Thus the window size sum in x is at most W.

This reduction runs in polynomial time since the only thing to calculate is ϑ.

At last, we show `WinSumBR` $\in \mathcal{NP}$. Given an instance G, W, φ, τ and a drawing $\mathring{x}$. Computing the window sum runs in time linear in the number of edges. Checking if $\mathring{x}$ yields frame placement φ runs simultaneously. □

Variant: We now have $\mathcal{NP}$-completeness for the variant where both sides are allowed to move in three variants: In parallel drawings, we proved it for window minimization (Definition 4.9) and for minimum edge length sum (Definition 5.9). In this section, we showed that in radial drawing, window minimization in this way is complex, too. So the combination left over is if edge length minimization is also $\mathcal{NP}$-complete in radial drawings.

Definition 6.14 (`MinimumEdgeSumBothRadial` (`EdgeSumBothR`))**.** *Suppose we are given a bipartite graph $G = (P \cup C, E)$, an integer $W \in \mathbb{N}$ and a frame size $\varphi \leq \frac{\tau}{4}$ for a fraction τ. Is there a radial drawing $\mathring{x} : P \cup C \to \mathbb{Z}_\tau$ holding the frame size φ such that the following holds?*

$$\sum_{e \in E} \mathring{\lambda}(e) \leq W.$$

Transferring the problem `EdgeSumBoth` to `EdgeSumBothR` works one by one the same. The same techniques can be used for this. We use the same ϑ and create the same large-radial drawing from parallel drawing and vice versa. In conclusion, this leads to our last $\mathcal{NP}$-completeness statement:

Corollary 6.15. `MinimumEdgeSumBothRadial` *is $\mathcal{NP}$-complete.*

6.3 Minimal frame size

This concept of drawing bipartite graphs on concentric circles also gives us new questions to ask. In the last section, we focused on minimizing windows and edges with fixed frame size. However, can we compute a minimal frame size, too? Edge lengths and windows are correlated with frame size. An edge length can be, at most, as large as the frame, and no window can become bigger than twice the frame size. So with a reduced frame, the windows and edges shrink simultaneously.

We no longer care about window sizes and only focus on achieving minimal frame sizes. That helps us in two aspects of visualizing the bipartite graph. In the concentric drawing, the frame size corresponds to the angle of the sectors around the parent, including all adjacent children. This angle is related to the angle the parent forms with their children, so a smaller frame size also leads to a smaller angle.

Additionally, a reasonable restriction on a drawing is straight edges without intersecting the inner circle. So if we have a parent in the inner circle and want to know all possible positions for children with this restriction, we draw a tangent through this point. This tangent subdivides the outer circle into two parts from which the minor marks all positions to which this parent can "see", i.e., can have valid connections (Figure 6.4). The size of this smaller part is proportional to the ratio between the outer and inner circle radii. Suppose we are given a children's circle radius. The upper bound for the parent radius is this radius itself since it has to be smaller. In this case, the parent can see only the opposite point. The lower bound for the radius is zero. When the inner circle's radius tends toward zero, the cut-off part approaches half a circle. So the most a parent can see is the semicircle without the first and last node.

Once again, it is not necessary to know information about all children. It is sufficient to know the marginal children's location. If the start and the end of a span are inside the frame of their parent, then the other children between them are, too.

All this motivates the last problem in this thesis:

Definition 6.16 (`MinimumFrameSizeParents` (`MinFP`))**.** *Suppose we are given a set of parents P, the set of their marginal children $S = \{(f_p, l_p) \mid p \in P\}$ in a partial drawing $x : S \to \mathbb{Z}$, and an integer $W \in \mathbb{N}$. What is the smallest frame size φ, such that there is a completing drawing $\mathring{x} : P \to \mathbb{Z}_\tau$ holding φ?*

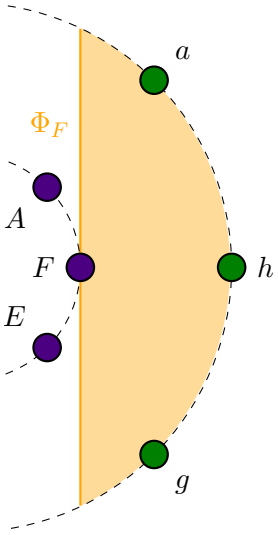


Figure 6.4: An radial drawing of a graph with tangent Φ_F of F. At this position, F can see children a, h, and g.

Suppose we are given G and the partial drawing $\mathring{x}$ for the children. We already have an algorithm that computes the minimal window sum for G and a fixed frame size in polynomial time. So we can use this algorithm and guess a frame size. If it is possible to draw G with this frame, we continue with a smaller frame size; otherwise, we increment it. Eventually, this will give us the smallest possible frame size. However, this takes for each step $\mathcal{O}(|P|^{2+o(1)})$ time, so let us find a better algorithm.

When we think about the relation of the frame size to the other problems, we see that the smallest possible frame size equals the minimal longest edge. In this thesis, we calculate projected edge length, so a radial drawing yields a frame size φ if and only if all edges are at most of this length, i.e., the longest edge has lengths of at most φ. In Variant 2 of Section 6.1, we showed that there is an algorithm that returns a drawing with a minimal longest edge. So we create this algorithm and modify it to return the minimum frame size, too.

Per definition is $0 < \varphi \leq \frac{\tau}{4}$. The span size sets an additional lower bound. If we place a parent in the center of its span, the maximum distance to its children is minimal, so the lower bound for φ is $\frac{\mathring{s}^*}{2}$, where $\mathring{s}^*$ is the largest span of all parents.

So we edit our algorithm so that the weight is equal to the minimal frame size needed to put a parent at this position instead of the window size. In this case, a placement with any frame size is not guaranteed. A node can have at most $2\varphi + 1$ neighbors, and since the frame size can be at most a quarter of the fraction rate, a node can not be adjacent to all other nodes. If there is a child adjacent to more than $\frac{\tau}{2} + 1$ parents or a parent has a span larger than $\frac{\tau}{2}$, it is not possible to find a frame size. However, when one wants to draw a bipartite graph with one of these attributes, the radial drawing may not be the best visualization method to choose.

Algorithm: Suppose we are given a bipartite graph $G = (P \cup C, E)$ and a partial radial drawing $\mathring{x} : C \to \mathbb{Z}_\tau$ as an instance of `MinFP`. Set $n \coloneqq \max\{|P|, \frac{\tau}{2} + 1\}$ to the number of positions to match. For each parent $p \in P$, repeat the following steps:

If the span is larger than $\frac{\tau}{2}$, return `false` and stop the algorithm. If the span is smaller than $n - 1$, start k at $f_p - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil \bmod \tau$, otherwise start at f_p. For this and the following

Algorithm 6.2: MINIMIZING FRAME SIZE

Input: A bipartite graph $\mathsf{G} = (\mathsf{P} \cup \mathsf{C}, \mathsf{E})$ with a partial radial drawing $\mathring{\mathsf{x}} : \mathsf{C} \to \mathbb{Z}_\tau$
Data: Weighted bipartite graph $\mathsf{H} = (\mathsf{P} \cup \mathsf{T}, \mathsf{E}', \xi)$
Output: Minimum frame size and radial drawing with this frame size, if it is possible to place the parents; otherwise, false.

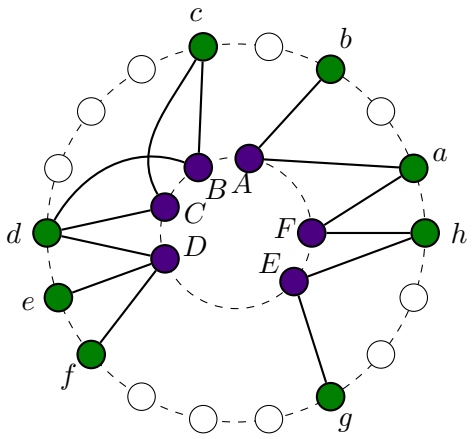
```
// Initialization
E' ← ∅
n ← max{|P|, τ − 1}
// Adding the weighted edges
for p ∈ P do
    if s_p > τ/2 then
        RETURN false
    else if s_p < n − 1 then
        k ← f_p − ⌈(n−(s_p+1))/2⌉ mod τ
    else
        k ← f_p
    for t ← k to k + n − 1 do
        e ← (p, t mod τ)
        E'.ADD(e)
        ξ(e) ← max{(l_p − t) mod τ, (t − f_p) mod τ}
// Calculating the minimal weight bottleneck matching and creating
   the placing
H ← (P ⊎ Z_τ, E', ξ)
M ← MINBOTTMATCH {H}
if |M| < n then
    RETURN false
φ for (p, t) ∈ M do
    x̊(p) ← t
    φ ← max{φ, ξ((p, t))}
RETURN x̊, φ
```

$n-1$ positions, add an edge to the weighted bipartite graph from p to this position. The weight of this edge is set equal to the distance to l_p or f_p, whichever is further away.

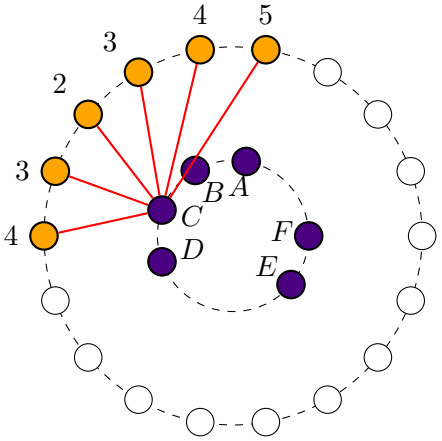
With this set of weighted edges, create a weighted bipartite graph as an instance of `BottMatch` (See Definition 2.12). This matching returns a drawing with the longest edge as short as possible. If the matching is not perfect and not every parent is matched, return `false`. Return a drawing where all parents are placed at the matched positions. The minimum frame size equals the heaviest edge. The pseudo-code algorithm is described in Algorithm 6.2.

Example: Suppose we are given a graph $G = (P \cup C, E)$ and a radial partial drawing $\mathring{x} : C \to \mathbb{Z}_\tau$ as an instance of `MinFP` like in Figure 6.5(a). For each parent, we repeat the following, exemplarily shown by parent C:

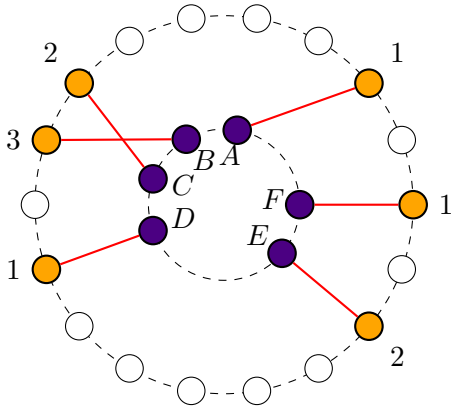
Since $|P| = 6$ and $6 \leq \frac{\tau}{2} + 1$, we have to match 6 positions for each parent. We start at $f_p - \left\lceil \frac{n-(s_p+1)}{2} \right\rceil \bmod \tau = c - \left\lceil \frac{6-(4+1)}{2} \right\rceil \bmod \tau = c - 1$ as shown in Figure 6.5(b). The distance to f_p is 1, and the distance to l_p is 5, so this position has weight 5. We continue



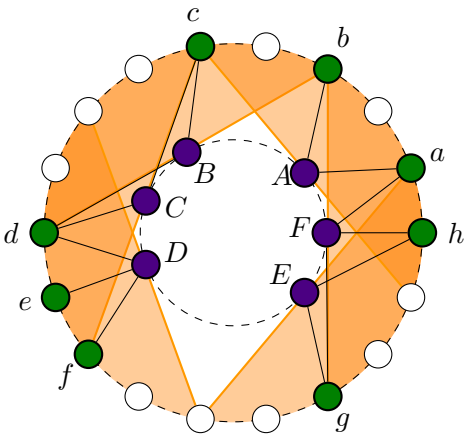
(a) A graph G in an arbitrary drawing as instance of `MinFP`



(b) For visualization, we show only the matched positions for C, but this happens for the other nodes, too.



(c) The bottleneck matching with the shortest longest edge, i.e., smallest frame size



(d) A drawing of G with minimal frame size

Figure 6.5: The algorithm to determine the minimum frame size of a graph G

by adding the following five positions. The first three have l_p as their furthest children, so the weights are $4, 3, 2$. Then the distance to f_p is higher, so we have frame sizes 3 and 4.

After adding all positions to the weighted graph, we compute a `MinMatch` (Definition 2.11) as shown in Figure 6.5(c). This matching gives us the minimum frame size achievable. Though B and C have the same positions as optimum, one has to be placed one-off. So the minimum frame size that a drawing can hold is 3.

Figure 6.5(d) shows the radial drawing with this frame size. As mentioned at the beginning of this section, the minimum frame size can be used to determine the biggest ratio of the inner to the outer radius. The maximum inner radius to draw all edges straight without intersecting the inner circle is shown in the figure, too.

Theorem 6.17. `MinimumFrameSizeParents` *is solvable in* $\mathcal{O}(|P|^{2.5})$ *time.*

Proof. Suppose we are given a graph $G = (P \dot\cup C, E)$ and a partial radial drawing $\mathring{x} : C \to \mathbb{Z}_\tau$ as an instance of `MinFP`. Let φ be the frame size and $\mathring{x}$ the drawing the algorithm returns. We have to prove the following:

(Z1) The algorithm detects if there is no placement with any frame size.

(Z2) The weight of an added edge (t, p) equals the frame size, necessary to place a p at t.

(Z3) There is a radial drawing of G holding frame size φ and no drawing holding a smaller frame size.

Statement (Z1): There are two possible situations where a graph has no placement with any frame size: There is a parent with a span size of more than $\frac{\tau}{2}$, but the algorithm would have stopped in this case. If there is no possible matching, e.g., when a child has a degree larger than $\frac{\tau}{2} + 1$, it is impossible to find a perfect matching, and the algorithm stops, too.

Statement (Z2): Whenever adding an edge to the graph, the weight equals the maximum distance to f_p or l_p. These are the only candidates to be the furthest children since all the other children are between them.

Statement (Z3): The matching in the algorithm has no edge with a weight larger than φ, so the drawing $\mathring{x}$ has maximum edge length φ. So if we place all parents in their matched positions, the drawing yields the frame size φ. Additionally, there is at least one parent with weight φ, so it is not possible to place all parents with frame size $\varphi - 1$. Hence the algorithm calculates the minimum frame size.

Running time: Creating the weighted graph needs at most $n \coloneqq |P|$ position candidates for every parent, so this part runs in $\mathcal{O}(n^2)$ time. The `BottMatch` as described in Definition 2.12 needs $\mathcal{O}(n \cdot \sqrt{mn})$ time with $m \coloneqq |E'|$ [PL88]. Since we added for each parent at most n positions, the number of edges equals n^2. So $\mathcal{O}(n \cdot \sqrt{mn}) = \mathcal{O}(n^{2.5})$. □

7. Conclusion

7.1 Summary

In this thesis, we proved different problems to be $\mathcal{NP}$-complete. Meanwhile, we found algorithms for several problems solvable in polynomial time. A quick overview of all results with hardness or running time grouped by chapter is given in Table 7.1. For the problems in $\mathcal{P}$, a Python Code is provided. [1]

The first problem was `SpanSum`, of which we saw that it is very close to the `LAP`. This problem is $\mathcal{NP}$-complete whether we look for a sum-of-the-span or the largest span minimizing. Furthermore, we went to minimizing windows. If we want to have each window strictly at its minimum, the span, then we have to iterate over the span, and with greedy choice, we can solve this decision question. It gets more complicated as soon as we allow parents to be placed outside their span. Though, with the window size of a parent only depending on its fixed children and position, it is possible to calculate the cost for each position upfront. This property allows the matching of each parent to the locations with different weights. Thus we can use a known algorithm and create the proper input. Knowing that we only need a limited number of positions to match and not all possible ones, we reduce the running time by presorting them. This algorithm can even be used to find the best solution with parents limited to specific locations to look prettier. It is also possible to compute the minimum widest window with a similar algorithm. Whereas if we allow both parents and children to move, computing the minimum window sum is $\mathcal{NP}$-complete.

The next chapter looked at a slightly different way to improve readability. With short edges, a graph looks less overloaded, and additionally, we also save ink to print the graph, so why not create the minimum possible sum of the edge lengths? By moving both sides, we realized that it is $\mathcal{NP}$-complete to compute the minimum edge sum, similar to the window sum. However, if we fix one side, we can use the algorithm we already discovered. We need a more complex presorting mechanism, but the running time does not get much worse. We realized that this could also be used for minimizing the longest edge. So there are multiple ways to pretty up bipartite graphs drawn on parallel lines.

Though this may be the most common way of visualizing bipartite graphs, there are other ways to draw bipartite graphs. The only restriction we should make is that the two disjoint sets do not intersect each other in the drawing. Thus we experimented with a different kind of representing bipartite graphs: By drawing the sets on two concentric circles. This

[1] `https://github.com/DaKassian/window-minimization`

Chapter	Problem	Result	Ref
Span Minimization	SpanSum	$\mathcal{NP}$-complete	3.3
	MaxSpan	$\mathcal{NP}$-complete	3.5
Window Minimizing	PinS	$\mathcal{O}(\lvert P\rvert \log \lvert P\rvert)$	4.2
	WinSumP	$\mathcal{O}(\lvert P\rvert^{2+o(1)})$	4.4
	WinSumLoc	$\mathcal{O}(\lvert L\rvert^{1+o(1)})$	4.6
	MaxWinP	$\mathcal{O}(\lvert P\rvert^{2.5})$	4.8
	WinSumB	$\mathcal{NP}$-complete	4.13
Edge Minimization	EdgeSumOne	$\mathcal{O}(\lvert P\rvert^{2+o(1)} + \lvert P\rvert \cdot S_P)$	5.4
	EdgeSumOneLoc	$\mathcal{O}(\mathcal{O}(\lvert L\rvert^{1+o(1)} + d_{\max}\lvert L\rvert))$	5.6
	MaxEdgeOne	$\mathcal{O}(\lvert P\rvert^{2.5})$	5.8
	EdgeSumBoth	$\mathcal{NP}$-complete	5.11
Radial Drawing	WinSumPR	$\mathcal{O}(\lvert P\rvert^{2+o(1)})$	6.6
	EdgeSumOneR	$\mathcal{O}(\lvert P\rvert^{2+o(1)} + \lvert P\rvert \cdot S_P)$	6.8
	MaxWinPR	$\mathcal{O}(\lvert P\rvert^{2.5})$	6.11
	MaxEdgeOneR	$\mathcal{O}(\lvert P\rvert^{2.5})$	6.11
	WinSumBR	$\mathcal{NP}$-complete	6.13
	EdgeSumBothR	$\mathcal{NP}$-complete	6.15
	MinFP	$\mathcal{O}(\lvert P\rvert^{2.5})$	6.17

Table 7.1: Summary of all solved problems grouped by chapter.

kind of visualization opens up the door to new problems. On the one hand, we made it possible to connect the first and the last node without crossing the other nodes; on the other hand, it is now more complicated to draw the edges. An edge from a parent to a child on the other side of the graph can not be drawn by intersecting the inner circle or bending the edge.

Firstly we asked the questions about the window sum again, this time in a radial drawing: If we limit the number of children a parent can see, can we use algorithms based on our technique to get the best solution? We discovered that with slight modifications, the algorithms work for radial drawing, too. The most significant difference is that it is not guaranteed anymore that there is a solution. Even our other algorithms for minimizing the maximum window and the longest edge and edge length sum can be transformed for radial drawings. Unfortunately, minimizing the window size sum or the edge length sum by moving both sides does not get more manageable when we draw bipartite graphs on concentric circles.

However, we discovered an entirely new question, too. If we want to draw the inner circle as large as possible, can we do that in polynomial time? We demonstrated that this problem is closely related to finding the longest edge in a radial drawing.

7.2 Open problems

In the thesis, we discussed and mentioned multiple criteria for improved readability of bipartite graphs. When we look back at the question we asked in the introduction in Section 1.2, we can now answer many distinct problems resolving from different answers to the questions. In Figure 7.1, we present the combinations and the answers we know. We have 20 combinations to answer these questions. In this thesis, we newly answered the problems of eleven combinations. Additionally, we have two variants where we already knew the answer from Bekos et al. [BFK$^+$23]. That leaves seven unanswered combinations. Let us look at the open problems now.

The first problem, *Drawing parallel* × *Calculating Window* × *Minimizing Sum* × *Children*, is assumed to be $\mathcal{NP}$-hard, similar to the variant with maximal window, as mentioned in the introduction.

None of the four combinations *Max* with *Both Sides* are proven now, but their pendants with minimizing the sum are proven to be $\mathcal{NP}$-complete. With `LAP` used in the proofs, the assumption may be close that `BandwidthProblem` can be reduced on these problems.

The problem of minimizing the maximum window by moving children is proven to be $\mathcal{NP}$-hard by Bekos et al. [BFK+23]. It may be easy to transform this problem with radial drawing.

These are already the open combinations with these questions. Additionally, there are more possible combinations with minimizing the span and the frame size. The problem `PinS` can also be transformed in radial drawing problems.

However, there are more ways to vary the problems. Take the open problem *Window Minimization* × *Moving Children*. It seems to be $\mathcal{NP}$-hard. Is it still hard if we restrict to permuting the children's positions instead of moving them freely? Or if we allow only one-degree children? The last one seems solvable, but when does it break? Does it break as soon as we add second-degree children? These are some of the open problems worth pursuing.

Let us look at *Window Minimization* by *Moving Children*. We could also allow placing multiple parents in the same locations and try to minimize the windows by moving children. This problem seems easy to reduce to the `LAP` if, for every edge, we create a parent with the two adjacent children as we did in the problem `SpanSum` (Definition 3.3). The last problem `MinFP` in Section 6.3 can also be transformed into problems where children move with fixed parents or both partitions move.

When looking again at the *ASCT+B Reporter*[2], there is a desire to generalize this problem to multiple layers. It can be wished to improve the windows from the lowest layer to the first one, to minimize the window in this graph, Can this be achieved by ordering clustering the even and the odd layers? In this case, we have a bipartite graph, too, but the first layer has the same significance as the third. This graph is not a tree, but would that improve things if it were? We cannot just swap the children in the lowest two layers and continue to compare two layers up to the top. With that approach, we would need the parents in the second last set to block the positions of their children, too. Thus instead of having positions placed on integer coordinates, let us assign each node a width and sort the intervals with different lengths. Then we can arrange them on a real-valued line, too, and do not have to restrict them to integer values anymore.

When looking at window sizes, we looked at one side, the parents, and measured the distance to their children. What if we want to minimize window sizes for both partitions such that the sum of the window sizes of the parents plus the sum of the window sizes of the children is minimal?

When we add these questions to our graph in Figure 7.1 additional layer makes the tree grow, and more combinations will be possible. Thus the more questions are answered in this field, the more new questions appear. So we leave these problems to further research.

[2] `https://hubmapconsortium.github.io/ccf-asct-reporter/`

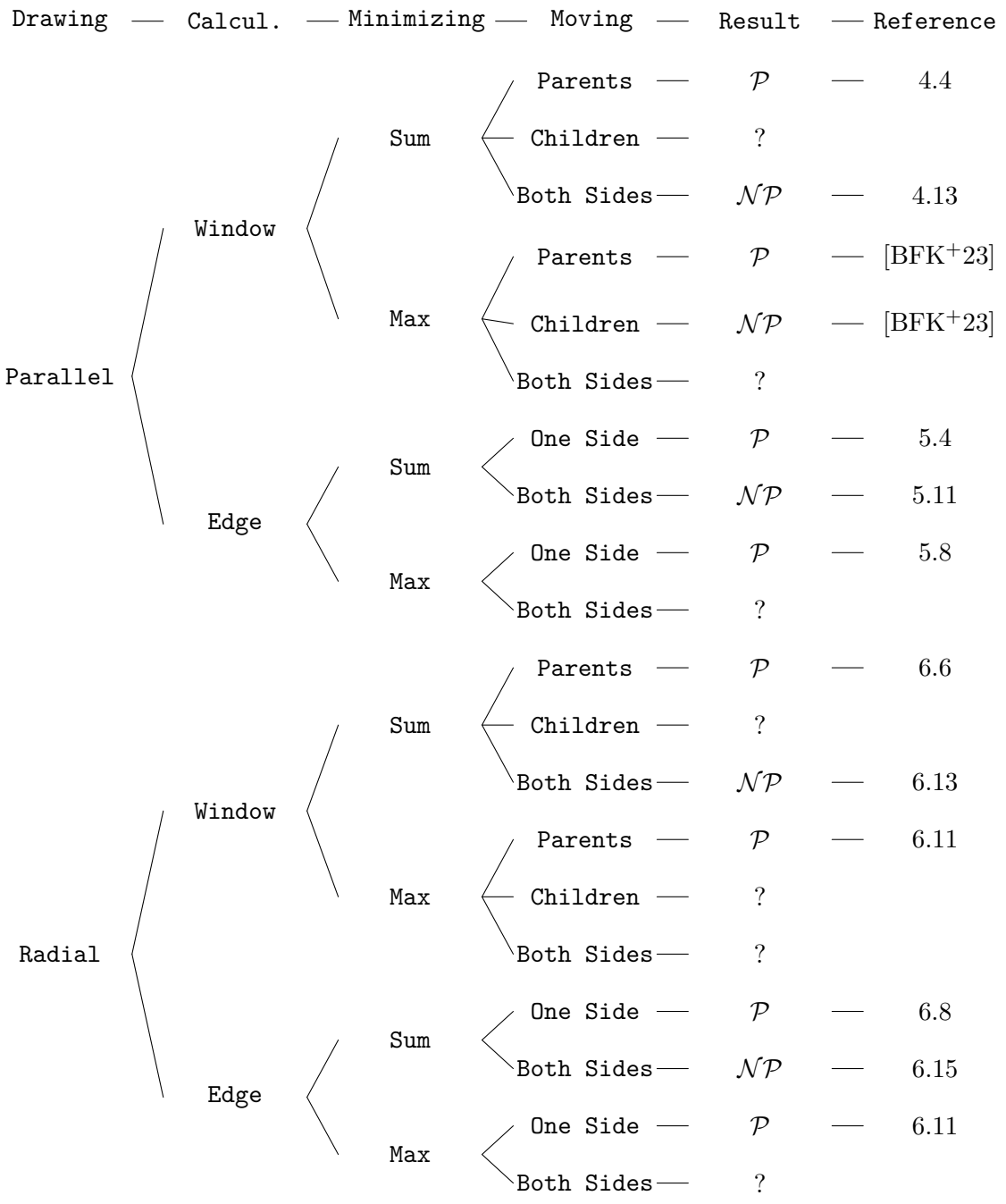


Figure 7.1: The hardness or membership in $\mathcal{P}$ depending on the problem specifications. "?" stands for unknown results.

Bibliography

[ADH98] Armen S Asratian, Tristan MJ Denley, and Roland Häggkvist. *Bipartite graphs and their applications*, volume 131. Cambridge university press, 1998.

[BFK+23] Michael A Bekos, Henry Förster, Michael Kaufmann, Stephen Kobourov, Myroslav Kryven, Axel Kuckuk, and Lena Schlipf. On the 2-layer window width minimization problem. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 209–221. Springer, 2023.

[ÇEKS09] Olca A Çakıroğlu, Cesim Erten, Ömer Karataş, and Melih Sözdinler. Crossing minimization in weighted bipartite graphs. *Journal of Discrete Algorithms*, 7(4):439–452, 2009.

[CKL+22] Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *arXiv preprint arXiv:2203.00671*, 2022.

[DBETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[DGGL07] Emilio Di Giacomo, Luca Grilli, and Giuseppe Liotta. Drawing bipartite graphs on two curves. In *Graph Drawing: 14th International Symposium, GD 2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers 14*, pages 380–385. Springer, 2007.

[DGGL08] Emilio Di Giacomo, Luca Grilli, and Giuseppe Liotta. Drawing bipartite graphs on two parallel convex curves. *Journal of Graph Algorithms and Applications*, 12(1):97–112, 2008.

[DLR09] Geoffrey M Draper, Yarden Livnat, and Richard F Riesenfeld. A survey of radial methods for information visualization. *IEEE transactions on visualization and computer graphics*, 15(5):759–776, 2009.

[DMRW12] Maxime Dumas, Michael J McGuffin, Jean-Marc Robert, and Marie-Claire Willig. Optimizing a radial layout of bipartite graphs for a tool visualizing security alerts. In *Graph Drawing: 19th International Symposium, GD 2011, Eindhoven, The Netherlands, September 21-23, 2011, Revised Selected Papers 19*, pages 203–214. Springer, 2012.

[DP14] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM (JACM)*, 61(1):1–23, 2014.

[DRM12] Maxime Dumas, Jean-Marc Robert, and Michael J McGuffin. Alertwheel: radial bipartite graph visualization applied to intrusion detection system alerts. *Ieee Network*, 26(6):12–18, 2012.

[EK72] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[EW94] Peter Eades and Nicholas C Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica*, 11(4):379–403, 1994.

[FAL+06] Stefano Foresti, James Agutter, Yarden Livnat, Shaun Moon, and Robert Erbacher. Visual correlation of network alerts. *IEEE Computer Graphics and Applications*, 26(2):48–59, 2006.

[GJ83] Michael R Garey and David S Johnson. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 4(3):312–316, 1983.

[GJS76] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some Simplified $\mathcal{NP}$-Complete Graph Problems. *Theoretical Computer Science*, 1:237–267, 1976.

[Hal98] Marshall Hall. *Combinatorial theory*, volume 71. John Wiley & Sons, 1998.

[HEH14] Weidong Huang, Peter Eades, and Seok-Hee Hong. Larger crossing angles make graphs easier to read. *Journal of Visual Languages & Computing*, 25(4):452–465, 2014.

[HGKW16] Xiangnan He, Ming Gao, Min-Yen Kan, and Dingxian Wang. Birank: Towards ranking on bipartite graphs. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):57–71, 2016.

[HM97] James Haralambides and Fillia Makedon. Approximation algorithms for the bandwidth minimization problem for a large class of trees. *Theory of Computing Systems*, 30(1):67–90, 1997.

[KLST99] Ming-Yang Kao, Tak-Wah Lam, Wing-Kin Sung, and Hing-Fung Ting. A decomposition theorem for maximumweight bipartite matchings with applications to evolutionary trees. In *European Symposium on Algorithms*, pages 438–449. Springer, 1999.

[Kuh55] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[Pap76] Ch H Papadimitriou. The np-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976.

[PFH+18] Nicola Pezzotti, Jean-Daniel Fekete, Thomas Höllt, Boudewijn PF Lelieveldt, Elmar Eisemann, and Anna Vilanova. Multiscale visualization and exploration of large bipartite graphs. In *Computer Graphics Forum*, volume 37, pages 549–560. Wiley Online Library, 2018.

[PKP+18] Georgios A Pavlopoulos, Panagiota I Kontou, Athanasia Pavlopoulou, Costas Bouyioukos, Evripides Markou, and Pantelis G Bagos. Bipartite graphs in systems biology and medicine: a survey of methods and applications. *GigaScience*, 7(4):giy014, 2018.

[PL88] Paul A Peterson and Michael C Loui. The general maximum matching algorithm of micali and vazirani. *Algorithmica*, 3(1-4):511–533, 1988.

[PN94] Abraham P Punnen and KPK Nair. Improved complexity bound for the maximum cardinality bottleneck bipartite matching problem. *Discrete Applied Mathematics*, 55(1):91–93, 1994.

[Pur02] Helen C Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing*, 13(5):501–516, 2002.

[San09] Piotr Sankowski. Maximum weight bipartite matching in matrix multiplication time. *Theoretical Computer Science*, 410(44):4480–4488, 2009.

[Shi79] Yossi Shiloach. A minimum linear arrangement algorithm for undirected trees. *SIAM Journal on Computing*, 8(1):15–32, 1979.

[SSW05] Justus Schwartz, Angelika Steger, and Andreas Weißl. Fast algorithms for weighted bipartite matching. In *International Workshop on Experimental and Efficient Algorithms*, pages 476–487. Springer, 2005.

[Tas12] Tamir Tassa. Finding all maximally-matchable edges in a bipartite graph. *Theoretical Computer Science*, 423:50–58, 2012.

[Uno97] Takeaki Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In *International Symposium on Algorithms and Computation*, pages 92–101. Springer, 1997.

[WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.