REPORTS IN INFORMATICS

ISSN 0333-3590

Bandwidth of bipartite permutation graphs in polynomial time

Pinar Heggernes Dieter Kratsch
Daniel Meister

REPORT NO 356 July 2007

Department of Informatics
UNIVERSITY OF BERGEN
Bergen, Norway

This report has URL
http://www.ii.uib.no/publikasjoner/texrap/pdf/2007-356.pdf

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at http://www.ii.uib.no/publikasjoner/texrap/.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

Bandwidth of bipartite permutation graphs in polynomial time*

Pinar Heggernes† Dieter Kratsch‡ Daniel Meister†

Abstract

We give the first polynomial-time algorithm that computes the bandwidth of bipartite permutation graphs. Bandwidth is an NP-complete graph layout problem that is notorious for its difficulty even on small graph classes. For example, it remains NP-complete on caterpillars of hair length at most three, a special subclass of trees. Much attention has been given to designing approximation algorithms for computing the bandwidth, as it is NP-hard to approximate the bandwidth of general graphs with a constant factor guarantee. The problem is considered important even for approximation on restricted classes, with several distinguished results in this direction. Prior to our work, polynomial-time algorithms for exact computation of bandwidth were known only for caterpillars of hair length at most 2, chain graphs, cographs, and most interestingly, interval graphs.

1 Introduction

The bandwidth problem asks, given a graph G and an integer k, whether there exists a linear layout of the vertices of G such that no edge of G has its endpoints mapped to positions with difference more than k. The problem is motivated from Sparse Matrix Computations, where given an $n \times n$ matrix A and an integer k, the goal is to decide whether there is a permutation matrix P such that PAP^T is a matrix with all nonzero entries on the main diagonal or on the k diagonals on either side of the main diagonal. This problem is of great importance in many engineering applications since matrices arising in practice are often sparse. Standard matrix operations like inversion and multiplication as well as Gaussian elimination can be sped up considerably if the input matrix A is transformed into a matrix PAP^T of small bandwidth [14]. The graph and the matrix version of the bandwidth problem are equivalent, and both have been studied extensively in the last 40 years. Bandwidth is also one of the most studied graph layout problems and intuitively it seems to be harder to attack than various other graph layout problems, like vertex separation, pathwidth, and profile. (For a survey on graph layout problems we refer to [8]. See [18, 2] for recent generalisations of bandwidth.)

The bandwidth problem is NP-complete [23], and it remains NP-complete even on very restricted subclasses of trees, like caterpillars of hair length at most 3 [22]. Bandwidth is a benchmark problem known for its difficulty among the often studied NP-hard graph problems.

*This work is supported by the Research Council of Norway through grant 166429/V30.

†Department of Informatics, University of Bergen, N-5020 Bergen, Norway. {pinar, danielm}@ii.uib.no

‡Laboratoire d'Informatique Théorique et Appliquée, Université Paul Verlaine - Metz, 57045 Metz Cedex 01, France. kratsch@univ-metz.fr

With respect to parameterized complexity (see e.g. [9] for an introduction), the bandwidth problem (with parameter k) is $W[k]$-hard [5]. Thus, not only is it unlikely that an $\mathcal{O}(f(k) \cdot p(n))$ algorithm exists for its solution with an arbitrary function f and a polynomial p, but it is also much harder than most other well-studied graph problems with respect to parameterized complexity. Considering exact exponential-time algorithms the situation is similar: for various graph layout problems an $\mathcal{O}(2^n)$ dynamic programming algorithm can be obtained (see e.g. [6]); however this approach does not seem to work for the bandwidth problem. The best known exact algorithm for computing the bandwidth of general graphs has running time $\mathcal{O}(11^n)$, and it uses very clever techniques [11].

Due to the difficulty of the bandwidth problem, approximation algorithms for it attracted much attention. For any constant c, it is NP-hard to compute a c-approximation of the bandwidth of general graphs [26]. In fact, even the bandwidth of trees is hard to approximate within some constant factor [3]. Furthermore, Unger [26] claimed (without proof) that the bandwidth of caterpillars is hard to approximate within some constant factor. Consequently approximation algorithms on restricted graph classes have been received with great interest by the research community, and approximation algorithms have been given for the bandwidth of trees and even caterpillars [12, 16, 17]. Constant factor approximation algorithms for the bandwidth of AT-free graphs and subclasses of them, including permutation graphs, exist [21]. There are approximation algorithms for the bandwidth of general graphs using advanced techniques [4, 10].

Given the hardness of the bandwidth problem under various algorithmic approaches and the interest in small graph classes even with respect to approximation, it is not surprising that polynomial-time algorithms to compute the bandwidth exactly are known for only a few and very restricted graph classes. In most cases such algorithms are established using the structural properties of the graph class through standard techniques. Polynomial-time algorithms are known for caterpillars with hair length at most two [1], chain graphs [20], and cographs [27]. The outstanding result is the polynomial-time algorithm of Kleitman and Vohra that computes the bandwidth of interval graphs [19]. The knowledge on the algorithmic complexity of bandwidth on particular graph classes did not advance much during the last decade. The only, though small, progress was made in [21] were the NP-completeness of bandwidth of cocomparability graphs was observed, and a simple 2-approximation algorithm for the bandwidth of permutation graphs was established. Permutation graphs are precisely those graphs for which the graph and its complement are cocomparability, and thus a subclass of cocomparability graphs. However, the algorithmic complexity of the bandwidth of permutation graphs remained open for a long time, and is still open. Despite various attempts, not even the computational complexity of bandwidth of bipartite permutation graphs was resolved prior to our work.

In this paper, we give the first polynomial time algorithm to compute the bandwidth of bipartite permutation graphs. Our algorithm is based on structural properties of bipartite permutation graphs, in particular the use of strong orderings. Moreover we heavily rely on a deep theorem concerning linear extensions and bandwidth of posets, that for cocomparability graphs guarantees the existence of an

optimal bandwidth layout which is a cocomparability ordering [13]. Finally, a novel local exchange algorithm to find so called normalised (partial) k-layouts is the key algorithmic idea of our work. No tools from previous bandwidth algorithms for special graph classes have been used; rather, our algorithm is especially tailored for bipartite permutation graphs through non-standard techniques.

In the next section we give the necessary definitions, notation, and background on bandwidth of bipartite permutation graphs. In Section 3 we explain the main idea behind our algorithm, and identify the challenging tasks. Section 4 is devoted to the solution of these tasks, after which we present the full algorithm with proof of correctness and polynomial running time in Section 5. Finally Section 6 concludes the paper.

2 Preliminaries

A graph is denoted by $G = (V, E)$, where V is the set of vertices with $n = |V|$ and E is the set of edges with $m = |E|$. The set of neighbours of a vertex v is denoted by $N(v)$, and $N[v] = N(v) \cup \{v\}$. Similarly, for $S \subseteq V$, $N[S] = \bigcup_{v \in S} N[v]$, and $N(S) = N[S] \setminus S$. The subgraph of G induced by the vertices in S is denoted by $G[S]$. For $G' = G[S]$ and $v \in V \setminus S$, $G'+v$ denotes $G[S \cup \{v\}]$, and for any $v \in V$, $G-v$ denotes $G[V \setminus \{v\}]$.

For a given graph $G = (V, E)$ with $V = \{v_1, v_2, ..., v_n\}$, a *layout* $\beta : \{1, \ldots, n\} \to V$ of G is an ordering $(v_{\pi(1)}, \ldots, v_{\pi(n)})$ where π is a permutation of $\{1, \ldots, n\}$. The *distance* between two vertices u, v in a layout β is $d_\beta(u, v) = |\beta^{-1}(u) - \beta^{-1}(v)|$. For a given layout (or ordering) β, we write $u \prec_\beta v$ when $\beta^{-1}(u) < \beta^{-1}(v)$. For a vertex v in G, every vertex u with $u \prec_\beta v$ is *to the left* of v, and every vertex w with $v \prec_\beta w$ is *to the right* of v in β. We will also informally write *leftmost* and *rightmost* vertices accordingly.

For an integer $k \geq 0$, we call β a *k-layout* for G if, for every edge uv of G, $d_\beta(u, v) \leq k$. The *bandwidth* of G, $bw(G)$, is the smallest k such that G has a k-layout. In this paper a layout will be called *optimal* if it is a $bw(G)$-layout for G.

Bipartite permutation graphs are permutation graphs that are bipartite. For the definition and properties of permutation graphs, we refer to [7]. Let $G = (A, B, E)$ be a bipartite graph. Sets A and B are called *colour classes*. A *strong ordering* for G is a pair of orderings (σ_A, σ_B) on respectively A and B such that for every pair of edges ab and $a'b'$ in E with $a, a' \in A$ and $b, b' \in B$, $a \prec_{\sigma_A} a'$ and $b' \prec_{\sigma_B} b$ imply that ab' and $a'b$ are in E. The following characterization of bipartite permutation graphs is the only property that we will need in this paper.

Theorem 1 ([24]). *A bipartite graph is a bipartite permutation graph if and only if it has a strong ordering.*

Spinrad et al. give a linear time recognition algorithm for bipartite permutation graphs that produces a strong ordering if the input graph is bipartite permutation [24]. It follows from the definition of a strong ordering that if $G = (A, B, E)$ is a connected bipartite permutation graph then any strong ordering (σ_A, σ_B) satisfies the following. For every vertex a of A, the neighbours of a appear consecutively in

σ_B. Furthermore, if $N(a) \subseteq N(a')$ for two vertices $a, a' \in A$ then a is adjacent to the leftmost or the rightmost neighbour of a' with respect to σ_B.

An ordering σ of a graph G is called a *cocomparability ordering* if for all u, v, w with $u \prec_\sigma v \prec_\sigma w$, $uw \in E$ implies that $uv \in E$ or $vw \in E$. A graph that has a cocomparability ordering is called a *cocomparability graph*. (Bipartite) permutation graphs are cocomparability graphs [7].

Let V be a set, and let $\prec_P$ be a binary reflexive, antisymmetric and transitive relation over V. Then $P = (V, \prec_P)$ is called a *partially ordered set*. A *linear extension* β of $P = (V, \prec_P)$ is a layout of V satisfying $a \prec_P b \Rightarrow a \prec_\beta b$. Hence for all pairs of elements of V, a linear extension preserves their order relation of P. For an integer $k \geq 0$, a *k-linear labelling* for $P = (V, \prec_P)$ is a linear extension β of P such that for every pair a, b of elements of V, $a \neq b$: $a \prec_P b \Rightarrow d_\beta(a, b) \leq k$. Fishburn et al. showed an interesting connection between linear labellings of partially ordered sets and the bandwidth of graphs [13]. The *incomparability graph* $G = G(P)$ of a partially ordered set $P = (V, \prec_P)$ has vertex set V, and two vertices are adjacent if and only if the corresponding elements $a \neq b$ of V are not in relation in P (neither $a \prec_P b$ nor $b \prec_P a$). It is well-known that if β is a linear extension of $P = (V, \prec_P)$ then β is a cocomparability ordering of the incomparability graph $G = G(P)$, and vice versa.

Theorem 2 ([13]). *Let $P = (V, \prec_P)$ be a partially ordered set, where V is finite. Let $k \geq 0$. Then, $P = (V, \prec_P)$ has a k-linear labelling if and only if the incomparability graph of $P = (V, \prec_P)$ has bandwidth at most k.*

For each cocomparability graph G, there is a partially ordered set P such that G is the incomparability graph of P. Therefore, Theorem 2 implies that every cocomparability graph G has an optimal layout β such that β is a linear extension of P, and thus a cocomparability ordering of G. We shall heavily rely on the following consequence of this for connected bipartite permutation graphs.

Corollary 3. *Let $G = (A, B, E)$ be a connected bipartite permutation graph, and let $k \geq 0$ be a number. Let (σ_A, σ_B) be a strong ordering for G. If G has a k-layout then G has a k-layout β (a cocomparability ordering) satisfying the following two conditions:*

- (C1) *for every pair a, a' of vertices from A, $a \prec_{\sigma_A} a'$ implies $a \prec_\beta a'$, and for every pair b, b' of vertices from B, $b \prec_{\sigma_B} b'$ implies $b \prec_\beta b'$*
- (C2) *for every triple a, b, b' of vertices of G where $a \in A$ and $b, b' \in B$ and $ab \in E$ and $ab' \notin E$, neither $a \prec_\beta b' \prec_\beta b$ nor $b \prec_\beta b' \prec_\beta a$.*

For more information on partially ordered sets, graph classes and vertex orderings we refer to [7, 15, 25].

3 Bandwidth of bipartite permutation graphs

We call a layout of a connected bipartite permutation graph *normalised* if it satisfies conditions (C1) and (C2) with respect to some given strong ordering. Hence, to

decide whether a given connected bipartite permutation graph has a k-layout for some $k \geq 0$, it suffices to check for normalised k-layouts. In this section we give the main idea behind our algorithm, and these arguments will be used to prove the correctness of the final algorithm that is presented in Section 5.

Observation 1. *Let $G = (A, B, E)$ be a connected bipartite permutation graph, let (σ_A, σ_B) be a strong ordering of G with $A = \{a_1, \ldots, a_s\}$ and $B = \{b_1, \ldots, b_t\}$ where $a_1 \prec_{\sigma_A} \cdots \prec_{\sigma_A} a_s$ and $b_1 \prec_{\sigma_B} \cdots \prec_{\sigma_B} b_t$, and let $G_i = G[N[\{a_1, a_2, ..., a_i\}]]$ for $1 \leq i \leq s$. Assume that a G_{i-1} has a normalised k-layout β. Then G_i has k-layout if and only if it has a normalised k-layout α where (compared to β)*

- *the internal order of the A-vertices of G_{i-1} is not changed,*
- *a_i appears to the right of a_{i-1},*
- *a_i does not appear to the left of any B-vertex that it is not adjacent to,*
- *the internal order of the B-vertices of G_{i-1} is not changed, and*
- *vertices of $N(a_i) \setminus V(G_{i-1})$ appear to the right of all vertices of G_{i-1}.*

Proof Observe that since G is connected and by the mentioned properties of strong orderings, a_i is adjacent to the last vertex in β. With this, the first four conditions listed follow from the results mentioned in the preliminaries. For the last condition, definitely $N(a_i) \setminus V(G_{i-1})$ must appear to the right of all B-vertices of G_{i-1}, and since these vertices have no neighbours in G_{i-1} but are all adjacent to a_i, indeed no vertex of G_{i-1} can appear to the right of any of them in a normalised layout. ∎

For the rest of the paper, we let $G_i = G[N[\{a_1, a_2, ..., a_i\}]]$ for $1 \leq i \leq s$. If there is a desired k-layout α for G_i as described in Observation 1 then α satisfies one of the following two conditions, which we will analyse separately.

1. The position of a_i in α is at least $|V(G_{i-1})| + 1$.
2. The position of a_i in α is at most $|V(G_{i-1})|$.

1. $\alpha^{-1}(a_i) > |V(G_{i-1})|$: To check whether a normalised k-layout α exists for G_i that satisfies this condition, we place all vertices of $N[a_i] \setminus V(G_{i-1})$ to the right of the rightmost vertex of G_{i-1} according to β. The order of the newly added B-vertices is according to σ_B, and we place a_i as far to the left as its rightmost neighbour allows (at most k positions away from the end), but not further left than position $|V(G_{i-1})| + 1$. Let us call this new layout for G_i, β'. Let b be the leftmost neighbour of a_i in β'. If $d_{\beta'}(a_i, b) \leq k$ then β' is the desired k-layout and we are done. In the opposite case, we need to move $d_{\beta'}(a_i, b) - k$ A-vertices between b and a_i to the left of b. If there are fewer A-vertices between b and a_i, it means that there are more than $d_{\beta'}(a_i, b) - k$ B-vertices between them. In this case, the distance between b and a_i cannot be reduced, and since a_i is to the right of all B-vertices of G_{i-1} by our assumption on α, we conclude that a desired k-layout does not exist. Otherwise, let a be the $(d_{\beta'}(a_i, b) - k)$th closest A-vertex to b to the right of b. *A normalised k-layout α as assumed exists if and only if there is a normalised k-layout β_* of G_{i-1} where b appears to the right of a.* To see this, observe that none of the vertices in $N(a_i) \setminus V(G_{i-1})$ has neighbours in G_{i-1}, and

b is the leftmost neighbour of a_i in every normalised k-layout of G_i, since b cannot exchange places with other B-vertices in such a layout. Thus appending the layout of $N[a_i] \setminus V(G_{i-1})$ as described above to the end of β_* gives a k-layout for G_i. Checking whether β_* exists for G_{i-1} and how to compute it from β is one of the two key points of our algorithm, and the next section is devoted to this task. The case that we have explained in this paragraph will be resolved by Theorem 8.

2. $\alpha^{-1}(a_i) \leq |V(G_{i-1})|$: To check whether a normalised k-layout α exists for G_i that satisfies this condition, we place all vertices of $N(a_i) \setminus V(G_{i-1})$ to the right of the rightmost vertex of G_{i-1} according to β. The order of the newly added B-vertices is according to σ_B. *A normalised k-layout α for G_i as assumed in this case exists if and only if there is a normalised k-layout β_* for $G_{i-1}+a_i$ where a_i is placed between b_{p-1} and b_p* (there might be other A-vertices between b_{p-1} and b_p as well) *for some B-vertex b_p in G_{i-1} with $p \geq 2$, and $\beta_*^{-1}(a_i) \geq |V(G_i)| - k$.* To see this, observe that a_i is the only vertex in $G_{i-1}+a_i$ that has neighbours in $N(a_i) \setminus V(G_{i-1})$. Hence if β_* is a k-layout for $G_{i-1}+a_i$, the ordering obtained by appending the vertices of $N(a_i) \setminus V(G_{i-1})$ to the end of β_* gives a k-layout for G_i. The condition $\beta_*^{-1}(a_i) \geq |V(G_i)| - k$ is necessary since a_i is adjacent to the rightmost vertex in every normalised k-layout of G_i satisfying the condition of this case. To see that we do not need to consider the case where a_i is moved to the left of b_1, notice that if there is a normalised k-layout for $G_{i-1}+a_i$ where a_i appears to the left of b_1, then all A-vertices appear before all B-vertices, and exchanging the places of a_i and b_1 results also in a normalised k-layout since a_i has no neighbours to its left, and b_1 has no neighbours to its right. Thus we need to check for each B-vertex b_p of G_{i-1} whether there is a normalised layout β_* for $G_{i-1}+a_i$ where a_i is placed between b_{p-1} and b_p. We check this for at most k of the rightmost B-vertices in β, since otherwise the distance between a_i and its rightmost neighbour will be too large. Placing a_i between b_{p-1} and b_p might of course require moving other A-vertices. How to check whether such a layout exists for $G_{i-1}+a_i$ is the second of the two key points of our algorithm, and it will be handled in the next section. The case that we have explained in this paragraph will be resolved by Theorem 9.

4 Deciding the existence of desired layouts

We need to decide, given a normalised k-layout for G_{i-1}, whether there exists a normalised k-layout for G_{i-1} where a given B-vertex b is required to appear to the left of a given A-vertex a, and whether there exists a normalised k-layout for $G_{i-1}+a_i$ where a_i appears between two given B-vertices b_{p-1} and b_p. To check this, our approach is to indeed place the vertices as desired, and then check if the modified layout can be repaired to become a normalised k-layout. For the first question, we want to place b immediately to the right of a, forbid b to move left, and check whether this layout can be turned into a k-layout with this restriction. For the second question, we want to place a_i between b_{p-1} and b_p, forbid it to move in any direction, and check whether this layout can be turned into a k-layout with this restriction. Hence, we need an algorithm that takes as input a given layout with restrictions on how the vertices are allowed to move, and checks whether this

can be turned into a normalised k-layout.

In this section we present exactly such an algorithm. The algorithm starts from a layout that satisfies certain properties of a desired layout. By moving vertices, we want to make the given layout into a k-layout or find an easily detectable pattern showing that there is no desired layout. According to Corollary 3, it suffices to restrict to normalised layouts. Before we give the algorithm, the following lemma shows that a certain layout modification of a normalised layout yields again a normalised layout. Let $G = (A, B, E)$ be a bipartite permutation graph, and let β be a normalised layout for G. Let u and v be adjacent vertices of G. We want to obtain another normalised layout by moving u one position closer to v. This is possible if and only if there is a vertex of the colour class of v between u and v in β. Let w be such a vertex that is closest to u. We define *the layout obtained from β by moving u one position closer to v* to be layout β' which we obtain by exchanging the position of w with the vertex next to it in the direction towards u repeatedly until w is next to u, and then exchanging the positions of u and w. It is important to note that whenever two consecutive vertices exchange positions, they are neighbours in G, as will be clear in the proof of the lemma. The orderings defined by β and β' restricted to A or B are equal, v has the same position in β and β', and $d_{\beta'}(u, v) = d_\beta(u, v) - 1$.

Lemma 4. *Let $G = (A, B, E)$ be a connected bipartite permutation graph, and let (σ_A, σ_B) be a strong ordering for G. Let β be a normalised layout for G. Let u and v be adjacent vertices, and let there be a vertex of the colour class of v between u and v in β. The layout obtained from β by moving u one position closer to v is normalised.*

Proof Without loss of generality, we can assume that u is an A-vertex and v is a B-vertex, since β is normalised also for graph (B, A, E). We assume that $u \prec_\beta v$, which means that we execute a right move operation on u. The case $v \prec_\beta u$ is analogous, particularly, since the reverse of β is a normalised layout for G (using the reversed orderings σ_A and σ_B). To improve readability, let $a =_{\text{def}} u$ and $b =_{\text{def}} v$. Hence, by the premises of the lemma, there is at least one B-vertex between a and b. Let b' be the first B-vertex to the right of a, meaning that b' is closest to a of all B-vertices between a and b. By Condition (C2) of Corollary 3, ab' is an edge. If b' is immediately to the right of a then exchanging the positions of a and b' gives exactly the layout obtained from β by moving a one position closer to b, and it is normalised since a and b' are adjacent. Otherwise, there are only A-vertices between a and b', and again by condition (C2) of Corollary 3 all A-vertices between a and b' are adjacent to b. Let a' be the furthest to the right of these A-vertices; hence b' is immediately to the right of a'. By the same argument that we applied on a above, we can exchange the positions of b' and a' and obtain a normalised layout. We repeat this for all A-vertices between a' and a, and exchange the position of b' with each A-vertex to its immediate left, obtaining a new normalised layout each time. Finally when b' is immediately to the right of a, we exchange the positions of b' and a to obtain a normalised layout which is exactly the layout obtained from β by moving a one position closer to b. ∎

Now we describe the algorithm which we call `MoveRepair`. Input is a graph G,

an integer $k \geq 0$, a layout β of G, and a direction assignment h in which every vertex is assigned one of the following four symbols: $\cdot, \leftarrow, \rightarrow, \leftrightarrows$. These symbols stand for directions in which a vertex can be moved. For example, a vertex with direction symbol $\leftarrow$ can be moved only to the left. Symbol $\leftrightarrows$ does not specify a special direction, thus these vertices can be moved in any direction. Symbol $\cdot$ does not allow any move. Algorithm `MoveRepair` detects patterns in the layout and works according to a set of rules, presented in Figure 1. Let β and h be the layout and assignment before a rule is applied, and let β_* and h_* be the modified layout and assignment as a result of the applied rule. The interpretation is as follows: let u and v be adjacent vertices with $u \prec_\beta v$ and $d_\beta(u, v) > k$. If $h(u) \in \{\cdot, \leftarrow\}$ and $h(v) \in \{\cdot, \rightarrow\}$ then the first rule is applied and the algorithm rejects β. If $h(u) = \leftrightarrows$ and $h(v) \in \{\cdot, \rightarrow\}$ then the second rule is applied, and the direction symbol of u is changed to $\rightarrow$, hence β_* is the same as β, whereas h_* is the same as h except that $h_*(u) = \rightarrow$. Braces in the table offer different possibilities. Note that there are four patterns that are not contained in the table; these patterns do not imply any action.

(Os1)	$\overset{\leftarrow}{\cdot}\}$	$\xrightarrow{>k}$	$\{\overset{\rightarrow}{\cdot}$	`reject`
(Os2)	$\leftrightarrows$	$\xrightarrow{>k}$	$\{\overset{\rightarrow}{\cdot}$	replace by $\rightarrow \xrightarrow{>k} \{\overset{\rightarrow}{\cdot}$
(Os3)	$\overset{\leftarrow}{\cdot}\}$	$\xrightarrow{>k}$	$\leftrightarrows$	replace by $\overset{\leftarrow}{\cdot}\} \xrightarrow{>k} \leftarrow$
(Os4)	$\rightarrow$	$\xrightarrow{>k}$	$\{\overset{\rightarrow}{\cdot}$	`MoveAttempt` left vertex to right
(Os5)	$\overset{\leftarrow}{\cdot}\}$	$\xrightarrow{>k}$	$\leftarrow$	`MoveAttempt` right vertex to left

Figure 1: Patterns and rules for deciding the existence of a desired layout.

Algorithm `MoveRepair`
Input: A graph G, a layout β for G, a direction symbol assignment h to the vertices of G and an integer $k \geq 0$.
Output: A layout β_* and a reply `accept` or `reject`.

while there is an edge uv in G satisfying one of the given patterns in Figure 1 **do**
  execute the corresponding operation on edge uv with input (β, h) and output (β_*, h_*);
  $\beta =_{\text{def}} \beta_*$; $h =_{\text{def}} h_*$;
end-while
`accept`;

We should mention that whenever a `reject` is executed, the algorithm terminates with output `reject`, and the consecutive instructions are not executed. It only remains to describe the `MoveAttempt` operation. Assume that $u \prec_\beta v$. We describe the operation `MoveAttempt` *left vertex to right*. (The `MoveAttempt` *right vertex to left* operation is the symmetric case and defined analogously.) Since `MoveAttempt` left vertex to right is invoked, we know that u has symbol $\rightarrow$ and v has $\cdot$ or $\rightarrow$. If there is a vertex between u and v belonging to the colour class of v then u can be moved one position closer to v if the direction symbols of vertices between u and w allow this, otherwise not. Let w be a vertex of the colour class of v that is to the right of u and closest to u of all such vertices. As described in the beginning of this section, moving u one position closer to v means that w moves to the left of u, and

all vertices between u and w move one position to the right. Hence this move is only possible if w has symbol $\leftarrow$ or $\leftrightarrows$, and all vertices between u and w (all from the colour class of u) have symbol $\rightarrow$ or $\leftrightarrows$.

Operation `MoveAttempt` left vertex to right
Input: A graph G, a layout β and a direction symbol assignment h on G, two adjacent vertices u, v with distance more than k in β that satisfy the condition of (Os4) in Figure 1.
Output: A layout β_* and a direction symbol assignment h_* on G, or a reply `reject`.

if there is no vertex of the colour class of v between u and v in β **then**
    `reject`
else
    let w be the closest vertex to the right of u belonging to the colour class of v;
    if $h(w) \notin \{\leftarrow, \leftrightarrows\}$ **then**
        `reject`
    else
        if all vertices between u and w in β have symbol $\rightarrow$ or $\leftrightarrows$ in h **then**
            $h_* =_{\text{def}} h$;
            for every vertex x between u and w in β **do** $h_*(x) =_{\text{def}} \rightarrow$ **end-for**
            $h_*(w) =_{\text{def}} \leftarrow$;
            $\beta_* =_{\text{def}}$ the layout obtained from β by moving u one position closer to v;
        else
            `reject`
        end-if
    end-if
end-if

By Lemma 4 and the description of the algorithm, it follows that if the input layout to Algorithm `MoveRepair` is normalised then the layout produced after each single operation, and hence the output layout, is also normalised. It is important to note that whenever a vertex is moved in one direction, its direction symbol is fixed to indicate this direction, and it is not allowed to move in the other direction during the same execution of Algorithm `MoveRepair`.

A bad situation would occur if an edge of distance more than k between its endpoints had on both its endpoints "inward" arrows or $\leftrightarrows$, which would give several possibilities to repair this edge and too many possibilities in total. We will ensure that this situation never occurs, and we introduce the following definition which encapsulates this idea. We say that (β, h) has the *outward arrows* property if the following is true for every edge uv of G with $u \prec_\beta v$: if $d_\beta(u, v) > k$ then $h(u) \in \{\cdot, \leftarrow\}$ or $h(v) \in \{\cdot, \rightarrow\}$. Observe that the rules of Algorithm `MoveRepair` apply precisely to those edges. Algorithm `MoveRepair` will always be called with input that has the outward arrows property. Lemma 5 shows that this property will be maintained throughout all layouts produced during the algorithm.

Let $G = (A, B, E)$ be a connected bipartite permutation graph, and let (σ_A, σ_B) be a strong ordering for G. Let $k \geq 0$. Let β be a normalised layout for G and let h be a direction symbols assignment for G. We define $\Delta_{G,k}(\beta, h)$ to be the set of normalised k-layouts γ for G satisfying the following three properties for every vertex x of G: if $h(x) = \cdot$ then $\gamma^{-1}(x) = \beta^{-1}(x)$; if $h(x) = \leftarrow$ then $\gamma^{-1}(x) \leq \beta^{-1}(x)$; if $h(x) = \rightarrow$ then $\gamma^{-1}(x) \geq \beta^{-1}(x)$. For simplicity we write

$\Delta(\beta, k)$, since the context G and k is always clear. With the following two lemmas we show that if (β, h) has the outward arrows property then Algorithm `MoveRepair` correctly decides whether $\Delta(\beta, h)$ is empty.

Lemma 5. *Let $G = (A, B, E)$ be a connected bipartite permutation graph, and let (σ_A, σ_B) be a strong ordering for G. Let $k \geq 0$. Let β be a normalised layout for G, and let h be a direction symbol assignment for G that has the outward arrows property. Then each of the layout-assignment pairs generated throughout Algorithm* `MoveRepair` *has the outward arrows property, and if Algorithm* `MoveRepair` *accepts input (β, h) and outputs β_* then $\beta_* \in \Delta(\beta, h)$.*

Proof Let $(\beta, h)=(\beta_0, h_0), \ldots, (\beta_l, h_l)=(\beta_*, h_*)$ be the layout-assignment pairs generated by the algorithm on input (β, h). The following is clear from the definition of the algorithm regardless of the properties of β and h, for every vertex x of G and $i \in \{1, \ldots, l\}$:

1. if $h_i(x) = \leftarrow$ then $h_{i-1}(x) \in \{\leftarrow, \leftrightarrows\}$ and $\beta_i^{-1}(x) \leq \beta_{i-1}^{-1}(x)$
2. if $h_i(x) = \rightarrow$ then $h_{i-1}(x) \in \{\rightarrow, \leftrightarrows\}$ and $\beta_i^{-1}(x) \geq \beta_{i-1}^{-1}(x)$
3. if $h_i(x) \in \{\cdot, \leftrightarrows\}$ then $h_{i-1}(x) = h_i(x)$ and $\beta_i^{-1}(x) = \beta_{i-1}^{-1}(x)$.

We first show that (β_i, h_i) either is a k-layout or has the outward arrows property, for each $i \in \{0, ..., l\}$. By the premises of the lemma this is true for $i = 0$. Assume that it is true for $i - 1$ but not for i for some $i > 0$. If β_i is not a k-layout for G, there are adjacent vertices u and v, $u \prec_{\beta_i} v$, such that $h_i(u) \in \{\rightarrow, \leftrightarrows\}$ and $h_i(v) \in \{\leftarrow, \leftrightarrows\}$ and $d_{\beta_i}(u, v) > k$. With the listed properties above, we obtain that $\beta_{i-1}^{-1}(u) \leq \beta_i^{-1}(u)$ and $\beta_{i-1}^{-1}(v) \geq \beta_i^{-1}(v)$ and $h_{i-1}(u) \in \{\rightarrow, \leftrightarrows\}$ and $h_{i-1}(v) \in \{\leftarrow, \leftrightarrows\}$. Then $d_{\beta_{i-1}}(u, v) > k$, which contradicts that h_{i-1} has the outward arrows property. Hence, either β_i is a k-layout for G or (β_i, h_i) has the outward arrows property. Since the algorithm stops and accepts, no edge uv with $u \prec_{\beta_*} v$ and $d_{\beta_*}(u, v) > k$ satisfies $h_*(u) \in \{\cdot, \leftarrow\}$ or $h_*(v) \in \{\cdot, \rightarrow\}$; otherwise the algorithm would continue. Since h_* has the outward arrows property, there cannot be any other edge with distance more than k between its endpoints in β_*, and therefore β_* is a k-layout. Furthermore, as argued before, (β_i, h_i) is a normalised layout for every $i \in \{0, ..., l\}$, and we conclude that $\beta_* \in \Delta(\beta, h)$. ∎

The following lemma is the negative case counterpart of Lemma 5 and shows that the actions taken by Algorithm `MoveRepair` are *forced*, meaning that if the algorithm moves a vertex in one direction to repair an edge, then there is no k-layout in $\Delta(\beta, h)$ in which this vertex is moved in the other direction. The claim appearing inside the proof states this formally and is the key to understand the idea.

Lemma 6. *Let $G = (A, B, E)$ be a connected bipartite permutation graph, and let (σ_A, σ_B) be a strong ordering for G. Let β be a normalised layout for G and let h be a direction symbols assignment for G. If Algorithm* `MoveRepair` *rejects input (β, h) then $\Delta(\beta, h)$ is empty.*

Proof To prove the lemma, we assume that the algorithm rejects but $\Delta(\beta, h)$ is non-empty. Let $(\beta, h)=(\beta_0, h_0), \ldots, (\beta_l, h_l)$ be the layout-assignment pairs generated by the algorithm, where (β_i, h_i) is the result after algorithm step i. In step $l+1$,

the algorithm decides rejection. We first show properties relating assigned direction symbols and vertex positions in layouts in $\Delta(\beta, h)$.

Claim. *Let $\Delta(\beta, h)$ be non-empty. For every $i \in \{0, \ldots, l\}$ and every vertex x of G, the following holds:*

- *if $h_i(x) = \cdot$ then $\gamma^{-1}(x) = \beta_i^{-1}(x)$ for every $\gamma \in \Delta(\beta, h)$*
- *if $h_i(x) = \leftarrow$ then $\gamma^{-1}(x) \leq \beta_i^{-1}(x)$ for every $\gamma \in \Delta(\beta, h)$*
- *if $h_i(x) = \rightarrow$ then $\gamma^{-1}(x) \geq \beta_i^{-1}(x)$ for every $\gamma \in \Delta(\beta, h)$.*

Proof Let γ be a layout in $\Delta(\beta, h)$. We show by induction on i that the claim holds for γ. The claim holds for $i = 0$ by the definition of $\Delta(\beta, h)$. Let the claim be true for (β_{i-1}, h_{i-1}) for some $i > 0$. Let (β_i, h_i) be obtained from (β_{i-1}, h_{i-1}) by application of operation o. For all vertices x such that $\beta_i^{-1}(x) = \beta_{i-1}^{-1}(x)$ and $h_i(x) = h_{i-1}(x)$, the claim holds for (β_i, h_i) by the induction hypothesis. For the other vertices, we distinguish between cases according to o. Let uv be the edge to which o is applied, $u \prec_{\beta_{i-1}} v$. Let o be one of the two operations of (Os2). Then, $\beta_i = \beta_{i-1}$ and h_i differs from h_{i-1} only for u. By definition, $h_{i-1}(v) \in \{\cdot, \rightarrow\}$, and $\gamma^{-1}(v) \geq \beta_{i-1}^{-1}(v)$ by the induction hypothesis. Since $d_{\beta_{i-1}}(u, v) > k$, u must be further to the right in γ, i.e., $\gamma^{-1}(u) \geq \beta_{i-1}^{-1}(u)$. Since $h_i(u) = \rightarrow$, the claim holds for this case. Analogously, it is proved that the claim holds in case o is an operation from (Os3). Let o be from (Os4) or (Os5). Since both cases are symmetric, we consider an operation from (Os4). Since (β_i, h_i) is defined, u is moved one position closer to v. Since $\gamma^{-1}(v) \geq \beta_{i-1}^{-1}(v)$ by induction hypothesis and $d_{\beta_{i-1}}(u, v) > k$ by the assumption about application of o, $\gamma^{-1}(u) > \beta_{i-1}^{-1}(u)$. And since $\beta_i^{-1}(u) = \beta_{i-1}^{-1}(u) + 1$ and $h_i(u) = \rightarrow$, we conclude that the claim holds for u. Let w be the closest vertex to the right of u in β_{i-1} from the colour class of v. Remember that $w \prec_{\beta_{i-1}} v$. The only further vertices that change position or direction symbol in (β_i, h_i) with respect to (β_{i-1}, h_{i-1}) are w and the vertices between u and w in β_{i-1}. Since γ is a normalised layout, we conclude that w cannot be to the right of u in γ; otherwise, $\gamma^{-1}(u) \leq \beta_{i-1}^{-1}(u)$, which contradicts the conclusion above. So, $\gamma^{-1}(w) \leq \beta_{i-1}^{-1}(u) = \beta_i^{-1}(w)$, and the claim holds for w with $h_i(w) = \leftarrow$. Correctness for the vertices between u and w in β_{i-1} then immediately follows from the restriction of γ to a normalised layout and the correctness for u, since all these vertices are assigned direction symbol $\rightarrow$ by h_i. □

Now, let uv be the edge which is considered by the algorithm in step $l + 1$. Let $u \prec_{\beta_l} v$. Since the algorithm rejects in step $l + 1$, the executed operation is one of the set (Os1) or (Os4–5). We first consider set (Os1). According to the definition of (Os1) and the claim, $\gamma^{-1}(u) \leq \beta_l^{-1}(u)$ and $\beta_l^{-1}(v) \leq \gamma^{-1}(v)$ for all $\gamma \in \Delta(\beta, h)$. Since $d_{\beta_l}(u, v) > k$, γ cannot be a k-layout for G. Let the executed operation now be from set (Os4). The case (Os5) is analogous. According to the definition of `MoveAttempt` left vertex to right, we have to distinguish between three cases which can imply rejection. Let w be the closest vertex to u from the colour class of v to the right of u in β_l. Note that w exists. If $w = v$ then all vertices between u and v in β_l are from the colour class of u, and u can be closer to v only by moving v closer to u. This, however, is not possible for a layout in $\Delta(\beta, h)$ and $h_l(v) = \rightarrow$. So, let $w \prec_{\beta_l} v$. Let $h_l(w) \in \{\cdot, \rightarrow\}$. Then, $\beta_l^{-1}(w) \leq \gamma^{-1}(w)$ for all $\gamma \in \Delta(\beta, h)$.

By definition of normalised layouts and since there are only vertices from the colour class of u between u and w in β_l, it follows that $\gamma^{-1}(u) \leq \beta_l^{-1}(u)$, which means $\gamma^{-1}(u) = \beta_l^{-1}(u)$ according to the claim and with $h_l(u) = \rightarrow$. This, however, is not possible for layouts in $\Delta(\beta, h)$. Finally, let x be a vertex between u and w in β_l and let $h_l(x) \in \{\cdot, \leftarrow\}$. This particularly means that $x \prec_\gamma w$ for all $\gamma \in \Delta(\beta, h)$. We conclude like in the previous case that $\gamma^{-1}(u) = \beta_l^{-1}(u)$ for all $\gamma \in \Delta(\beta, h)$, which is a contradiction to γ being a k-layout for G. Since we showed contradictions for every possible case, $\Delta(\beta, h)$ cannot be non-empty. ∎

Now, we discuss implementation and running time aspects of `MoveRepair`. From the definition, `MoveRepair` is a nondeterministic algorithm, since vertex pairs to which an operation can be applied can be chosen arbitrarily. We do not want to give a deterministic algorithm that simulates `MoveRepair`, since this would take too much time and is not necessary. Instead, it suffices to simulate a single computation path of `MoveRepair`. Lemmata 5 and 6 show that the answer ('accept' or 'reject') is the same on every computation path for inputs satisfying the outward arrows property. A simple and straightforward implementation already gives $\mathcal{O}(n^5)$ time: it takes $\mathcal{O}(n^2)$ time to find a vertex pair to which an operation can be applied, a single operation can be executed in time $\mathcal{O}(n)$, and every vertex can be moved at most n times, which means that at most n^2 operations are executed. A more intelligent strategy for choosing vertex pairs and a more careful running-time analysis gives a much better result. The input to the algorithm is a normalised layout of a connected bipartite permutation graph and a table containing leftmost and rightmost neighbour of each vertex. Note that this succinct representation contains all information about neighbourhoods.

Theorem 7. *There is an algorithm that can be implemented to run in time $\mathcal{O}(kn)$ on normalised layouts of connected bipartite permutation graphs for every $k \geq 1$ and simulates a possible computation of algorithm* `MoveRepair`*.*

Proof Let $k \geq 1$. We partition the running time analysis into three parts. We first show that, after `MoveRepair` moved some vertex by more than k positions, an easily detectable vertex pair is created to which an operation from set (Os1) can be applied. Let $G = (A, B, E)$ be a connected bipartite permutation graph with strong ordering (σ_A, σ_B). Let β be a normalised layout for G, and let h be a direction symbol assignment for G. Let $(\beta_0, h_0), \ldots, (\beta_l, h_l)$ be the sequence of layout-direction symbol assignment pairs generated by `MoveRepair` on input (G, β, h, k); hence if the algorithm accepts, β_l is the output. Let x be a vertex, and assume that there is $j \leq l$ such that $|\beta^{-1}(x) - \beta_j^{-1}(x)| > k$. Without loss of generality, we can assume that j is smallest possible. This means that $|\beta^{-1}(x) - \beta_{j-1}^{-1}(x)| \leq k$, which particularly means that a `MoveAttempt` operation involving x was (successfully) executed in step j. By repeatedly applying the arguments at the beginning of the proof of Lemma 5, we obtain that $h_j(x) \in \{\leftarrow, \rightarrow\}$. We consider the case $h_j(x) = \rightarrow$; the other case is analogous. By the same arguments and the definition of the operation sets, it follows that there is an integer $j' < j$ such that $\beta_i^{-1}(x) = \beta^{-1}(x)$ for all $i \leq j'$, and $\beta_i^{-1}(x) \geq \beta_{i-1}^{-1}(x)$ and $h_i(x) = \rightarrow$ for all $i \in \{j' + 1, \ldots, j\}$. Let p be smallest such that $\beta_p^{-1}(x) > \beta_{p-1}^{-1}(x)$; clearly, $j' < p < j$. Then, in step p of `MoveRepair`, a `MoveAttempt` operation was executed, and there is a vertex y such

that $\beta_p^{-1}(y) < \beta_{p-1}^{-1}(y)$ and $\beta_{p-1}^{-1}(x) < \beta_{p-1}^{-1}(y)$ and $\beta_p^{-1}(y) < \beta_p^{-1}(x)$. We assume that y is leftmost with respect to β, which also implies that $\beta_p^{-1}(y) \leq \beta^{-1}(x)$. It is important to note that x and y are adjacent in G, which follows from β_{p-1} and β_p being normalised. By the definition and properties of `MoveAttempt` and `MoveRepair`, $h_p(y) = \cdots = h_j(y) = \leftarrow$ and $\beta_j^{-1}(y) \leq \beta_p^{-1}(y)$. Hence, $\beta_j^{-1}(x) - \beta_j^{-1}(y) \geq k+1$. And since $h_j(x) = \rightarrow$ and $h_j(y) = \leftarrow$, an operation from set (Os1) can be applied to the pair x, y in the next step. For further considerations, we give a name to y: we call y the *partner vertex* of x. So, the partner vertex of a vertex is determined when it is moved the first time.

In the second part, we describe a strategy for choosing vertex pairs to which the next operation is applied, that implements the idea above about not moving a single vertex too far. The basic idea is to fix vertex pairs for moving and for verifying. In a queue, we store the pairs to which `MoveAttempt` is applied. The strategy is based on the following two observations.

- Let u, v be a pair of vertices to which a replace or a `MoveAttempt` operation is applied. We assume that u is to the left of v. Depending on the case, u is assigned direction symbol $\rightarrow$ or v is assigned direction symbol $\leftarrow$. Let the former be the case; the latter is analogous. Suppose that u is not the leftmost neighbour of v; let y be leftmost neighbour. Then, `MoveRepair` can apply an operation also to the pair y, v, which is from (Os1), (Os2) or (Os4).
- After execution of a step by `MoveRepair`, pairs of vertices to which an operation can be applied are pairs of vertices to which an operation could have been applied before the last operation execution or one of the two vertices has been actively involved (thus, has been moved or obtained a new direction symbol) in the last operation execution.

These two observations lead to the following algorithm: from the queue, pick the first vertex, say u. The direction symbol of u is not $\leftrightarrows$. If the direction symbol of u is $\rightarrow$, find the leftmost neighbour v of u. If v is to the left of u and at distance more than k, an operation can be applied to the pair v, u. If this operation is from set (Os2), apply the operation from (Os4) right after. After execution of a `MoveAttempt` operation, that moved and did not reject, check whether an operation from set (Os1) can be applied to a moved vertex together with its partner vertex. (Remember that every moved vertex indeed has a partner vertex.) If `MoveRepair` does not reject, add all moved vertices and u itself to the queue and continue. If v is not to the left of u or if u and v are at distance at most k, continue with the next vertex in the queue. If the direction symbol of u is $\leftarrow$, the procedure is analogous. And if the direction symbol of u is $\cdot$, apply the procedure for $\leftarrow$ and $\rightarrow$ to u. It remains to initialize the queue. All operations require a vertex whose direction symbol is not $\leftrightarrows$. So, initialize the queue by inserting exactly these vertices. This completes the definition of the strategy. We have to show now that `MoveRepair` accepts an input if and only if it accepts an input being restricted to our strategy. It is clear that if `MoveRepair` with strategy rejects then `MoveRepair` without strategy also rejects. The converse is more complex. From the definition of the strategy, it is clear that after every "round" the queue contains exactly the vertices whose assigned

direction symbols are different from $\leftrightarrows$. Suppose now `MoveRepair` with strategy accepts but there is still a vertex pair x, y to which an operation of `MoveRepair` can be applied. Then, x or y is not assigned direction symbol $\leftrightarrows$. This means that x or y was in the queue at the end of the last round. Let (β_*, h_*) be the last pair computed by `MoveRepair` with strategy. Without loss of generality, $\beta_*^{-1}(x) < \beta_*^{-1}(y)$ can be assumed. If $h_*(x) = \cdot$ or $h_*(y) = \cdot$, then an operation can be applied to x with its rightmost neighbour of to y with its leftmost neighbour, which contradicts the assumption. Hence, $h_*(x), h_*(y) \in \{\leftarrow, \rightarrow, \leftrightarrows\}$. If $h_*(x) = \leftarrow$ or $h_*(y) = \rightarrow$, again an operation can be applied to x with its rightmost neighbour or to y with its leftmost neighbour. Therefore, $h_*(x) \in \{\rightarrow, \leftrightarrows\}$ and $h_*(y) \in \{\leftarrow, \leftrightarrows\}$. However, no operation can be applied in any of these cases, which contradicts the assumption that an operation can be applied to x, y. We conclude that `MoveRepair` with strategy correctly implements a possible computation of `MoveRepair`.

In the third part, we address running-time aspects. Using arrays as tables the position in the layout and the direction symbol of every vertex can be determined in constant time. Furthermore, leftmost and rightmost neighbour of a vertex are the same vertices in every normalised layout, so the ones that are given in tables as input. For a given pair of vertices, it can be determined in constant time whether an operation can be executed and, if so, which one. Operations from the sets (Os1–3) can be executed in constant time. For operations from the sets (Os4–5), i.e., `MoveAttempt` operations, we analyse as follows. Let `MoveAttempt` be applied to the vertex pair u, v. The algorithm first checks whether there is a vertex from the colour class of v between u and v in the layout; by starting at u, the algorithm finds the closest such vertex w. The two further conditions can be checked for each passed vertex in constant time, which means that the verification step takes time proportional to the distance between u and w. If the verification step succeeds, the moving step also takes time proportional to the distance of u and w, i.e., `MoveAttempt` takes time proportional to $d(u, w)$. If the verification step fails, `MoveAttempt` runs in $\mathcal{O}(n)$ time. The crucial observation for the successful case, however, is that `MoveAttempt` moves $d(u, w) + 1$ vertices each by at least one position.

The total running time of `MoveRepair` with strategy mainly depends on the number of executed `MoveAttempt` operations and the time for checking for a vertex pair to which an operation from set (Os1) can be applied. From the result in the first part of the proof we know that every vertex can be moved at most k positions without creating a vertex pair that leads to rejection (operation from set (Os1)). It is immediately clear then that the total running time of `MoveAttempt` operations is $\mathcal{O}(kn)$ (including the time for the last, possibly unsuccessful execution). Since the vertices put into the queue after a `MoveAttempt` is proportional to the running time of the single `MoveAttempt` `MoveRepair` with strategy runs in $\mathcal{O}(kn)$ time not considering the verification part at the end of a round. For the verification part, only vertices that were moved in the current round are considered, and the partner vertex is fixed, so that $\mathcal{O}(kn)$ is the running time of the total algorithm. Remember that the partner vertex of a vertex is determined when it is moved first. This completes the proof. ∎

We are ready to formulate the two main results of this section. In particular

we show that we can use Algorithm `MoveRepair` to decide the existence of desired layouts. We use a simplified representation for normalised layouts which is particularly convenient to describe modifications of a layout. Let $G = (A, B, E)$ be a bipartite permutation graph, and let (σ_A, σ_B) be a strong ordering for G. Let $A = \{a_1, \ldots, a_s\}$ where $a_1 \prec_{\sigma_A} \cdots \prec_{\sigma_A} a_s$. Let β be a normalised layout for G with respect to (σ_A, σ_B). For every vertex a in A, let $n_\beta(a)$ be the number of vertices from B that appear to the left of a in β. Then, $(n_\beta(a_1), \ldots, n_\beta(a_s))$ is a unique representation of β, which we call the *left-neighbour representation*. Note that $0 \leq n_\beta(a_1) \leq \cdots \leq n_\beta(a_s) \leq |B|$ and that every sequence of s numbers with this monotonicity property corresponds to a layout of G that satisfies condition (C1) of Corollary 3.

Theorem 8. *Let $G = (A, B, E)$ be a connected bipartite permutation graph, and let (σ_A, σ_B) be a strong ordering for G. Let $k \geq 0$. Assume that the following holds:*

(A1) *$A = \{a_1, \ldots, a_s\}$ and $B = \{b_1, \ldots, b_t\}$ where $a_1 \prec_{\sigma_A} \cdots \prec_{\sigma_A} a_s$ and $b_1 \prec_{\sigma_B} \cdots \prec_{\sigma_B} b_t$.*

(A2) *b_q is a neighbour of a_s.*

(A3) *$r \in \{1, \ldots, s\}$.*

Then, given a normalised k-layout for G, there is a polynomial-time algorithm that decides whether there is a normalised k-layout for G such that b_q is to the right of a_r. In the positive case, the algorithm outputs such a layout.

Proof Let $\beta' = (d'_1, \ldots, d'_s)$ be the given layout. The idea is to modify β' and to define an appropriate direction symbols assignment for G having the outward arrows property. If $d'_r < q$, b_q is to the right of a_r in β', so that we can immediately answer YES and output β'. So, let $d'_r \geq q$. Let r' be smallest such that $d_{r'} \geq q$. We define $\beta =_{\text{def}} (d'_1, \ldots, d'_{r'-1}, q-1, \ldots, q-1, d'_{r+1}, \ldots, d'_s)$. Note that in every normalised layout γ satisfying $a_r \prec_\gamma b_q$, $\gamma^{-1}(a) \leq \beta^{-1}(a)$ for $a \in \{a_{r'}, \ldots, a_r\}$ and $\beta^{-1}(b) \leq \gamma^{-1}(b)$ for $b \in \{b_q, \ldots, b_{d_r}\}$. This is due to the restriction to normalised layouts and by construction: $a_{r'}, \ldots, a_r$ appear consecutively in β and $b_q, \ldots, b_{d_r}$ appear consecutively in β and b_q is the vertex to the right of a_r. We define the following direction symbols assignment h: $h(a_{r'}) =_{\text{def}} \cdots =_{\text{def}} h(a_r) =_{\text{def}} \leftarrow$, $h(b_q) =_{\text{def}} \cdots =_{\text{def}} h(b_{d_r}) =_{\text{def}} \rightarrow$ and $h(x) =_{\text{def}} \leftrightarrows$ for all other vertices x. We have to show that β is normalised and that (β, h) has the outward arrows property If $r < s$, β is obtained from β' by iteratively moving b_q one position closer to a_s. Since b_q and a_s are adjacent according to assumption (A2) and β' is normalised, β is normalised according to Lemma 4. If $r = s$, we iteratively move b_q one position closer to a_s such that b_q is to the right of a_{s-1}. Then, we iteratively move a_s one position closer to b_q and stop when a_s is the right vertex of b_q. Lemma 4 shows that the obtained layout is normalised. In a last step, we exchange a_s and b_q and obtain β. It is easy to see from the definition of normalised layouts that β then is a normalised layout.

Now, we show that (β, h) has the outward arrows property. Let uv be an edge of G, where $u \prec_\beta v$, and let $d_\beta(u, v) > k$. Then, u or v was moved for obtaining β from β', since otherwise $d_{\beta'}(u, v) > k$, contradicting β' being a k-layout. Remember that A-vertices are moved to the left only and B-vertices are moved to the right only. So,

v cannot be an A-vertex and u cannot be a B-vertex. Hence, $h(u) \in \{\leftarrow, \leftrightarrows\}$ and $h(v) \in \{\rightarrow, \leftrightarrows\}$ and $h(u) \neq h(v)$. Thus, (β, h) has the outward arrows property.

Combining the proved results, Lemma 5 shows that, if the algorithm on input (β, h) accepts, the output is a normalised k-layout for G. If the algorithm on input (β, h) rejects, Lemma 6 shows that $\Delta(\beta, h)$ is empty. According to the definition of (β, h), $\Delta(\beta, h)$ is the set of normalised k-layouts for G where b_q is to the right of a_r. Since the algorithm runs in polynomial time and gives the correct answer, we conclude the proof. ∎

Theorem 9. *Let $G = (A, B, E)$ be a connected bipartite permutation graph, and let (σ_A, σ_B) be a strong ordering for G. Let $k \geq 0$. Assume that the following holds:*

(A1) *$A = \{a_1, \ldots, a_s, a_{s+1}\}$ and $B = \{b_1, \ldots, b_t\}$ where $a_1 \prec_{\sigma_A} \cdots \prec_{\sigma_A} a_{s+1}$ and $b_1 \prec_{\sigma_B} \cdots \prec_{\sigma_B} b_t$.*

(A2) *$N(a_{s+1}) \subseteq N(a_s)$.*

(A3) *b_p is a neighbour of a_{s+1}, where $p \geq 2$.*

Then, given a normalised k-layout for $G - a_{s+1}$, there is a polynomial-time algorithm that decides whether there is a normalised k-layout for G such that a_{s+1} is between b_{p-1} and b_p. In the positive case, the algorithm outputs such a layout.

Proof The outline of the proof is analogous to the proof of Theorem 8. Let $\beta' = (d'_1, \ldots, d'_s)$ be the given layout for $G - a_{s+1}$. Let r be smallest such that $d'_r \geq p$; if r is undefined, set $r =_{\text{def}} s + 1$. We define $\beta'' =_{\text{def}} (d'_1, \ldots, d'_{r-1}, p-1, \ldots, p-1)$. With the same arguments as in the proof of Theorem 8, β'' is a normalised layout for $G - a_{s+1}$ (choose $r = s$ in Theorem 8). We obtain β from β'' by adding a_{s+1} as the left vertex of b_p, i.e., $\beta = \beta'' \circ (p-1)$. According to the definition of normalised layouts, β is also normalised. The direction symbols assignment h is defined as follows: $h(a_{s+1}) =_{\text{def}} h(b_p) =_{\text{def}} \cdots =_{\text{def}} h(b_t) =_{\text{def}} \cdot$ and all other vertices are assigned $\leftrightarrows$. We show that (β, h) has the outward arrows property. Let uv be an edge of G, where $u \prec_\beta v$, such that $d_\beta(u, v) > k$. If $v \prec_\beta a_r$ then u and v have the same positions in β and β', i.e., $d_\beta(u, v) \leq k$. If $v \in \{a_r, \ldots, a_s\}$ then $\beta^{-1}(v) < (\beta')^{-1}(v)$ and $\beta^{-1}(u) = (\beta')^{-1}(u)$, so that $d_\beta(u, v) \leq k$. Hence, $v \in \{a_{s+1}, b_p, \ldots, b_t\}$, and $a_{s+1}, b_p, \ldots, b_t$ are assigned direction symbol $\cdot$. So, (β, h) has the outward arrows property. Hence, the algorithm accepts on input (β, h) if and only if $\Delta(\beta, h)$ is non-empty. It is not hard to see that the layouts in $\Delta(\beta, h)$ are exactly the normalised k-layouts for G where a_{s+1} is the left vertex of b_p. ∎

We give a remark on the proof of Theorem 9. We apply this result to find a normalised k-layout for G such that a_{s+1} is to the right of b_{p-1}. For the definition of $\Delta(\beta, h)$, it would be sufficient to assign $\rightarrow$ to a_{s+1} and $\leftarrow$ to $b_p, \ldots, b_t$. Then, however, the condition of Lemma 5 cannot be satisfied, i.e., the algorithm might accept and output a layout that is not a k-layout. Fixing the positions of these vertices by assigning $\cdot$ is the only possibility.

5 A polynomial-time algorithm for computing the bandwidth of bipartite permutation graphs

In this section we present the polynomial-time algorithm for computing the bandwidth of bipartite permutation graphs. In the algorithm, symbol $\circ$ denotes the concatenation operation.

Algorithm `BPG-Bandwidth` (Bipartite Permutation Graphs Bandwidth)
Input: A connected bipartite permutation graph $G = (A, B, E)$, a strong ordering (σ_A, σ_B) for G and an integer k.
Output: A reply `accept` and a k-layout β for G if $\mathrm{bw}(G) \leq k$, or a reply `reject` if $\mathrm{bw}(G) > k$.

let $A = \{a_1, \ldots, a_s\}$ and $B = \{b_1, \ldots, b_t\}$ where $a_1 \prec_{\sigma_A} \cdots \prec_{\sigma_A} a_s$ and $b_1 \prec_{\sigma_B} \cdots \prec_{\sigma_B} b_t$;
if $|N(a_1)| > 2k$ **then**
  `reject`; **stop**;
end-if
let β be a normalised k-layout for G_1;
for $i = 2$ **to** s **do** (*)
  let δ be a normalised k-layout for $G[N[a_i] \setminus V(G_{i-1})]$ with a_i leftmost possible;
  let δ' be a normalised k-layout for $G[N(a_i) \setminus V(G_{i-1})]$;
  $\beta' =_{\mathrm{def}} \beta \circ \delta$;
  if β' is a k-layout for G_i **then**
    $\beta =_{\mathrm{def}} \beta'$; **goto** the next iteration of main for-loop (*);
  else
    let b be the leftmost neighbour of a_i in β';
    if there are less than $d_{\beta'}(a_i, b) - k$ A-vertices between a_i and b **then**
      `reject`; **stop**;
    else
      let a be the $(d_{\beta'}(a_i, b) - k)$th A-vertex closest to b between b and a_i in β';
(1)   **if** there is a normalised k-layout β_* for G_{i-1} in which b appears to the right of a **then**
        $\beta =_{\mathrm{def}} \beta_* \circ \delta$; **goto** the next iteration of main for-loop (*);
      end-if
    end-if
  end-if
  for each of the k last B-vertices b_p in β **do**
(2)   **if** there is a normalised k-layout β_* for $G_{i-1}+a_i$ where a_i appears between b_{p-1} and b_p **then**
        $\beta' =_{\mathrm{def}} \beta_* \circ \delta'$;
(3)   **if** the distance between a_i and its rightmost neighbour in β' is at most k **then**
          $\beta =_{\mathrm{def}} \beta'$; **goto** the next iteration of main for-loop (*);
        end-if
      end-if
  end-for
  `reject`; **stop**;
end-for (*)
`accept`;

Lemma 10. *Let $G = (A, B, E)$ be a connected bipartite permutation graph, and let (σ_A, σ_B) be a strong ordering for G. Let $k \geq 0$. Then,* `BPG-Bandwidth` *on input G, (σ_A, σ_B) and k accepts if and only if* $\mathrm{bw}(G) \leq k$. *In the accepting case, the output layout is a k-layout for G.*

Proof Let us call an iteration of the main for-loop marked by (*), *a step* of the algorithm. We prove the lemma by induction on the number of steps i of

the algorithm. The induction hypothesis is the following: if `BPG-Bandwidth` computes a k-layout β for G_i then this is a correct normalised k-layout for G_i, and if `BPG-Bandwidth` rejects at step i, then G_i has no k-layout and hence $bw(G) > k$. For $i = 1$, G_1 is a star with center in a_1. If the algorithm rejects then $|N(a_1)| > 2k$, which means that $|V(G_1)| > 2k + 1$, so that every layout for G_1 has a vertex at distance more than k to a_1. Hence G cannot have a k-layout and rejection is correct. If $|N(a_1)| \leq 2k$ then any k-layout of G_1 is a normalised k-layout; for example the one that places a_1 in the middle. Assume now that $i \geq 2$ and that the induction hypothesis holds for all previous steps. Hence we know that G_{i-1} has a normalised k-layout β. By the way vertices a, b, and b_p are chosen by the algorithm, and by the arguments of Section 3, G_i has a normalised k-layout if and only if condition (1) of the algorithm is satisfied or both conditions (2) and (3) of the algorithm are satisfied. We use the algorithm described in the proof of Theorem 8 to decide condition (1). Note that a_i is adjacent to b in this case. We use the algorithm described in the proof of Theorem 9 to decide condition (2). By the correctness of these algorithms the theorem follows. ∎

Theorem 11. *There is a polynomial-time algorithm that computes the bandwidth of a bipartite permutation graph and outputs a corresponding layout.*

Proof Let $G = (A, B, E)$ be a connected bipartite permutation graph. As mentioned in the preliminaries, bipartite permutation graphs can be recognized in linear time. Also in linear time, a strong ordering (σ_A, σ_B) for G can be computed. The time consuming operations of `BPG-Bandwidth` are the algorithms of Theorems 8 and 9. Both algorithms involve running Algorithm `MoveRepair` once, and hence require $\mathcal{O}(kn)$ time each. Due to the second for-loop in `BPG-Bandwidth` the algorithm of Theorem 9 might be called at most k times at each iteration. Since Algorithm `BPG-Bandwidth` has $\mathcal{O}(n)$ iterations of the main for-loop (*), its running time for a given fixed k is $\mathcal{O}(k^2n^2)$.

A graph has bandwidth at most k if and only if all its connected components have bandwidth at most k. For a disconnected graph, a k-layout can be obtained from concatenating k-layouts for the connected components. Thus, it suffices to consider the problem of determining the bandwidth of a connected bipartite permutation graph. By binary search, we determine the smallest integer $k \in \{1, \ldots, n\}$ such that `BPG-Bandwidth` accepts input G, (σ_A, σ_B), and k. Due to Lemma 10, $k = bw(G)$. Since $k = \mathcal{O}(n)$, the total running time is $\mathcal{O}(n^4 \log n)$. ∎

6 Concluding remarks

An important open question is whether the bandwidth of permutation graphs can be computed in polynomial time. Another interesting open problem is to find a simple argument proving the lower bound for the bandwidth of a bipartite permutation graph. In other words, can our algorithm be modified so that it becomes certifying?

Acknowledgments

We would like to thank Fedor V. Fomin and Saket Saurabh for useful discussions on the importance of the bandwidth problem.

References

[1] S. F. Assmann, G. W. Peck, M. M. Sysło, and J. Zak. The bandwidth of caterpillars with hairs of length 1 and 2. *SIAM J. Algebraic and Discrete Methods* 2:387–393, 1981.

[2] M. Badoiu, K. Dhamdhere, A. Gupta, Y. Rabinovich, H. Räcke, R. Ravi, and A. Sidiropoulos. Approximation algorithms for low-distortion embeddings into low-dimensional spaces. *Proceedings of SODA 2005*, pages 119-128, ACM and SIAM, 2005.

[3] G. Blache, M. Karpinski, and J. Wirtgen. On approximation intractability of the bandwidth problem. Technical report, University of Bonn, 1997.

[4] A. Blum, G. Konjevod, R. Ravi, and S. Vempala. Semi-Definite Relaxations for Minimum Bandwidth and other Vertex-Ordering Problems. *Proceedings of STOC 1998*, pages 100-105, ACM, 1998.

[5] H. L. Bodlaender, M. R. Fellows, and M. T. Hallet. Beyond NP-completeness for problems of bounded width (extended abstract): hardness for the W hierarchy. *Proceedings of STOC 1994*, pages 449–458, ACM, 1994.

[6] H. L. Bodlaender, F. Fomin, D. Kratsch, A. Koster, and D.Thilikos. Exact algorithms for treewidth. *Proceedings of ESA 2006*, pages 672–683, Lecture Notes in Computer Science, vol. 4168, Springer, 2006.

[7] A. Brandstädt, V. B. Le, and J. Spinrad. *Graph Classes: A Survey*. SIAM, Philadelphia, 1999.

[8] J. Diaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356, 2002.

[9] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, New York, 1999.

[10] U. Feige. Approximating the Bandwidth via Volume Respecting Embeddings. *Proceedings of STOC 1998*, pages, 90–99, ACM, 1998.

[11] U. Feige. Coping with the NP-Hardness of the Graph Bandwidth Problem. *Proceedings of SWAT 2000*, pages 10–19, Lecture Notes in Computer Science, vol. 1851, Springer, 2000.

[12] U. Feige and K. Talwar. Approximating the Bandwidth of Caterpillars. *Proceedings of APPROX 2005*, pages 62–73, Lecture Notes in Computer Science, vol. 3624, Springer, 2005.

[13] P. Fishburn, P. Tanenbaum, and A. Trenk. Linear discrepancy and bandwidth. *Order*, 18:237–245, 2001.

[14] J. A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[15] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[16] A. Gupta. Improved bandwidth approximation for trees. *Proceedings of SODA 2000*, pages 788-793, ACM and SIAM, 2000.

[17] J. Haralambides, F. Makedon, and B. Monien. Bandwidth Minimization: An Approximation Algorithm for Caterpillars. *Mathematical Systems Theory* 24(3): 169–177, 1991.

[18] C. Kenyon, Y. Rabani, and A. Sinclair. Low distortion maps between point sets. *Proceedings of STOC 2004*, pages 272-280, ACM, 2004.

[19] D. J. Kleitman and R. V. Vohra. Computing the bandwidth of interval graphs. *SIAM J. Disc. Math.*, 3:373–375, 1990.

[20] T. Kloks, D. Kratsch, and H. Müller. Bandwidth of chain graphs. *Information Processing Letters* 68:313-315, 1998.

[21] T. Kloks, D. Kratsch, and H. Müller. Approximating the bandwidth for AT-free graphs. *Journal of Algorithms* 32:41–57, 1999.

[22] B. Monien. The bandwidth minimization problem with hair length 3 is NP-complete. *SIAM J. Alg. Disc. Meth.*, 7:505–512, 1986.

[23] C. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[24] J. Spinrad, A. Brandstädt, and L. Stewart. Bipartite permutation graphs. *Disc. Appl. Math.*, 18:279–292, 1987.

[25] W. T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory.* Hopkins University Press, Baltimore, 1992.

[26] W. Unger. The Complexity of the Approximation of the Bandwidth Problem. *Proceedings of FOCS 1998*, pages 82–91, IEEE, 1998.

[27] J. H. Yan. The bandwidth problem in cographs. *Tamsui Oxf. J. Math. Sci.*, 13:31–36, 1997.