



# Succinct Permutation Graphs

Konstantinos Tsakalidis<sup>1</sup> · Sebastian Wild<sup>1</sup> · Viktor Zamaraev<sup>1</sup>

Received: 1 September 2021 / Accepted: 10 September 2022 / Published online: 24 September 2022  
© The Author(s) 2022

## Abstract

We present a succinct data structure for permutation graphs, and their superclass of circular permutation graphs, i.e., data structures using optimal space up to lower order terms. Unlike concurrent work on circle graphs (Acan et al. in Theor Comput Sci, <https://doi.org/10.1016/j.tcs.2022.06.022>, 2022), our data structure also supports distance and shortest-path queries, as well as adjacency and neighborhood queries, all in optimal time. We present in particular the first succinct exact distance oracle for (circular) permutation graphs. A second succinct data structure also supports degree queries in time independent of the neighborhood's size at the expense of an  $O(\log n / \log \log n)$ -factor overhead in all running times. Furthermore, we develop a succinct data structure for the class of bipartite permutation graphs. We demonstrate how to run algorithms directly over our succinct representations for several problems on permutation graphs: CLIQUE, COLORING, INDEPENDENT SET, HAMILTONIAN CYCLE, ALL-PAIR SHORTEST PATHS, and others. Finally, we initiate the study of *semi-distributed* graph representations; a concept that smoothly interpolates between distributed (labeling schemes) and centralized (standard data structures). We show how to turn some of our data structures into semi-distributed representations by storing only  $O(n)$  bits of additional global information, circumventing the lower bound on distance labeling schemes for permutation graphs.

**Keywords** Succinct data structures · Distance oracles · Permutation graphs · Bipartite permutations graphs · Circular permutation graphs · Graph compression · Graph encoding

---

✉ Sebastian Wild  
Sebastian.Wild@liverpool.ac.uk

Konstantinos Tsakalidis  
K.Tsakalidis@liverpool.ac.uk

Viktor Zamaraev  
Viktor.Zamaraev@liverpool.ac.uk

<sup>1</sup> Department of Computer Science, University of Liverpool, Ashton Building, Ashton Street, Liverpool, Merseyside L69 3BX, UK

## 1 Introduction

As a result of the rapid growth of data sets, memory requirements become a bottleneck in many applications; in particular when data structures do no longer fit into faster levels of the memory hierarchy of computer systems. Research on *succinct data structures* has led to optimal-space data structures for many types of data [27], significantly extending the size of data sets that can be analyzed efficiently on commodity hardware. A data structure is called *succinct* when its space usage is optimal up to lower order terms, i. e., optimal up to a factor of  $1 + o(1)$ .

Graphs are one of the most widely used types of data. In this paper, we study succinct representations of specific classes of graphs, namely permutation graphs and related families of graphs. A graph is a *permutation graph* (PG) if it can be obtained as the intersection graph of chords (line segments) between two parallel lines [29], i. e., the vertices corresponding to two such chords are adjacent, if and only if the chords intersect. PGs are a well-studied class of graphs; they are precisely the comparability graphs of two-dimensional partial orders, and the class of comparability graphs whose complement graph is also a comparability graph [13] (see Sect. 2 for definitions of these concepts). Many generally intractable graph problems can be solved efficiently on PGs, for instance CLIQUE [22, 23], INDEPENDENT SET [22, 23], COLORING [22, 23], CLIQUE COVER [22, 23], DOMINATING SET [6], HAMILTONIAN CYCLE [11], and GRAPH ISOMORPHISM [8]. ALL-PAIR SHORTEST PATHS on PGs can be solved faster than in general graphs [4, 24]. Moreover, PGs can be recognized in linear time [22].

In this paper we study how to succinctly encode permutation graphs, while supporting the following queries efficiently:

- $\text{adj}(u, v)$ : whether vertices  $u$  and  $v$  are adjacent;
- $\text{deg}(v)$ : the degree of vertex  $v$ , i. e., the number of vertices adjacent to  $v$ ;
- $\text{nbrhood}(v)$ : the vertices adjacent to vertex  $v$ ;
- $\text{next\_nbr}(u, v)$ : the successor of vertex  $v$  in the adjacency list of vertex  $u$ ;
- $\text{spath}(u, v)$ : listing a shortest path from vertex  $u$  to vertex  $v$ ;
- $\text{spath\_succ}(u, v)$ : the first vertex after vertex  $u$  on a shortest path from  $u$  to vertex  $v$ ;
- $\text{dist}(u, v)$ : the length of the shortest path from vertex  $u$  to vertex  $v$ .

### Data structures

A succinct data structure is space optimal in the sense that it stores a given combinatorial object using asymptotically only the information-theoretic minimum of bits. Specifically, given a class of graphs  $\mathcal{C}$  and denoting by  $\mathcal{C}_n$  for the set of graphs  $G \in \mathcal{C}$  on  $|V(G)| = n$  vertices, a succinct data structure for  $\mathcal{C}$  is allowed to spend  $(1 + o(1)) \lg |\mathcal{C}_n|$  bits of space when representing a graph in  $\mathcal{C}_n$ . We present the first succinct data structures that support the above queries on a PG (Theorem 3.1), as well as on its generalization, the *circular permutation graphs* (CPGs, see Theorem 6.4).

Moreover, we present the first succinct data structure for the special case of a *bipartite permutation graph* (BPG, see Theorem 5.1). Table 1 summarizes these results.<sup>1</sup>

To our knowledge, the only centralized data structures that store PGs are presented by Gustedt et al. [18] and by Crespelle and Paul [9]. The former are not succinct (using  $O(n \lg n)$  words of space), but are parallelizable [18]. The latter support only `adj` queries (in constant time), but are dynamic (supporting insertions and deletions of vertices/chords and edges). We are not aware of previous work on data structures for CPGs, or on space-efficient data structures for BPGs.

Bazzaro and Gavaille [4] present *distance labeling schemes* for PGs, a distributed distance oracle, where the distance of two vertices can be computed solely from the two labels of the vertices. Their scheme uses labels of  $\sim 9 \lg n$  bits per vertex<sup>2</sup>, and their `dist` queries take constant time. By concatenating all labels, their labeling scheme implies a data structure with matching time complexity and total space of  $\sim 9n \lg n$  bits. Our data structures (Theorem 3.1) improve upon that space, while simultaneously supporting further queries besides `dist`.

Interestingly, Bazzaro and Gavaille [4] further give a *lower bound* of  $3 \lg n - O(\lg \lg n)$  bits per vertex for `dist` labeling schemes on PGs. Comparing our data structures to this lower bound reveals a separation in terms of total space between their distributed and our centralized model: giving up the distributed storage requirement, a data structure using the optimal  $\sim n \lg n$  bits of space, i.e.,  $\lg n$  per vertex, becomes possible, proving that the centralized model is strictly more powerful.

## Semi-distributed graph representations

To further explore the boundary of the above separation between standard centralized data structures and fully distributed labeling schemes, we introduce a *semi-distributed* model of computation for graph data structures that smoothly interpolates between these two extremes: in a  $\langle L(n), D(n) \rangle$ -space semi-distributed representation, each vertex locally stores a label of  $L(n)$  bits, but all vertices also have access to a “global” data structure of  $D(n)$  bits to support the queries. Such a representation uses a total of  $nL(n) + D(n)$  bits of space, but apart from the global part, only the labels of queried vertices are accessible to compute the answer.

The lower bound from [4] implies that when  $D(n) = 0$ , we must have  $L(n) \geq 3 \lg n - O(\lg \lg n)$  to support `dist` on PGs, making the total space at least a factor 3 worse than the information-theoretic lower bound. But what happens if we allow a small amount of global storage on top of the labels? Is access to global storage inherently more powerful, even if insufficient encode the entire PG? If so, what is the least amount of global storage that is necessary to overcome the labeling-scheme lower bound?

We do not comprehensively answer the latter question, but settle the former in the affirmative: we show that PGs admit a  $\langle 2 \lg n, O(n) \rangle$ -space semi-distributed rep-

<sup>1</sup> Throughout this paper, running times assume the word-RAM model with word size  $w = \Theta(\log n)$ , where  $n$  denotes the number of vertices of the input PG.

<sup>2</sup> By  $\sim$  we denote a leading-term asymptotic approximation, i.e.,  $f(n) \sim g(n)$  iff  $f(n)/g(n) \rightarrow 1$  as  $n \rightarrow \infty$ .

**Table 1** Our data structure results for (variants of) permutation graphs with  $n$  vertices

	Permutation graphs		Bipartite Permutation	Circular Permutation
	(a) By grid	(b) By array		
Space usage	$n \lg n + o(n \log n)$	$n \lg n + O(n)$	$2n + o(n)$	$n \lg n + O(n)$
Lower bound	$\sim n \lg n$ [4]	$\sim n \lg n$ [4]	$\sim 2n$ [32]	$\sim n \lg n$ [4]
adj	$\log n / \log \log n$	1	1	1
deg	$\log n / \log \log n$	$deg + 1$	1	$deg + 1$
nbrhood	$(deg + 1) \log n / \log \log n$	$deg + 1$	$deg + 1$	$deg + 1$
next_nbr	$(deg + 1) \log n / \log \log n$	1 (amortized)	1	1 (amortized)
spath	$(dist + 1) \log n / \log \log n$	$dist + 1$	$dist + 1$	$dist + 1$
spath_succ	$\log n / \log \log n$	1	1	1
dist	$\log n / \log \log n$	1	$dist + 1$	1
Theorem	Theorem 3.1(a)	Theorem 3.1(b)	Theorem 5.1	Theorem 6.4

Space is in bits. Query times are  $O(\cdot)$  bounds;  $deg$  denotes the queried vertex' degree and  $dist$  the shortest-path distance between the queried vertices

resentation that answers distance queries in constant time, i.e., although the global space cannot distinguish all possible PGs, it suffices to circumvent the lower bound for labeling schemes in terms of total space and label size. Thus having access even to limited amounts of global space is inherently more powerful than a fully distributed data structure.

## Applications

Our data structures can replace the standard (space-inefficient) representation by adjacency lists in graph algorithms. For several known algorithms on PGs that make explicit use of their special structure (namely, linear-time algorithms for computing minimum colorings, maximum cliques, maximum independent sets, or minimum clique covers), we show that they can be run with minimal extra space directly on top of our succinct representation.

Moreover, our data structures immediately yield an optimal-time all-pairs shortest-paths algorithm on PGs: For a PG with  $n$  vertices and  $m$  edges we can report all pairwise distances in  $O(n^2)$  time, matching the result of Mondal et al. [24]; however, our approach is more flexible in that we can report the distances of any  $k$  specified pairs of vertices in just  $O(n + m + k)$  total time. Furthermore, we can report the shortest paths (not just their lengths) in total time  $O(n + m + s)$ , where  $s$  is the size of the output; this does not immediately follow from [24]. The labeling scheme of [4] yields the same running times, but uses more space.

## Further related work

Similar to our work on PGs, succinct data structures that support the considered set of queries have been presented for *chordal graphs* [25] and *interval graphs* [3, 19]. The latter also consider the special class of unit/proper interval graphs and the generalization to circular interval graphs.

Concurrently<sup>3</sup> to this work, Acan et al. [2] presented succinct data structures for *circle graphs* (i.e., the intersection graph of chords of a (single) circle) and related classes (specifically  $k$ -polygon circle graphs and trapezoid graphs). They show space lower bounds for these classes and data structures with asymptotically matching space usage. Since a PG is also a circle graph, their data structures can be applied to PGs, but this is not known for CPGs. Superficially, their grid-based representation [2, Thm. 4.4] is similar to ours, but the construction uses a different point set with different properties for queries: Acan et al. support navigational operations `adj`, `deg`, and `nbrhood`, but none of their data structures offer `dist` or `spath`, which are a main technical challenge of our work. A further difference is that for general circle graphs, no succinct data structures with constant query time are known, whereas for PGs, we can use our array-based data structure, offering constant-time support for `adj`, `next_nbr`, `spath_succ`, `dist`.

<sup>3</sup> The preprint [1] (now published as [2]) appeared shortly after an initial version of this article [37] was published on arXiv.

## Outline

The rest of this paper is organized as follows. Section 2 collects previous results on PGs and succinct data structures. In Sect. 3, we describe our main result: the succinct data structures for PGs. Our other results extend the techniques established in that section. Section 4 describes how to simulate various algorithms on top of our succinct representation. Section 5 discusses our data structure for bipartite PGs, and Sect. 6 extends our approach to circular PGs. Finally, Sect. 7 introduces semi-distributed graph representations and our corresponding results. Section 8 concludes the paper.

## 2 Preliminaries

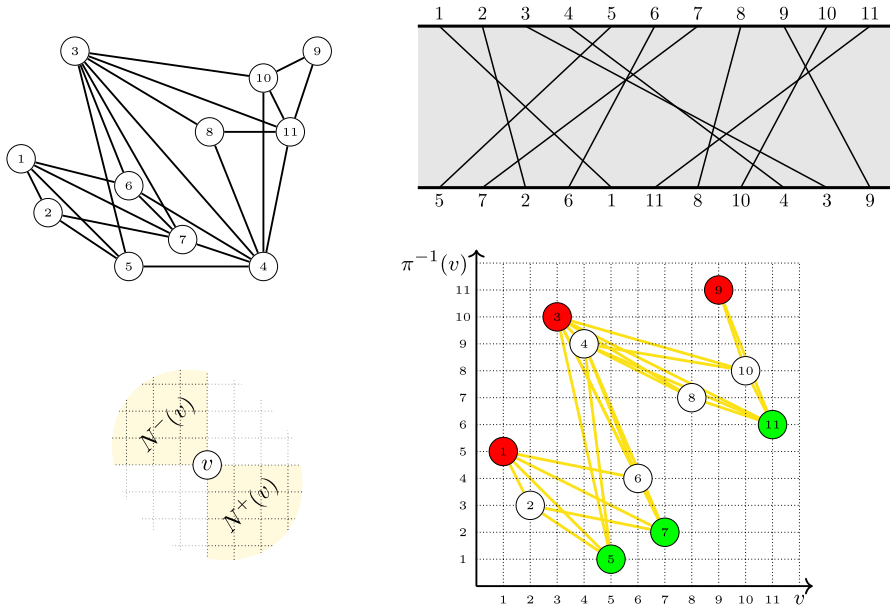
We write  $[n..m]$  for  $\{n, \dots, m\} \subset \mathbb{Z}$  and  $[n] = [1..n]$ . We use standard notation for graphs, in particular (unless stated otherwise)  $n$  denotes the number of vertices,  $m$  the number of edges.  $N(v)$  is the neighborhood of  $v$ , i.e., the set of vertices adjacent to  $v$ . In a directed graph  $G = (V, E)$ , we distinguish out-neighborhood  $N^+(v) = \{u : (v, u) \in E\}$  and in-neighborhood  $N^-(v) = \{u : (u, v) \in E\}$  of a vertex  $v \in V$ . The complement graph of  $G$  is denoted by  $\overline{G}$ . We use the “Iverson bracket” notation:  $[cond]$  is 1 if  $cond$  is true and 0 otherwise.

### 2.1 Permutation Graphs

It is easy to see from the intersection model of a PG  $G$  (as intersections of chords between parallel lines) that only the relative order of upper (resp. lower) endpoints of the chords are relevant (cf. Fig. 1). Hence, a graph  $G$  is a PG if there exists a permutation  $\pi$  and a bijection between the vertices of  $G$  and the elements of  $\pi$ , such that two vertices are adjacent if and only if the corresponding elements are reversed by  $\pi$ ; that explains the name.

To avoid confusion in counting results, we carefully distinguish three related notions for PGs. First, given a permutation  $\pi : [n] \rightarrow [n]$ , the *ordered PG* induced by  $\pi$ , denoted  $G_\pi = (V, E)$ , has vertices  $V = [n]$  and (undirected) edges  $\{i, j\} \in E$  for all  $i > j$  with  $\pi^{-1}(i) < \pi^{-1}(j)$ , i.e., if  $\pi$  has an *inversion*  $(i, j)$ . Given an ordered PG  $G$ , we can uniquely reconstruct the permutation  $\pi$  with  $G_\pi = G$ : By setting  $b_j$ , for each vertex  $j$ , to the number of its neighbors  $i$  with  $i > j$ , we obtain the inversion table  $b_1, \dots, b_n$  of the permutation, from which there is a well-known bijection to  $\pi$  itself [21, §5.1.1]. Hence, ordered PGs and permutations are in bijection. This yields a simple recognition algorithm for ordered PGs: Compute  $\pi$  as above and check if the given graph equals  $G_\pi$ .

The ordered PG  $G_\pi$  can be characterized by its *grid representation*, which is a collection of integer points in the plane associated with the vertices of  $G_\pi$ : a vertex  $v$  is associated with the unique point  $(v, \pi^{-1}(v))$  (see Fig. 1). A useful property of the grid representation is that the neighbors of the vertex  $v$  are exactly those vertices whose points are located in the top left or the lower right quadrant around the point of  $v$ .



**Fig. 1** Example permutation graph (top left) from [4] in different representations: a representation as intersections of chords between two parallels (top right), corresponding to the permutation  $\pi = (5, 7, 2, 6, 1, 11, 8, 10, 4, 3, 9)$ , and the points  $(v, \pi^{-1}(v))$  on a 2D grid (bottom right). A point in the grid can “see” (is adjacent to) all points in the top left resp. lower right quadrant around it as illustrated from the bottom left [4]

A graph  $G = ([n], E)$  is a *labeled PG*, written  $G \in \mathcal{P}^n$ , if there is set of  $n$  chords between two parallel lines *and* an assignment of vertices to chords, so that  $\{i, j\} \in E$  iff chords  $i$  and  $j$  intersect. In other words,  $G \in \mathcal{P}^n$  iff there are *two* permutations  $\pi : [n] \rightarrow [n]$  and  $\rho : [n] \rightarrow [n]$ , so that  $\rho(G) = ([n], \rho(E)) = G_\pi$ , where  $\rho(E) = \{\{\rho(u), \rho(v)\} : \{u, v\} \in E\}$ ; in short:  $G$  is a labeled PG iff it is isomorphic to some ordered PG  $G_\pi$ .

The set of *unlabeled PGs* of size  $n$ , denoted by  $\mathcal{P}_n$ , is the family of equivalence classes of labeled graphs in  $\mathcal{P}^n$  under (graph) isomorphism. To illustrate the notions of ordered, labeled, and unlabeled PGs, and to make the distinction between them clear, we consider a few simple examples. Both the empty or complete *unlabeled graph* correspond to a single ordered PG, namely with  $\pi$  the sorted (resp. reverse sorted) permutation. Similarly, there is only one labeled empty or complete graph; in this case, the three notions coincide. However, the unlabeled graph with just a single edge corresponds to  $n - 1$  ordered PGs, namely all  $n - 1$  permutations with a single inversion; and there are  $\binom{n}{2}$  labeled graphs with a single edge. We can always select a representative (a labeled PG) for an isomorphism class (the unlabeled PG) that is an ordered PG, but in general, there are more ordered PGs than unlabeled PGs.

A graph is *comparability* if its edges can be oriented such that if there are edge  $(a, b)$  and  $(b, c)$ , then there is an edge  $(a, c)$ . We will use the following classical characterization of PGs.

**Theorem 2.1** (PG & comparability, [13]) *A graph  $G$  is a PG if and only if both  $G$  and  $\overline{G}$  are comparability graphs.*

Finally, for the construction of our data structures, we will assume that an ordered PG  $G_\pi$  is given; the following result allows to compute such from a given PG in linear time.

**Theorem 2.2** (PG recognition, [22]) *There is an algorithm that given a graph  $G = (V, E)$  on  $n = |V|$  vertices and  $m = |E|$  edges computes in  $O(n + m)$  time two bijections  $\pi : [n] \rightarrow [n]$  and  $\rho : V \rightarrow [n]$  with  $\rho(G) = G_\pi$ , or determines that  $G$  is not a PG.*

## 2.2 Space Lower Bounds

Recall that  $\mathcal{P}_n$  denotes the set of unlabeled PGs on  $n$  vertices. We obtain information-theoretic lower bounds for storing an unlabeled PG from known counting results [4].

**Corollary 2.3**  $\lg |\mathcal{P}_n| \geq n \lg n - O(n \log \log n)$  *bits are necessary to represent an unlabeled permutation graph on  $n$  vertices.*

**Proof** Recall that we write  $\mathcal{P}^n$  for the set of labeled PGs on  $n$  vertices and  $\mathcal{P}_n$  for the set of unlabeled PGs on  $n$  vertices. [4, Thm. 5.2] shows that  $\lg |\mathcal{P}^n| \geq 2n \lg n - O(n \log \log n)$ . Clearly  $|\mathcal{P}^n| \leq n! |\mathcal{P}_n|$  since there are at most  $n!$  ways of assigning labels  $[n]$ . Using the Stirling approximation,  $\lg(n!) = n \lg n - O(n)$ , we obtain that  $\lg |\mathcal{P}_n| \geq 2n \lg n - O(n \log \log n) - \lg(n!) \geq n \lg n - O(n \log \log n)$ .  $\square$

Up to lower order terms, this lower bound coincides with  $\lg(n!)$ , so succinctly storing a given grid representation of an ordered PG in our data structures suffices for a succinct PG data structure.

Generalizing a construction from Acan et al. [2], we can strengthen the above lower bound.

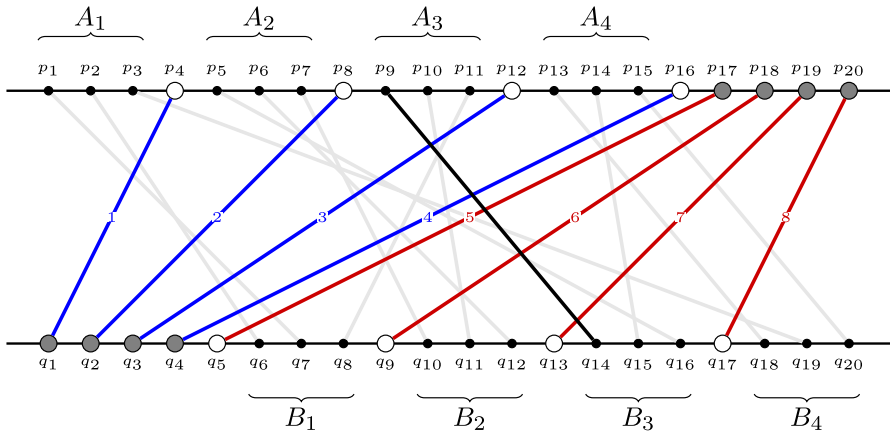
**Theorem 2.4** (Space lower bound)  $\lg |\mathcal{P}_n| \geq n \lg n - O(n)$  *bits are necessary to represent an unlabeled permutation graph on  $n$  vertices.*

*Proof Sketch* We build on the proof of Thm. 4.2 of [2]; we reproduce the parts that need amendment here.

We construct a specific family of vertex-colored PGs that is large enough so that – even after discounting the overcounting due to counting *colored* graphs – it corresponds to  $2^{n \lg n - O(n)}$  distinct unlabeled PGs, yielding the claim. We represent the colored graphs via their (colored) permutation diagram. We begin with two parallel lines and place  $n$  “chord slots” (points) on each line; we will later connect these to  $n$  disjoint chords. Let  $p_1, \dots, p_n$  resp.  $q_1, \dots, q_n$  denote these points on the upper resp. lower line, numbered from left to right; cf. Fig. 2.

As in [2], we fix parameters  $k$  and  $\ell$  so that  $k\ell + 2k = n$ . Now fix  $2k$  special cords as follows: The first  $k$  special cords connect  $q_1, \dots, q_k$  to the points  $p_{\ell+1}, p_{2(\ell+1)}, p_{3(\ell+1)}, \dots, p_{k(\ell+1)}$ , the second  $k$  special cords connect





**Fig. 2** The colored PG construction from Theorem 2.4 for  $\ell = 3$  and  $k = 4$ , and hence  $n = k\ell + 2k = 20$ . Special chords are shown in blue and red. The highlighted chord  $(p_9, q_{14})$  intersects the special chords  $[i..j] = \{3, 4, 5, 6, 7\}$  and has endpoints in  $A_i = A_3$  and  $B_{j-k} = B_{7-4} = B_3$  (Color figure online)

$p_{n-k+1}, \dots, p_n$  with  $q_{k+1}, q_{k+1+(\ell+1)}, q_{k+1+2(\ell+1)}, \dots, q_n$ . Each of the  $2k$  special chords is colored using a unique color in  $[2k]$ , assigned from left to right; all other chords (added below) have color 0.

We have so far used  $4k$  of the chord slots; the remaining  $2n - 4k$  slots are partitioned by the special chords into  $2k$  intervals of  $\ell$  chord slots each:  $k$  on the upper line,  $k$  on the lower line, each separated by an endpoint of a special chord. We name these intervals  $A_1, \dots, A_k$  on the upper line and  $B_1, \dots, B_k$  on the lower line (see Fig. 2).

We now consider *matchings* of the remaining  $k\ell$  slots on the upper line with the remaining  $k\ell$  slots on the lower line. Each such matching corresponds to one way of adding the remaining  $k\ell$  chords; Fig. 2 shows an example (gray lines). In general, different matchings can correspond to the same unlabeled colored graph, but we will see that this can only happen for *bad matchings* [2]: a matching is *bad* if it contains 3 or more of chords connecting the same  $A_i$  with the same  $B_j$ ; otherwise it is *good*. A good matching can be uniquely reconstructed from its induced colored PG: First, each colored vertex is unique and its color uniquely determines which special chord it corresponds to. Next, each 0-colored vertex must be adjacent to special chords with colors from a contiguous range  $[i..j] \subset [2k]$ ; its upper endpoint then lies on  $A_i$  and its lower endpoint on  $B_{j-k}$ . Hence we can uniquely reconstruct the intervals each chord’s endpoints belong to. Finally, if two chords  $u, v$  both end in the same  $A_i$ , their relative order is determined by whether or not they are adjacent. Since the matching is good, there is at most one such pair  $u, v$  where the relative order of endpoints on the bottom line is not already determined, so we can work out a total order of the endpoints within  $A_i$  from the colors and adjacencies. The argument for two chords ending in the same  $B_j$  is similar.

Using Lem. 4.1 of [2], which shows that for  $k = n^{3/4+\epsilon}$ ,  $\epsilon > 0$  fixed, a  $1 - o(1)$  fraction of all possible matchings is good, we can now finish the proof as in [2]:

$$\begin{aligned}
|\mathcal{P}_n| \binom{n}{2k} (2k)! &\geq \# \text{PGs with } 2k \text{ vertices assigned unique colors in } [2k] \\
&\geq \# \text{colored PGs obtained from above construction} \\
&\geq \# \text{good matchings} \\
&= (1 - o(1))(k\ell)! = (1 - o(1))(n - 2k)!;
\end{aligned}$$

hence we have, denoting by  $n^k = \prod_{i=0}^{k-1} (n - i)$  the  $k$ th falling power of  $n$ , that

$$\begin{aligned}
\lg |\mathcal{P}_n| &\geq \lg((n - 2k)!) + \lg(1 - o(1)) - \lg(n^{2k}) \\
&\geq \lg((n - 2k)!) - \lg(n^{2k}) - O(1) \\
&= ((n - 2k) \lg(n(1 - 2k/n)) - (n - 2k) \lg(e) + \Theta(\log n)) - 2k \lg(n) - o(1) \\
&\geq n \lg n - \lg(e)n - O(k \log n).
\end{aligned}$$

This concludes the proof.  $\square$

**Remark 2.5** We note that in the data-structures and graph-labeling-schemes communities, the above approach for proving space optimality of graph representations via lower bounds on the number of unlabeled graphs in the class is quite typical [3, 4, 16, 26]: One establishes a lower bound on the number  $|\mathcal{X}_n|$  of unlabeled graphs in a given class  $\mathcal{X}$  by first deriving a lower bound on the number  $|\mathcal{X}^n|$  of labeled (or colored) graphs in the class, and then applying the obvious relation  $|\mathcal{X}^n| \leq n! |\mathcal{X}_n|$  (or a similar one for partially colored graphs). The non-trivial part in this approach is the former one, and it usually boils down to an ad-hoc construction of a large family of labeled graphs. For leading-term estimates, a recent work of Sauermaun [33] provides a uniform framework for deriving tight lower bounds on the number of labeled graphs in any *semi-algebraic* graph class. The family of semi-algebraic graph classes contains many geometric intersection graph classes, including interval graphs and PGs.

### 2.3 Succinct Data Structures

For the reader's convenience, we collect used results on succinct data structures here. First, we cite the compressed bit vectors of Pătraşcu [28].

**Lemma 2.6** (Compressed bit vector) *Let  $B[1..n]$  be a bit vector of length  $n$ , containing  $m$  1-bits. For any constant  $c > 0$ , there is a data structure using  $\lg \binom{n}{m} + O\left(\frac{n}{\log^c n}\right) \leq m \lg \left(\frac{n}{m}\right) + O\left(\frac{n}{\log^c n} + m\right)$  bits of space that supports in  $O(1)$  time operations (for  $i \in [1, n]$ ):*

1.  $\text{access}(B, i)$ : return  $B[i]$ , the bit at index  $i$  in  $B$ .
2.  $\text{rank}_\alpha(B, i)$ : return the number of bits with value  $\alpha \in \{0, 1\}$  in  $B[1..i]$ .
3.  $\text{select}_\alpha(B, i)$ : return the index of the  $i$ -th bit with value  $\alpha \in \{0, 1\}$ .

**Remark 2.7** (Simpler bitvectors) The result of Pătraşcu has the best theoretical guarantees, but requires rather complicated data structures. Compressed bitvectors with

space

$$\begin{aligned} \lg \binom{n}{m} + O\left(\frac{n \log \log n}{\log n}\right) &\leq nH\left(\frac{m}{n}\right) + O\left(\frac{n \log \log n}{\log n}\right) \\ &= m \lg\left(\frac{n}{m}\right) + O\left(\frac{n \log \log n}{\log n} + m\right) \end{aligned}$$

have been proposed by Raman et al. [30] and implemented [17].

For our application, indeed a plain (uncompressed) bitvector with  $O(1)$ -time support for rank and select and using  $n + O(n/\log \log n)$  bits of space is sufficient (see, e.g., [27, §4.2.2 & §4.3.3], originally proposed in [7, 20]).

Using wavelet trees, based on above bitvectors, we can also handle non-binary arrays.

**Lemma 2.8** (Wavelet trees for constant  $\sigma$ ) *Let  $S[1..n]$  be a static array with entries  $S[i] \in \Sigma = [1..\sigma]$  for  $\sigma$  a fixed constant. There is a data structure using  $\lg(\sigma)n + o(n)$  bits of space that supports the following queries in  $O(\log \sigma) = O(1)$  time (without access to  $S$  at query time)*

1.  $\text{access}(S, i)$ : return  $S[i]$ , the symbol at index  $i$  in  $S$ .
2.  $\text{rank}_\alpha(S, i)$ : return the number of indices with value  $\alpha \in \Sigma$  in  $S[1..i]$ .
3.  $\text{select}_\alpha(S, i)$ : return the index of the  $i$ -th occurrence of value  $\alpha \in \Sigma$  in  $S$ .

**Proof** Wavelet trees [27, §6.2] support these operations in the stated time. For the case of a small fixed  $\sigma$  that we need, we can use a separate compressed bitvector Lemma (2.6) for each of the  $O(\sigma)$  nodes in the wavelet tree. By the aggregation property of the entropy, the overall space is bounded by  $nH_0 + o(\sigma n) \leq n \lg(\sigma) + o(n)$ , where  $H_0$  is the (zeroth-order) empirical entropy of  $S$  (see, e.g., [27, §6.2.4]).  $\square$

Given an array  $A[1..n]$  of comparable elements, (e.g., numbers), range-minimum queries (resp. range-maximum queries) are defined for  $1 \leq i \leq j \leq n$  by

$$\begin{aligned} \text{rm}_{\text{QA}}(i, j) &= \arg \min_{i \leq k \leq j} A[k], \\ \text{rM}_{\text{QA}}(i, j) &= \arg \max_{i \leq k \leq j} A[k]. \end{aligned}$$

In both cases, ties are broken by the index, i.e., the index of the leftmost minimum (resp. maximum) is returned.

**Lemma 2.9** (RMQ index, [14, Thm. 3.7]) *For any constant  $\epsilon > 0$  the following holds. Given a static array  $A[1..n]$  of comparable elements, there is a data structure using  $\epsilon n$  bits of space on top of  $A$  that answers range-minimum queries in  $O(1/\epsilon)$  time (making as many queries to  $A$ ).*

Clearly, the same data structure can also be used to answer range-maximum queries by building the data structure w.r.t. the reverse ordering.

**Remark 2.10** (Sublinear RMQ) Indeed,  $\epsilon$  can be chosen smaller than constant, yielding sublinear extra space, at the cost of increasing the query time to superconstant; we only need  $\epsilon = \Omega(n^{-1+\delta})$  for some  $\delta > 0$ .

Given a static set of points in the plane, *orthogonal range reporting* asks to find all points in the point set that lie inside a query rectangle  $[x_1, x_2] \times [y_1, y_2]$ . Range counting queries only report the number of such points.

**Lemma 2.11** (Succinct point grids, [5, Thm. 1]) *A set  $N$  of  $n$  points in an  $n \times n$  integer grid can be represented using  $n \lg n + o(n \log n)$  bits of space so that*

1. *orthogonal-range-counting queries are answered in  $O(\log n / \log \log n)$  time, and*
2. *orthogonal-range-reporting queries are answered in  $O((k + 1) \log n / \log \log n)$  time, where  $k$  is the output size.*

**Lemma 2.12** (Permutation grid) *Given a permutation  $\pi : [n] \rightarrow [n]$ , we can represent the point set  $P = P(\pi) = \{(x, \pi(x)) : x \in [n]\}$  using  $n \lg n + o(n \log n)$  bits of space so that we can answer the following queries:*

1. *orthogonal-range-counting queries,  $\text{RCount}_P(x_1, x_2; y_1, y_2) = |P \cap [x_1, x_2] \times [y_1, y_2]|$  in  $O(\log n / \log \log n)$  time;*
2. *orthogonal-range-reporting queries,  $\text{RPoints}_P(x_1, x_2; y_1, y_2) = P \cap [x_1, x_2] \times [y_1, y_2]$  in  $O((k + 1) \log n / \log \log n)$  time, where  $k = \text{RCount}_P(x_1, x_2; y_1, y_2)$ ;*
3. *application of  $\pi$ ,  $\text{YForX}_P(\pi)(x) = \pi(x)$  for  $x \in [n]$  in  $O(\log n / \log \log n)$  time;*
4. *inverse of  $\pi$ ,  $\text{XForY}_{P(\pi)}(y) = \pi^{-1}(y)$  for  $y \in [n]$  in  $O(\log n / \log \log n)$  time.*

**Proof** We use the grid data structure from Lemma 2.11 on  $P$ ; counting and reporting queries are immediate, and for others we use that  $\text{YForX}_{P(\pi)}(x) = \text{RPoints}_P(x, x; 1, n).y$  and  $\text{XForY}_{P(\pi)}(y) = \text{RPoints}_P(1, n; y, y).x$ . Here we write  $Q.x$  to denote the projection of point set  $Q$  to the  $x$ -coordinates of the points.  $\square$

**Remark 2.13** (Iterate over range) It is not clear if we can iterate over the result of  $\text{RPoints}$  with  $O(\log n / \log \log n)$  time per point instead of obtaining all points in one go.

**Remark 2.14** (Simpler alternatives) At the slight expense of increasing running times by a  $O(\log \log n)$  factor, we can replace the grid data structure by a wavelet tree, which is likely to be favorable for an implementation [3, 27].

A last ingredient for our data structures is a recent result on succinct distance oracles for proper interval graphs. Here, an interval graph is the intersection graph of a set of intervals on the real line, and a proper interval graph is one that has an interval realization where no interval strictly contains another one.

**Lemma 2.15** (Succinct proper interval graphs [19, Thm. 12]) *A proper interval graph on  $n$  vertices can be represented in  $3n + o(n)$  bits of space so that  $\text{dist}(u, v)$  for  $u, v \in [n]$  can be computed in  $O(1)$  time, and vertices are identified by the rank of the left endpoints of their interval in some realization of the proper interval graph. We can also answer  $\text{adj}$ ,  $\text{deg}$ ,  $\text{nbrhood}$  in  $O(1)$  time and  $\text{spath}(u, v)$  in  $O(\text{dist}(u, v))$  time. For connected graphs, the space can be reduced to  $2n + o(n)$  bits.*

**Remark 2.16** (*O(1) time neighborhood*) It might sound impossible to do `neighborhood` in constant time independent of the output size; this is possible in proper interval graphs since neighborhoods are contiguous intervals (of vertex labels) and thus can be encoded implicitly in a constant number of words.

**Remark 2.17** (*Routing*) By inspection of the proof, the data structure from [19] can also support `spath_succ(u, v)` in constant time. Thus, not just can `spath(u, v)` be answered in optimal overall time, but we can output the path step by step in optimal time per edge.

### 3 Data Structures for Permutation Graphs

In this section, we assume a permutation  $\pi : [n] \rightarrow [n]$  is given and we describe how to answer queries on  $G_\pi$ , i.e., we describe our data structures for *ordered* PGs. We present two approaches: the first solution uses a grid data structure that can support all queries, albeit with superconstant running time; the second solution stores  $\pi$  as an array and achieves optimal query times for all operations except `deg`. Our formal result is as follows.

**Theorem 3.1** (Succinct PG) *A permutation graph can be represented*

- (a) *using  $n \lg n + o(n \lg n)$  bits of space while supporting `adj`, `deg`, `dist`, `spath_succ` in  $O(\log n / \log \log n)$  time, `neighborhood(v)` in  $O((\deg(v) + 1) \cdot \log n / \log \log n)$  time, and `spath(u, v)` in  $O((\text{dist}(u, v) + 1) \log n / \log \log n)$  time; or*
- (b) *using  $n \lg n + (6.17 + \epsilon)n + o(n)$  bits of space (for any constant  $\epsilon > 0$ ) while supporting `adj`, `dist`, `spath_succ`, `next_nbr` in  $O(1)$  time, `neighborhood(v)`, `deg(v)` in  $O(\deg(v) + 1)$  time, and `spath(u, v)` in  $O(\text{dist}(u, v) + 1)$  time. The time for `next_nbr(v)` is amortized  $O(1)$  over iterating through `neighborhood(v)`.*

#### 3.1 Grid-Based Data Structure

We first present the simpler grid-based data structure. Here, we store  $P(\pi) = \{(v, \pi^{-1}(v)) : v \in [n]\}$  in the data structure of Lemma 2.12 and identify vertices with the  $x$ -coordinates of these points (the rank of the vertex' chord endpoint on the upper line).

##### 3.1.1 Adjacent

Given two vertices  $u$  and  $v$ , w.l.o.g.  $u < v$ . We compute  $\pi^{-1}(u) = \text{YForX}(u)$  and  $\pi^{-1}(v) = \text{YForX}(v)$ ; then `adj(u, v) =  $[\pi^{-1}(u) > \pi^{-1}(v)]$` .

##### 3.1.2 Neighborhood

We separate the neighbors of a vertex  $v$  into `neighborhood(v) =  $N^-(v) \cup N^+(v)$`  where  $N^-(v) = \text{neighborhood}(v) \cap [1..v - 1]$  and  $N^+(v) = \text{neighborhood}(v) \cap [v + 1..n]$ .

Using the graphical representation of neighborhoods from Fig. 1, we immediately obtain  $N^-(v) = \text{RPoints}(1, v - 1; \text{YForX}(v), n)$  and  $N^+(v) = \text{RPoints}(v + 1, n; 1, \text{YForX}(v))$ .

### 3.1.3 Degree

Replacing the range-reporting queries from `nbrhood` by range-counting queries yields  $\text{deg}(v) = |N^-(v)| + |N^+(v)|$ .

## 3.2 Array-Based Data Structure

To improve the query time, we now give an alternative representation. A key observation is that we never compute  $\pi$ ; only  $\pi^{-1}$  is needed. Hence we simply store an array  $\Pi[1..n]$  with  $\Pi[i] = \pi^{-1}(i)$  using  $n \lceil \lg n \rceil \leq n \lg n + n$  bits of space. At the expense of a slightly more complicated data structure, one can improve this space usage to  $\lceil n \lg n \rceil = n \lg n + O(1)$  using the techniques of Dodis et al. [12], still retaining access to  $\Pi$  in constant time.

For legibility, we continue to write  $\pi^{-1}(i)$  in operations, but it is understood that this is indeed an access to  $\Pi[i]$ .

### 3.2.1 Adjacency

`adj` queries only use  $\pi^{-1}$ , and thus they are solved exactly as above.

### 3.2.2 Neighborhood

Like in our previous approach, we separately handle the neighbors  $u$  of  $v$  with  $u < v$  (in  $N^-(v)$ ) and with  $u > v$  (in  $N^+(v)$ ). Even though we do not explicitly store the point set  $P(\pi)$  in our data structure, we can still answer the above range queries, because these are effectively two-sided range queries (dominance queries):

For  $N^-(v) = \text{RPoints}(1, v - 1; \pi^{-1}(v), n)$ , we maintain the range-maximum index from Lemma 2.9 on  $\Pi[1..n]$  using  $\epsilon n$  bits of space. We can then iterate through the vertices in  $N^-(v)$  using the standard algorithm for three-sided orthogonal range reporting that uses priority search trees: We compute  $i = \text{rMQ}_{\Pi}(1, v - 1)$ ; if  $\pi^{-1}(i) \geq \pi^{-1}(v)$ , we report  $i$  as a neighbor and recursively continue in the ranges  $[1..i - 1]$  and  $[i + 1..v - 1]$ . Otherwise, if  $\pi^{-1}(i) < \pi^{-1}(v)$ , we terminate the recursion. (We also terminate recursive calls on empty ranges). Each recursive call only takes constant time and either terminates or outputs a new neighbor of  $v$ , so we can iterate through  $N^-(v)$  with constant amortized time per vertex.

For  $N^+(v) = \text{RPoints}(v + 1, n; 1, \pi^{-1}(v))$ , we use the same technique, reflected: we store a range-minimum index on  $\Pi[1..n]$ , starting with the range  $[v + 1, n]$  and continue as long as the returned minimum is at most  $\pi^{-1}(v)$ .

### 3.2.3 Next neighbor

The above method can easily be used to *iterate* over neighbors one at a time, instead of generating and returning the full neighborhood. The order of iteration is implementation-defined (ultimately by the RMQ index), but fixed for any  $G_\pi$ . An easy argument shows that reporting the  $k$ th neighbor with the above algorithm can take  $\Theta(k)$  time, but amortized over the entire neighborhood of a vertex, iteration takes constant time per neighbor. However, if done naively, it would require  $O(k)$  extra working space to store the  $k$  ranges wherein the  $k$ th neighbor might be found.

We can improve the extra space to  $O(1)$  (words) and support starting at an arbitrary given neighbor  $w$  to find `next_nbr(v, w)` in the traversal. For that, we have to look into the black box that is the RMQ index from Lemma 2.9. Indeed, what we describe here is modification of the construction of Fischer and Heun [14, Thm. 3.7] that has the same asymptotic performance characteristics as in Lemma 2.9, but allows to iterate over values above a threshold.

**Lemma 3.2** (RMQ index with next-above) *Let  $A[1..n]$  be a static array of comparable elements. For any constant  $\epsilon > 0$ , there is a data structure using  $\epsilon n$  bits of space on top of  $A$  that supports the following queries in  $O(1/\epsilon)$  time (making as many queries to  $A$ ) and using  $O(1)$  words of working memory:*

- (a) *range-maximum queries,  $\text{rMq}_A(\ell, r)$ ,*
- (b) *next-above queries,  $\text{next\_above}_A(\ell, r, y; i)$ , enumerating  $\{i \in [\ell, r] : A[i] \geq y\}$  in amortized  $O(1/\epsilon)$  time.*

*Formally, `next_above` implicitly defines a sequence  $(i_j)_{j \geq 0}$  via  $i_0 = \text{rMq}_A(\ell, r)$  if  $A[i_0] \geq y$  and  $i_0 = \text{null}$  otherwise, and  $i_{j+1} = \text{next\_above}_A(\ell, r, y; i_j)$  if  $i_j \neq \text{null}$  and  $i_{j+1} = \text{null}$  otherwise. Then we require  $\{i_j : i_j \neq \text{null}\} = \{i \in [\ell, r] : A[i] \geq y\}$ .*

This index can be used to iterate over the result of 3-sided orthogonal range queries with amortized constant delay and using constant working memory by computing the sequence  $(i_j)$ .

**Proof** A  $2\epsilon n + o(n)$  bit RMQ index for an array  $A[1..n]$  can be obtained by (conceptually) dividing  $A$  into  $\epsilon n$  blocks of  $\lceil 1/\epsilon \rceil$  elements each and storing the Cartesian tree [15, 38] of the block maxima as a succinct binary tree [10, Thm. 3] in  $2\epsilon n + o(n)$  bits. This tree data structure allows in constant time to (a) map between nodes and their corresponding block indices in  $A$ , (b) map between nodes and preorder indices, (c) find the lowest common ancestor (LCA) of two nodes, and (d) return the number of descendants of a node. We first discuss how to solve the problem for  $\epsilon = 1$ , i.e., when all elements are part of the tree. We will identify nodes in the Cartesian tree  $T$  with their inorder number, i.e., the index in  $A$ . To answer  $\text{rMq}_A(\ell, r)$ , we simply use the Cartesian tree operations to find the nodes (of inorder index)  $\ell$  and  $r$  and return (the inorder index of) their LCA.

To iterate through all indices  $i \in [\ell, r]$  with  $A[i] \geq y$ , we will now show how to compute the next such index,  $\text{next\_above}_A(\ell, r, y; i)$ , given only a current such index  $i$  (and  $\ell, r$  and  $y$ ); if no further such index exists, we will return “null”.

First, we compute  $i_0 = \text{rMq}_A(\ell, r)$ . We will iterate through indices in the order of a *preorder* traversal of the subtree rooted at  $i_0$ , starting from the current node  $i$ . The

challenge is to, in constant time, skip over parts of the tree that are outside of the range  $[\ell, r]$  or have all  $A$ -values below  $y$ . More specifically, the first step is to find the next candidate index  $s \in [\ell, r]$ , for which  $A[s] \geq y$  might hold, given the current index  $i$ . We initialize  $s$  to the successor of  $i$  in preorder.

Now, we repeat the following steps until we have either found the next index or have determined that none exists. If  $s$  is *not* a descendant of  $i_0$  in  $T$ , then there are no more indices to report and we return null; we can check this condition in constant time by comparing the preorder index of  $s$  to the sum of the preorder index of  $i_0$  and  $i_0$ 's subtree size.

If  $s$  is within  $i_0$ 's subtree, we check whether  $s \in [\ell, r]$ ; if not,  $s$  is too far left or too far right, and we have to find the next node (in preorder) that lies inside  $[\ell, r]$ . If  $s < \ell$  and  $i > \ell$ , then  $s$  is the left child of  $i$ , and following right-child links from  $s$  eventually brings us back into the range  $[\ell, r]$  since node  $i - 1 \in [\ell, r]$  must lie in  $s$ 's right subtree. In this case, we update  $s$  to the LCA of  $\ell$  and  $i - 1$  to obtain, in  $O(1)$  time, the first node (in preorder) where this sequence of right-child links from  $s$  enters the inorder range  $[\ell, r]$  again. If  $s < \ell$  and  $i = \ell$ ,  $i$  is the leftmost node in the range and we have to skip its left subtree. We can do this by advancing from  $s$  (in preorder) by as many nodes as  $s$ 's right child has descendants; the tree data structure again supports this in constant time. The symmetric case of  $s > r$  is handled similarly. If  $i < r$ , we set  $s$  to LCA of  $i + 1$  and  $r$ ; if  $s > r$  and  $i = r$ ,  $i$  was the last node in preorder with inorder index in range  $[\ell, r]$ , so we can return null.

In all cases, after  $O(1)$  time, we either terminate or arrive at the next candidate node  $s$ . If  $A[s] \geq y$  we return  $s$  and are done. Otherwise, i.e., if  $A[s] < y$ , then  $s$  and its entire subtree have to be skipped; the tree data structure supports this in constant time (as above). Then we repeat the above steps with the new  $s$ .

We note that the accesses to  $A$  are the same as in the naive implementation of three-sided range reporting, and only constant time is needed between two such accesses; hence the same time bounds hold.

When we use blocks of  $c = \lceil 1/\epsilon \rceil$  elements and only construct  $T$  based on the block minima, we modify this procedure as follows. When we are given a current index  $i$ , we first check the indices  $j > i$  in  $i$ 's block. If any  $j$  has  $A[j] \geq y$ , we return it. Only if none of the indices in  $i$ 's block are returned, we continue with the above procedure to find the next candidate node  $s$ . When we compare the candidate node " $A[s] \geq y$ ", we now iterate through the block corresponding to node  $s$  and compare each array entry with  $y$ . When we find  $i$  with  $A[i] \geq y$ , we return this index; if none of the elements in the block were big enough, we continue as if  $A[s] < y$  held.  $\square$

From the discussion above, it is clear that `next_nbr` corresponds exactly to `next_above` queries (separately for  $N^+$  and  $N^-$ ), and so using Lemma 3.2, we can support `next_nbr`( $u, w$ ) with constant words of extra working memory and amortized constant running time (amortized over the iteration over all neighbors of  $u$ ).

**Remark 3.3** (*Easy degrees*) We can compute  $\deg(v)$  as  $|\text{nbrhood}(v)|$  in  $O(\deg(v) + 1)$  time, but this is not particularly efficient for high-degree vertices. We can obviously also add support for  $\deg$  in constant time by storing the degrees of all vertices explicitly in an array. This occupies an additional  $n \lceil \lg n \rceil$  bits of space and is thus not succinct,



but might in implementations be preferable to the grid data structure (and offers all queries in optimal time complexity).

### 3.3 Distance and Shortest Paths

Both of the above data structures can be augmented to support distance and shortest-path queries; the only difference will be the running time to compute  $\pi^{-1}(v)$ .

For that, we follow the idea of [4]; we sketch their approach here and give a more formal definition below. A shortest path from  $u$  to  $v$  in a PG can always be found using only *left-to-right maxima* (“type  $A$ ” vertices) and *right-to-left minima* (“type  $B$ ” vertices) as intermediate vertices; moreover, these are strictly alternating. Hence, after removing an initial segment of at most 2 edges on either end of the path, such a shortest path has either type  $A(BA)^*A$  or  $B(AB)^*B$ . For example, a shortest path from vertex 15 to vertex 25 in Fig. 3 is 15–14–23–22–25. Finally, how many intermediate  $B$ -vertices are needed to move from one  $A$  vertex to another is captured by a *proper interval graph*  $G_A$ , and likewise for  $B$ -vertices in  $G_B$ . We can hence reduce the shortest-path queries to proper interval graphs and use Lemma 2.15. We present the details below.

### 3.4 Distance

A vertex  $v \in [n]$  is a *type- $A$  vertex* iff  $\pi^{-1}$  has a left-to-right maximum at position  $v$ , i.e., when  $\pi^{-1}(v) \geq \pi^{-1}(u)$  for all  $u < v$ . Note that 1 is always a left-to-right maximum. Similarly, a vertex  $v \in [n]$  is *type  $B$*  iff  $\pi^{-1}$  has a right-to-left minimum at  $v$ , i.e.,  $\pi^{-1}(v) \leq \pi^{-1}(u)$  for all  $u > v$ ; vertex  $n$  is always type  $B$ . As in [4], we use  $A$  and  $B$  to denote the set of  $A$ -vertices and  $B$ -vertices, respectively, and we define:

$$\begin{aligned} a^-(v) &= \min(\text{nbrhood}(v) \cap A), \\ a^+(v) &= \max(\text{nbrhood}(v) \cap A), \\ b^-(v) &= \min(\text{nbrhood}(v) \cap B), \\ b^+(v) &= \max(\text{nbrhood}(v) \cap B). \end{aligned}$$

If we are computing a shortest path from  $u$  to  $v$ , then either  $u$  and  $v$  are adjacent, or there is a shortest path whose first vertex after  $u$  is one of  $a^+(u)$  and  $b^+(u)$ , if  $v > u$ , or one of  $a^-(u)$  and  $b^-(u)$ , if  $v < u$ . It is therefore vital to be able to compute these four functions. For that, we store four bitvectors with rank/select support Lemma (2.6) that encode which points belong to  $A$  (resp.  $B$ ) given an  $x$ - (resp.  $y$ -)coordinate:

$$\begin{aligned} A_x[1..n] & \text{ with } A_x[u] = [u \in A], \\ B_x[1..n] & \text{ with } B_x[u] = [u \in B], \\ A_y[1..n] & \text{ with } A_y[i] = [\pi(i) \in A], \\ B_y[1..n] & \text{ with } A_y[i] = [\pi(i) \in B]. \end{aligned}$$

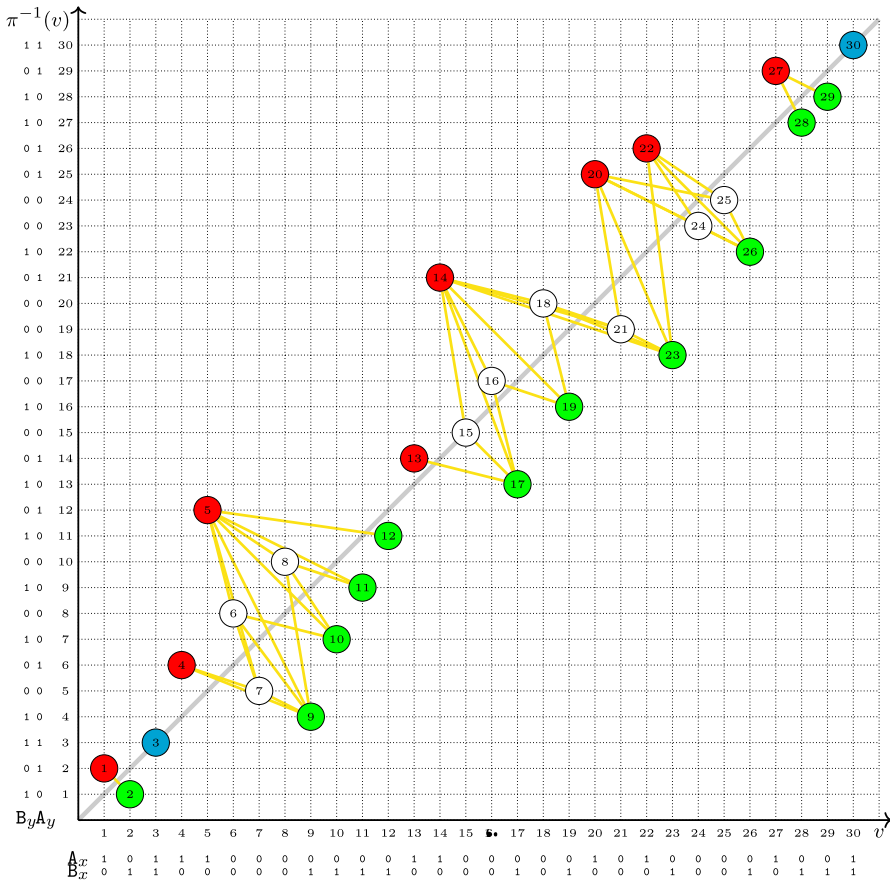


Fig. 3 Example of a permutation graph with  $n = 30$  vertices, shown as the points  $P(\pi)$ . A-vertices are shown red, B-vertices are green and vertices that have both type A and B (isolated vertices) are shown blue. Edges in  $G_\pi$  are drawn yellow (Color figure online)

Figures 3 and 4 show examples of these bitvectors. We can now use these to compute the extremal A and B neighbors of a vertex  $v$  as follows:

$$\begin{aligned} a^+(v) &= \text{select}_1(A_x, \text{rank}_1(A_x, v)), \\ a^-(v) &= \text{select}_1(A_x, \text{rank}_1(A_y, \pi^{-1}(v) - 1) + 1), \\ b^+(v) &= \text{select}_1(B_x, \text{rank}_1(B_y, \pi^{-1}(v))), \\ b^-(v) &= \text{select}_1(B_x, \text{rank}_1(B_x, v - 1) + 1). \end{aligned}$$

The computation takes  $O(1)$  time plus at most one evaluation of  $\pi^{-1}(v)$ .

**Remark 3.4** ( $\pi^{-1}$  for A/B-vertices) We note here (for later reference) that for  $a \in A$  we can compute  $\pi^{-1}(a) = \text{select}_1(A_y, \text{rank}_1(A_x, a))$  just from the bitvectors

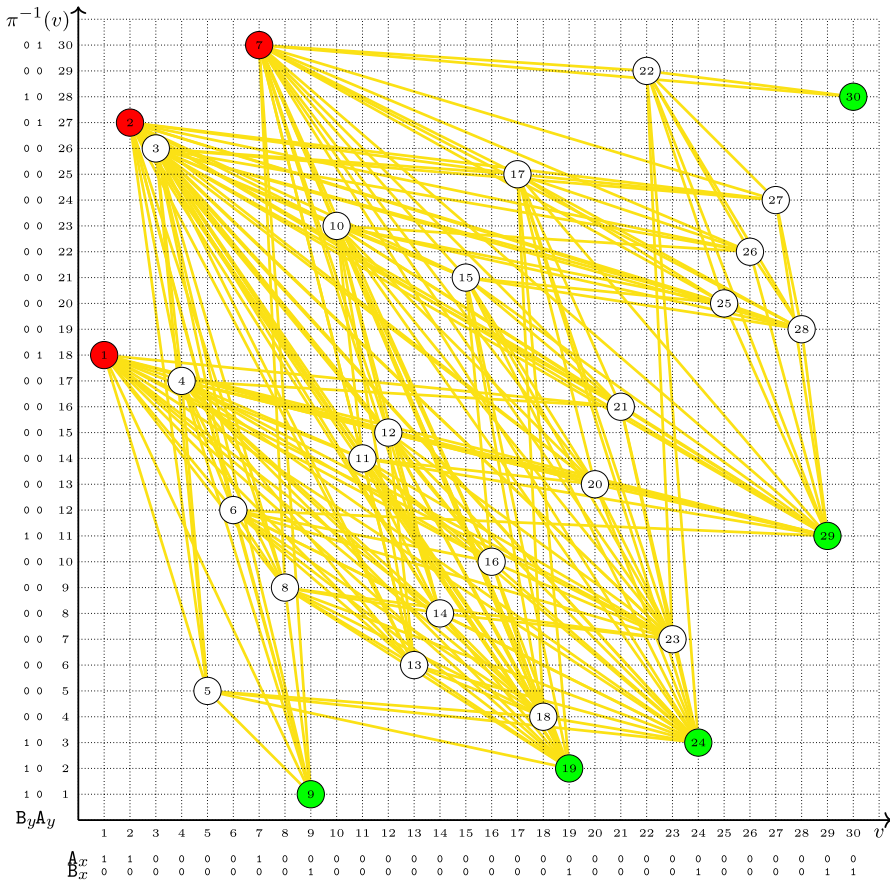


Fig. 4 Another example of a permutation graph; the drawing is as in Fig. 3. This graph is a typical graph when the  $\pi$  is chosen uniformly at random

without access to  $\Pi$ , because  $\pi^{-1}$  is monotonically increasing on  $A$ ; similarly for  $b \in B$ :  $\pi^{-1}(b) = \text{select}_1(B_y, \text{rank}_1(B_x, b))$ .

In [4, Thm. 2.1], Bazzaro and Gavaille show that the distances/shortest paths in a PG can now be found by testing for the special cases of distance at most 3 (using  $a^\pm$  or  $b^\pm$ ) or by asking a distance query in a proper interval graph. More specifically, let  $u < v$ .

1. If  $\text{adj}(u, v)$ , the distance is 1 and we are done.
2. Otherwise, if  $\text{adj}(a^+(u), v)$  or  $\text{adj}(b^+(u), v)$ , which can equivalently be written as  $a^-(v) \leq a^+(u) \vee b^-(v) \leq b^+(u)$ , the distance is 2 and we are done.
3. Otherwise, if  $\text{adj}(a^+(u), b^-(v))$  or  $\text{adj}(b^+(u), a^-(v))$ , which is equivalent to  $a^-(v) \leq a^+(b^-(u)) \vee b^-(v) \leq b^+(a^-(u))$ , the distance is 3 and we are done.
4. Otherwise, the distance is the minimum of the following four cases:  $2 + 2 \cdot \text{dist}_{G_B}(b^+(u), b^-(v))$ ,  $3 + 2 \cdot \text{dist}_{G_B}(b^+(a^+(u)), b^-(v))$ ,  $2 + 2 \cdot \text{dist}_{G_A}(a^+(u), a^-(v))$ ,  $3 + 2 \cdot \text{dist}_{G_A}(a^+(b^+(u)), a^-(v))$ .

Here  $G_A$  is the interval graph (intersection graph) defined by intervals  $[b^-(v), b^+(v)]$  for all  $v \in A$  and  $G_B$  by intervals  $[a^-(v), a^+(v)]$  for all  $v \in B$ . In general, these intervals share endpoints, but they can be transformed into a proper realization by breaking ties by vertex  $v$ , e.g., for  $G_A$ , we use  $[b^-(v) - (n - v) \cdot \epsilon, b^+(v) + v \cdot \epsilon]$  instead of  $[b^-(v), b^+(v)]$  for, say,  $\epsilon = 1/n^2$ . Then all endpoints are disjoint and no interval properly contains another; moreover, the  $i$ th smallest left endpoint corresponds to the  $i$ th smallest vertex in  $A$ .

We compute the data structure of Lemma 2.15 for  $G_A$  and  $G_B$ ; to map vertex  $v \in A$  to the corresponding vertex in  $G_A$ , we simply compute  $\text{rank}_1(A_x, v)$ ; recall that the data structure of Lemma 2.15 identifies vertices with the rank of their left endpoints. With that, we can compute the four distances above and return the minimum.

The running time for  $\text{dist}$  is the time needed for a constant number of extremal neighbor queries ( $O(1)$  for the array-based data structure,  $O(\log n / \log \log n)$  for the grid-based one), a constant number of adjacency checks (same running times), a constant number of rank-queries ( $O(1)$  each), and finally a constant number of  $\text{dist}$  queries in proper interval graphs (again  $O(1)$ ). The running time for  $\text{dist}$  is thus dominated by the time for evaluating  $\pi^{-1}$ .

### 3.4.1 Shortest paths

Suppose  $u < v$ . As noted by Bazzaro and Gavaille [4], the above case distinction does not only determine the distance, but also determines in each case a next vertex  $w$  after  $u$  on a shortest path from  $u$  to  $v$ . We output  $u$  and unless  $u = v$ , we recursively call  $\text{spath}(w, v)$ .

Since the running time for all checks above is dominated by  $\pi^{-1}(v)$ , we can iterate through the vertices on  $\text{spath}(u, v)$  in  $O(1)$  time per vertex for the array-based data structure, and in  $O(\log n / \log \log n)$  time per vertex for the grid-based data structure.

### 3.4.2 Space

The four bitvectors  $A_x, B_x, A_y,$  and  $B_y$  require no more than  $4n + o(n)$  bits of space including the support for rank and select operations.

When we allow ourselves to modify  $\pi$ , we can slightly improve upon this: We first move all isolated vertices to the largest indices. Note that any connected components can be freely permuted without changing the graph; in the point grid this has to be done by shifts along the  $y = x$  line. We now store the number  $w$  of isolated vertices. Each of the remaining nodes,  $[n - w]$ , can either be an  $A$ -node, a  $B$ -node, or neither, which can be encoded as a string over  $\{A, B, N\}$ . We store this string as a wavelet tree Lemma (2.8) with support for rank and select, using at most  $\lg(3)n + o(n)$  bits of space per dimension ( $x$  and  $y$ ), for a total of at most  $3.16993n + o(n)$  bits. (The data structure can sometimes achieve even better compression since it compresses to the empirical entropy of the string).

$G_A$  and  $G_B$  have no more than  $n$  vertices in total, so the data structures from Lemma 2.15 will use at most  $3n + o(n)$  bits of space. In addition to that, we need  $\epsilon n$  bits of space for the range-maximum and range-minimum indices, for a total of

$(6.17 + \epsilon)n + o(n)$  bits of space on top of storing  $\Pi$ . Assuming we using the data structure of [12] for the latter, the total space is  $n \lg n + (6.17 + \epsilon)n + o(n)$ .

This concludes the proof of Theorem 3.1.

### 4 Algorithms on Succinct Permutation Graphs

Clearly, `next_nbr` is equivalent to an adjacency-list based representation of a graph, so our succinct data structures can replace them in standard graph algorithms, like traversals. Beyond that, there are a few more properties specific to PGs that known algorithms for this class build on and which are not reflected in our list of standard operations. Fortunately, as we will show in the following, our data structures are capable of providing this more specialized access, as well; we formulate these as remarks for later reference.

**Remark 4.1** (*Transitive orientations & topological sort*) A graph is a comparability graph iff it admits a transitive orientation, i.e., an orientation of all its edges so that if there is a directed path from  $u$  to  $v$ , we must also have the “shortcut edge”  $(u, v)$ . In any ordered PG  $G_\pi$ , orienting all edges  $\{u, v\}$  with  $u < v$  as  $(u, v)$  yields such a transitive orientation as is immediate from the point-grid representation. Denote the resulting directed graph by  $G_\pi^\rightarrow$ .

It follows that the partition of the neighborhood into  $N^-(v)$  and  $N^+(v)$  introduced above coincides with in-neighborhood and out-neighborhood of  $v$  in  $G_\pi^\rightarrow$ , respectively. Since both our data structures for PGs handle  $N^-(v)$  and  $N^+(v)$  separately, our data structure can indeed answer `adj`, `neighborhood`, `deg`, `dist`, and `spath` queries w.r.t. digraph  $G_\pi^\rightarrow$  instead of  $G_\pi$  at no extra cost and in the same running time. (Note that `dist` and `spath` are trivial in a transitively oriented digraph: All shortest directed paths are single edges.)

It is immediate from the definition that  $1, \dots, n$ , i.e., listing the vertices by (increasing)  $x$ -coordinate in the point grid, is a topological sort of the vertices in  $G_\pi^\rightarrow$ . It is also easy to see that the same is true for decreasing  $y$ -coordinate, i.e.,  $\pi(n), \pi(n - 1), \dots, \pi(1)$  is a second topological sort of  $G_\pi^\rightarrow$ . Indeed, PGs are *exactly* the comparability graphs of posets of dimension two, i.e., the edge set of  $G_\pi^\rightarrow$  is obtained as the (set) intersection of two linear orders (namely  $1, \dots, n$  and  $\pi(n), \dots, \pi(1)$ ).

**Remark 4.2** (*One data structure for  $G$  and  $\overline{G}$* ) PGs are exactly the graphs where both  $G$  and the complement graph  $\overline{G}$  are comparability graphs. That immediately implies that  $\overline{G}$  is also a PG, when  $G$  is such.

We can extend our data structure with just  $O(n)$  additional bits of space so that we can also answer all queries in  $\overline{G}$  that the data structure could answer for  $G$ ; in fact, only the distance-related data structures  $(A_x, A_y, B_x, B_y$  and  $G_A, G_B)$  need to be duplicated for  $\overline{G}$ .

With these preparations, we can show how several known algorithms for PGs [22, 23] can efficiently run directly on top of our data structure (without storing  $G$  separately).

#### 4.1 Maximum Clique and Minimum Coloring

While computing (the size of) a maximum clique is NP-complete for general graphs, in comparability graphs, they can be found efficiently: we transitively orient the graph and then find a longest (directed) path. Note that any directed path in the transitive orientation is actually a clique in the comparability graph.

Since our data structures already maintain  $G_\pi$  in oriented form Remark (4.1), the textbook dynamic-programming algorithm for longest paths in DAGs [34] suffices: For each vertex  $v$ , we store the length of the longest directed path *ending* in  $v$  seen so far in an array  $L[v]$ . We iterate through the vertices in a topological sort; say  $v = 1, \dots, n$  (in that order). To process vertex  $v$ , we iterate through its in-neighbors  $N^-(v)$  and compute  $L[v] = \max(\{L[u] + 1 : u \in N^-(v)\} \cup \{1\})$ . Then,  $\ell = \max_v L[v]$  is the length of the longest path in  $G_\pi^\rightarrow$ , and the path can be computed by backtracing. The same  $\ell$  vertices then form a clique in  $G$ . As McConnell and Spinrad [22] noted,  $L[v]$  is simultaneously a valid coloring for  $G$  with  $\ell$  colors, so no larger clique can possibly exist.

The running time of above algorithm is  $O(n + m)$ , where  $m$  is the number of edges in  $G_\pi$ ; the extra space on top of our data structure is just  $n$  words to store the colors.

#### 4.2 Maximum Independent Set and Minimum Clique Cover

Clearly, a maximum independent set in  $G$  is a maximum clique in  $\overline{G}$ , and similarly, a minimum clique cover of  $G$  equals a minimum coloring of  $\overline{G}$ . As discussed in Remark 4.2, our data structure can without additional space support to iterate through  $N_{\overline{G}}^-(v)$ , the in-neighbors of  $v$  in  $\overline{G}$ , which is enough to run the above max-clique/min-coloring algorithm on  $\overline{G}$ .

### 5 Bipartite Permutation Graphs

Bipartite permutation graphs (BPGs) are permutation graphs that are also bipartite. While our data structures for general PGs clearly apply to BPGs, their special structure allows to substantially reduce the required space.

**Theorem 5.1** (Succinct BPG) *A bipartite permutation graph can be represented*

- using  $2n + o(n)$  bits of space while supporting  $\text{adj}$ ,  $\text{deg}$ ,  $\text{spath\_succ}$  in  $O(1)$  time and  $\text{nbrhood}(v)$  in  $O(\text{deg}(v))$  time,*
- using  $5n + o(n)$  bits of space while supporting  $\text{adj}$ ,  $\text{deg}$ ,  $\text{spath\_succ}$ ,  $\text{dist}$  in  $O(1)$  time and  $\text{nbrhood}(v)$  in  $O(\text{deg}(v))$  time.*

By iterating  $\text{spath\_succ}$ , we can answer  $\text{spath}(u, v)$  in optimal  $O(\text{dist}(u, v) + 1)$  time.

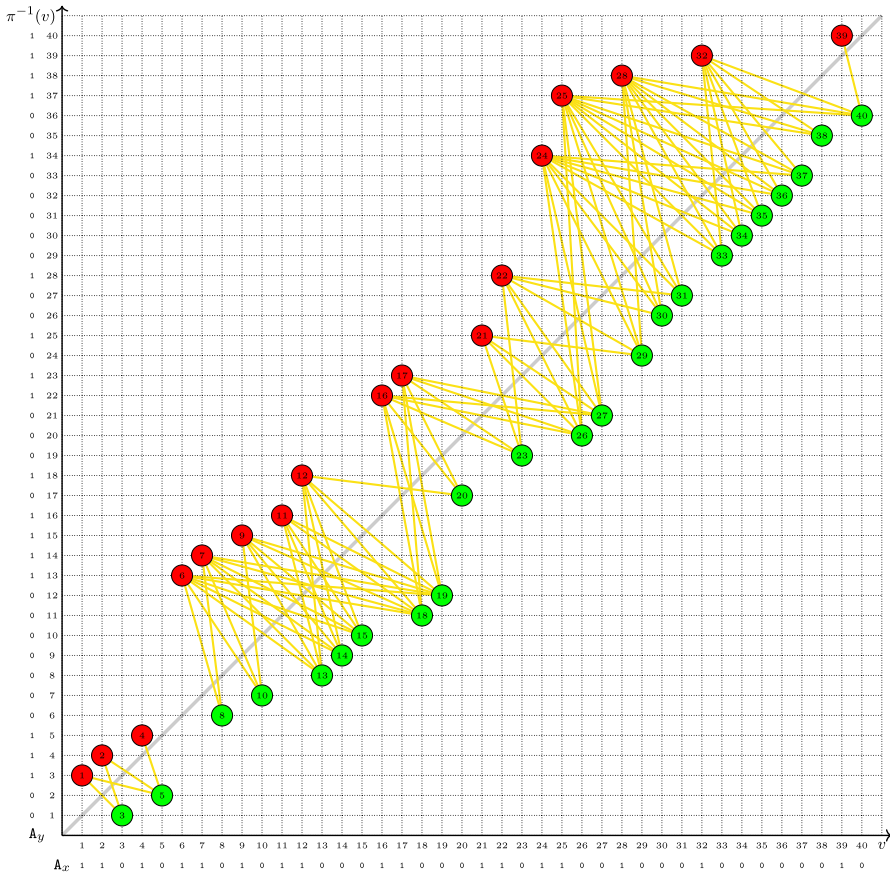


Fig. 5 An exemplary bipartite permutation graph, shown as the grid  $P(\pi)$

### 5.1 Data Structure

As already observed in [4], BPGs consist of only  $A$  and  $B$  vertices. Isolated vertices are formally of both type  $A$  and  $B$ ; thus it is convenient to assign them to the highest possible indices and to exclude them from further discussion. (All operations on them are trivial.)

All vertices being of type  $A$  or  $B$  means that every vertex corresponds to a left-to-right maximum or a right-to-left minimum. The permutation  $\pi^{-1}$  thus consists of two shuffled increasing subsequences and can be encoded using the bitvectors  $A_x$  and  $A_y$  (introduced in Sect. 3.3) in just  $2n$  bits. We add rank and select support to both bitvectors (occupying  $o(n)$  additional bits of space). Figure 5 shows an example.

The key operation is to simulate access to  $\pi^{-1}(v)$  based on the above representation:

$$\pi^{-1}(v) = \begin{cases} \text{select}_1(A_y, \text{rank}_1(A_x, v)) & \text{if } A_x[v] = 1 \\ \text{select}_0(A_y, \text{rank}_0(A_x, v)) & \text{if } A_x[v] = 0 \end{cases}$$

Computation of  $\pi^{-1}$  is thus supported in constant time. That immediately allows to compute  $\text{adj}(u, v)$  as before; moreover,  $a^-(v)$ ,  $a^+(v)$ , are directly supported, too. For  $b^-(v)$ ,  $b^+(v)$ , we exploit that in BPGs,  $\mathbb{E}_x[v] = 1 - \mathbb{A}_x[v]$  so  $b^+(v) = \text{select}_0(\mathbb{A}_x, \text{rank}_0(\mathbb{A}_y, \pi^{-1}(v)))$ , and similarly for  $b^-(v)$ .

It is easy to see that for a  $B$ -vertex  $v$ , its neighbors are exactly all  $A$ -vertices in  $[a^-(v), a^+(v)]$ ; similarly for  $A$ -vertex  $v$ , we have  $N(v) = [b^-(v), b^+(v)] \cap B$ . We can iterate through these (in sorted order) using rank/select on  $\mathbb{A}_x$ , so  $\text{neighborhood}$  can be answered in constant time per neighbor.

The degree of a vertex can be computed in  $O(1)$  time. If  $v$  is a  $B$ -vertex,  $\text{deg}(v) = \text{rank}_1(\mathbb{A}_x, a^+(v)) - \text{rank}_1(\mathbb{A}_x, a^-(v)) - 1$ , and similarly for an  $A$ -vertex.

Finally, shortest paths in BPGs are particularly simple since there is only one candidate successor vertex left: Let  $u < v$  and assume  $u$  is an  $A$ -vertex. Then either  $u$  and  $v$  are adjacent, or  $\text{spath\_succ}(u, v) = b^+(u)$ . The situation where  $u$  is a  $B$ -vertex is symmetric.

Computing  $\text{dist}(u, v)$  faster than  $\Theta(\text{dist}(u, v))$  seems only possible using the distance oracles for  $G_A$  and  $G_B$ , which require  $3n + o(n)$  additional bits of space. The query itself is as for general PGs.

This concludes the proof of Theorem 5.1.

### 5.2 Space Lower Bound

A known counting result for unlabeled BPGs implies that our data structure from Theorem 5.1 is succinct. Let us denote by  $b_n$  the number of unlabeled BPGs and by  $\bar{b}_n$  the number of unlabeled *connected* BPGs. Saitoh et al. [32, Thm. 3.14] showed that

$$\begin{aligned} \bar{b}_n &= \begin{cases} \frac{1}{4}(C_{n-1} + C_{n/2-1} + \binom{n}{n/2}) & \text{if } n \text{ is even} \\ \frac{1}{4}(C_{n-1} + \binom{n-1}{(n-1)/2}) & \text{if } n \text{ is odd} \end{cases} \\ &= C_{n-2}(1 + o(1)), \end{aligned}$$

for  $n \geq 2$ , where  $C_n$  is the  $n$ th Catalan number. Hence  $\lg b_n \geq \lg \bar{b}_n = 2n - O(\log n)$  bits are necessary to represent an unlabeled BPG on  $n$  vertices. This is asymptotically equivalent to the amount of space required by our data structure.

### 5.3 Algorithms

Our data structure for BPGs can be used to solve the HAMILTONIAN PATH and the HAMILTONIAN CYCLE problems in  $O(n)$  time with no extra space. A Hamilton path (resp. Hamiltonian cycle) in a graph is a simple path (resp. simple cycle) which contains every vertex of the graph. Given a graph  $G$ , the HAMILTONIAN PATH (resp. HAMILTONIAN CYCLE) problem asks whether the graph  $G$  contains a Hamiltonian path (resp. Hamiltonian cycle). These problems are NP-complete even when restricted to several special classes of bipartite graphs, but can be solved efficiently in the class of BPGs (see [35] and references therein). We will show how our data structure can be used to execute the algorithms from [35] in  $O(n)$  time without using extra space.



In order to explain the algorithms and their execution on the data structure, we need to introduce some preliminaries from [35]. A *strong ordering* of the vertices of a bipartite graph  $G = (A, B, E)$  consists of an ordering of  $A$  and an ordering of  $B$  such that for all  $\{a, b\}, \{a', b'\}$  in  $E$ , where  $a, a'$  are in  $A$  and  $b, b'$  are in  $B$ ,  $a < a'$  and  $b > b'$  imply  $\{a, b'\}$  and  $\{a', b\}$  are in  $E$ . The algorithms are based on the following characterization of BPGs.

**Theorem 5.2** (Strong ordering, [35]) *A graph  $G = (A, B, E)$  is BPG if and only if there exists a strong ordering of  $A \cup B$ .*

Let  $G = (A, B, E)$  be a BPG, where  $A = \{a_1, a_2, \dots, a_k\}$ ,  $B = \{b_1, b_2, \dots, b_s\}$ , and the vertices are indexed according to a strong ordering of  $A \cup B$ . Then using the characterization from Theorem 5.2, the following results were proved in [35].

**Theorem 5.3** (Hamiltonian path, [35]) *Graph  $G$  contains a Hamiltonian path if and only if*

- *either  $s = k - 1$  and  $a_1, b_1, a_2, b_2, \dots, b_{k-1}, a_k$  is a Hamiltonian path,*
- *or  $s = k$  and  $a_1, b_1, a_2, b_2, \dots, b_{k-1}, a_k, b_k$  is a Hamiltonian path,*
- *or  $s = k + 1$   $b_1, a_1, b_2, a_2, \dots, a_k, b_{k+1}$  is a Hamiltonian path,*
- *or  $s = k$  and  $b_1, a_1, b_2, a_2, \dots, a_{k-1}, b_k, a_k$  is a Hamiltonian path.*

**Theorem 5.4** (Hamiltonian cycle, [35]) *Graph  $G$  contains a Hamiltonian cycle if and only if  $k = s \geq 2$  and  $a_i, b_i, a_{i+1}, b_{i+1}$  is a cycle of length four for  $1 \leq i \leq k - 1$ .*

In order to make use of these results, we will show that in our data structure, vertices of a given ordered BPG are stored in a *strong ordering*. Recall, that given a permutation  $\pi : [n] \rightarrow [n]$ , the ordered PG induced by  $\pi$ , denoted  $G_\pi = (V, E)$ , has vertices  $V = [n]$  and edges  $\{i, j\} \in E$  for all  $i > j$  with  $\pi^{-1}(i) < \pi^{-1}(j)$ .

**Claim 5.5** Let  $G_\pi = (A, B, E)$  be an ordered BPG, then the ordering  $1 < 2 < \dots < n - 1 < n$  (restricted to  $A$  and  $B$ , respectively) is a strong ordering of  $A \cup B$ .

**Proof** As before, we assume that  $A$  is the set of  $A$ -vertices and  $B$  is the set of  $B$ -vertices of  $G$ . Let  $a, a' \in A$  and  $b, b' \in B$  be such that  $\{a, b\}$  and  $\{a', b'\}$  are in  $E$ , and  $a < a'$  and  $b > b'$ . We will show that in this case  $\{a, b'\}$  and  $\{a', b\}$  are also in  $E$ . By definition, we need to establish:

- (1)  $a < b'$  and  $\pi^{-1}(a) > \pi^{-1}(b')$ ; and
- (2)  $a' < b$  and  $\pi^{-1}(a') > \pi^{-1}(b)$ .

We will show only (1), as (2) is proved similarly. Since  $\{a', b'\} \in E$  and  $a'$  is an  $A$ -vertex, we have that  $a' < b'$  and hence  $a < b'$  (as, by assumption,  $a < a'$ ). To prove the second part of (1), we note that  $\pi^{-1}(b) < \pi^{-1}(a)$  and  $b > a$  because  $\{a, b\} \in E$ . Furthermore, since  $\{b', b\} \notin E$  and  $b' < b$ , we have that  $\pi^{-1}(b') < \pi^{-1}(b)$ . Consequently,  $\pi^{-1}(b') < \pi^{-1}(a)$ . □

### 5.3.1 Hamiltonian Path

Using Theorem 5.3 and Claim 5.5 the problem can be solved by going in constant time from the first  $A$ -vertex  $a_1$  to its first  $B$ -neighbor  $b_1 = b^-(a_1)$ , then going in constant time from  $b_1$  to its first  $A$ -neighbor  $a_2 = a^-(b_1)$ , and so on until we can no longer move. If we made  $n$  moves, then we have visited all the vertices of the graph following a Hamiltonian path. Otherwise, we try to do the same but this time starting from the first  $B$ -vertex. Similarly, if we made  $n$  moves, then the graph has a Hamiltonian path. If both attempts fail, the graph does not contain a Hamiltonian path. This algorithm works in  $O(n)$  time.

### 5.3.2 Hamiltonian Cycle

First, we check that the number of  $A$ -vertices is equal to the number of  $B$ -vertices. If so, we check next if the graph contains a Hamiltonian path using the previous algorithm (this will ensure that  $A$ - and  $B$ -vertices alternate). In the case of success, at the final stage of the algorithm, we iterate through  $A$ -vertices following the strong ordering, and for every  $A$ -vertex  $a_i$  calculate in constant time the vertices  $b_{i,1} = b^-(a_i)$ ,  $a_{i,2} = a^-(b_{i,1})$ ,  $b_{i,2} = b^-(a_{i,2})$  and check if the vertices  $a_i$  and  $b_{i,2}$  are adjacent, (i.e., whether all the four vertices induce a cycle on four vertices), which is equivalent to  $\pi^{-1}(a_i) > \pi^{-1}(b_{i,2})$ . Theorem 5.4 and Claim 5.5 imply that the graph contains a Hamiltonian cycle if and only if all stages of the algorithm were successful. Overall, the algorithm works in  $O(n)$  time.

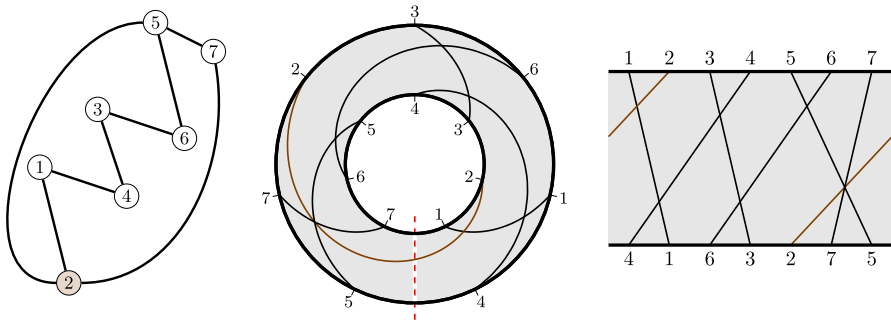
## 6 Circular Permutation Graphs

Circular permutation graphs (CPGs) are a natural generalization of PGs first introduced by Rotem and Urrutia [31]. In this section, we show how to extend our data structure to CPGs.

### 6.1 Preliminaries

CPGs results from PGs by allowing *circular/cyclic* permutation diagrams, i.e., in the intersecting chords representation, we connect the right and left end of the gray ribbon to form a cylinder. The cylinder can be smoothly transformed into two concentric circles with chords in the annular region between them; Fig. 6 shows an example.

By cutting the annulus open again, we obtain the *permutation diagram with crossings*, i.e., where some chords *cross* the cut and continue from the opposite end; (Fig. 6 right). This induces a linear order of the endpoints on both circles (in counterclockwise direction starting at the cut) and hence a permutation  $\pi : [n] \rightarrow [n]$  as before; e.g., for Fig. 6, we have  $\pi = (4, 1, 6, 3, 2, 7, 5)$ . Note that for CPGs, though,  $\pi$  no longer uniquely determines a graph because chords between circles can wrap around the inner circle in clockwise or counterclockwise direction and this affects intersections. The representation becomes unique again upon adding an assignment of chord types



**Fig. 6** Small circular permutation graph on 7 vertices (left) that is not a standard permutation graph, shown as the intersection of chords between concentric circles (middle), and as intersections of chords on a cylinder that has been cut open (note that chord 2 wraps around the cut)

$t : [n] \rightarrow \{N, F, B\}$  to  $\pi$  with the following meaning:  $N$ -chords do not cross the cut at all.  $F$ -chords do cross the cut, namely in forward direction, i.e., when following the chord from the upper endpoint to the lower endpoint, we move to the right. Finally,  $B$ -chords also cross the cut, but in backward direction, i.e., following the chord top down moves us to the left. A larger example with all types of crossings is shown in Fig. 8 (page 30).

Note that every PG is also a CPG (setting  $t(v) = N$  for all vertices), so the lower bounds from Sect. 2.2 applies here as well.

**Remark 6.1** (*Improper diagrams*) The original definition of CPGs required the permutation diagram to be “proper”, meaning that no two chords intersect more than once. All our permutation diagrams are required to be proper in this sense. (Later works [36] achieved a similar effect by defining vertices adjacent iff their chords intersect *exactly* once.)

Note that monotonic/straight chords and forbidding double crossings of the cut are *not* sufficient: not all combinations of  $\pi$  and  $t$  lead to a proper permutation diagram. Indeed, the pair  $(\pi, t)$  is valid iff no pair of chords  $u, v$  has one of the following forbidden combinations of crossing type and relative location:

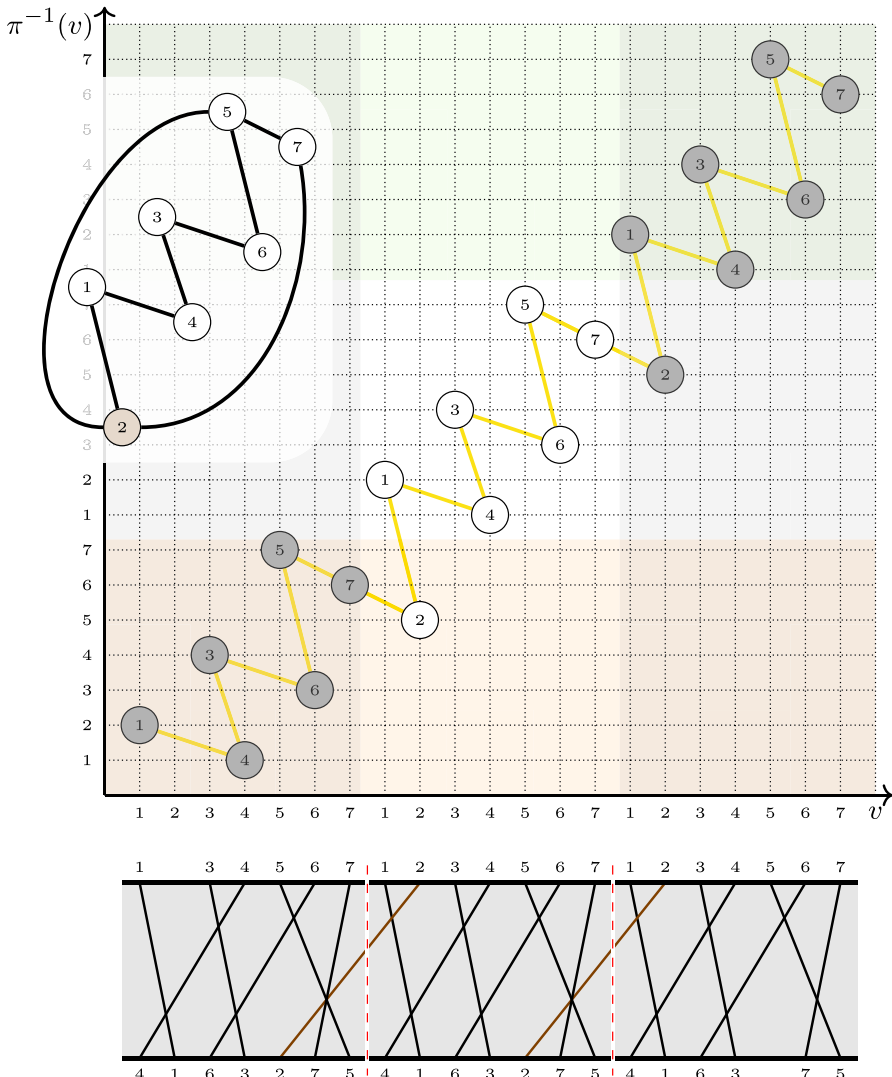
1.  $u < v, \pi^{-1}(u) > \pi^{-1}(v)$  (inversion),  $t(v) = N$ , and  $t(u) = F$ .
2.  $u < v, \pi^{-1}(u) > \pi^{-1}(v)$  (inversion),  $t(v) = B$ , and  $t(u) = N$ .
3.  $u < v, \pi^{-1}(u) < \pi^{-1}(v)$  (no inversion), and  $N \neq t(v) \neq t(u) \neq N$ .

Each of these cases implies a double crossing and a chord length  $> n$  after “pulling one chord straight” (by turning the two circles against each other).

Sritharan [36] gave a linear-time algorithm for recognizing CPGs, which also computes the circular permutation diagram if the input is a CPG.

### 6.2 Ordered CPGs and the Thrice-Unrolled PG

In analogy to ordered PG  $G_\pi$ , we define the *ordered CPG*  $G_{\pi,t}$  for a (valid combination of) permutation  $\pi : [n] \rightarrow [n]$  and chord types  $t : [n] \rightarrow \{N, F, B\}$ .



**Fig. 7** The circular permutation graph from Fig. 6 and its thrice-unrolled PG  $G_3$  as a permutation diagram and in the grid representation

From now on, we assume such a graph  $G_{\pi,t}$  is given. In preparation of our succinct data structure for CPGs, we again define a planar point set based on which we support all queries:

$$\begin{aligned}
 P(\pi, t) = & \{ (v + kn, \pi^{-1}(v) + kn) : v \in [n], k \in \{0, 1, 2\}, t(v) = N \} \\
 & \cup \{ (v + kn, \pi^{-1}(v) + (k + 1)n) : v \in [n], k \in \{0, 1\}, t(v) = F \} \\
 & \cup \{ (v + kn, \pi^{-1}(v) + (k - 1)n) : v \in [n], k \in \{1, 2\}, t(v) = B \}
 \end{aligned}$$

$P(\pi, t)$  lies in a  $3n \times 3n$  grid and  $2n \leq |P(\pi, t)| \leq 3n$ . Intuitively,  $P(\pi, t)$  is obtained by *unrolling* the circular permutation diagram of  $G_{\pi,t}$  *three times*: We record the times at which we see a chord’s endpoints during this unrolling process and output a point for these times. We only output chords when we have seen both endpoints during this process, so each noncrossing chord is output three times, whereas the crossing chords are only present twice. See Fig. 7 for an example.

Clearly,  $P(\pi, t)$  corresponds to the grid representation of a (larger) PG, denoted by  $G_3 = G_3(\pi, t)$ , which “contains”  $G_{\pi,t}$  in the sense detailed in Lemma 6.2 below. Denote the vertices in  $G_3$  by  $\ell_j, c_j,$  and  $r_j, j \in [n]$ , respectively, where  $\ell_j$  is the vertex corresponding to point  $(x, y)$  with  $x = j \in [n], c_j$  is the vertex for  $(x, y)$  with  $x = j + n \in (n..2n]$  and  $r_j$  is the vertex for  $(x, y)$  with  $x = j + 2n \in (2n..3n]$ . (Note that in general not all  $\ell_j$  (resp.  $r_j$ ) will be present.) We call  $c_v$  the main copy of vertex  $v$  in  $G_{\pi,t}$ , and  $\ell_v$  and  $r_v$  are the left (resp. right) copies of  $v$ .

**Lemma 6.2** (Neighborhood from  $G_3$ ) *Let  $v$  be a vertex in  $G_{\pi,t}$  and  $c_v$  its main copy in  $G_3(\pi, t)$ . Then  $v$ ’s neighbors (in  $G_{\pi,t}$ ) can be deduced from  $c_v$ ’s neighbors in  $G_3(\pi, t)$  as follows:*

$$N^-(v) = \{w : c_w \in N^-(c_v) \vee \ell_w \in N^-(c_v)\},$$

$$N^+(v) = \{w : c_w \in N^+(c_v) \vee r_w \in N^+(c_v)\}.$$

**Proof** First note that by construction, any edge in  $G_3$  between copies of  $u$  and  $v$  in  $G_3$  (i.e., any edge between  $\ell_u, c_u, r_u,$  resp.  $\ell_v, c_v, r_v$ ) implies an edge in  $G$  between  $u$  and  $v$ . Hence we never report non-neighbors in the set for  $N^-(v)$  and  $N^+(v)$  above. Moreover, for any combination of  $\ell, c, r$  where both copies of  $u$  and  $v$  exist, these copies are adjacent in  $G_3$ . It remains to show that any edge in  $G$  is witnessed by at least one pair of copies. For that, consider the permutation diagram of  $G_3$  and note that it contains a complete copy of the permutation diagram with crossings of  $G$  in its middle third (see Fig. 7), so every neighbor of  $v$  in  $G$  can be witnessed from  $c_v$  in  $G_3$ .  $\square$

**Remark 6.3** (*Thrice or twice?*) It follows directly from the definition of a proper permutation diagram that the upper endpoints of all backward-crossing chords must precede all upper endpoints of forward-crossing chords, and vice versa for lower endpoints. As a consequence, we can remove further copies from  $G_3$  without affecting Lemma 6.2; one can show that at most two copies of every noncrossing chord are always sufficient. Since the size of  $G_3$  will only affect lower-order terms of space, we omit this optimization here for ease of presentation.

### 6.3 Succinct CPGs

With this preparation, we can now describe our succinct data structure for CPGs. Conceptually, we store our succinct PG data structure for  $G_3$  and reduce the queries to it. For the space-dominant part, i.e., the inverse permutation  $\pi^{-1}$ , we store it implicitly, exploiting the special structure of  $G_3$ .

**Theorem 6.4** (Succinct CPGs) *An (unlabeled) circular permutation graph on  $n$  vertices can be represented using  $n \lg n + O(n)$  bits of space while supporting  $\text{adj}$ ,  $\text{dist}$ ,  $\text{spath\_succ}$  in  $O(1)$  time and  $\text{nbrhood}(v)$  and  $\text{deg}(v)$  in  $O(\text{deg}(v) + 1)$  time.*

As always, we can add constant-time degree support at the expense of another  $n \lceil \lg n \rceil$  bits of space.

We are now ready to give the proof of Theorem 6.4. Let a valid pair  $(\pi, t)$  be given and consider  $G_{\pi, t}$ . As for PGs, we store the array  $\Pi[1..n]$  with  $\Pi[i] = \pi^{-1}(i)$ ; additionally, we store the sequence  $t = t(1), \dots, t(n)$  over alphabet  $\{N, F, B\}$  for constant-time access; (two bitvectors suffice for the claimed space).

For the operations, we will show how to simulate access to the grid representation of  $G_3$ ; the reader will find it useful to consult the larger example CPG in Fig. 8 when following the description.

Mapping between vertex  $v$  in  $G$  and the  $x$ -coordinates of  $\ell_v, c_v, r_v$  in  $G_3$  is trivial. To access the  $y$ -coordinate for a point  $(x, y)$ ,  $y(x)$ , we consult the type of the corresponding vertex  $v$ :

$$y(\ell_v) = \begin{cases} \Pi[v] & \text{if } t[v] = N \\ \Pi[v] + n & \text{if } t[v] = F \end{cases}$$

$$y(c_v) = \begin{cases} \Pi[v] + n & \text{if } t[v] = N \\ \Pi[v] + 2n & \text{if } t[v] = F \\ \Pi[v] & \text{if } t[v] = B \end{cases}$$

$$y(r_v) = \begin{cases} \Pi[v] + 2n & \text{if } t[v] = N \\ \Pi[v] + n & \text{if } t[v] = B \end{cases}$$

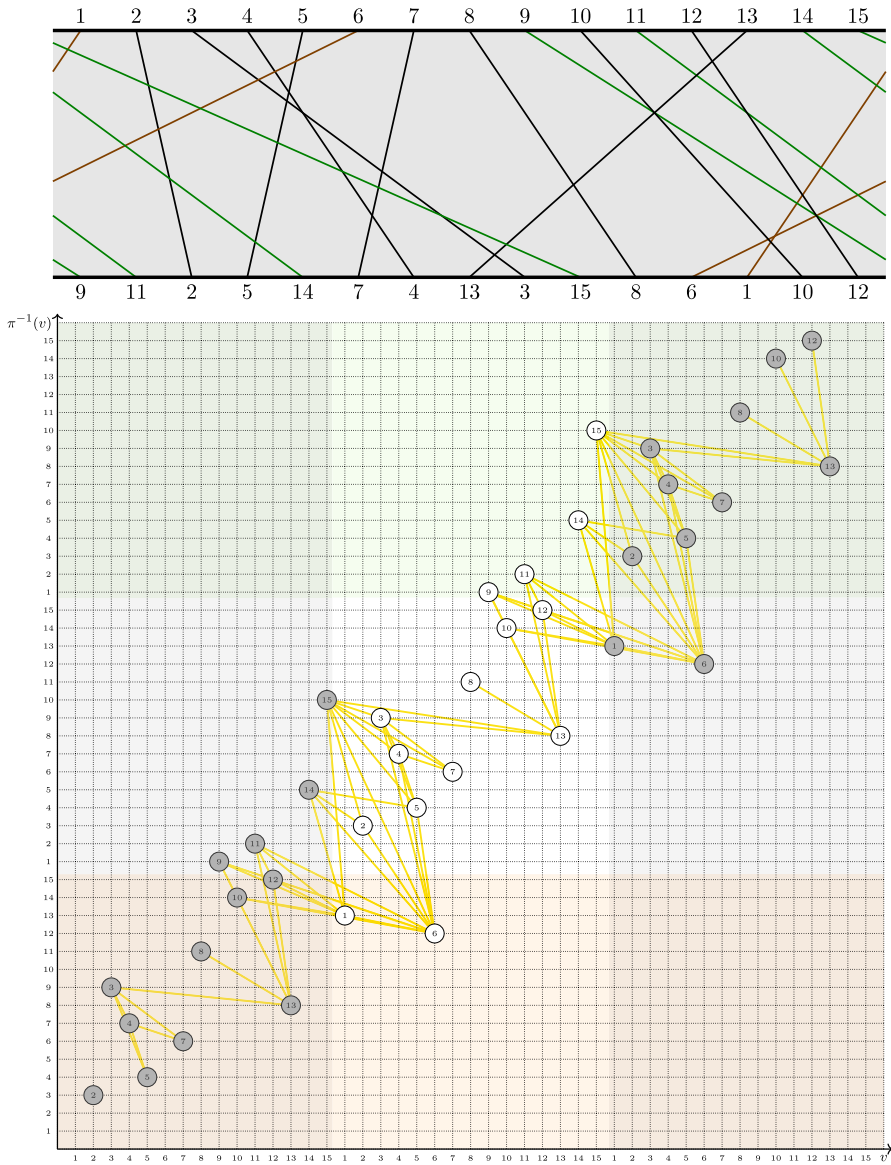
All can be answered in  $O(1)$  time. Based on that, we can answer the main queries.

### 6.3.1 Adjacency

$u < v$  are adjacent (in  $G_{\pi, t}$ ) iff  $y(c_u) > y(c_v) \vee y(\ell_v) > y(c_u) \vee y(c_v) > y(r_u)$ ; if any of the involved copies does not exist, that part of the condition is considered unfulfilled.

### 6.3.2 Neighborhood

Given a vertex  $v$ , we use Lemma 6.2 to reduce the query to neighborhood queries on  $G_3$ . To compute the neighborhood of  $c_v$  in the PG  $G_3$ , we use the same method as in Sect. 3.2; for that we store the range-minimum/maximum index from Lemma 2.9 for the sequence of  $y$ -values of all vertices in  $G_3$  (filling empty slot from missing copies with  $+\infty$ , resp.  $-\infty$ , values). Note that this index only requires access to individual values in the sequence of  $y$ -values (which we can provide in constant time); it does not require the values to be stored explicitly in an array. The additional space cost for



**Fig. 8** A larger circular permutation graph with  $n = 15$  vertices, represented by the permutation diagram with crossings (top) and the grid representation of the thrice-unrolled PG (bottom). In the permutation diagram, noncrossing chords are drawn black, forward crossing chords are green (vertices 9, 11, 14, 15) and backward crossing chords are brown (vertices 1, 6) (Color figure online)

constant-time range-min/max queries is only  $\epsilon n$  bits. The time stated for `deg` follows from counting the neighbors one by one.

### 6.3.3 Distance and shortest paths

As for neighborhood, we augment our data structure with the additional data structures from Sect. 3.3 for the PG  $G_3$ , i.e., we define  $A, B, a^\pm(v), b^\pm(v)$ , and  $G_A, G_B$  as before for  $G_3$ . All now have up to  $3n$  vertices instead of  $n$ , but only occupy  $O(n)$  bits in total.

By construction, two vertices  $u$  and  $v$  in  $G_3$  are only adjacent if the corresponding vertices in  $G$  are adjacent. Therefore, the distance between  $u$  and  $v$  can be found as the minimum over all combinations of copies of  $u$  and  $v$  in  $G_3$  (at most 9).

For (the first vertex on) a shortest path, the minimal distance pair of copies can be used with the `spath_succ` query on  $G_3$ .

This concludes the proof of Theorem 6.4.

## 7 Semi-Distributed Graph Representations

While Bazzaro and Gavoille [4] report that no distance labeling scheme for PGs exists with less than  $3 \lg(n)(1 - o(1))$  bits per label, our succinct data structure with overall  $n \lg(n)(1 + o(1))$  bits of space clearly demonstrates that this lower bound can be overcome in “centralized” data structures. An interesting question is whether this lower bound can also be circumvented using only a *small amount of global memory* on top of the local labels.

More formally, a *semi-distributed* (distance) oracle consists of a vertex labeling  $\ell : V \rightarrow \{0, 1\}^*$  and a data structure  $\mathcal{D}$ , so that `dist`( $u, v$ ) can be computed from  $(\ell(u), \ell(v), \mathcal{D})$ . If we allow arbitrary data structures  $\mathcal{D}$ , this notion is not very interesting; one could simply ask  $\mathcal{D}$  to compute all queries. But if we restrict  $\mathcal{D}$  to less space than necessary to simply encode the graph, we obtain an interesting model of computation that interpolates between standard data structures and labeling schemes.

Let us call a representation an  $(L(n), D(n))$ -space semi-distributed representation if for every  $n$ -vertex graph we have  $|\ell(v)| \leq L(n)$  for all vertices  $v$  and  $|\mathcal{D}| \leq D(n)$ . Our question can then be formulated as follows: *What is the smallest  $D(n)$  that permits a  $((3 - \epsilon) \lg n, D(n))$  space semi-distributed distance oracle for permutation graphs?*

The known distance labeling scheme from [4] implies a  $(9 \lg n, 0)$ -space semi-distributed representation, and our succinct data structure constitutes a  $(\lg n, n \lg(n)(1 + o(1)))$ -space semi-distributed representation.

A closer look at Sect. 3 reveals that the dominant space in our (array-based) data structure comes from storing  $\pi^{-1}$ . In particular, all further data structures required to answer `dist` queries occupy only  $O(n)$  bits of space. Moreover, all computations to determine distances, and even the entire shortest path, require only  $\pi^{-1}$  of the original endpoints (cf. Remark 3.4). We can thus move  $\pi^{-1}(v)$  into the label of node  $v$ , thereby making it inaccessible from any other vertex without affecting the queries. We hence obtain the following result.



**Theorem 7.1** (Semi-distributed PGs) *Permutation graphs admit a  $\langle 2 \lg n, O(n) \rangle$ -space semi-distributed representation that allows to answer the following queries:  $\text{adj}$ ,  $\text{dist}$ , and  $\text{spath\_succ}$  in  $O(1)$  time and  $\text{spath}(u, v)$  in  $O(\text{dist}(u, v) + 1)$  time.*

**Proof** The label  $\ell(v)$  consists of the pair of  $(v, \pi^{-1}(v))$ , i.e., the  $x$ - and  $y$ -coordinate in the grid representation of  $G$ . All remaining data structures from Sect. 3 occupy  $O(n)$  bits of space. As discussed above, for the listed operations access to  $\pi^{-1}$  is only needed for the queried vertices.  $\square$

**Remark 7.2** (*Who stores the labels*) Note that in our succinct data structures, we identify vertices with the (left-to-right) ranks of the upper endpoints of their chords in the permutation diagram. That means that the user of our data structure is willing to let (the construction algorithm of) our succinct data structure decide how to label vertices, and vertices are henceforth referred to using these labels. In a (semi-)distributed representation, we have to assign *and store* a unique label for each vertex, because queries are computed only from the *labels* of the vertices (and potentially  $\mathcal{D}$ ). The semi-distributed scheme derived from our succinct representation therefore takes up a total of  $\sim 2n \lg n$  bits.

This  $\langle 2 \lg n, O(n) \rangle$  scheme circumvents the lower bound for distance labelings in label length and overall space; it thus gives a novel trade-off beyond the fully distributed and fully centralized representations. In particular, it shows that access to global storage, even a fairly limited amount, is inherently more powerful than a fully-distributed labeling scheme.

## 8 Conclusion

We presented the first space-efficient data structures for permutation graphs (PGs), circular permutation graphs (CPGs), and bipartite permutation graphs (BPGs). They use space close to the information-theoretic lower bound for these classes of graphs, while supporting many queries in optimal time. The use of our data structures as space-efficient exact distance oracles improves the state of the art and proves a separation between standard, centralized data structures and distributed graph labeling schemes for distance oracles in permutation graphs. Our notion of semi-distributed graph representations interpolates between these two extremes; an initial result shows that access to global memory is inherently more powerful even if we cannot store the entire graph there.

There are several interesting directions for future research.

1. Is it possible to support degree queries in constant time and succinct space, together with the queries covered by our data structures? With our current approach, this seems to require improvements to range searching in succinct grids, but the queries are of a restricted form.
2. What is the least amount of global storage in a semi-distributed representation for distances in permutation graphs that overcomes the lower bound for distance label-

ing schemes? Is there a smooth trade-off between the “amount of decentralization” and total space, or does it exhibit a sharp threshold?

3. Comparability graphs of dimension  $k$ . These graphs have representations with  $k - 1$  chord segments per vertex; PGs correspond to  $k = 2$ . It is known [4] that for  $k \geq 3$ , distance labels require  $\Omega(n^{1/3})$  bits. Is a succinct distance oracle with efficient queries possible for these graphs?
4. Circle graphs. While navigational operations are possible [2], efficient distance queries remain an open problem.

**Acknowledgements** We thank the TCS Open Problem Group of the Department of Computer Science of University of Liverpool for creating the productive environment from which this research originated.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

1. Acan, H., Chakraborty, S., Jo, S., Nakashima, K., Sadakane, K., Satti, S.R.: Succinct navigational oracles for families of intersection graphs on a circle. [arXiv:2010.04333](https://arxiv.org/abs/2010.04333) (2020)
2. Acan, H., Chakraborty, S., Jo, S., Nakashima, K., Sadakane, K., Satti, S.R.: Succinct navigational oracles for families of intersection graphs on a circle. *Theor. Comput. Sci.* (2022). <https://doi.org/10.1016/j.tcs.2022.06.022>
3. Acan, H., Chakraborty, S., Jo, S., Rao Satti, S.: Succinct encodings for families of interval graphs. *Algorithmica* (2020). <https://doi.org/10.1007/s00453-020-00710-w>
4. Bazzaro, F., Gavaille, C.: Localized and compact data-structure for comparability graphs. *Discret. Math.* **309**(11), 3465–3484 (2009). <https://doi.org/10.1016/j.disc.2007.12.091>
5. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: *Workshop on Algorithms and Data Structures (WADS)*, pp. 98–109. Springer, Berlin, Heidelberg (2009) [https://doi.org/10.1007/978-3-642-03367-4\\_9](https://doi.org/10.1007/978-3-642-03367-4_9)
6. Chao, H.S., Hsu, F.R., Lee, R.C.T.: An optimal algorithm for finding the minimum cardinality dominating set on permutation graphs. *Discret. Appl. Math.* **102**(3), 159–173 (2000). [https://doi.org/10.1016/s0166-218x\(98\)00145-0](https://doi.org/10.1016/s0166-218x(98)00145-0)
7. Clark, D.R.: Compact PAT Trees. Ph.d. thesis (1996)
8. Colbourn, C.J.: On testing isomorphism of permutation graphs. *Networks* **11**(1), 13–21 (1981). <https://doi.org/10.1002/net.3230110103>
9. Crespelle, C., Paul, C.: Fully dynamic algorithm for recognition and modular decomposition of permutation graphs. *Algorithmica* **58**(2), 405–432 (2010)
10. Davoodi, P., Raman, R., Satti, S.R.: On succinct representations of binary trees. *Math. Comput. Sci.* **11**(2), 177–189 (2017). <https://doi.org/10.1007/s11786-017-0294-4>
11. Deogun, J.S., Steiner, G.: Polynomial algorithms for hamiltonian cycle in cocomparability graphs. *SIAM J. Comput.* **23**(3), 520–552 (1994). <https://doi.org/10.1137/s0097539791200375>
12. Dodis, Y., Pătraşcu, M., Thorup, M.: Changing base without losing space. In: *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, pp. 593–602 (2010)
13. Dushnik, B., Miller, E.W.: Partially ordered sets. *Am. J. Math.* **63**(3), 600–610 (1941)
14. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.* **40**(2), 465–492 (2011). <https://doi.org/10.1137/090779759>

15. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Symposium on Theory of Computation (STOC). ACM Press (1984) <https://doi.org/10.1145/800057.808675>
16. Gavoille, C., Paul, C.: Optimal distance labeling for interval graphs and related graph families. *SIAM J. Discret. Math.* **22**(3), 1239–1258 (2008). <https://doi.org/10.1137/050635006>
17. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: International Symposium on Experimental Algorithms (SEA), pp. 326–337. [https://doi.org/10.1007/978-3-319-07959-2\\_28](https://doi.org/10.1007/978-3-319-07959-2_28) (2014)
18. Gustedt, J., Morvan, M., Viennot, L.: A compact data structure and parallel algorithms for permutation graphs. In: Graph-Theoretic Concepts in Computer Science, pp. 372–380. Springer Berlin, Heidelberg. [https://doi.org/10.1007/3-540-60618-1\\_89](https://doi.org/10.1007/3-540-60618-1_89) (1995)
19. He, M., Munro, J.I., Nekrich, Y., Wild, S., Wu, K.: Breadth-first rank/select in succinct trees and distance oracles for interval graphs. [arXiv:2005.07644](https://arxiv.org/abs/2005.07644) (2020)
20. Jacobson, G.: Space-efficient static trees and graphs. In: Symposium on Foundations of Computer Science (FOCS), pp. 549–554 (1989)
21. Knuth, D.E.: *The Art of Computer Programming: Searching and Sorting*, 2nd edn. Addison Wesley, New York (1998)
22. McConnell, R.M., Spinrad, J.P.: Modular decomposition and transitive orientation. *Discret. Math.* **201**(1–3), 189–241 (1999). [https://doi.org/10.1016/s0012-365x\(98\)00319-7](https://doi.org/10.1016/s0012-365x(98)00319-7)
23. Möhring, R.H.: Algorithmic aspects of comparability graphs and interval graphs. In: *Graphs and Order*, pp. 41–101. Springer, Netherlands. [https://doi.org/10.1007/978-94-009-5315-4\\_2](https://doi.org/10.1007/978-94-009-5315-4_2) (1985)
24. Mondal, S., Pal, M., Pal, T.K.: An optimal algorithm to solve the all-pairs shortest paths problem on permutation graphs. *J. Math. Model. Algor.* **2**(1), 57–65 (2003)
25. Munro, J.I., Wu, K.: Succinct data structures for chordal graphs. In: 29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16–19, 2018, Jiaoxi, Yilan, Taiwan, pp. 67:1–67:12. <https://doi.org/10.4230/LIPIcs.ISAAC.2018.67> (2018)
26. Munro, J.I., Wu, K.: Succinct data structures for chordal graphs. In: 29th International Symposium on Algorithms and Computation (ISAAC 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
27. Navarro, G.: *Compact Data Structures—A practical approach*. Cambridge University Press, Cambridge (2016)
28. Pătraşcu, M.: Succincter. In: Symposium on Foundations of Computer Science (FOCS). IEEE. <https://doi.org/10.1109/focs.2008.83> (2008)
29. Pnueli, A., Lempel, A., Even, S.: Transitive orientation of graphs and identification of permutation graphs. *Can. J. Math.* **23**(1), 160–175 (1971). <https://doi.org/10.4153/cjm-1971-016-5>
30. Raman, R., Raman, V., Rao Satti, S.: Succinct indexable dictionaries with applications to encoding  $k$ -ary trees, prefix sums and multisets. *ACM Trans. Algorithms* **3**(4), 43 (2007). <https://doi.org/10.1145/1290672.1290680>
31. Rotem, D., Urrutia, J.: Circular permutation graphs. *Networks* **12**(4), 429–437 (1982). <https://doi.org/10.1002/net.3230120407>
32. Saitoh, T., Otachi, Y., Yamanaka, K., Uehara, R.: Random generation and enumeration of bipartite permutation graphs. *J. Discrete Algorithms* **10**, 84–97 (2012). <https://doi.org/10.1016/j.jda.2011.11.001>
33. Sauerermann, L.: On the speed of algebraically defined graph classes. *Adv. Math.* **380**, 107593 (2021)
34. Sedgewick, R., Wayne, K.: *Algorithms*, 4th edn. Addison-Wesley, New York (2011)
35. Spinrad, J., Brandstädt, A., Stewart, L.: Bipartite permutation graphs. *Discret. Appl. Math.* **18**(3), 279–292 (1987)
36. Sriharan, R.: A linear time algorithm to recognize circular permutation graphs. *Networks* **27**(3), 171–174 (1996). [https://doi.org/10.1002/\(sici\)1097-0037\(199605\)27:3<171::aid-net1>3.0.co;2-f](https://doi.org/10.1002/(sici)1097-0037(199605)27:3<171::aid-net1>3.0.co;2-f)
37. Tsakalidis, K., Wild, S., Zamaraev, V.: Succinct permutation graphs. [arXiv:2010.04108](https://arxiv.org/abs/2010.04108) (2020)
38. Vuillemin, J.: A unifying look at data structures. *Commun. ACM* **23**(4), 229–239 (1980). <https://doi.org/10.1145/358841.358852>