Efficient neighborhood encoding for interval graphs and permutation graphs and $O(n)$ Breadth-First Search

Christophe Crespelle[1] and Philippe Gambette[2⋆]

[1] CNRS - Univ. Paris 6, christophe.crespelle@lip6.fr
[2] LIRMM, Univ. Montpellier 2 - CNRS, gambette@lirmm.fr

Abstract. In this paper we address the problem of designing $O(n)$ space representations for permutation and interval graphs that provide the neighborhood of any vertex in $O(d)$ time, where d is its degree. To that purpose, we introduce a new parameter, called linearity, that would solve the problem if bounded for the two classes. Surprisingly, we show that it is not. Nevertheless, we design representations with the desired property for the two classes, and we implement the Breadth-First Search algorithm in $O(n)$ time for permutation graphs; thereby lowering the complexity of All Pairs Shortest Paths and Single Source Shortest Path problems for the class.

Introduction

Interval graphs are the intersection graphs of intervals of the real line, and permutation graphs are the intersection graphs of segments joining two parallel lines. They are extensively studied graph classes. One of the reasons is that they naturally appear in many contexts such as scheduling, genomics, phylogeny and archeology. On these two classes, a lot of usually NP-complete problems (e.g. coloring, maximum clique, domination) admit very efficient and elegant solutions (see [7, 11]). These algorithms are based on geometric definitions of the two classes, which give rise to $O(n)$ space representations providing adjacency information[3] between an arbitrary pair of vertices in $O(1)$ time, while the adjacency matrix takes $O(n^2)$ space, where n is the number of vertices in the graph.

Though it seems a natural question, the issue of designing $O(n)$ space data structures providing the neighborhood of an arbitrary vertex in $O(d)$ time, where d is its degree, has never been investigated for any of the two classes. For arbitrary graphs, the question of finding compact representations providing optimal time neighborhood queries is actually a practical issue [12]. The compactness of such representations allows to store the graph entirely in memory, and preserve the complexity of algorithms using neighborhood

⋆ This work was supported by the French ANR project ANR-06-BLAN-0148-01 (GRAAL).
[3] That is, answering the question "Is x adjacent to y?"

queries. The conjunction of these two advantages has great impact on the running time of algorithms managing large amount of data.

Does there exist a $O(n)$ space structure providing neighborhoods in $O(d)$ time for interval and permutation graphs? There is a natural approach that one cannot avoid to consider. It consists in trying to extend some known results for subclasses of interval graphs or permutation graphs that are known to have very good properties with regard to neighborhood encoding. The proper (or unit) interval graphs are the subclass of interval graphs that admit a model whose intervals all have the same length. They are also characterized as the graphs admitting a linear order on their vertices such that the closed[4] neighborhood of each vertex is an interval [10]. Some compression techniques are based on this notion [1, 2]: they try to find orders of the vertices that group the neighborhoods together, as much as possible. If one uses one single order on the vertices and allows the closed neighborhoods of the graph to be split in at most k intervals, the minimum value of k which make possible to encode the graph in this way is a known parameter called *closed contiguity*. Let us mention that [6, 14] showed that deciding whether a graph has closed contiguity at most k is NP-complete for any fixed $k \geq 2$, and [5] gave an upper bound on the value of the parameter for arbitrary graphs. Another possible way of generalization is to use at most k orders on the vertices such that the closed neighborhood of a vertex is the union of one interval in each of the k orders. It gives rise to a new parameter, that we call *closed linearity*, which is always less or equal to the closed contiguity. Concerning the corresponding notions for open neighborhoods, it is known that bipartite permutation graphs have[5] *open contiguity* 1 [3], or equivalently *open linearity* 1. Then, it seems unavoidable to ask whether interval and permutation graphs have one of the four parameters mentioned above bounded. Unfortunately, none of them is bounded for any of the two classes. We show that the linearity (closed or open) can be up to $\Omega(\log n / \log \log n)$. However, following another approach, we devise $O(n)$ space data structures, for both interval graphs and permutation graphs, that provide neighborhoods in $O(d)$ time, and that can be computed from an intersection model of the graph in $O(n)$ time. This gives new possibilities for the applications dealing with big interval or permutation graphs on which neighborhood queries are needed.

The fact that the neighborhood representation question had not been risen before for interval and permutation graphs is even more surprising considering that, on arbitrary graphs, many algorithmic problems are efficiently solved thanks to adjacency lists, which provide $O(d)$ time neighborhood queries.

[4] The closed neighborhood is the classic neighborhood augmented with the vertex itself. We will refer to the classic neighborhood as the open neighborhood, in order to avoid confusion between the two notions.

[5] The graphs having open contiguity 1 are exactly biconvex graphs, which is a subclass of bipartite graphs that properly contains bipartite permutation graphs.

Part of the reason is that the structure induced by the intersection models of these two classes often allows to avoid addressing the problem. However, we believe that the problem of efficiently managing neighborhood in interval and permutation graphs may lead to new algorithmic developments for the two classes by improving or simplifying some algorithms. As an illustration of the interest of considering this question, we show how to implement an $O(n)$ time Breadth-First Search algorithm (BFS for short) for permutation graphs. This lowers the complexity of finding All Pair Shortest Paths and Single Source Shortest Paths in a non weighted graph of the class to respectively $O(n^2)$ and $O(n)$.

Outline of the paper. Section 1 gives some basic definitions and notations. In Section 2, we formally define the closed linearity and show that this parameter can be up to $\Omega(\log n/\log\log n)$ for both interval graphs and permutation graphs. In Section 3, we design an $O(n)$ space data structure providing the neighborhood of any vertex in $O(d)$ time. Finally, in Section 4 we implement the BFS algorithm in $O(n)$ time for permutation graphs.

1 Preliminaries.

All graphs considered here will be finite, undirected, loopless and simple. In the following, G denotes for a graph, V for its vertex set and E for its edge set, we denote $G = (V, E)$. The set of subsets of V is denoted 2^V. Throughout the paper, n stands for $|V|$ and m for $|E|$. An edge between vertices x and y will be arbitrarily denoted xy or yx. The (open) neighborhood of x is denoted $N(x)$ and the closed neighborhood $N[x] = N(x) \cup \{x\}$. For a rooted tree T and a vertex $u \in T$, we denote $T(u)$ for the subtree of T rooted at u, and $Anc_T(u)$ for the ancestors of u in T ($u \in Anc_T(u)$). The depth of u in T, is the number of edges in the path from the root to u (the root has depth 0). The depth of T, denoted $depth(T)$, is the greatest depth of its leaves. The set of vertices at depth i in T will be denoted T^i. For a linear ordering σ on a set S, we denote $min(\sigma)$ (resp. $max(\sigma)$) for the first (resp. last) element of σ. For any $s \in S$, we denote $\sigma(s)$ for the rank of s in σ ($min(\sigma)$ has rank 1, and $max(\sigma)$ rank $|S|$), and for any $i \in [1, |S|]$, we denote $\sigma^{-1}(i)$ for the element $s \in S$ such that $\sigma(s) = i$. For $s \in S$, s^- (resp. s^+) denotes for the predecessor (resp. successor) of s in σ. $\overline{\sigma}$ denotes for the reverse order of σ. The list L containing elements $x_1, \ldots, x_k$ is denoted $L = [x_1, \ldots, x_k]$. For two lists L_1, L_2 with $L_1 = [x_1, \ldots, x_k]$ and $L_2 = [y_1, \ldots, y_k]$, we denote $L_1.L_2$ for the concatenated list $L_1.L_2 = [x_1, \ldots, x_k, y_1, \ldots, y_k]$.

An interval model of a graph G is a set of intervals of the real line together with a one to one mapping onto the set of vertices of G, such that there is an edge between vertices x and y in G iff their corresponding intervals intersect

(see Fig. 1(b)). An interval graph is a graph admitting such a model. The class remains the same if the intervals are required to be closed and to have integer endpoints between 1 and $2n$, all models considered in the following satisfy this restriction. Associating each vertex with the endpoints of its interval provides an efficient encoding of the graph that takes $O(n)$ space and allows to answer adjacency queries between any pair of vertices in $O(1)$ time.

A permutation model of a graph is a set of segments joining two given parallel lines along with a one to one mapping onto the set of vertices of G, such that there is an edge between vertices x and y in G iff their corresponding segments intersect (see Fig. 1(c)). A graph G is a permutation graph iff it admits a permutation model. The class remains the same if the endpoints of the segments are required to be pairwise distinct. We denote π_1 and π_2 for the orders on the vertices induced by the order of their endpoints respectively on the first and second line. Vertices x and y are adjacent in G iff $(\pi_1(y) - \pi_1(x)) \times (\pi_2(y) - \pi_2(x)) < 0$; that is iff x and y do not appear in the same relative order in the first and in the second linear ordering. Associating with each vertex x of G the couple $(\pi_1(x), \pi_2(x))$ results in a $O(n)$ representation of G providing adjacency in $O(1)$ time. We will refer to $\pi_1(x)$ and $\pi_2(x)$ as the endpoints of x, identifying x and its corresponding segment.

2 Interval graphs and permutation graphs have unbounded closed linearity

The aim of this section is to prove that interval graphs and permutation graphs have unbounded contiguity and linearity (both open and closed). Thanks to the relationships between the four parameters, we will derive the result from the case of closed linearity, for which we give a formal definition.

Definition 1. *We call a* closed p-line-model *of a graph $G = (V, E)$ a tuple $(\sigma_1, \ldots, \sigma_p)$ of linear orders on V such that $\forall v \in V, \exists (I_1, \ldots, I_p) \in (2^V)^p, (\forall i \in [1, p], I_i$ is an interval of $\sigma_i)$ and $N[x] = \bigcup_{1 \leq i \leq p} I_i$.*
The closed linearity *of G, denoted $cl(G)$ is the minimum integer p such that there exists a closed p-line-model of G.*

We now exhibit a family of graphs which shows that closed linearity is unbounded for interval and permutation graphs.

Theorem 1. *For any $k \in \mathbb{N}$, there exists a graph G that is both an interval graph and a permutation graph, and such that $cl(G) > k$.*

Proof. Consider the transitive closure of the rooted directed $2k+1$-ary tree T_k of depth k, for $k \geq 1$. Let G_k be its underlying undirected graph.

We first prove that G_k is a permutation graph: for every internal node v of T_k, choose an arbitrary order π_v on its $2k+1$ children. Let π_1 (resp. π_2)

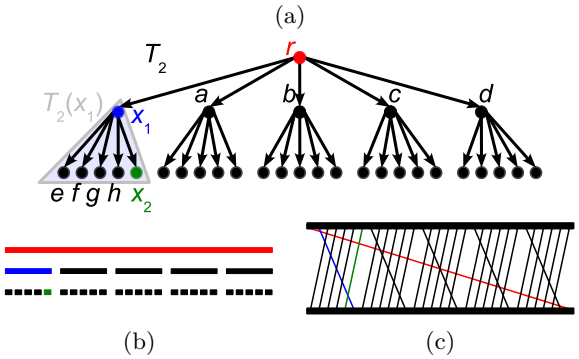

Fig. 1. The interval model (b) and the permutation model (c) of the undirected underlying graph of the transitive closure of the rooted directed tree T_2 (a). This graph has linearity strictly greater than 2, because two orders are not sufficient to represent the closed neighborhood $\{r, x_1, x_2\}$ of x_2.

be the order in which one discovers the nodes of T_k in a Depth-First Search that respects the orders π_v (resp. $\overline{\pi_v}$) for all $v \in T_k$. $(\pi_1, \overline{\pi_2})$ is a permutation model of G_n, as illustrated for G_2 in Fig. 1(c). It is easy to see that G_k is also an interval graph: for any vertex $v \in T$, choose an interval included in its father's one and disjoint from its siblings'one, as in Fig. 1(b).

We now prove that $cl(G_k) > k$. Assume for contradiction that G_k admits a closed k-line-model $\mathcal{M}$. We prove by recursion that for all $i \in [0, k]$, there is a vertex x_i at depth i in T_k such that for all $u \in Anc_{T_k}(x_i) \setminus \{x_i\}$ and for all $y \in T_k(u)$, y is not next to u in any order of $\mathcal{M}$. This is trivially true for $i = 0$ since the root is at depth 0 and has no strict ancestor. We now suppose this is true for some $i \in [0, k-1]$, and prove it for $i+1$. Since there are k orders in $\mathcal{M}$ and since x_i has $2k+1$ children, there exists one child x_{i+1} of x_i such that for all $y \in T_k(x_{i+1})$, y is not next to x_i in any order of $\mathcal{M}$. Then, x_{i+1} shows that the inductive hypothesis also holds for $i+1$.

Consider the leaf x_k given by the statement above, proved by recursion. The closed neighborhood of x_k is exactly $Anc_{T_k}(x_k)$. Let σ be a linear order of $\mathcal{M}$. From the statement, no vertex of $Anc_{T_k}(x_k)$ is next to another vertex of $Anc_{T_k}(x_k)$ in σ. It follows that the interval associated to x_k in σ contains only one vertex. As there are $k+1$ vertices in the closed neighborhood of x_k and only k orders in $\mathcal{M}$, we get a contradiction. Thus, $cl(G_k) > k$. □

Since the graphs G_k used in the proof of Theorem 1 have $(2k+1)^k$ vertices and since $cl(G_k) > k$, it follows that $cl(G_k) = \Omega(\log n / \log\log n)$. This lower bound also holds for open linearity, open contiguity and closed contiguity, which are respectively denoted $ol(G)$, $oc(G)$ and $cc(G)$. Indeed, one can obtain a closed $p+1$-line-model from an open p-line-model by adding a linear order in which the interval associated to a vertex is reduced to the vertex itself. It

follows that $ol(G) \geq cl(G)-1$. In addition, if G has contiguity p, make p copies of a linear ordering π realizing this p, and, in each of the p copies, associate to all the vertices of G one of their at most p intervals in π. This results in a k-line-model of G and shows that $cc(G) \geq cl(G)$ and $oc(G) \geq ol(G) \geq cl(G) - 1$. Thus, the four parameters have value $\Omega(\log n / \log\log n)$ for G_n. Let us mention that there are examples showing that the contiguity (open and closed) can even be up to $\Omega(\log n)$ for both interval and permutation graphs.

3 Encoding Neighborhoods in Interval Graphs and Permutation Graphs

Here, we aim at providing $O(n)$ space representations of interval graphs and permutation graphs that allow to answer neighborhood queries on any vertex x in $O(d)$ time, where d is the degree of x. Moreover, the structures we propose can be constructed in $O(n)$ time from the interval or permutation model of the graph. We first show that, for interval graphs, the encoding problem we consider can be reduced to the same problem on permutation graphs. Then, we describe an encoding of permutation graphs satisfying the desired properties.

Interval Graphs The neighborhood of a vertex x of an interval graph can be divided into three (not necessarily disjoint) parts: the subset $L(x)$ of vertices whose interval left endpoint lies in the interval of x, the subset $R(x)$ of vertices whose interval right endpoint lies in the interval of x, and the vertices whose interval either contains or is included in the one of x. Let π_1 (resp. π_2) be the order on the vertices of G obtained by sorting them with increasing interval left (resp. right) endpoints, breaking the ties in an arbitrary way. It is not difficult to see that $L(x)$ is an interval I_1 of π_1 and $R(x)$ is an interval I_2 of π_2. Concerning the last part of the neighborhood, it is known that the containment relationship of intervals (i.e. I is in relation with J iff $I \subseteq J$ or $J \subseteq I$) is a permutation graph, denoted G', whose permutation model is (π_1, π_2). Therefore, a neighborhood query on x in G will be treated by augmenting the result of the query in G' with the vertices of $I_1 \cup I_2$.

Note that this encoding of interval graphs usually contains redundant information. During the query on x, a vertex whose interval is included in that of x will appear in the result of the query in G', as well as in I_1 and in I_2. This drawback can be avoided by simply parsing the output list to remove repetitions. Anyway, this redundancy only introduces a constant multiplicative factor and we still achieve expected time and space complexity, provided that there exists a structure achieving it for permutation graphs. We now concentrate on building such a structure.

Permutation Graphs The difficulty of encoding neighborhoods in permutation graphs comes from the fact that the neighborhood of a vertex can be spread everywhere in the two orders of the permutation model. Then, scanning the orders to collect the neighborhood of x may take up to $O(n)$ time. However, we show that it is possible to extract the neighborhood of x from the permutation model, without scanning it, in $O(d)$ time, where d is the degree of x. This can be achieved thanks to *augmented Cartesian trees*, introduced in [4], which is based on *Cartesian trees* [13] and constant time nearest common ancestor queries [8, 9]. This structure provides, in constant time, the maximum, on any given interval, of an integer function f defined on a linear order; and it can be computed in $O(n)$ time for a linear order on n elements. Notice that changing f for its opposite provides the same features for minimum queries. We denote $max(f, I)$ (resp. $min(f, I)$) for the maximum (resp. minimum) of function f over interval I.

MaxNeighbor(x, I)
1. $N \leftarrow \varnothing$
2. $y \leftarrow \pi_2^{-1}(max(\pi_2, I))$
3. **If** $y >_{\pi_2} x$ **Then**
4. $\quad N \leftarrow N \cup \{y\}$
5. $\quad$ **If** $l_I <_{\pi_1} y$ **Then** $N \leftarrow N \cup$ MaxNeighbor$(x, [l_I, y^-])$
6. $\quad$ **If** $y <_{\pi_1} r_I$ **Then** $N \leftarrow N \cup$ MaxNeighbor$(x, [y^+, r_I])$
7. Return N

Neighborhood(x)
8. $N \leftarrow \varnothing$
9. **If** $min(\pi_1) <_{\pi_1} x$ **Then** $N \leftarrow N \cup$ MaxNeighbor$(x, [min(\pi_1), x^-])$
10. **If** $x <_{\pi_1} max(\pi_1)$ **Then** $N \leftarrow N \cup$ MinNeighbor$(x, [x^+, max(\pi_1)])$
11. **Return** N

Fig. 2. Routines `MaxNeighbor` and `Neighborhood`. x is a vertex of G and $I = [l_I, r_I]$ is an interval of π_1.

We now detail the algorithm listing the neighborhood of a vertex (cf. Fig 2). We assume that the model (π_1, π_2) of the permutation graph G is given, as well as the augmented Cartesian tree of the function π_2 on the linear order π_1, which provides the maximum of π_2 on any interval of π_1. The prerequisites of Routine `MaxNeighbor`(x, I) are: 1) x is a vertex of G and 2) $I = [l_I, r_I]$ is a non-empty interval of π_1 such that $r_I <_{\pi_1} x$. Lemma 1 below states that, when they are satisfied, the routine returns all the neighbors y of x that belong to I, and runs in $O(d)$ time.

Lemma 1. *When its prerequisites are satisfied, Routine* `MaxNeighbor` *returns the neighbors of x belonging to the interval I of π_1, in $O(d)$ time.*

Proof. The prerequisite 2) guarantees that $r_I <_{\pi_1} x$. Then, a vertex $z \in I$ is adjacent to x iff $z >_{\pi_2} x$. Since the test of Line 3 occurs on the vertex $y \in I$ having the greatest value for $\pi_2(y)$, it exactly determines whether x is adjacent to some vertex of I; and in the positive, y is added to N. Then, the search recursively goes on in the two pieces made by the removal of x in I. Since these two pieces still satisfy the prerequisites and since the search do not forget any part of I, it discovers all the neighbors of x belonging to I.

Thanks to the augmented Cartesian tree, the maximum query of Line 2 takes constant time, and so does each recursive call to `MaxNeighbor`. Then the complexity of the routine is its number of recursive calls. Each vertex $y \in N(x) \cap I$ discovered during a recursive call, then called a *discovering call*, has not been discovered before, since the neighbors discovered are excluded from the search. Then the number of discovering calls is $O(d)$. When a recursive call does not discover any neighbor of x, it does not call Routine `MaxNeighbor`. It follows that all calls are made by a discovering call, called its *parent call*. Since Routine `MaxNeighbor` contains at most two recursive calls during an execution, a discovering call is the parent of at most two non-discovering calls. Thus, the number of non-discovering calls is also $O(d)$, and so is the total running time of Routine `MaxNeighbor`. □

Similarly to `MaxNeighbor`, we can design a Routine `MinNeighbor` that discovers all the neighbors of x belonging to an interval I of π_1 lying entirely to the right of x: simply replace max with min at Line 2 and reverse the inequality of Line 3. Then, from Lemma 1, it is clear that Routine `Neighborhood` discovers all the neighbors of x in $O(d)$ time, as stated by Theorem 2.

Theorem 2. *Routine* `Neighborhood` *returns the neighborhood of x in $O(d)$ time.*

4 Breadth-First Search of Permutation Graphs

The input of our algorithm is a permutation graph G given by its model (π_1, π_2) and a linear order σ on V. It computes the BFS tree T_σ resulting from the BFS of G with priority order σ, that is, the BFS starting with vertex $min(\sigma)$ and where the neighbors y of a vertex are examined with increasing $\sigma(y)$. Our algorithm does not discover the vertices in the same order as the standard BFS would do. Nevertheless, the tree T produced is the same, except that the children of a vertex of T are not ordered; which we fix, in a final step, by sorting all the children lists according to σ. The total complexity of the process is $O(n)$.

BuildTree(π_1, π_2, σ)

1. $x \leftarrow min(\sigma)$; $ord(x) \leftarrow 1$
2. initialize I and Γ with (x, x, x, x)
3. **For** all $y \in N(x)$ **Do**
4. $\quad parent(y) \leftarrow x$; color y in gray; $update(\Gamma, y)$
5. **While** $I \neq \Gamma$ **Do**
6. $\quad \Gamma_{new} \leftarrow \Gamma$; $Exam \leftarrow \varnothing$
7. $\quad$ **If** $\alpha_1 < a_1$ **Then** `PutInTree`$(\alpha_1, a_1^-, \pi_1, Q_1^l, Exam)$
8. $\quad$ **If** $\beta_1 > b_1$ **Then** `PutInTree`$(\beta_1, b_1^+, \pi_1, Q_1^r, Exam)$
9. $\quad$ **If** $\alpha_2 < a_2$ **Then** `PutInTree`$(\alpha_2, a_2^-, \pi_2, Q_2^l, Exam)$
10. $\quad$ **If** $\beta_2 > b_2$ **Then** `PutInTree`$(\beta_2, b_2^+, \pi_2, Q_2^r, Exam)$
11. $\quad AssignOrd(Q_1^l, Q_1^r, Q_2^l, Q_2^r)$; color $Exam$ in gray
12. $\quad I \leftarrow \Gamma$; $\Gamma \leftarrow \Gamma_{new}$

Fig. 3. Routine `BuildTree`. Lists Q_1^l, Q_1^r, Q_2^l, Q_2^r and $Exam$ are local variables of the main loop, as well as Γ_{new} which is a quadruplet of integers. I and Γ are global variables.

The Algorithm Routine `BuildTree` (cf. Fig. 3) computes the BFS tree of the connected component of the first visited vertex. It can be applied iteratively on the first non-visited vertex in σ in order to get the complete BFS forest. From now on, we concentrate on building a single tree T. In the description of the algorithm, $parent(y)$ denotes for the parent of vertex y in T. Before the algorithm starts, $parent(y)$ is initialized with $\perp$, and y is colored in white. $\Gamma = (\alpha_1, \beta_1, \alpha_2, \beta_2)$ and $I = (a_1, b_1, a_2, b_2)$ are quadruplets of vertices. We denote I_1 for the interval $[a_1, b_1]$ of π_1 and I_2 for $[a_2, b_2]$ in π_2. Similarly, $\Gamma_1 = [\alpha_1, \beta_1]$ in π_1, and $\Gamma_2 = [\alpha_2, \beta_2]$ in π_2. Procedure $update(X, y)$, where $y \in V$ and $X = (l_1, r_1, l_2, r_2)$ is a quadruplet of vertices, executes the instructions $l_j \leftarrow min_{\pi_j}(l_j, y)$ and $r_j \leftarrow max_{\pi_j}(r_j, y)$, for $j \in \{1, 2\}$. Function ord takes integer values assigned by Procedure $AssignOrd$. We denote $<_{ord}$ the order defined by $u <_{ord} v$ iff $ord(u) < ord(v)$. The arguments of $AssignOrd$ are four lists (not necessarily disjoint), which are merged by the procedure into a single one Q_{glob} without repetition and sorted according to order $<_{prior}$ (defined below), the first element of the list being the least one for $<_{prior}$. Then, for each vertex x in Q_{glob}, the procedure assigns $ord(x)$ with the rank of x in Q_{glob}. For any $i \geq 0$, we denote $\mathcal{O}^i$ for $\{y \in T^i \mid ord(y) \text{ is defined}\}$. Order $<_{prior}$ is defined by $u <_{prior} v$ iff $ord(parent(u)) < ord(parent(v))$ or $\big(ord(parent(u)) = ord(parent(v))$ and $\sigma(u) < \sigma(v)\big)$. By convention, element $\perp$ is the greatest for order $<_{prior}$.

Routine `BuildTree` builds T level by level from the root to the leaves. The i^{th} iteration of the main loop (starting at Line 5) builds T^{i+1} by parsing the vertices of $(\Gamma_1 \setminus I_1) \cup (\Gamma_2 \setminus I_2)$, thanks to the four calls to Routine `PutInTree` (cf. Fig. 4). This routine affects to the encountered white vertices their parent

in T (Line 4). A vertex is colored gray (Line 11 of `BuildTree`) only when it has been placed correctly in T; note that a vertex may be assigned a parent twice, once in π_1 and once in π_2. The main loop stops when $I = \Gamma$ (Line 5), that is, when the two intervals of π_1 and π_2 corresponding to the connected component C of $min(\sigma)$ have been entirely parsed. Then, all the vertices have been assigned a parent, and the construction of T is over.

PutInTree(u, v, π, Q, F)
1. $p \leftarrow u$; $Q \leftarrow [p]$
2. **For** y from u to v in π **Do**
3. **If** y is white and $p <_{prior} parent(y)$ **Then**
4. $parent(y) \leftarrow p$; $F \leftarrow F.[y]$; $update(\Gamma_{new}, y)$
5. **If** y is gray and $y <_{prior} p$ **Then**
6. $p \leftarrow y$; $Q \leftarrow [p].Q$

Fig. 4. Routine `PutInTree`. π is an order, u and v two vertices, and Q and F are lists. Γ_{new} is a variable of Routine `BuildTree`.

Correctness It is straightforward that during all the algorithm, $I_1 \subseteq \Gamma_1$ and $I_2 \subseteq \Gamma_2$. The key of the correctness of our algorithm is the following invariants. At the beginning of the i^{th} iteration of the main loop of `BuildTree` (Line 5), T has been built correctly until depth i, the set of gray vertices is exactly $\bigcup_{0 \leq j \leq i} T^j$, and the properties below hold.

1. the vertices of $\bigcup_{0 \leq j < i} T^j$ have their two endpoints in I_1 and I_2;
2. the vertices of T^i have their two endpoints in Γ_1 and Γ_2;
3. $I_1 \cup I_2$ contains all the vertices of $\bigcup_{0 \leq j \leq i} T^j$ and no others;
4. the bounds of Γ_1 and Γ_2 belong to $\bigcup_{0 \leq j \leq i} T^j$.
5. the order $<_{ord}$ restricted to $\mathcal{O}^{i-1}$ is exactly the order of visit of the vertices of $\mathcal{O}^{i-1}$ by the standard BFS algorithm.

During the main loop, the four calls to `PutInTree` parse the vertices of $B = (\Gamma_1 \setminus I_1) \cup (\Gamma_2 \setminus I_2)$. The white vertices of B are exactly the vertices of T^{i+1} and the gray ones are in T^i. `PutInTree` affects to the white vertices of B their parent in T, which is among the gray vertices of B. The vertices of T^i that have their two endpoints in I_1 and I_2 are leaves of T. Let us examine more precisely the first call to `PutInTree`, when $\alpha_1 < a_1$. From Invariant 1 and 4 (Inv. for short), $\alpha_1 \in T^i$. Let w be a white vertex of $[\alpha_1, a_1^-]$. Since all the vertices of $I_1 \cup I_2$ are gray (Inv. 3) and since w is not adjacent to $min(\sigma)$, necessarily $w <_{\pi_2} a_2$. Moreover, since $\alpha_1 \in T^i$, α_1 has an endpoint

in I_2 (Inv. 3). It follows that w and α_1 are adjacent and $w \in T^{i+1}$. Similarly, w is adjacent to all the gray vertices $z <_{\pi_1} w$ and to none of the gray vertices $z >_{\pi_1} w$. From Inv. 5 and the definition of $<_{prior}$, we have that, for $u, v \in T^i$, $u <_{prior} v$ iff u is visited before v by the standard BFS. Then, the test at Line 5 together with the affectation at Line 6 maintain variable p as the gray node of $[\alpha_1, y]$ which is the first visited by the standard BFS, where y is the current vertex in the loop starting at Line 2. Thus, any white vertex $w \in [\alpha_1, a_1^-]$ is assigned a parent which is the gray vertex of $[\alpha_1, a_1^-] \cap N(w)$ being the first one visited by the standard BFS (remind that $[w, a_1^-] \cap N(w) = \varnothing$). More generally, this is true for any white node discovered in any of the four calls to `PutInTree`. Note that a white vertex w may be seen in two different calls to `PutInTree`, once in π_1 and once in π_2. In this case, the test at Line 3 guarantees that the second possible parent is affected if and only if it is visited by the standard BFS before the first one. Thus, during the main loop, any white vertex w of B is affected its correct parent in T. We showed that $w \in T^{i+1}$; note that conversely, every vertex $w \in T^{i+1}$ is white and has necessarily an endpoint in B: otherwise, it would either be adjacent to none of the vertices of $\bigcup_{0 \leq j \leq i} T^j$, or adjacent to $min(\sigma)$; both contradict the fact that $w \in T^{i+1}$. As a conclusion, the i^{th} iteration of the main loop properly computes level T^{i+1} and colors its vertices in gray (Line 11).

The fact that Γ_{new} is updated, at Line 4 of `PutInTree`, every time a white vertex (which will become gray at Line 11) is visited, together with the two affectations at Line 12 of `BuildTree`, imply that Inv. 1 to 4 are still true at the beginning of the next iteration. Finally, since order $<_{prior}$ on set T^i is the order of visit of the vertices of T^i by the standard BFS, and since $AssignOrd$ (Line 11) assigns the values of $ord(v)$ to the vertices $v \in T^i$ according to $<_{prior}$, it follows that Inv. 5 is also maintained during the loop.

Complexity The total running time of our BFS algorithm is $O(n)$. First, we build the structure of Section 3 that allows to answer neighborhood queries in $O(d)$ time. It takes $O(n)$ time. Then, Routine `BuildTree` gives the BFS tree (except the order on the children) of the connected component C of the first vertex examined, in $O(|C|)$ time. We repeat Routine `BuildTree`, starting from the first non-visited vertex in σ, until all the vertices of the graph have been visited. As the sets of vertices visited during each call to `BuildTree` are disjoint, the total running time of all the calls to `BuildTree` is $O(n)$. At last, we order the lists of children of the vertices in all the trees produced: set the lists of children of all the vertices in the forest to $\varnothing$, and scan σ placing each vertex in the list of its parent. At the end of the scan, all the children lists have been rebuilt and sorted according to σ. This process takes $O(n)$ time.

Let us detail the $O(|C|)$ time complexity of Routine `BuildTree`. Remind that B denotes for $(\Gamma_1 \setminus I_1) \cup (\Gamma_2 \setminus I_2)$. Thanks to the data structure of

Section 3, at Line 3, we can get $N(x)$ in $O(|N(x)|)$ time, which is also the running time of the initialization loop. In `PutInTree`, all instructions take $O(1)$ time, and the Routine runs in $O(|\pi(u) - \pi(v)|)$ time. That is, $O(|B|)$ time for the four calls of the main loop. Coloring $Exam$ also takes $O(|B|)$ time. The complexity of procedure $AssignOrd$ is a crucial point. It is worth to note that any list Q' being one of its arguments is already sorted according to $<_{prior}$. This is a property of `PutInTree`, which produces Q', guaranteed by the test $y <_{prior} p$ at Line 5 and the affectations of Line 6. It follows that $AssignOrd$ can be implemented to merge the four lists in a single one, sorted according to $<_{prior}$, in $O(|Q_1^l| + |Q_1^r| + |Q_2^l| + |Q_2^r|) = O(|B|)$ time. Then, the running time of an iteration of the loop is $O(|B|)$, and since all the B's considered until the end of the loop are pairwise disjoint, it follows that the main loop, as well as Routine `BuildTree`, runs in $O(|C|)$ time.

References

1. P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW'04*, pages 595–602. ACM, 2004.
2. P. Boldi and S. Vigna. Codes for the world wide web. *Internet Mathematics*, 2(4):407–429, 2005.
3. A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph Classes: a Survey*. SIAM Monographs on Discrete Mathematics and Applications, 1999.
4. H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *STOC'84*, pages 135–143, 1984.
5. C. Gavoille and D. Peleg. The compactness of interval routing. *SIAM Journal on Discrete Mathematics*, 12(4):459–473, 1999.
6. P.W. Goldberg, M.C. Golumbic, H. Kaplan, and R. Shamir. Four strikes against physical mapping of DNA. *Journal of Computational Biology*, 2(1):139–152, 1995.
7. M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. Elsevier, second edition, 2004.
8. D. Harel. A linear time algorithm for the lowest common ancestors problem (extended abstract). In *FOCS'80*, pages 308–319, 1980.
9. D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
10. F.S. Roberts. *Representations of Indifference Relations*. PhD thesis, Stanford University, 1968.
11. J. P. Spinrad. *Efficient graph representations*, volume 19 of *Fields Institute Monographs*. American Mathematical Society, 2003.
12. G. Turan. On the succinct representation of graphs. *Discr. Appl. Math.*, 8:289–294, 1984.
13. J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.
14. R. Wang, F.C.M. Lau, and Y. Zhao. Hamiltonicity of regular graphs and blocks of consecutive ones in symmetric matrices. *Discr. Appl. Math.*, 155(17):2312–2320, 2007.