Unrestricted and complete Breadth-First Search of trapezoid graphs in O(n) time

Christophe Crespelle, Philippe Gambette

Unrestricted and Complete Breadth-First Search of Trapezoid Graphs in $O(n)$ Time

Christophe Crespelle*

LIP6 - Université Pierre & Marie Curie, CNRS

Philippe Gambette*

LIRMM - Université Montpellier 2, CNRS

Abstract

We present an $O(n)$ Breadth-First Search algorithm for trapezoid graphs, which takes as input a trapezoid model and any priority order on the vertices. Our algorithm is the first able to produce *any* BFS-tree, and not only one specific to the model given as input, within this complexity. Moreover, it produces *all* the shortest paths from the root of the BFS-tree to the other vertices of the graph.

Keywords: Breadth-First Search, trapezoid graphs, permutation graphs, interval graphs, shortest paths, graph algorithms

1. Introduction

Breadth-First Search (BFS for short) is probably the most famous graph algorithm, and also one of the most basic ones. Nevertheless, it turns out to be useful in many more sophisticated graph algorithms and in many contexts. One may think that there is nothing to do to lower the $O(n + m)$ time complexity of the BFS algorithm, as, anyway, every edge of the graph should be traversed in order to perform the search correctly. This is not required for some well-structured graph classes that admit $O(n)$ space representations, such as trapezoid graphs.

This class is a proper generalization of both permutation graphs and interval graphs, which are extensively studied; the main reason being that they naturally appear in many contexts such as scheduling, genomics, phylogeny and archeology [6]. Moreover, because of their strong but non-trivial structure, they also constitute a case worth to be studied before generalizing results to wider graph classes.

Here, we design the first algorithm able to produce *any* Breadth-First Search of a trapezoid graph, regardless of the model given as input, in $O(n)$ time. This is achieved by making part of the input a priority order σ on the vertices, according to which the ties between neighbors of the vertex currently examined by the BFS will be broken. Moreover, our algorithm gives not only one but *all* possible shortest paths between any vertex of the graph and the root of the BFS-tree, which is a feature of high interest in practice for many problems.

Efficient implementations of BFS for these classes of graphs have already been studied: [11] showed that BFS can be implemented in $O(n)$ time for permutation graphs, but their algorithm can only produce a single BFS-tree, which is not general at all and reflects the particular structure of their algorithm: it is always a caterpillar (a tree with a unique internal node at each level) rooted at the vertex whose upper endpoint is leftmost in the permutation diagram. [7] improved this result by giving an algorithm, having the same complexity, that is general to trapezoid graphs and can produce a tree T rooted at any vertex r of the graph. But again, the tree T produced is always the same and is strongly constrained: it is the union of four caterpillars having the same root r. [3] showed that a parallel algorithm achieves a better complexity for the problem, but the tree produced is exactly the same as in [11]. Concerning Depth-First Search, the situation is similar: [11, 4] give efficient implementations ($O(n)$ time in [4]) but can only produce a single tree whose structure is very particular among all possible trees.

2. Preliminaries

Without loss of generality, we can make some restrictions and simplifications in the description of our algorithm. First, unlike the standard BFS algorithm, the tree first produced by our algorithm is unordered, that is, there is no order on the children of a vertex in the tree, while, in the standard BFS-tree, these children are ordered according to their discovery time in the search. In order to get the desired ordered tree, we can proceed as follows at the end of the algorithm. We parse order σ from the first to

*Corresponding author

Email addresses: christophe.crespelle@lip6.fr (Christophe Crespelle), philippe.gambette@lirmm.fr (Philippe Gambette)

the last vertex, and for each vertex, we remove it from the list of children of its father and insert it again at the end of this list. This takes $O(n)$ time and orders the lists of children with regard to σ, as desired. As a consequence, in the following, we only consider the construction of the unordered tree.

In addition, we focus on the case where the input graph is connected. If it is not, our algorithm performs the BFS of the connected component of the first vertex in σ, without any modification of its description. Then, the algorithm can go on starting from the first non-visited vertex of σ. Finding this vertex can be done in constant time by maintaining along the algorithm the list of non-visited vertices, ordered with regard to σ.

Definitions and Notations. All graphs considered here are finite, undirected, loopless and simple. V is the vertex set of graph G and E is its edge set, which we denote by $G = (V, E)$. Throughout the paper, n stands for $|V|$. An edge between vertices x and y will be arbitrarily denoted xy or yx. The neighborhood of x is denoted $N(x)$. T denotes the BFS-tree to be produced. The depth of node u in T is its distance from the root. The set of vertices at depth i in T will be denoted T^i (T^0 contains only the root of T). 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 $s \in S$, s^- (resp. s^+) denotes for the predecessor (resp. successor) of s in σ. The interval of σ made of vertices between a and b, with $a \leq_\sigma b$, is denoted $[\![a, b]\!]$. The list L containing elements $x_1, \ldots, x_k$ is denoted $L = [x_1, \ldots, x_k]$. The concatenation of two lists L_1, L_2 is denoted $L_1.L_2$.

A trapezoid model of a graph G is a set of trapezoids between two horizontal lines (two endpoints on the upper line and two endpoints on the lower 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 trapezoids intersect. A trapezoid graph is a graph admitting such a model. Note that a trapezoid model can be computed from the graph in time $O(n^2)$ using the algorithm of [9]. The class remains the same if the trapezoids are required to be closed and to have distinct integer endpoints between 1 and $2n$ on both lines. All models considered here satisfy this restriction. More precisely, in the following, a trapezoid model will be considered as a couple (π_1, π_2) of orders on some subsets of trapezoid endpoints of the model: π_1 is the left to right order of the endpoints on the upper line, and π_2 is the left to right order of the endpoints on the bottom line. Each vertex v is associated with the four endpoints of its corresponding trapezoid, which are denoted v_1^l, v_1^r, v_2^l and v_2^r for the top-left, top-right, bottom-left and bottom-right endpoint respectively. This 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. In the sequel, we often identify vertices and their associated trapezoids.

BuildTree(π_1, π_2, σ)
1. $x \leftarrow min(\sigma)$; $ord(x) \leftarrow 1$; color x in gray
2. initialize I and Γ with $(x_1^l, x_1^r, x_2^l, x_2^r)$
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. $\quad \Gamma_{new} \leftarrow \Gamma$; $Ex \leftarrow \varnothing$
7. $\quad$ **If** $\alpha_1^l <_{\pi_1} a_1^l$ **Then** `PutInTree`$(\alpha_1^l, a_1^{l-}, \pi_1, l, Q_1^l)$
8. $\quad$ **If** $\beta_1^r >_{\pi_1} b_1^r$ **Then** `PutInTree`$(\beta_1^r, b_1^{r+}, \pi_1, r, Q_1^r)$
9. $\quad$ **If** $\gamma_2^l <_{\pi_2} c_2^l$ **Then** `PutInTree`$(\gamma_2^l, c_2^{l-}, \pi_2, l, Q_2^l)$
10. $\quad$ **If** $\delta_2^r >_{\pi_2} d_2^r$ **Then** `PutInTree`$(\delta_2^r, d_2^{r+}, \pi_2, r, Q_2^r)$
11. $\quad AssignOrd(Q_1^l, Q_1^r, Q_2^l, Q_2^r)$; color Ex in gray
12. $\quad I \leftarrow \Gamma$; $\Gamma \leftarrow \Gamma_{new}$

Figure 1: Routine `BuildTree`. Lists Q_1^l, Q_1^r, Q_2^l, Q_2^r and Ex are global variables, as well as $I = (a_1^l, b_1^r, c_2^l, d_2^r)$, $\Gamma = (\alpha_1^l, \beta_1^r, \gamma_2^l, \delta_2^r)$ and Γ_{new} which contain quadruplets of endpoints.

PutInTree$(u_k^{side}, v_k^t, \pi_k, side, Q)$
1. $p \leftarrow u$; $Q \leftarrow [p]$
2. **For** y_k^s from u_k^{side} to v_k^t in π_k **Do**
3. $\quad$ **If** y white & $s \neq side$ & $p <_{prior} parent(y)$ **Then**
4. $\quad\quad parent(y) \leftarrow p$; $Ex \leftarrow Ex.[y]$; $update(\Gamma_{new}, y)$
5. $\quad$ **If** y gray & $s = side$ & $y <_{prior} p$ **Then**
6. $\quad\quad p \leftarrow y$; $Q \leftarrow [p].Q$

Figure 2: Routine `PutInTree`. $side$ and t belongs to $\{l, r\}$, u_k^{side} and v_k^t are two endpoints, and Q is a list. Γ_{new} and Ex are global variables initialized in Routine `BuildTree`. Note that according to our previously introduced notation u is the vertex corresponding to endpoint u_k^{side} and y is the vertex corresponding to endpoint y_k^s.

3. Breadth-First Search of Trapezoid Graphs

The input of our algorithm is a trapezoid graph G (assumed to be connected w.l.o.g.) given by its model (π_1, π_2) and a linear order σ on V. The first vertices of σ are those having highest priority, in particular, the first visited vertex is the minimum of σ. We first focus our attention on the algorithm building the unordered BFS-tree T. The structure computing and storing all shortest paths from the root of T to the other vertices of the graph is easy to design afterwards. This is done in Section 3.3.

3.1. Overview of the algorithm

Our algorithm computes the unordered BFS-tree T of G level by level: iteration number i of the main loop of Routine `BuildTree` (Line 5, cf. Fig. 1) builds level T^{i+1}. To that purpose, we parse the model of the graph pieces by pieces as explained below. We maintain along the algorithm two intervals in each of the two orders of the model: Γ_1 is the minimal interval of π_1 containing the upper endpoints of the vertices placed in the tree so far, namely

$\bigcup_{0\leq j\leq i} T^j$, and I_1 is the interval containing the upper endpoints of the vertices of $\bigcup_{0\leq j\leq i-1} T^j$; intervals Γ_2 and I_2 are similarly defined in π_2 for lower endpoints.

Each level of the tree is built by parsing the endpoints in $\Gamma_1 \backslash I_1$ and $\Gamma_2 \backslash I_2$, which are split in four intervals, thanks to the four calls to Routine `PutInTree` (Lines 7 to 10 of `BuildTree`). The new vertices encountered during this parse are exactly those of T^{i+1}, which are then assigned their parent in T. It is important to note that our algorithm does not discover the vertices in the same order as the standard BFS would do. Despite of this, we are able to determine, for each vertex, its correct parent in the BFS-tree, thanks to function *ord*. *ord* numbers a subset of vertices in each level of the tree computed so far, except in the last one T^i. Its essential property is that, at the beginning of the i^{th} iteration of the main loop, every vertex of level T^{i-1} having children in T is numbered and the order induced by *ord* on these vertices is exactly the order in which the standard BFS discovers them. Together with order σ, it allows to determine the parent of the vertices of level T^{i+1} and to assign their *ord* value to vertices of T^i (at least to those having children in T). Note that a vertex may be assigned a parent twice by our algorithm: in this case, the second one is the correct one.

Before the algorithm starts, for every vertex y, $parent(y)$ is initialized with $\perp$, and y is colored white. During the algorithm, vertices are colored gray once they have been assigned their correct parent in the tree. This is done at the end of the main loop of Routine `BuildTree` (Line 11). We denote by (a_1^l, b_1^r) (resp. (α_1^l, β_1^r), (c_2^l, d_2^r), (γ_2^l, δ_2^r)) the left and right endpoint of I_1 (resp. Γ_1, I_2, Γ_2). Remark that this notation is coherent, since it is a straightforward property of our algorithm that the left (resp. right) endpoint of any of these four intervals is always the left (resp. right) endpoint of some trapezoid of the model. For $y \in V$ and a quadruplet $X = (l_1, r_1, l_2, r_2)$ of endpoints such that $l_1, r_1 \in \pi_1$ and $l_2, r_2 \in \pi_2$, procedure $update(X, y)$ executes the instructions $l_j \leftarrow min_{\pi_j}(l_j, y_j^l)$ and $r_j \leftarrow max_{\pi_j}(r_j, y_j^r)$, for $j \in \{1, 2\}$. Function *ord* takes integer values which are assigned by Procedure *AssignOrd*. Order $<_{prior}$ is defined by $u <_{prior} v$ iff $ord(parent(u)) < ord(parent(v))$ or $\big(ord(parent(u)) = ord(parent(v))$ and $u <_\sigma v\big)$. By convention, element $\perp$ is the greatest for order $<_{prior}$.

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 above), the first element of the list being the least one for $<_{prior}$. Once list Q_{glob} is obtained, for each vertex x contained in it, the procedure assigns $ord(x)$ with the rank of x in Q_{glob}.

3.2. Correctness

Our proof of correctness is based on two invariants. The first one below shows that the set of new vertices discovered during an iteration of the main loop of Routine `BuildTree` is exactly the next level of the BFS-tree

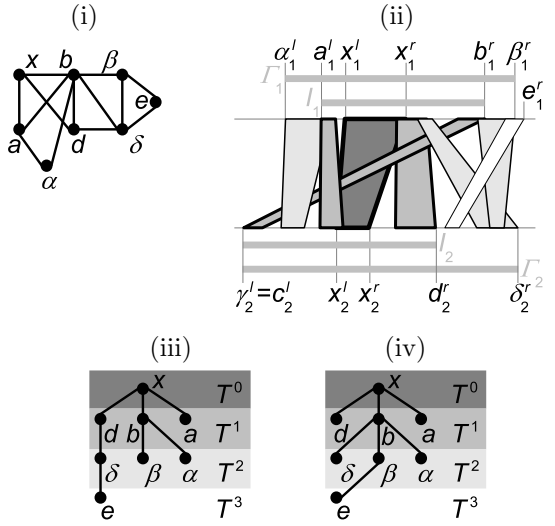

Figure 3: A trapezoid graph (i), its trapezoid model (ii) labeled as in the beginning of the second iteration of the main loop of Algorithm 1, and two BFS trees: one respecting priority order ($x\beta\delta dbea\alpha$) (iii) and the other respecting priority order ($x\beta\delta bdea\alpha$) (iv).

T. The second one implies that those vertices are assigned their correct parent in T by the algorithm.

Invariant 1. *For any $i \geq 2$, at the beginning of the i^{th} iteration of the main loop, the following properties hold.*

1. *I_1 (resp. I_2) is the minimal interval of π_1 (resp. π_2) containing both upper endpoints (resp. lower endpoints) of the vertices of $\bigcup_{0\leq j\leq i-1} T^j$; and*
2. *Γ_1 (resp. Γ_2) is the minimal interval of π_1 (resp. π_2) containing both upper endpoints (resp. lower endpoints) of the vertices of $\bigcup_{0\leq j\leq i} T^j$; and*
3. *set $\bigcup_{0\leq j\leq i} T^j$ is equal to the set of vertices with at least one endpoint in $I_1 \cup I_2$, and is exactly the set of gray vertices.*

Remark: For $i = 1$, the first two properties of Invariant 1 are satisfied, but even though $\bigcup_{0\leq j\leq i} T^j$ is the set of gray vertices, Property 3 is not satisfied.

Nevertheless, for any $i \geq 2$ the three properties hold. In order to prove it, we will need the following lemma.

Lemma 1. *Let $i \geq 1$. At the beginning of the i^{th} iteration of the main loop of Routine `BuildTree`, if the conditions of Invariant 1 are satisfied or if $i = 1$, then the set of white vertices with at least one endpoint in $\Gamma_1 \cup \Gamma_2$ is exactly T^{i+1}.*

Proof. Let y be a white vertex with at least one endpoint y_k^s in $\Gamma_1 \cup \Gamma_2$. If Invariant 1 is satisfied, then from Property 3, vertices having at least one endpoint in $I_1 \cup I_2$ are gray. This is still true for $i = 1$ since those vertices belong to $N(x)$. It follows that the endpoint y_k^s of y belonging to $\Gamma_1 \cup \Gamma_2$ is in $(\Gamma_1 \setminus I_1) \cup (\Gamma_2 \setminus I_2)$. Let us examine the case where $k = 1$ and y_1^s is such that $b_1^r <_{\pi_1} y_1^s <_{\pi_1} \beta_1^r$ (illustrated in Figure 3(ii) with $y_k^s = e_1^l$), the other cases are

similar. Since y is white, $y \notin T^i$ and therefore is not adjacent to any vertex of T^{i-1}, which implies that $y_1^l >_{\pi_1} b_1^r$ and $y_2^l >_{\pi_2} d_2^r$. Moreover, since $b_1^r \neq \beta_1^r$, Property 1 and 2 of the invariant ensures that $\beta \in T^i$. If $i \neq 1$, from Property 3 of the invariant, β has at least one endpoint in $I_1 \cup I_2$. It follows that $\beta_1^l <_{\pi_1} b_1^r$ or $\beta_2^l <_{\pi_2} d_2^r$. In the former case, since $\beta_1^l <_{\pi_1} y_1^l <_{\pi_1} \beta_1^r$, y is adjacent to β. In the latter case, since $y_1^l <_{\pi_1} \beta_1^r$ and $\beta_2^l <_{\pi_2} d_2^r <_{\pi_2} y_2^l$, y is again adjacent to β. If $i = 1$, $\beta \in T_1$ is adjacent to x, which implies that $\beta_1^l <_{\pi_1} x_1^r$ or $\beta_2^l <_{\pi_1} x_2^r$. As above, it follows that β is adjacent to y. Thus, in every case, $y \in T^{i+1}$.

Conversely, let $y \in T^{i+1}$. From Property 3 of the invariant, y is white, which is also clearly the case if $i = 1$, since $y \notin N(x)$. Suppose for contradiction that y has no endpoint in $\Gamma_1 \cup \Gamma_2$. Since y is not adjacent to x, the two endpoints of y are on the same side of Γ_1 (resp. Γ_2) in π_1 (resp. π_2). Moreover, the endpoints of y cannot lie on one side of Γ_1 in π_1 and on the other side of Γ_2 in π_2. Since, from Property 2 of the invariant, vertices of T^i have both upper endpoints in Γ_1 and both lower endpoints in Γ_2, it follows that y is not adjacent to any vertex of T^i. This is a contradiction with the fact that $y \in T^{i+1}$. Thus, y has at least one endpoint in $\Gamma_1 \cup \Gamma_2$. □

Let us now prove that Invariant 1 holds for any $i \geq 2$. Let $i \geq 1$ such that either $i = 1$ or the invariant holds, we show that the invariant holds for $i + 1$. Since Property 2 holds at the beginning of the i^{th} iteration of the loop and since I is set to the old value of Γ at the end of the loop, it follows that Property 1 holds at the end of the i^{th} iteration. ¿From Lemma 1, the white vertices encountered during the i^{th} iteration of the loop are exactly the vertices of T^{i+1}. Moreover, at the end of the loop, Γ is assigned the value of Γ_{new} that has been computed by the calls to $update(\Gamma_{new}, y)$ at Line 4 of `PutInTree` when the endpoint of a new white vertex is encountered. Consequently, at the end of the i^{th} iteration, Γ_1 and Γ_2 are the least intervals containing both endpoints of vertices of T^{i+1}, respectively in π_1 and π_2. Thus, Property 2 holds. Lemma 1 ensures that, at the beginning of the i^{th} iteration, the set of vertices with at least one endpoint in $\Gamma_1 \cup \Gamma_2$ is $\bigcup_{0 \leq j \leq i+1} T^j$. Since, at the end of the loop, I is set to the old value of Γ and vertices of T^{i+1} are colored in gray, then Property 3 holds at the end of the i^{th} iteration. And so does the invariant at the beginning of the $i+1^{th}$ iteration, which proves that it holds for any $i \geq 2$.

Invariant 2 claims that function *ord* gives the order in which the standard BFS algorithm discovers vertices of the graph. In order to prove it, we need the following lemma, which characterizes the neighborhood in T^i of the vertices of T^{i+1} during the i^{th} execution of the main loop.

Lemma 2. *Let $i \geq 1$, let y be a vertex of T^{i+1} and z be a vertex of T^i. y and z are adjacent iff there exists some call to `PutInTree`$(u_k^{side}, v_k^t, \pi_k, side, Q)$ occurred during the i^{th} iteration of the main loop such that the non-*side *endpoint of y and the* side *endpoint of z both lie between u_k^{side} and v_k^t in π_k, and z's endpoint is visited before y's.*

Before proving Lemma 2, it is worth to emphasize on the fact that during a call to `PutInTree`$(u_k^{side}, v_k^t, \pi_k, side, Q)$, we examine only the *side* endpoints of gray vertices (Line 5) and the non-*side* endpoints of white vertices (Line 3). All nodes encountered, white or gray, can be correctly handled regardless of their other endpoint. This is possible because the non-examined endpoint appears only in calls where the examined endpoint does as well, as stated by the following remark.

Remark: During any execution of `PutInTree`$(u_k^{side}, v_k^t, \pi_k, side, Q)$, if the non-*side* (resp. *side*) endpoint of a gray (resp. white) vertex is encountered, then its *side* (resp. non-*side*) endpoint is encountered too during the same execution of `PutInTree`.

Indeed, consider, without loss of generality, the case of the first call to `PutInTree` (Line 7 of `BuildTree`) between α_1^l and a_1^{l-}. If the right endpoint z_1^r of a gray vertex z is encountered, since all the endpoints of gray vertices in π_1 lie within Γ_1 (see Invariant 1), then necessarily the left endpoint z_1^l of z is such that $\alpha_1^l \leq_{\pi_1} z_1^l <_{\pi_1} z_1^r$ and therefore has been encountered before during the current execution of `PutInTree`. Similarly, let y be a white vertex whose left endpoint y_1^l is encountered. We have $\alpha_1^l <_{\pi_1} y_1^l <_{\pi_1} a_1^l$. Since y is white, $y \in T^{i+1}$ and y is not adjacent to x. Consequently, if $i = 1$, since $a_1^l = x_1^l$, we have $y_1^r <_{\pi_1} a_1^l$. If $i \geq 2$, since y has no endpoint in I_1 (see Invariant 1) and is not adjacent to x, we have again $y_1^r <_{\pi_1} a_1^l$. Thus, in every case, the right endpoint y_1^r of y is encountered later in the current execution of `PutInTree`.

Proof of Lemma 2: For any $i \geq 1$, vertices having an endpoint in $I_1 \cup I_2$ are gray at the beginning of the i^{th} iteration of the loop. Since $y \in T^{i+1}$ is white, y has no endpoints in $I_1 \cup I_2$. Moreover, since $y \in T^{i+1}$ and $i + 1 \geq 2$, y is not adjacent to x. It follows that either $b_1^r <_{\pi_1} y_1^l$ and $d_2^r <_{\pi_2} y_2^l$, or $y_1^r <_{\pi_1} a_1^l$ and $y_2^r <_{\pi_2} c_2^l$. Let us consider the first case, the second one is symmetric.

If y and z are adjacent, then clearly $y_1^l <_{\pi_1} z_1^r$ or $y_2^l <_{\pi_2} z_2^r$. Without loss of generality, assume that $y_1^l <_{\pi_1} z_1^r$. Since $z \in T^i$, from Property 2 of Invariant 1, $z_1^r \in \Gamma_1$, and so $y_1^l \in \Gamma_1$. Then, the endpoints of y and z are encountered in a same execution of `PutInTree`, and z is visited first within this execution.

Conversely, if y and z are not adjacent, the trapezoid of z lies either entirely to the right of the trapezoid of y, or entirely to its left. The former case is impossible as $z \in T^i$ is adjacent to some vertex of T^{i-1}, whose trapezoid, which is included in $I_1 \cup I_2$ from Property 1 of Invariant 1, lies entirely to the left of y. In the latter case, where the trapezoid of z lies entirely to the left of the trapezoid of y, no endpoint of z is visited before an endpoint of y in any execution of Routine `PutInTree`, which ends the proof of the lemma. □

Invariant 2. *For any $i \geq 1$, at the beginning of the i^{th} iteration of the main loop, the subset O^{i-1} of vertices of T^{i-1} for which ord is defined contains all the vertices of T^{i-1} that have children in T. And the order induced by ord on vertices of O^{i-1} is exactly the order of visit of those vertices by the standard BFS algorithm.*

Proof. Since $ord(x)$ is initialized to 1 at Line 1 of Routine `BuildTree`, the invariant clearly holds for $i = 1$. Suppose it holds for some $i \geq 1$. Then, since *ord* is defined for every parent of the vertices of T^i, order $<_{prior}$ is defined for all vertices of T^i and is precisely the BFS order of visit of those vertices. And since, during the i^{th} iteration of the loop, *AssignOrd* assigns the values of *ord* to a subset, denoted O^i, of vertices of T^i by increasing $<_{prior}$, it follows that the order induced by *ord* on O^i is the order of visit by the standard BFS algorithm.

Now, let $z \in T^i \setminus O^i$, we will show that z has no children in T. In other words, we want to prove that z is not the parent of any of its neighbors $y \in T^{i+1}$. From Lemma 2, since y and z are adjacent, they are encountered in a same execution of `PutInTree` occurred during the i^{th} iteration of the main loop. Assume without loss of generality that it was in the first call to `PutInTree` (Line 7 of `BuildTree`), between α_1^l and a_1^{l-}, the other cases are similar. By definition, since $z \notin O^i$, it is not assigned a value for *ord* in the i^{th} iteration of the main loop. Then, necessarily, z is not placed in Q at Line 6 of `PutInTree`, which implies that the test of Line 5 is negative. Since $z \in T^i$, z is already gray when this test is performed. It follows that the test is negative because there exists some gray vertex $u \in T^i$ such that $\alpha_1^l \leq_{\pi_1} u_1^l <_{\pi_1} z_1^l$ and $u <_{prior} z$. Lemma 2 states that the *side* endpoint of z is visited before the non-*side* endpoint of y in the considered execution of `PutInTree`. That is, in the present case, $z_1^l <_{\pi_1} y_1^r$. This implies that $u_1^l <_{\pi_1} y_1^r$, and Lemma 2 concludes that y is also adjacent to u. Finally, since $u <_{prior} z$, z is not the parent of y. □

We can now prove the correctness of our algorithm. Clearly, the first two levels of the tree (T^0 and T^1) are properly built by the initialization phase of Routine `BuildTree` (Line 1 to 4). Lemma 1 shows that, for all $i \geq 1$, the vertices discovered during the i^{th} iteration of the main loop of `BuildTree` are exactly those of T^{i+1}. In order to complete the proof of correctness, we have to show that these vertices are assigned their correct parent in T. Parent assignments occur at Line 4 of Routine `PutInTree`$(u_k^{side}, v_k^t, \pi_k, side, Q)$. Every vertex of T^{i+1} (white vertices) is assigned a parent. Indeed, when the non-*side* endpoint of a white vertex y is encountered, the test of Line 3 is true iff $p <_{prior} parent(y)$; and in particular, this is true if y has no parent (i.e. $parent(y) = \perp$), since, by definition, $\perp$ is the greatest element for $<_{prior}$.

The parent assigned to vertex y is p, which is initialized with u at Line 1 and is maintained to be the least gray node for order $<_{prior}$ among the gray nodes whose *side* endpoint has been visited so far during the current execution of `PutInTree` (Lines 5 and 6). From its definition, and from Invariant 2, order $<_{prior}$ is exactly the order in which the standard BFS algorithm discovers the vertices of T^i. Then, from Lemma 2, p is the first vertex visited by the standard BFS algorithm among the vertices of $T^i \cap J$ adjacent to x, where J is the set of vertices with an endpoint lying between u_k^{side} and v_k^t in π_k. The non-*side* endpoint of a white vertex y may be encountered in two different executions of Routine `PutInTree`, once in each of the two orders π_1 and π_2. The test $p <_{prior} parent(y)$ of Line 3 of `PutInTree` guarantees that it is assigned a new parent p iff p has been visited by the standard BFS algorithm before the previously assigned parent of y in T. Since from Lemma 2, every neighbor $z \in T^i$ of y is encountered in some call to `PutInTree` where y is encountered too, it follows that y is assigned as parent the first vertex visited by the standard BFS algorithm among vertices of $T^i \cap N(y)$. Thus, the tree T produced by our algorithm is the correct BFS-tree.

3.3. Computing all shortest paths

In the standard algorithm, it is possible to obtain, within $O(n + m)$ time complexity, a data-structure encoding all the shortest paths from the root x of T to the other vertices of the graph. This is done by storing, for each vertex y, the subset S_y of its neighbors that belong to a shortest path from y to x. This set S_y is nothing but the set of vertices of level T^{i-1}, with T^i being the level of y, that are adjacent to y. In the standard BFS algorithm, set S_y is stored in a list, and the total space needed to do so for all vertices of the graph is $O(n + m)$.

In our case, due to the structure of the intersection model, sets S_y can be stored more efficiently, within a total $O(n)$ space. Moreover, this data-structure can be computed without penalizing the complexity of our algorithm.

Indeed, from Lemma 2, set S_y is exactly the set of vertices of T^{i-1} whose *side* endpoint is encountered before the non-*side* endpoint of y in each of the executions of `PutInTree` where y is encountered, during the $i - 1^{th}$ iteration of the main loop. Thus, we can store S_y in the following way. For each level T^i, we build four lists L_c^i, with $c \in [\![1, 4]\!]$, corresponding to the four calls to `PutInTree` occurred during the $i - 1^{th}$ iteration. We build L_c^i by pushing on a stack (different for each call) each vertex of T^{i-1} whose *side* endpoint is encountered during execution number c of `PutInTree`. When we find the non-*side* endpoint of some vertex y of T^i, y is assigned a pointer to the cell of list L_c^i (under construction) which is the current top of the stack. Doing so, we are able to know which are the vertices of T^{i-1} whose *side* endpoint has been encountered before the non-*side* endpoint of y in the present execution of `PutInTree`. In this way, at the end of the computation, we can get set S_y in $O(|S_y|)$ time by merging the at most two partial lists given by the pointers assigned to y towards lists L_c^i, $c \in [\![1, 4]\!]$.

It is worth to note that set S_y can be handled by a constant space description consisting of the mentioned (at

most) two pointers. The total size of the four lists assigned to level T^i is dominated by the number of vertices in T^{i-1}. Thus the size of the additional space needed to store the whole data structure is $O(n)$.

In practice, it may be desirable to follow the shortest paths in the other way: from the root of T to the other vertices of G. To this purpose one must store, for each vertex at level i, its set of neighbors at level $i+1$. This can be done very similarly to what precedes. During the i^{th} iteration of the loop, for each execution of `PutInTree`, build the list of white vertices encountered by pushing them on a stack. When a gray vertex z is encountered, assign to it a pointer to the top of the stack: the neighbors of z at level $i+1$ are the white vertices that are above this pointer in the stack (Lemma 2).

3.4. Complexity

First of all, note that the computation of the lists storing all shortest paths from nodes of the graphs to the root x of T only requires constant additive extra-time in the treatment of an endpoint. This does not change the overall complexity of the algorithm and, for sake of simplicity, we do not take it into account anymore in the complexity analysis.

The complexity of Routine `BuildTree` is $O(n)$. Let B denote $(\Gamma_1 \setminus I_1) \cup (\Gamma_2 \setminus I_2)$. At Line 3, we can get $N(x)$ in $O(n)$ time, by scanning π_1 and π_2, and the running time of the initialization loop (Lines 3 and 4) is $O(|N(x)|) = O(n)$. In Routine `PutInTree`$(u_k^{side}, v_k^t, \pi_k, side, Q)$, all instructions take $O(1)$ time, and the routine runs in $O(|[\![u_k^{side}, v_k^t]\!]|)$ time. This gives an $O(|B|)$ time bound for the four calls of Lines 7 to 10 of `BuildTree`. Coloring list Ex also takes $O(|B|)$ time. The complexity of procedure *AssignOrd* is a crucial point. It is important to note that any list $Q' \in \{Q_1^l, Q_1^r, Q_2^l, Q_2^r\}$ being one of its arguments is already sorted according to order $<_{prior}$. 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, using classic technique for union of sorted lists. Finally, the running time of an iteration of the main 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 $O(n)$ time.

4. Conclusion and Perspectives

We showed that it is possible to achieve a complete and unrestricted BFS of a trapezoid graph in $O(n)$ time. It opens the way to improvement and simplification of several problems on this class of graphs, which make use of a BFS step in their solutions. Let us cite as example *All Pairs Shortest Paths* [10] and its variations (*Next-to-Shortest* [1], *Almost Shortest* [5]), *biconnected components* [8] and *betweeness* [2].

Another perspective is to generalize these results to other graph classes such as circular-arc graphs and circle graphs for example, which are the circular generalizations of respectively interval graphs and permutation graphs. We believe that only few work is left to be done for solving the problem in $O(n)$ time for circular-arc graphs, using the fact that deleting a vertex and its neighborhood from a circular-arc graph results in an interval graph. On the other hand, the case of circle graphs seems to be challenging, and solving it would certainly lead to new insight on the structure of this class of graphs.

[1] S. C. Barman, S. Mondal, and M. Pal. An efficient algorithm to find next-to-shortest path on trapezoid graphs. *Advances in Applied Mathematical Analysis*, 2(2):97–107, 2007.

[2] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[3] H. S. Chao, F. R. Hsu, and R. C. T. Lee. An optimal EREW parallel algorithm for computing breadth-first search trees on permutation graphs. *Information Processing Letters*, 61(6):311–316, 1997.

[4] H.-C. Chen and Y.-L. Wang. A linear time algorithm for finding depth-first spanning trees on trapezoid graphs. *Information Processing Letters*, 63(1):13–18, 1997.

[5] F. F. Dragan. Estimating all pairs shortest paths in restricted graph families: A unified approach. *Journal of Algorithms*, 57(1):103–116, 2005.

[6] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, volume 57 of *Annals of Discrete Mathematics*. Elsevier, second edition, 2004.

[7] Y. D. Liang. Steiner set and connected domination in trapezoid graphs. *Information Processing Letters*, 56:101–108, 1995.

[8] Y. D. Liang and C. Rhee. Finding biconnected components in $O(n)$ time for a class of graphs. *Information Processing Letters*, 60:159–163, 1996.

[9] T.-H. Ma and J.P. Spinrad. An $O(n^2)$ time algorithm for the 2-chain cover problem and related problems. *Proceedings of the Second Symposium on Discrete Algorithms*, pages 363–372, 1991.

[10] S. Mondal, M. Pal, and T.K. Pal. An optimal algorithm for solving all-pairs shortest paths on trapezoid graphs. *International Journal of Computational Engineering Science*, 3(2):103–116, 2002.

[11] C. Rhee, Y. Daniel Liang, S.K. Dhall, and S. Lakshmivarahan. Efficient algorithms for finding depth-first and breadth-first search trees in permutation graphs. *Information Processing Letters*, 49:45–50, 1994.