From Pathwidth to Connected Pathwidth

Dariusz Dereniowski[*1]

1 **Department of Algorithms and System Modeling, Gdańsk University of Technology, Narutowicza 11/12, 80-233 Gdańsk, Poland**
`deren@eti.pg.gda.pl`

Abstract

It is proven that the connected pathwidth of any graph G is at most $2 \cdot \mathrm{pw}(G) + 1$, where $\mathrm{pw}(G)$ is the pathwidth of G. The method is constructive, i.e. it yields an efficient algorithm that for a given path decomposition of width k computes a connected path decomposition of width at most $2k + 1$. The running time of the algorithm is $O(dk^2)$, where d is the number of 'bags' in the input path decomposition.

The motivation for studying connected path decompositions comes from the connection between the pathwidth and some graph searching games. One of the advantages of the above bound for connected pathwidth is an inequality $\mathtt{cs}(G) \leq 2\mathtt{s}(G) + 3$, where $\mathtt{cs}(G)$ is the connected search number of a graph G and $\mathtt{s}(G)$ is its search number, which holds for any graph G. Moreover, the algorithm presented in this work can be used to convert efficiently a given search strategy using k searchers into a connected one using $2k + 3$ searchers and starting at arbitrary homebase.

1998 ACM Subject Classification G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases connected pathwidth, connected searching, fugitive search games, graph searching, pathwidth

Digital Object Identifier 10.4230/LIPIcs.STACS.2011.416

1 Introduction

The notions of pathwidth and treewidth are receiving increasing interest since the series of Graph Minor articles by Robertson and Seymour. The importance of those parameters is due to their numerous practical applications, connections with several graph parameters and usefulness in designing graph algorithms. Informally speaking, the *pathwidth* of a graph G, denoted by $\mathrm{pw}(G)$, says how closely G is related to a path. Moreover, a path decomposition captures the linear path-like structure of G. (For a definition see Section 2.)

Here we briefly describe a graph searching game that is the main motivation for the results presented in this paper. A team of k searchers is given and the goal is to find an invisible and fast fugitive located in a given graph G. The fugitive has also the complete knowledge about the graph and about the strategy used by the searchers, and therefore he will avoid being captured as long as possible. The fugitive is captured when a searcher reaches his location. In this setting the game is equivalent to the problem of clearing all edges of a graph that is initially entirely contaminated. There are two main types of this graph searching problem. In the *node searching* two moves are allowed: placing a searcher on a vertex and removing a searcher from a vertex. An edge becomes clear whenever both of its

* Research partially supported by MNiSW grant N N206 379337 (2009-2011).

© Dariusz Dereniowski;
licensed under Creative Commons License NC-ND
28th Symposium on Theoretical Aspects of Computer Science (STACS'11).
Editors: Thomas Schwentick, Christoph Dürr; pp. 416–427
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCE

endpoints are simultaneously occupied by searchers. In the *edge searching* we have, besides to the two mentioned moves, a move of sliding a searcher along an edge. In this model an edge $\{u, v\}$ becomes clear if a searcher slides from u to v and either all the other edges incident to u have been previously cleared or another searcher occupies u. In both cases the goal is to find a search strategy (a sequence of moves of the searchers) that clears all the edges of G. The *node* (*edge*) *search number* of G, denoted by $\mathtt{ns}(G)$ ($\mathtt{s}(G)$, respectively), equals the minimum number of searchers sufficient to construct a node (edge, respectively) search strategy. An important property is that $\mathrm{pw}(G) = \mathtt{ns}(G) - 1$ for any graph G [13, 14, 15, 18]. The edge searching problem is closely related to node searching, i.e. $|\mathtt{s}(G) - \mathtt{ns}(G)| \leq 1$ [4], and consequently to pathwidth, $\mathrm{pw}(G) \leq \mathtt{s}(G) \leq \mathrm{pw}(G) + 2$.

In this work we are interested in special types of path decompositions called connected path decompositions. The motivation comes from the need of creating connected search strategies. An edge search strategy is *connected* if the subgraph of G that is clear is always connected. The minimum number of searchers that guarantees the capture of the fugitive by a connected (edge) search strategy, denoted by $\mathtt{cs}(G)$, is the *connected search number* of G. This model of graph searching receives recently a growing interest, because in many applications the connectedness is an requirement.

Related work. There are several results that give a relation between the connected and the 'classical' search numbers of a graph. Fomin et al. proved in [7] that the connected search number of an n-node graph of branchwidth b is bounded by $O(b \log n)$ and this bound is tight. One of the implications of this result is that $\mathtt{cs}(G) = O(\log n)\mathrm{pw}(G)$. Nisse proved in [19] that $\mathtt{cs}(G) \leq (\mathrm{tw}(G)+2)(2\mathtt{s}(G)-1)$ for any chordal graph G. Barrière et al. obtained in [2] a constant upper bound for trees, namely for each tree T, $\mathtt{cs}(T)/\mathtt{s}(T) \leq 2$. On the other hand, there exists an infinite family of graphs G_k such that $\mathtt{cs}(G_k)/\mathtt{s}(G_k)$ approaches 2 when k goes to infinity [3].

Fraigniaud and Nisse presented in [9] a $O(nk^3)$-time algorithm that takes a width k tree decomposition of a graph and returns a connected tree decomposition of the same width. (For definition of treewidth see e.g. [5, 21].) Therefore, $\mathrm{tw}(G) = \mathrm{ctw}(G)$ for any graph G. This method cannot be applied for proving the same result for connected path decompositions, because the decomposition that the algorithm in [9] constructs is not, in general, a path decomposition even when a path decomposition is given as an input. That result also yields an upper bound of $\mathtt{cs}(G) \leq (\log n+1)\mathtt{s}(G)$ for any graph G. The problems of computing the pathwidth (the search number) and the connected pathwidth (the connected search number) are NP-hard, also for several special classes of graphs, see e.g. [6, 11, 12, 16, 17, 20].

This work. This paper presents an efficient algorithm that takes a (connected) graph G and its path decomposition $\mathcal{P} = (X_1, \ldots, X_d)$ of width k as an input and finds in time $O(dk^2)$ a connected path decomposition $\mathcal{C} = (Z_1, \ldots, Z_m)$ of width at most $2k+1$, where $m \leq kd$. This solves an open problem stated in several papers, e.g. in [1, 2, 3, 7, 8, 9, 10, 22], since it implies that for any graph G, $\mathrm{cpw}(G) \leq 2\mathrm{pw}(G) + 1$, and improves previously known estimations [7, 19]. The path decomposition $\mathcal{C}$ can be turned into a monotone connected search strategy using at most $2k+3$ searchers. Thus, in terms of the graph searching terminology, the bound immediately implies that $\mathtt{mcs}(G) \leq \mathrm{cpw}(G)+2 \leq 2\mathrm{pw}(G)+3 \leq 2\mathtt{s}(G)+3$, where $\mathtt{mcs}(G)$ is the monotone connected search number of G. (A search strategy is *monotone* if the fugitive cannot reach a previously cleared edge.) Since $\mathtt{cs}(G) \leq \mathtt{mcs}(G)$, the bound can be restated for the connected search number of a graph, $\mathtt{cs}(G) \leq 2\mathtt{s}(G)+3$. Moreover, the factor 2 in the bound is tight [3]. The bound finds also applications in designing approximation algorithms, for it implies that the pathwidth and the connected pathwidth (the search

number, the connected search number, and some other search numbers not mentioned here, e.g. the internal search number) are within a constant factor of each other.

2 Preliminaries and basic definitions

Given a simple graph $G = (V(G), E(G))$ and its subset of vertices $X \subseteq V(G)$, the subgraph of G *induced* by X is $G[X] = (X, \{\{u, v\} \in E(G) \colon u, v \in X\})$. For a simple (not necessary connected) graph G, H is its connected component if H is connected, that is, there exists a path in H between each pair of vertices, and each proper supergraph of H is not a subgraph of G. For $X \subseteq V(G)$ let $N_G(X) = \{u \in V(G) \setminus X \colon \{u, x\} \in E(G) \text{ for some } x \in X\}$.

▶ **Definition 1.** A *path decomposition* of a simple graph $G = (V(G), E(G))$ is a sequence $\mathcal{P} = (X_1, \ldots, X_d)$, where $X_i \subseteq V(G)$ for each $i = 1, \ldots, d$, and

- $\bigcup_{i=1,\ldots,d} X_i = V(G)$,
- for each $\{u, v\} \in E(G)$ there exists $i \in \{1, \ldots, d\}$ such that $u, v \in X_i$,
- for each i, j, k, $1 \le i \le j \le k \le d$ it holds $X_i \cap X_k \subseteq X_j$.

The *width* of the path decomposition $\mathcal{P}$ is $\mathrm{width}(\mathcal{P}) = \max_{i=1,\ldots,d} |X_i| - 1$. The *pathwidth* of G, $\mathrm{pw}(G)$, is the minimum width over all path decompositions of G.

A path decomposition $\mathcal{P}$ is *connected* if $G[X_1 \cup \cdots \cup X_i]$ is connected for each $i = 1, \ldots, d$. Then, $\mathrm{cpw}(G)$ denotes the minimum width over all connected path decompositions of G.

▶ **Definition 2.** Given a graph G and its path decomposition $\mathcal{P} = (X_1, \ldots, X_d)$, a node-weighted graph $\mathcal{G} = (V(\mathcal{G}), E(\mathcal{G}), \omega)$ *derived* from G and $\mathcal{P}$ is the graph with vertex set $V(\mathcal{G}) = V_1 \cup \cdots \cup V_d$, where $V_i = \{v_i(H) \colon H \text{ is a connected component of } G[X_i]\}$, $i = 1, \ldots, d$, and edge set $E(\mathcal{G}) = \{\{v_i(H), v_{i+1}(H')\} \colon v_i(H) \in V_i, v_{i+1}(H') \in V_{i+1}, i \in \{1, \ldots, d - 1\}, \text{ and } V(H) \cap V(H') \neq \emptyset\}$. The weight of a vertex $v_i(H) \in V(\mathcal{G})$, $i \in \{1, \ldots, d\}$, is $\omega(v_i(H)) = |V(H)|$. The *width* of $\mathcal{G}$, denoted by $\mathrm{width}(\mathcal{G})$, equals $\mathrm{width}(\mathcal{P}) + 1$.

In the following we omit a subgraph H of G and the index $i \in \{1, \ldots, d\}$ whenever they are not important when referring to a vertex of $\mathcal{G}$ and we write v instead of $v_i(H)$. For brevity, $\omega(X) = \sum_{x \in X} \omega(x)$ for any subset $X \subseteq V(\mathcal{G})$.

Figures 1(a) and 1(b) present a graph G and its path decomposition $\mathcal{P}$, respectively, where the subgraph structure in each bag X_i is also given. Figure 1(c) depicts the derived graph $\mathcal{G}$. Note that $\mathcal{P}$ is not connected: the subgraphs $G[X_1 \cup \cdots \cup X_i]$ are not connected for $i = 2, 3, 4$. Let $C \subseteq V(\mathcal{G})$. The *border* $\delta(C)$ of the set C is its subset consisting of all the vertices $v \in C$ such that there exists $u \in V(\mathcal{G}) \setminus C$ adjacent to v in $\mathcal{G}$, i.e. $\delta(C) = N_{\mathcal{G}}(V(\mathcal{G}) \setminus C)$.

Given a set $X \subseteq V(\mathcal{G})$, $X \neq \emptyset$, we define the *left* (*right*) *extremity* of X as $l(X) = \min\{i \colon V_i \cap X \neq \emptyset\}$ ($r(X) = \max\{i \colon V_i \cap X \neq \emptyset\}$, respectively).

A path P in $\mathcal{G}$ is *progressive* if $|V(P) \cap V_i| \le 1$ for each $i = 1, \ldots, d$.

▶ **Definition 3.** Given $\mathcal{G}$, $C \subseteq V(\mathcal{G})$ and $X \subseteq \delta(C)$, a *left* (*right*) *branch* $\mathcal{B}_{\mathrm{L}}(C, X, i)$, where $1 \le i \le r(X)$ (respectively $\mathcal{B}_{\mathrm{R}}(C, X, i)$, where $l(X) \le i \le d$) is the subgraph of $\mathcal{G}$ induced by the vertices in X and by the vertices of all progressive paths contained in $(V(\mathcal{G}) \setminus C) \cup X$ and connecting $x \in X \cap V_j$ and $v \in V_k \setminus C$, where $i \le k \le j$ ($j \le k \le i$, respectively).

We sometimes write $\mathcal{B}$ to refer to a branch whenever its 'direction' or C, X, i are clear from the context. A branch $\mathcal{B} = \mathcal{B}_{\mathrm{L}}(C, X, i)$, $i \le r(X)$, ($\mathcal{B} = \mathcal{B}_{\mathrm{R}}(C, X, i)$, $i \ge l(X)$) is *continuous* if $V_j \cap V(\mathcal{B}) \neq \emptyset$ for each $j = i, \ldots, r(X)$ ($j = l(X), \ldots, i$, respectively). A vertex v of $\mathcal{B}$ is *external* if $N_{\mathcal{G}}(v) \nsubseteq C \cup V(\mathcal{B})$. The branch $\mathcal{B}$ is *proper* if it has no external vertices in

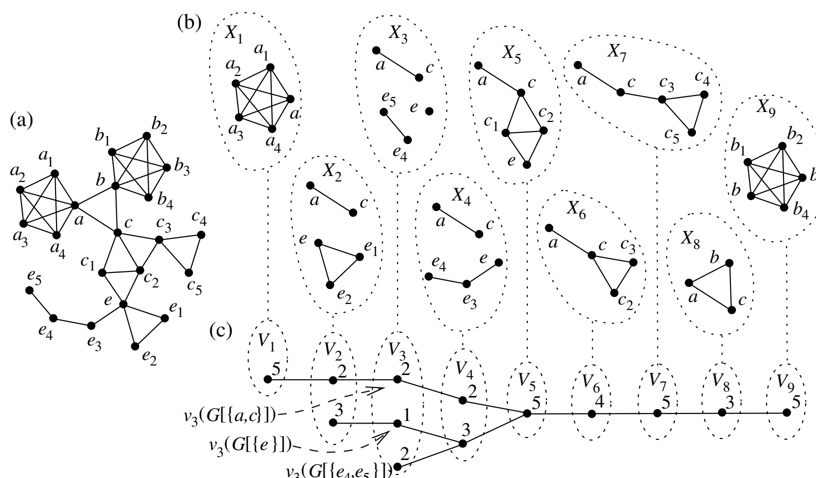

Figure 1 (a) a graph G; (b) a path decomposition $\mathcal{P}$ of G; (c) the graph $\mathcal{G}$ derived from G and $\mathcal{P}$

$V_{i+1} \cup \cdots \cup V_{r(X)-1}$ ($V_{l(X)+1} \cup \cdots \cup V_{i-1}$, respectively), while $\mathcal{B}$ is *maximal* if it is continuous, proper and $\mathcal{B}_{\mathrm{L}}(C, X, i-1)$ ($\mathcal{B}_{\mathrm{R}}(C, X, i+1)$, respectively) is not a proper branch. An integer j is a *cut* of the branch $\mathcal{B}$ if $i \leq j \leq r(X)$ ($l(X) \leq j \leq i$, respectively). The *weight* of the cut j of $\mathcal{B}$ is $\omega((V(\mathcal{B}) \cap V_j) \cup (X \cap Y))$, where $Y = V_{l(X)} \cup \cdots \cup V_{j-1}$ for a left branch, and $Y = V_{i+1} \cup \cdots \cup V_{r(X)}$ for a right branch. A cut of minimum weight is a *bottleneck* of $\mathcal{B}$.

Figure 2 illustrates the above definitions. (In all cases the branch is distinguished by the dark area.) Let $X = \{x_1, x_2, x_3\}$ be a subset of $\delta(C)$. Figure 2(a) gives $\mathcal{B}_{\mathrm{L}}(C, X, i_2)$ and this branch is continuous and proper, but not maximal for each i_2, $i_3 < i_2 \leq i_1$. The branch $\mathcal{B}_{\mathrm{L}}(C, X, i_3)$ (see Figure 2(b)) is maximal (thus continuous and proper), which follows from the fact that any branch $\mathcal{B}_{\mathrm{L}}(C, X, i_4)$, where $i_4 < i_3$ is not proper, because it contains an external vertex u_2, as shown in Figure 2(c). Note that the vertices of $\mathcal{G}_1$ and $\mathcal{G}_2$ (except for u_1 and u_2) do not belong to any branch $\mathcal{B}_{\mathrm{L}}(C, X, i)$, because they are not connected by progressive paths to x_2 or x_3. Figure 2(d) depicts a branch $\mathcal{B}_{\mathrm{L}}(C, X, i_7)$ that is not continuous for any $i_7 < i_5$, because $V_{i_5-1} \cap V(\mathcal{B}_{\mathrm{L}}(C, X, i_7)) = \emptyset$. In our algorithm we ensure that each branch we use is continuous and proper.

Given C, X and i, a left (respectively right) branch $\mathcal{B}$ can be calculated efficiently as follows. Initially $\mathcal{B}$ satisfies $V(\mathcal{B}) = X$. We start with $j = r(X)$ ($j = l(X)$, resp.) and we add to the vertex set of the branch all vertices in $V_{j-1} \setminus C$ ($V_{j+1} \setminus C$, resp.) that have a neighbor in $V_j \cap V(\mathcal{B})$. Then, we decrement (increment, resp.) j and repeat this step. The computation stops when $j < i$ ($j > i$, respectively).

3 The algorithm

The algorithm `CP` (*Connected Pathwidth*) for finding a connected path decomposition of a graph G takes G, a vertex v of G, and a path decomposition $\mathcal{P}$ of G as an input. The vertex v is guaranteed to belong to the first bag of the resulting connected path decomposition. This flexibility is provided due to the potential application of this algorithm: in graph searching games the bags of path decompositions correspond to the vertices occupied by the searchers while the search proceeds; in this way the selected vertex v can be the first one that becomes guarded in a connected search strategy and it is called the *homebase*. The first step performed by `CP` is the construction of the derived graph $\mathcal{G}$ and in the subsequent

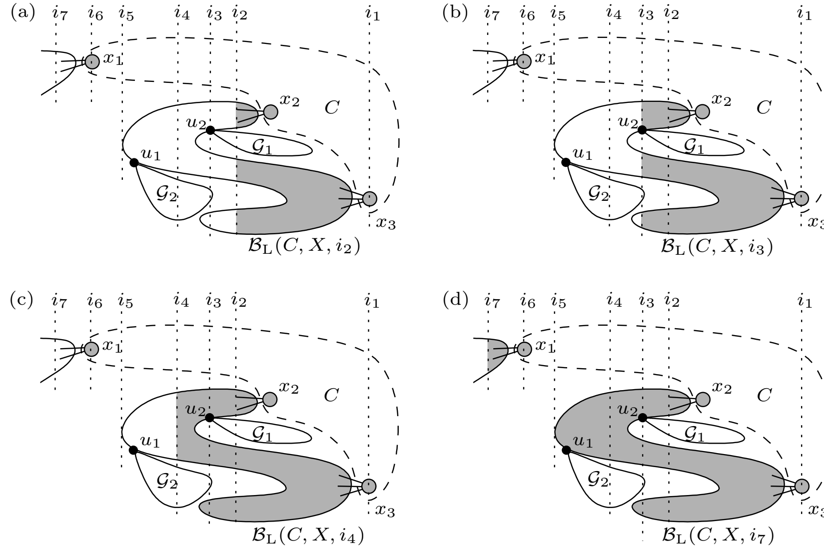

Figure 2 $\mathcal{G}$ with distinguished vertex sets $X = \{x_1, x_2, x_3\}$ and C, and the corresponding branches that are: (a) continuous and proper but not maximal; (b) maximal; (c) continuous but not proper (thus not maximal); (d) not continuous nor proper (thus not maximal)

steps the algorithm works on $\mathcal{G}$. (Also, most parts of our analysis use $\mathcal{G}$ rather than G.) The algorithm computes a sequence of sets $C_j \subseteq V(\mathcal{G})$, $j = 1, \ldots, m$, called *expansions*. The expansion C_1 consists of v and one of its neighbors, and $C_m = V(\mathcal{G})$ at the end of the execution of CP. Moreover, $C_j \subseteq C_{j+1}$ for each $j = 1, \ldots, m-1$. Informally speaking, C_{j+1} is obtained from C_j by adding to C_j some vertices from $N_{\mathcal{G}}(C_j)$. This guarantees that the final path decomposition obtained from $\delta(C_1), \ldots, \delta(C_m)$ is valid and is connected, as proved in Lemma 7. On the other hand, the particular vertices in $N_{\mathcal{G}}(C_j)$, used to obtain C_{j+1}, are selected in a way to guarantee that $\omega(\delta(C_j))$ is bounded by $2 \cdot \text{width}(\mathcal{G})$ for each $j = 1, \ldots, m$. By construction, $\omega(\delta(C_j))$ is the size of the corresponding jth bag in the resulting connected path decomposition.

In this section we give the statement of the algorithm and we prove that it computes a connected path decomposition $\mathcal{C}$. Then, in Section 4 we analyze the width of $\mathcal{C}$. Due to the space limitations, the proofs of several results (marked with $\triangle$) are omitted.

The algorithm computes for each expansion C_j two sets called the *left* and *right borders* of C_j, denoted by $\delta_{\rm L}(C_j)$ and $\delta_{\rm R}(C_j)$, respectively. It is guaranteed that $\delta_{\rm L}(C_j) \cup \delta_{\rm R}(C_j) = \delta(C_j)$ for each $j = 1, \ldots, m$ (see Lemma 6). As it is proven later, the left and right borders are special types of partitions of $\delta(C_j)$. In particular, there exists an integer $i \in \{1, \ldots, d\}$ such that the left border $\delta_{\rm L}(C_j)$ is contained in $V_1 \cup \cdots V_i$ and the right border $\delta_{\rm R}(C_j)$ is a subset of $V_{i+1} \cup \cdots \cup V_d$. For brevity let in the following $l(\delta_{\rm L}(C_j)) = r(\delta_{\rm L}(C_j)) = 0$ if $\delta_{\rm L}(C_j) = \emptyset$ and $l(\delta_{\rm R}(C_j)) = r(\delta_{\rm R}(C_j)) = d$ if $\delta_{\rm R}(C_j) = \emptyset$, where C_j is any expansion.

We start by describing a subroutine EE (*Extend Expansion*) that is used by the main procedure CP given below. The input to EE consists of two integers i and k, $i, k \in \{1, \ldots, d\}$. Informally speaking, the procedure adds, in its subsequent iterations, to the current expansion C_m each vertex in V_j that is connected by a progressive path to a vertex in $V_{j'} \cap \delta(C_m)$ for each $j = i$ to k and for some j', $i \leq j' \leq j$ if $i < k$, and for each $j = i$ down to k, $j \leq j' \leq i$ if $i > k$, which we formally prove in Lemma 4 below. All the 'intermediate' expansions are recorded as they will give us the corresponding bags in the final path decomposition. The procedures EE(i, k) and CP$(G, \mathcal{P})$ are as follows.

Procedure EE (*Extend Expansion*)
Input: integers i and k. ($\mathcal{G}$, m, C_m are used as global variables)
while $k \neq i$ **do**
if $k < i$ **then**
EL: Increment m, decrement i and set:
$C_m = C_{m-1} \cup (V_i \cap N_{\mathcal{G}}(C_{m-1}))$,
$\delta_{\mathrm{L}}(C_m) = (\delta_{\mathrm{L}}(C_{m-1}) \cup V_i) \cap \delta(C_m)$,
$\delta_{\mathrm{R}}(C_m) = \delta_{\mathrm{R}}(C_{m-1}) \cap \delta(C_m)$.
else ($k > i$)
ER: Increment m, increment i and set:
$C_m = C_{m-1} \cup (V_i \cap N_{\mathcal{G}}(C_{m-1}))$,
$\delta_{\mathrm{R}}(C_m) = (\delta_{\mathrm{R}}(C_{m-1}) \cup V_i) \cap \delta(C_m)$,
$\delta_{\mathrm{L}}(C_m) = \delta_{\mathrm{L}}(C_{m-1}) \cap \delta(C_m)$.
end if
end while.
end procedure EE.

Algorithm CP (*Connected Pathwidth*)
Input: a simple graph G, a path decomposition $\mathcal{P}$ of G, and a vertex $v \in V(G)$.
Output: a connected path decomposition $\mathcal{C}$ of G.

(*Initialization.*)

I.1: Use G and $\mathcal{P}$ to calculate the derived graph $\mathcal{G}$. Let v be any vertex of $\mathcal{G}$. Let $C_1 = \{x, y\}$, where $v \in C_1$, x, y are adjacent in $\mathcal{G}$, and $x \in V_i$, $y \in V_{i+1}$ for some $i \in \{1, \ldots, d-1\}$. Let $m = 1$.

I.2: If $x \in \delta(C_1)$, then set $\delta_{\mathrm{L}}(C_1) = \{x\}$, compute the maximal left branch $\mathcal{B}_{\mathrm{L}}(C_1, \delta_{\mathrm{L}}(C_1), a_0)$ with a bottleneck a'_0 ($a'_0 \geq a_0$) and with no external vertices in V_i and call EE(i, a'_0); otherwise $\delta_{\mathrm{L}}(C_1) = \emptyset$.

I.3: If $y \in \delta(C_1)$, then set $\delta_{\mathrm{R}}(C_1) = \{y\}$, compute the maximal right branch $\mathcal{B}_{\mathrm{R}}(C_1, \delta_{\mathrm{R}}(C_1), b_0)$ with a bottleneck b'_0 ($b'_0 \leq b_0$) and with no external vertices in V_{i+1} and call EE$(i+1, b'_0)$; otherwise $\delta_{\mathrm{R}}(C_1) = \emptyset$.

(*Main loop.*)
while $C_m \neq V(\mathcal{G})$ **do**
if $\omega(\delta_{\mathrm{L}}(C_m)) > \omega(\delta_{\mathrm{R}}(C_m))$ **then**

L.1: Compute the maximal left branch $\mathcal{B}_1 = \mathcal{B}_{\mathrm{L}}(C_m, \delta_{\mathrm{L}}(C_m), k_1)$. If $\mathcal{B}_1$ has no external vertex in V_i, $i = r(\delta_{\mathrm{L}}(C_m))$, then call EE$(r(\delta_{\mathrm{L}}(C_m)), k_1)$, otherwise let $k_1 = r(\delta_{\mathrm{L}}(C_m))$.

L.2: Compute the maximal right branch $\mathcal{B}_2 = \mathcal{B}_{\mathrm{R}}(C_m, \delta_{\mathrm{R}}(C_m) \cup (V_{k_1} \cap \delta_{\mathrm{L}}(C_m)), k_2)$. Let k'_2 be its minimum weight cut such that $k'_2 > k_1$. Call EE(k_1, k'_2).

L.3: If $r(\delta_{\mathrm{L}}(C_m)) = k_1$, then compute the maximal left branch $\mathcal{B}_3 = \mathcal{B}_{\mathrm{L}}(C_m, \delta_{\mathrm{L}}(C_m), k_3)$ with bottleneck k'_3 and call EE(k_1, k'_3).

else ($\omega(\delta_{\mathrm{L}}(C_m)) \leq \omega(\delta_{\mathrm{R}}(C_m))$)

R.1: Compute the maximal right branch $\mathcal{B}_1 = \mathcal{B}_{\mathrm{R}}(C_m, \delta_{\mathrm{R}}(C_m), k_1)$. If $\mathcal{B}_1$ has no external vertex in V_i, $i = l(\delta_{\mathrm{R}}(C_m))$, then call EE$(l(\delta_{\mathrm{R}}(C_m)), k_1)$, otherwise let $k_1 = l(\delta_{\mathrm{R}}(C_m))$.

R.2: Compute the maximal left branch $\mathcal{B}_2 = \mathcal{B}_{\mathrm{L}}(C_m, \delta_{\mathrm{L}}(C_m) \cup (V_{k_1} \cap \delta_{\mathrm{R}}(C_m)), k_2)$. Let k_2' be its minimum weight cut such that $k_2' < k_1$. Call $\mathtt{EE}(k_1, k_2')$.

R.3: If $l(\delta_{\mathrm{R}}(C_m)) = k_1$, then compute the maximal right branch $\mathcal{B}_3 = \mathcal{B}_{\mathrm{R}}(C_m, \delta_{\mathrm{R}}(C_m), k_3)$ with bottleneck k_3' and call $\mathtt{EE}(k_1, k_3')$.

end if

end while.

Let $Z_j = \bigcup_{v_k(H) \in \delta(C_j)} V(H)$ for each $j = 1, \ldots, m$. **Return** $\mathcal{C} = (Z_1, \ldots, Z_m)$.

end procedure CP.

First we briefly discuss the initialization stage of CP. In Step I.1 an expansion C_1 is constructed in such a way that it contains any two adjacent vertices (the adjacency guarantees the connectedness of the final path decomposition) such that one of them is the input vertex v. (W.l.o.g. v has a neighbor in G, because otherwise $\mathcal{G}$ contains a single vertex and therefore $\mathcal{P}$ is connected.) Steps I.2 and I.3 are symmetric. In Step I.2 (I.3) the algorithm finds the maximal left (right) branch 'emanating' from x (resp. y) provided that the vertex belongs to the border of C_1, otherwise the left (right, respectively) border is empty.

In the following, one *iteration* of CP or EE means one iteration of the 'while' loop in the corresponding procedure. Thus, in the case of CP, one iteration reduces to executing Steps L.1-L.3 or R.1-R.3 within the 'if' statement, while in the procedure EE one iteration results in executing the instructions in Step EL or in Step ER. We use the symbols $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$, a_0, a_0', b_0, b_0', k_i, k_i' to refer to the variables used in CP, where in the case of k_1 we refer to its value at the end of Step L.1 or Step R.1. In what follows we denote for brevity $\mathcal{B}_2' = \mathcal{B}_{\mathrm{R}}(C_m, \delta_{\mathrm{R}}(C_m) \cup (V_{k_1} \cap \delta_{\mathrm{L}}(C_m)), k_2')$ and $\mathcal{B}_3' = \mathcal{B}_{\mathrm{L}}(C_m, \delta_{\mathrm{L}}(C_m), k_3')$ if Steps L.1-L.3 have been executed in this particular iteration of CP, and $\mathcal{B}_2' = \mathcal{B}_{\mathrm{L}}(C_m, \delta_{\mathrm{L}}(C_m) \cup (V_{k_1} \cap \delta_{\mathrm{R}}(C_m)), k_2')$, $\mathcal{B}_3' = \mathcal{B}_{\mathrm{R}}(C_m, \delta_{\mathrm{L}}(C_m), k_3')$ otherwise (i.e. in Steps R.1-R.3). Informally speaking, $\mathcal{B}_2'$ and $\mathcal{B}_3'$ are the branches $\mathcal{B}_2$ and $\mathcal{B}_3$, respectively, restricted to the vertices up to the corresponding cut k_2' or k_3'. The vertex v selected to be in C_1 is called the *starting vertex*.

The branches are used in the subsequent iterations of the algorithm in the way presented in Figure 3, where C_{s_i} refers to the expansion obtained at the end of Step L.i of an iteration of CP, $i = 1, 2, 3$ (the execution of Steps R.1-R.3 is symmetric), and C_{s_0} is the expansion from the beginning of the iteration. First, a branch $\mathcal{B}_1$ is used to obtain C_{s_1} from C_{s_0} (during the execution of Step L.1 of CP). It holds in particular $C_{s_1} = C_{s_0} \cup V(\mathcal{B}_1)$, as stated in Lemma 5 below. It is guaranteed that $r(\delta_{\mathrm{L}}(C_{s_1})) \leq k_1$. The (external) vertices in $V(\mathcal{B}_1) \cap V_{k_1}$ have some neighbors in $V_{k_1+1} \setminus C_{s_1}$ and the algorithm calculates the right branch $\mathcal{B}_2'$ ('emanating' from k_1) in Step L.2. Its right extremity, k_2', may by strictly less than the left extremity of the new right border $\delta_{\mathrm{R}}(C_{s_2})$ if $\mathcal{B}_2'$ has no external vertices in $V_{k_2'}$. Finally, a branch $\mathcal{B}_3'$ is calculated in a symmetric way (this step is omitted if the vertices in $C_{s_1} \cap V_{k_1}$ have no neighbors in $V_{k_1-1} \setminus C_{s_1}$, and in such case $\delta_{\mathrm{L}}(C_{s_3}) \subseteq \delta_{\mathrm{L}}(C_{s_0})$).

The following lemmas are used to prove that the computation stops and they also demonstrate how the expansions change between the subsequent calls of EE.

▶ **Lemma 4.** *Given an expansion C_j and $X \subseteq \delta(C_j)$, after the execution of the ith iteration of the procedure* $\mathtt{EE}(r(X), k)$*, where $k \leq r(X)$ (respectively* $\mathtt{EE}(l(X), k)$*, where $k \geq l(X)$) it holds $C_{j+i} = C_j \cup V(\mathcal{B}_{\mathrm{L}}(C_j, X, r(X) - i))$ ($C_{j+i} = C_j \cup V(\mathcal{B}_{\mathrm{R}}(C_j, X, l(X) + i))$, respectively), $i \geq 1$.* △

▶ **Lemma 5.** *Let C_{s_0} be an expansion from the beginning of an iteration of* CP*, and let C_{s_i}, $i = 1, 2, 3$, be the expansions obtained at the end of Steps L.1, L.2 and L.3 or R.1, R.2*

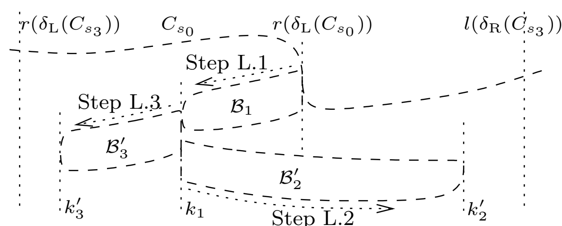

Figure 3 The expansion C_{s_3} obtained from C_{s_0} by including three branches $\mathcal{B}_1, \mathcal{B}'_2$ and $\mathcal{B}'_3$ calculated in Steps L.1,L.2 and L.3 of CP, respectively

and R.3 in this iteration, respectively. Then, $C_{s_1} = C_{s_0} \cup V(\mathcal{B}_1)$*,* $C_{s_2} = C_{s_1} \cup V(\mathcal{B}'_2)$ *and* $C_{s_3} = C_{s_2} \cup V(\mathcal{B}'_3)$*. Moreover,* $C_{s_3} \neq C_{s_0}$*.* △

Figure 4 gives an example of the execution of CP. In all cases (including the following figures) ♢ and □ are used to denote the vertices of the right and left borders, respectively. In particular Figure 4(a) presents a graph $\mathcal{G}$ and C_2 (this is the expansion obtained at the end of initialization of CP, where the starting vertex and its neighbor in C_1 are among the three vertices in C_2). Figures 4(b)-(d) depict the state of the algorithm at the end of the first three iterations. (The fourth iteration executes the Steps L.1-L.3, which ends the computation.)

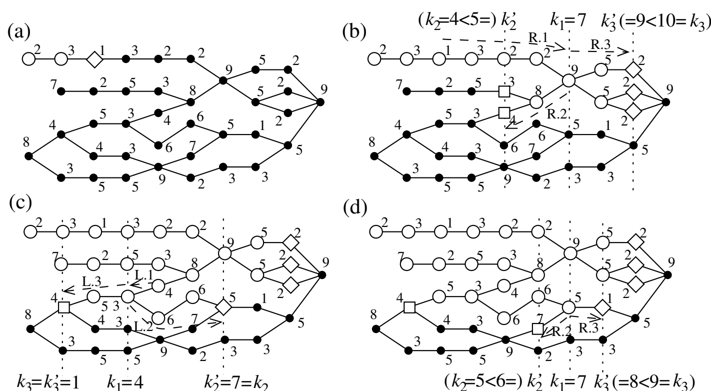

Figure 4 a graph $\mathcal{G}$ (the integers are vertex weights) with distinguished vertices in C_m representing the state of CP after: (a) the initialization; (b) first iteration with Steps R.1-R.3 executed; (c) second iteration with Steps L.1-L.3 executed; (c) third iteration with Steps R.1-R.3 executed

The lemma below follows directly from the instructions in procedure EE.

▶ **Lemma 6.** $\delta(C_j) = \delta_{\mathrm{L}}(C_j) \cup \delta_{\mathrm{R}}(C_j)$ *for each* $j = 1, \ldots, m$*.* △

The connectedness of $\mathcal{C}$ is due to the fact that $\mathcal{G}[C_j]$ is connected for each $j = 1, \ldots, m$, while the fact that $\mathcal{C}$ is a path decomposition follows from the definition of $\mathcal{G}$.

▶ **Lemma 7.** *Given a simple graph* G *and its path decomposition* $\mathcal{P} = (X_1, \ldots, X_d)$*,* CP *returns a connected path decomposition* $\mathcal{C} = (Z_1, \ldots, Z_m)$ *of* G*.* △

4 The approximation guarantee of the algorithm

In this section we analyze the width of the path decomposition $\mathcal{C}$ calculated by CP for the given G and $\mathcal{P}$. First we introduce the concept of a nested expansion, which, informally

speaking, is as follows. The first condition for C to be nested states that the weight of $V_i \cap C$ for any i 'between' the right extremity of the left border and the left extremity of the right border (by Lemma 9 the former is less than the latter) is greater than or equal to the weight of the left or the right border of C. The remaining conditions refer to the situation 'inside' borders and are analogous in both cases. The condition (ii) for the left border requires that the weight of $V_i \cap C$, where $i \leq r(\delta_{\mathrm{L}}(C))$, is not less than the weight of the left border restricted to the vertices in $V_1 \cup \cdots \cup V_i$. Finally, condition (iii) gives a 'local' minimality, that is, if we take a left branch $\mathcal{B}_{\mathrm{L}}(C, \delta_{\mathrm{L}}(C), i)$ (where i by the definition is $\leq r(\delta_{\mathrm{L}}(C))$) and we include several vertices of the branch, as it is done in procedure EE, then we 'arrive' at some cut of this branch, and (iii) for C guarantees that the weight of the left border of the new expansion is greater than or equal to the weight of the left border of C.

We say that an expansion C is *nested* if it satisfies the following conditions:

(i) for each $i = r(\delta_{\mathrm{L}}(C)), \ldots, l(\delta_{\mathrm{R}}(C))$, $\min\{\omega(\delta_{\mathrm{L}}(C)), \omega(\delta_{\mathrm{R}}(C))\} \leq \omega(V_i \cap C)$,
(ii) for each $i \leq r(\delta_{\mathrm{L}}(C))$, $\omega(V_i \cap C) \geq \sum_{j \leq i} \omega(V_j \cap \delta_{\mathrm{L}}(C))$, and for each $i \geq l(\delta_{\mathrm{R}}(C))$, $\omega(V_i \cap C) \geq \sum_{j \geq i} \omega(V_j \cap \delta_{\mathrm{R}}(C))$,
(iii) $r(\delta_{\mathrm{L}}(C))$ ($l(\delta_{\mathrm{R}}(C))$) is a bottleneck of each branch $\mathcal{B}_{\mathrm{L}}(C, \delta_{\mathrm{L}}(C), i)$ (respectively, $\mathcal{B}_{\mathrm{R}}(C, \delta_{\mathrm{R}}(C), i)$).

Figure 5 presents a subgraph of $\mathcal{G}$ on the vertices that belong to an expansion C. For this expansion to be nested it holds in particular: (ii) implies $\omega(V_{i+1} \cap C) \geq \omega(\{x_1, x_2, x_3\})$, $\omega(V_{i+3} \cap C) \geq \omega(\{x_1, x_2, x_3\})$; (i) implies $\omega(V_{i+6} \cap C) \geq \min\{\omega(\delta_{\mathrm{L}}(C)), \omega(\delta_{\mathrm{R}}(C))\} = \min\{\omega(\{x_1, \ldots, x_4\}), \omega(\{y_1, \ldots, y_4\})\}$.

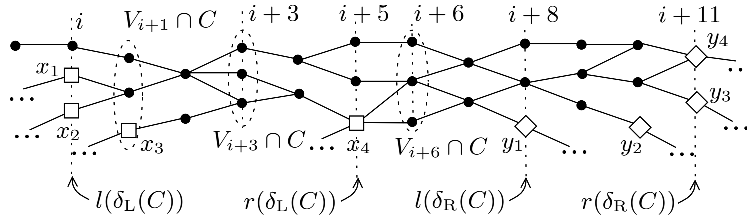

Figure 5 A nested expansion with left and right borders distinguished

Not all expansions computed by CP are nested, but we prove that all of them satisfy (ii) (Lemmas 10-12). The fact that the expansions obtained in Steps I.1-I.3 of CP satisfy (ii) follows from the observation that both the left and right border or each such expansion is a subset of a single set V_i. For this reason we focus on analyzing the subsequent iterations of CP, and we proceed with an assumption that an expansion from the beginning of an iteration (i.e. obtained at the end of the previous iteration, or at the end of Step I.3) is nested. We justify this assumption in Lemma 12.

First we introduce the following concept of moving the borders of an expansion. We say that C_j *moves* the right (left) border of C_{j-1} if $l(\delta_{\mathrm{R}}(C_j)) > l(\delta_{\mathrm{R}}(C_{j-1}))$ ($r(\delta_{\mathrm{L}}(C_j)) < r(\delta_{\mathrm{L}}(C_{j-1}))$, respectively). The following lemma states that in each iteration of CP at most one expansion may be computed that does not move the left or right border of its predecessor. Moreover, this is the first expansion calculated in Step L.2 or in Step R.2, depending on the condition checked in the 'if' statement in the main loop of CP.

▶ **Lemma 8.** *If C_j, $j \in \{2, \ldots, m\}$, is any expansion calculated in Step EL (Step ER) of* EE *invoked in an iteration of* CP*, except for an expansion obtained in the first iteration of* EE *executed in Step L.2 or in Step R.2 of* CP*, then C_j moves the left (right, resp.) border of C_{j-1}.* △

Lemma 6 states that $\delta(C_j) = \delta_{\rm L}(C_j) \cup \delta_{\rm R}(C_j)$ for each expansion obtained in CP, while the following lemma implies that the left and right borders of any expansion obtained during the execution of CP are disjoint.

▶ **Lemma 9.** *$r(\delta_{\rm L}(C_j)) < l(\delta_{\rm R}(C_j))$ for each $j = 1,\ldots,m$.* △

Provided that an expansion from the beginning of an iteration of CP is nested, the following, together with Lemma 8, implies that if the first expansion computed in Step L.2 or R.2 of CP satisfies (ii), then all expansions from the iteration satisfy (ii).

▶ **Lemma 10.** *Let $j \in \{2,\ldots,m\}$. If C_{j-1} satisfies (ii) and C_j moves the right or the left border of C_{j-1}, then C_j satisfies (ii).* △

Due to the above, we analyze the first expansion obtained in Step L.2 or R.2 of CP. Let C_{s_0} be the expansion from the beginning of an iteration of CP and let C_{s_1} be the expansion obtained at the end of Step L.1 or Step R.1 of this iteration. We obtain that if C_{s_0} is nested, then C_{s_1} satisfies (i) (the proof is omitted due to lack of space). This allows us to prove that the first expansion obtained in Step L.2 or R.2 of CP also satisfies (ii). Thus, due to Lemmas 8 and 10 we obtain that each expansion in an iteration of CP satisfies (ii), when the expansion from the beginning of the iteration is nested.

▶ **Lemma 11.** *If an expansion C_{s_0} from the beginning of an iteration of* CP *is nested, then each expansion computed by* CP *in this iteration satisfies (ii).* △

Finally, we finish our argument that each expansion calculated by CP satisfies (ii), by proving the following.

▶ **Lemma 12.** *Let C_{s_0} and C_{s_3} be the expansions from the beginning of two consecutive iterations of* CP*. If C_{s_0} is nested, then C_{s_3} is nested.* △

Therefore, an induction on the number of iterations of CP allows us to prove the claim that each expansion computed by CP satisfies (ii), as shown in Lemma 14. Lemma 13 below, together with Lemma 6, gives an upper bound for $\omega(\delta(C_j))$ for each $j = 1,\ldots,m$.

▶ **Lemma 13.** *If an expansion C satisfies (ii), then $\omega(\delta_{\rm L}(C)) \leq \mathrm{width}(\mathcal{G})$ and $\omega(\delta_{\rm R}(C)) \leq \mathrm{width}(\mathcal{G})$.*

Proof. Suppose w.l.o.g. that $\omega(\delta_{\rm R}(C)) \geq \omega(\delta_{\rm L}(C))$. Then, it is enough to argue that $\omega(\delta_{\rm R}(C)) \leq \mathrm{width}(\mathcal{G})$. To that end observe that by (ii), where $i = l(\delta_{\rm R}(C))$, $\omega(V_i \cap C) \geq \sum_{k\geq i} \omega(\delta_{\rm R}(C) \cap V_k) = \omega(\delta_{\rm R}(C))$. Since $\omega(V_i \cap C) \leq \omega(V_i) \leq \mathrm{width}(\mathcal{G})$, the thesis follows. ◀

▶ **Lemma 14.** *If $\mathcal{C} = (Z_1,\ldots,Z_m)$ is a path decomposition calculated by* CP *for the given G and $\mathcal{P}$, then $\mathrm{width}(\mathcal{C}) \leq 2\cdot\mathrm{width}(\mathcal{P}) + 1$.*

Proof. The expansion obtained at the end of Step I.3 of CP is nested. Indeed, (i) and (iii) follow from the fact that a_0' and b_0' are the bottlenecks of the corresponding branches used in Steps I.2 and I.3, respectively, while (ii) trivially holds, for both the left and right border is contained in a single set V_i. Using an induction (on the number of iterations of CP) we obtain by Lemma 12 that any expansion from the beginning of an iteration of CP is nested. Note that for each expansion C_j obtained in Steps I.1-I.3 of CP it holds $\delta_{\rm L}(C_j) \subseteq V_i$ and $\delta_{\rm R}(C_j) \subseteq V_{i'}$ for some $i, i' \in \{1,\ldots,d\}$, which implies (ii) for C_j. This, together with Lemma 11, implies that C_j satisfies (ii) for each $j = 1,\ldots,m$. By Lemmas 6 and 9, $\omega(\delta(C_j)) = \omega(\delta_{\rm L}(C_j)) + \omega(\delta_{\rm R}(C_j))$ for each $j = 1,\ldots,m$. By Lemma 13, $\omega(\delta(C_j)) \leq 2\cdot\mathrm{width}(\mathcal{G})$. By the definition, $\mathrm{width}(\mathcal{C}) = \max\{\omega(\delta(C_j))\colon j = 1,\ldots,m\} - 1$. Thus, by the definition, $\mathrm{width}(\mathcal{C}) \leq 2\cdot\mathrm{width}(\mathcal{G}) - 1 = 2\cdot\mathrm{width}(\mathcal{P}) + 1$. ◀

▶ **Lemma 15.** *Let G be a simple connected graph and let $\mathcal{P} = (X_1, \ldots, X_d)$ be its path decomposition of width k. The running time of* `CP` *executed for G and $\mathcal{P}$ is $O(dk^2)$.*

Proof. Since each edge of G is contained in one of the bags of $\mathcal{P}$, $|E(G)| \leq dk$. The number of vertices and edges in $\mathcal{G}$ is $O(kd)$ and $O(dk^2)$, respectively. Thus, the complexity of constructing $\mathcal{G}$ is $O(dk^2)$.

If a branch is given, then the weights of all its cuts can be calculated in time linear in the number of edges and vertices of the branch. The time of finding any branch $\mathcal{B}$ in an iteration of `CP` is $O(|E(\mathcal{B})|)$. The complexity of calculating the weight of all cuts of $\mathcal{B}$, and thus finding its bottleneck, is $O(|E(\mathcal{B})|)$. Whenever two branches overlap, we do not have to repeat the computation. Therefore, the time complexity of determining all branches and their bottlenecks is $O(dk^2)$. This includes the complexity of all executions of the procedure `EE`, because, by Lemma 5, the procedure 'follows' the previously calculated branches by including their vertices into the expansions C_j. It holds that $m \leq kd$, because (by Lemmas 4 and 5) $C_j \subseteq C_{j+1}$ and $C_j \neq C_{j+1}$ for each $j = 1, \ldots, m-1$. By Lemma 14, $\omega(C_j) = O(k)$ for each $j = 1, \ldots, m$. Thus, $\sum_{1 \leq j \leq m} |Z_j| = O(dk^2)$. Thus, the complexity of `CP` is $O(dk^2)$. ◀

▶ **Theorem 16.** *There exists a $O(dk^2)$-time algorithm that for given connected graph G and its path decomposition $\mathcal{P} = (X_1, \ldots, X_d)$ of width k returns a connected path decomposition $\mathcal{C} = (Z_1, \ldots, Z_m)$ such that $\mathrm{width}(\mathcal{C}) \leq 2 \cdot \mathrm{width}(\mathcal{P}) + 1$ and $m \leq kd$.*

Proof. The correctness of `CP` is due to Lemma 7. The inequality $\mathrm{width}(\mathcal{C}) \leq 2\mathrm{width}(\mathcal{P}) + 1$ follows from Lemma 14, and the complexity of `CP` is due to Lemma 15. As argued in the proof of Lemma 15, $m \leq kd$. ◀

▶ **Theorem 17.** *For each connected graph G,* $\mathrm{cpw}(G) \leq 2 \cdot \mathrm{pw}(G) + 1$. ◀

The inequalities $\mathrm{pw}(G) \leq \mathtt{s}(G) \leq \mathrm{pw}(G) + 2$ and $\mathrm{cpw}(G) \leq \mathtt{cs}(G) \leq \mathrm{cpw}(G) + 2$ [4] and Theorem 17 give the following

▶ **Corollary 18.** *For each graph G it holds* $\mathtt{cs}(G) \leq 2 \cdot \mathtt{s}(G) + 3$. ◀

5 Conclusions

The advances in graph theory presented in this paper are three-fold:

- A bound for connected pathwidth is given, $\mathrm{cpw}(G) \leq 2\mathrm{pw}(G) + 1$, where G is any graph, which bounds the connected search number of a graph by its search number, $\mathtt{cs}(G) \leq 2\mathtt{s}(G) + 3$. Moreover, the input vertex v that belongs to the first bag in the resulting connected path decomposition is selected arbitrarily, which implies a stronger fact, namely a connected $(2\mathtt{s}(G) + 3)$-search strategy can be constructed with any vertex of G playing the role of the homebase. This provides an efficient algorithm for converting a search strategy into a connected one with an arbitrary homebase.
- An efficient method is given for calculating a connected pathwidth of width at most $2k + 1$, provided that a graph G and its path decomposition of width k are given.
- It is a strong assumption that the algorithm requires a path decomposition to be given, because calculating $\mathrm{pw}(G)$ is a hard problem in general. However, this algorithm can be used to approximate the connected pathwidth for the classes of graphs for which the approximate algorithms for pathwidth exist.

References

1. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Capture of an intruder by mobile agents. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on parallel algorithms and architectures*, pages 200–209, New York, NY, USA, 2002. ACM.
2. L. Barrière, P. Fraigniaud, N. Santoro, and D.M. Thilikos. Connected and internal graph searching. Technical report, Technical Report, UPC Barcelona, 2002.
3. L. Barrière, P. Fraigniaud, N. Santoro, and D.M. Thilikos. Searching is not jumping. In *WG '03: Proceedings of the 29th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 34–45, 2003.
4. D. Bienstock. Graph searching, path-width, tree-width and related problems (a survey). *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, 5:33–49, 1991.
5. H.L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993.
6. D. Dereniowski. Connected searching of weighted trees. In *MFCS*, 2010. (to appear).
7. Fedor V. Fomin, Pierre Fraigniaud, and Dimitrios M. Thilikos. The price of connectedness in expansions. Technical report, Technical Report, UPC Barcelona, 2004.
8. F.V. Fomin and D.M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theor. Comput. Sci.*, 399(3):236–245, 2008.
9. P. Fraigniaud and N. Nisse. Connected treewidth and connected graph searching. In *Proc. of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06), LNCS*, volume 3887, pages 479–490, Valdivia, Chile, 2006.
10. P. Fraigniaud and N. Nisse. Monotony properties of connected visible graph searching. *Inf. Comput.*, 206(12):1383–1393, 2008.
11. J. Gustedt. On the pathwidth of chordal graphs. *Discrete Appl. Math.*, 45(3):233–248, 1993.
12. T. Kashiwabara and T. Fujisawa. Np-completeness of the problem of finding a minimum-clique-number interval graph containing a given graph as a subgraph. In *Proc. IEEE Inter. Symp. Circuits and Systems*, pages 657–660, 1979.
13. N.G. Kinnersley. The vertex separation number of a graph equals its path-width. *Inf. Process. Lett.*, 42(6):345–350, 1992.
14. L.M. Kirousis and C.H. Papadimitriou. Interval graphs and searching. *Discrete App. Math.*, 55:181–184, 1985.
15. L.M. Kirousis and C.H. Papadimitriou. Searching and pebbling. *Theor. Comput. Sci.*, 47(2):205–218, 1986.
16. N. Megiddo, S.L. Hakimi, M.R. Garey, D.S. Johnson, and C.H. Papadimitriou. The complexity of searching a graph. *J. ACM*, 35(1):18–44, 1988.
17. R. Mihai and I. Todinca. Pathwidth is NP-hard for weighted trees. In *FAW '09: Proceedings of the 3d International Workshop on Frontiers in Algorithmics*, pages 181–195, Berlin, Heidelberg, 2009. Springer-Verlag.
18. R. Möhring. Graph problems related to gate matrix layout and PLA folding. In *E. Mayr, H. Noltemeier, and M. Syslo eds, Computational Graph Theory, Computing Supplementum*, volume 7, pages 17–51, 1990.
19. N. Nisse. Connected graph searching in chordal graphs. *Discrete Applied Math.*, 157(12):2603–2610, 2008.
20. S.L. Peng, M.T. Ko, C.W. Ho, T.S. Hsu, and C.Y. Tang. Graph searching on chordal graphs. In *ISAAC '96: Proceedings of the 7th International Symposium on Algorithms and Computation*, pages 156–165, London, UK, 1996. Springer-Verlag.
21. N. Robertson and P.D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.
22. B. Yang, D. Dyer, and B. Alspach. Sweeping graphs with large clique number. *Discrete Mathematics*, 309(18):5770–5780, 2009.