Characterization of graphs and digraphs with small process number

David Coudert, Jean-Sébastien Sereni

To cite this version:

David Coudert, Jean-Sébastien Sereni. Characterization of graphs and digraphs with small process number. Discrete Applied Mathematics, Elsevier, 2011, 159 (11), pp.1094-1109. 10.1016/j.dam.2011.03.010 . inria-00587717

HAL Id: inria-00587717

https://hal.inria.fr/inria-00587717

Submitted on 21 Apr 2011

Characterization of graphs and digraphs with small process number

David Coudert
MASCOTTE, INRIA, I3S, CNRS, Univ. Nice Sophia
Sophia Antipolis, France
david.coudert@inria.fr

Jean-Sébastien Sereni
CNRS (LIAFA, Université Denis Diderot)
Paris, France
and
Department of Applied Mathematics (KAM)
Charles University
Prague, Czech Republic.
sereni@kam.mff.cuni.cz

April 21, 2011

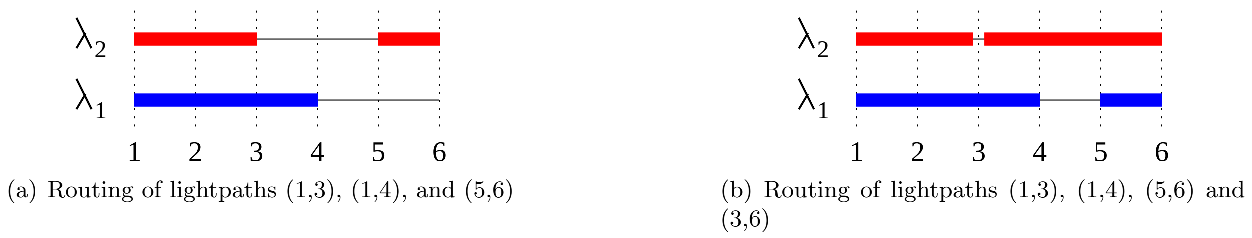
Abstract

We introduce the process number of a digraph as a tool to study rerouting issues in WDM networks. This parameter is closely related to the vertex separation (or pathwidth). We consider the recognition and the characterization of (di)graphs with small process number. In particular, we give a linear time algorithm to recognize (and process) graphs with process number at most 2, along with a characterization in terms of forbidden minors, and a structural description. As for digraphs with process number 2, we exhibit a characterization that allows one to recognize (and process) them in polynomial time.

Keywords: Rerouting, process number, vertex separation, pathwidth.

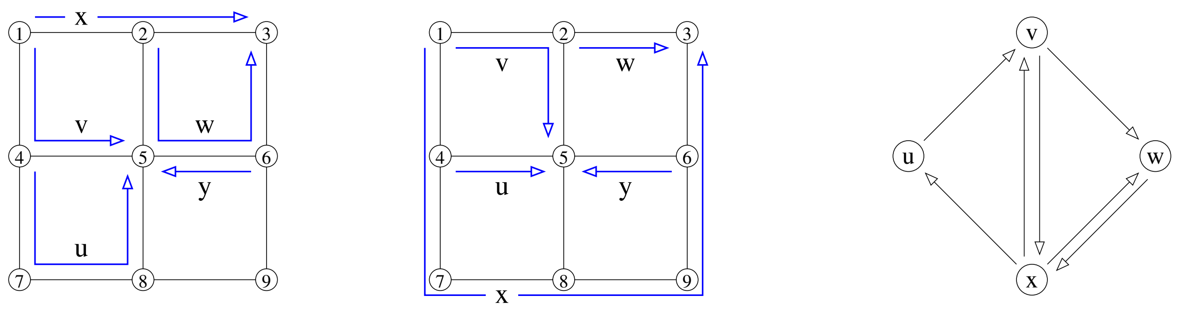
1 Introduction

In connection oriented networks such as Wavelength Division Multiplexing (WDM) networks, each connection request — called a *lightpath* in this context — is assigned a route in the network and a wavelength, under the constraint that two lightpaths sharing a link must have different wavelengths. Network operators have to change regularly (e.g., on a hourly or daily basis) the routing of the ligthpaths to improve the usage of resources with the evolution of the traffic patterns (addition and deletion of lightpaths), thereby reducing the *blocking probability*, or to stop using a particular link before a maintenance operation. For example, in Fig. 1 the new lightpath (3,6) cannot be accepted without modifying the routing of the lightpath (5,6). Another example is given in Fig. 2: a maintenance operation has to be performed on the link $\{5, 8\}$. With the initial routing of Fig. 2(a), the lightpath u has to be rerouted. However, there is no available route from node 4 to node 5 in the network with the current routing of lightpaths v, w, x, and y. Hence,



(a) Routing of lightpaths (1,3), (1,4), and (5,6) (b) Routing of lightpaths (1,3), (1,4), (5,6) and (3,6)

Figure 1: Example of a blocked lightpath in a WDM network. The network topology is a 6-node-path with 2 wavelengths. In Fig. 1(a), the lightpath (3,6) will be rejected although the routing of Fig. 1(b) is possible, up to the rerouting of lightpath (5,6).



(a) Routing R. (b) Routing R'. (c) Dependency digraph D.

Figure 2: Figs. 2(a) and 2(b) give two routings, R and R', for the set of lightpaths $\{u, v, w, x, y\}$ on the 3×3 grid. Each link of the grid is symmetric and has capacity 1 (i.e., a single wavelength) in both directions. Fig. 2(c) presents the dependency digraph for switching from routing R to routing R'.

lightpaths other than u also have to be rerouted so as to obtain an appropriate routing, like the one shown in Fig. 2(b). Therefore, a maintenance operation on a particular link of the network may impact more lightpaths than those using that link. This raises several questions, including *"how to compute the new routing knowing the current one?"* and *"how to perform the effective switching of lightpaths from the current routing to the target routing?"*. These questions arise in various connection oriented technological contexts such as circuit-switched telephone networks [1], WDM networks [20, 22, 2, 7, 9], or Multi-Protocol Label Switching (MPLS) networks [3, 16, 19].

Such questions have been widely addressed in the literature (See the surveys [29, 30]). A classical approach is based on the *Move-To-Vacant* (MTV) scheme [20, 22, 7]. It consists of a sequence of switching of lightpaths. Basically, the scheme is to choose a lightpath, compute a new route for it using available resources, move the lightpath to this new route and repeat with another lightpath until the measure of an appropriate cost function reaches a certain threshold (e.g., overall usage of resources, availability of a desired route). The difficulties here are thus to guarantee the convergence of the algorithm and to control the number of route changes (or convergence time). Integer linear programs to address this problem have been proposed [31, 19] as well as heuristic algorithms [20, 22, 16, 3, 4, 7]. However, such a scheme would work for the example of Fig. 1, but fail for the example of Fig. 2 since no such sequence exists in this case. Following previous works [15], we consider in this paper a different approach: we assume that both the initial and final routings are given and we focus on determining the best strategy to switch lightpaths from the initial routing to the final one, *possibly with interrupting some lightpaths*.

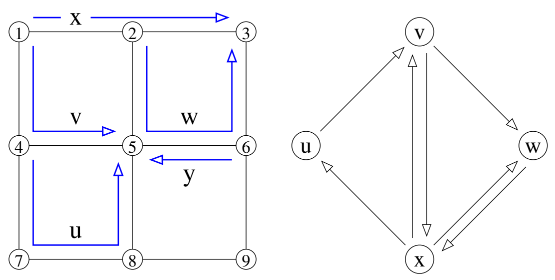
The concepts of *make-before-break* and *break-before-make* have been standardized for MPLS networks. A make-before-break consists in establishing the new route using available resources before effectively switching the lightpath, while a break-before-make starts by interrupting the lightpath before establishing the new route. Previous approaches have only considered the usage of make-before-break, but as we can see with the example of Fig. 2, it is not sufficient to switch lightpaths from one routing to another. On the other hand, if we are allowed to perform a break-before-make on lightpath x, then it is possible to reroute lightpaths u, v, and w using make-before-break's, as shown in Fig. 3.

To model the problem, Jose and Somani [15] have introduced the notion of *dependency digraphs*. Given a WDM network, a set of lightpaths I and two different routings for it in the network, R and R', the dependency digraph $D = (V, E)$ has one vertex for each lightpath with different routes in R and R', and there is an arc in E from vertex $u \in V$ to vertex $v \in V$ if the routing of u in R' aims to use resources that are in use by v in R, i.e., if $R'(u) \cap R(v) \neq \emptyset$, where $R(u)$ is the routing of the lightpath u in R. In other words, an arc $(u, v) \in E$ models the fact that the lightpath v must be switched before the lightpath u. In the example of Fig. 1, the dependency digraph contains the single vertex associated with the lightpath (5,6) and no arcs. In the example of Fig. 2, the dependency digraph is more complex and is given in Fig. 2(c).

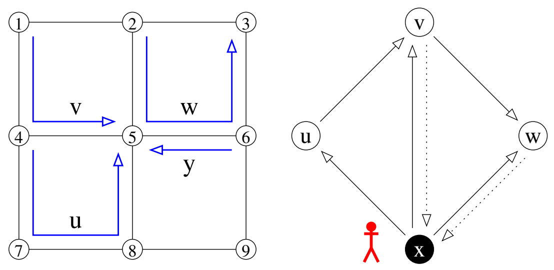
When the dependency digraph is acyclic (a directed acyclic graph, DAG), the scheduling of the sequence of rerouting is straightforward and no break-before-make is needed. Indeed, a vertex v of the dependency digraph without outneighbors means that the resources needed by the lightpath v in the new routing are available. So it can be switched directly using a make-before-break, as it is for instance the case in Fig. 1. Now, the lightpath associated with a vertex u of the DAG will be switched after the switching of all the outneighbors of u, and so the sequence of switching starts from the leafs and finishes at the roots.

However, as can be seen in Fig. 2(c), the dependency digraph may contain cycles, in which case the use of break-before-make is required. When we break (or interrupt) a lightpath, the corresponding resources are released and can be used by other lightpaths. To model the usage of break-before-make, we introduce the notion of *agents*. Placing an agent on a vertex (we then say that the vertex is *covered* by an agent) of the dependency digraph models the fact that we interrupt the associated lightpath. Furthermore, a lightpath can be rerouted if each of the outneighbors of the associated vertex in the dependency digraph either has already been rerouted or is covered by an agent. In the example of Fig. 3, placing an agent on vertex x allows us to reroute w, and then v, u, and finally x. For convenience, we say that a vertex u of the dependency digraph has been *processed* if the associated lightpath has been rerouted, and a vertex can be processed if and only if all its outneighbors are either processed or covered by an agent.

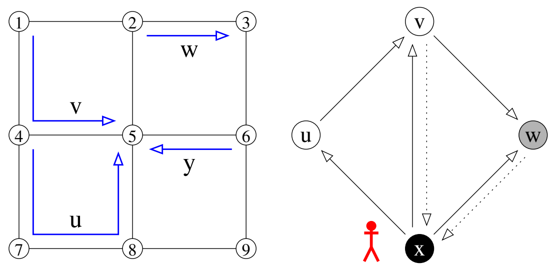
The role of an agent in the dependency digraph, and so the role of break-before-make's, is to break dependency cycles. Jose and Somani [15] proposed a heuristic algorithm to minimize the number of agents needed to break all the cycles. In fact, they actually design a heuristic algorithm for the *minimum feedback vertex set* (MFVS) problem [14], that is the size of a smallest set X of vertices such that every directed cycle contains a vertex of X. Following an earlier work [10], we consider the objective of minimizing the number of agents simultaneously placed in the dependency digraph. When all outneighbors of a vertex u covered by an agent are either processed or covered



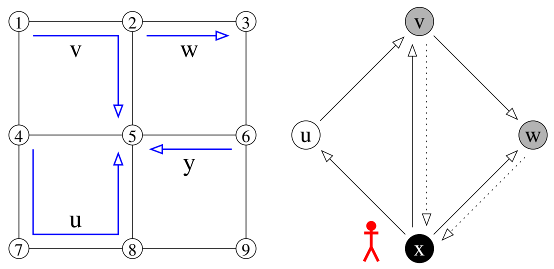
(a) Initial routing and dependency digraph from routing R to routing R'.



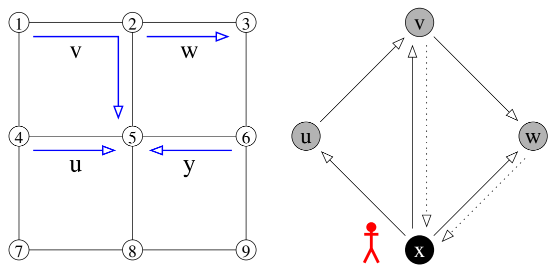
(b) Put an agent on vertex x, i.e., break request x.



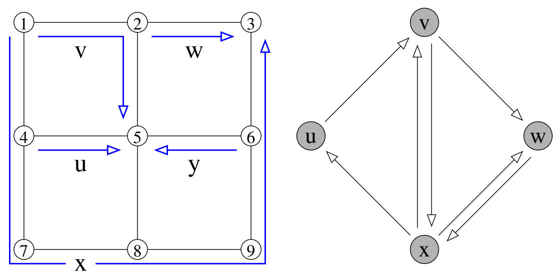
(c) Process vertex w, i.e., reroute request w.



(d) Process vertex v, i.e., reroute request v.



(e) Process vertex u, i.e., reroute request u.



(f) Remove the agent from vertex x and process it, i.e., restore and route request x.

Figure 3: Processing of the example of Fig. 2: processed vertices are in gray and vertices covered by an agent are in black. In Fig. 3(b), every vertex has at least one outneighbor in the initial state, so we must put an agent on some vertex. We choose vertex x in Fig. 3(b). Since vertex w now satisfies that all its outneighbors are either processed or covered by an agent, we process it in Fig. 3(c). Then we can process vertex v in Fig. 3(d) and vertex u in Fig. 3(e). Finally, we can process vertex x in Fig. 3(f), which frees the agent.

by an agent, then we can process u and release the agent, which can after be reused for another vertex if needed. The motivation here is to reduce the amount of traffic simultaneously blocked in the network, and so to keep the throughput of the network as high as possible at any step of the reconfiguration process.

While independent switching of requests can be made simultaneously, we consider, for matter of exposition, that only one request is switched per unit of time. So only one vertex of the dependency digraph is processed per unit of time. Also, observe that once covered by an agent, a vertex cannot recover its original state: it has to be processed. Nevertheless, it may be covered by the agent as long as desired. Processing a vertex covered by an agent frees the agent, so that it can immediately be used to cover another vertex. The digraph is *processed* when all its vertices have been processed. A *process strategy* is a sequence of the three following actions that leads to rerouting all the requests with respect to the constraints represented by the dependency digraph D.

(R_1) Put an agent on a vertex (*interrupt a connection*).

(R_2) Remove an agent from a vertex if each of its outneighbors is either processed or occupied by an agent (*reroute a connection to its final route when destination resources are available*). The vertex is now processed (*the connection has been rerouted*).

(R_3) Process a vertex if all its outneighbors are occupied by an agent (*destination resources are available, and so the connection can be rerouted*).

If we do not want to use any agents, then such a vertex ordering exists if and only if the digraph is acyclic; and in this case a processing order can be found in linear time. On the contrary, if we can use an arbitrary large number of agents, then we can first cover all the vertices by agents and next process them in any order. We aim at minimizing the number of agents simultaneously in use. The *process number* $\mathrm{pn}(D)$ of a digraph D is the minimum number of agents for which there exists a process strategy for D. Notice that the process number is upper bounded by $\textsc{mfvs}(D)$. A process strategy that uses p (at most p, at least p, respectively) agents is a *p-process strategy* (*($\leq p$)-process strategy*, *($\geq p$)-process strategy*, respectively).

The problem of determining the process number of a given dependency digraph has been proved to be NP-complete and APX-complete [10]. In this paper, we focus on the problem of recognition and characterization of digraphs and graphs with small process numbers. We start in Section 2 by recalling some general results on the process number, its links with other graph invariants like the *node search number*, and how to define it as a *cops-and-robber game*. Then, in Section 3, we first identify graphs with connectivity equal to the process number (Theorem 4). Then, we characterize graphs with process number at most 2 in terms of excluded minors. Techniques used to this end are close to those of Megiddo *et al.* [21], who gave the forbidden minors for graphs with search number at most 2. We also provide a structural description (Theorem 9), from which we design an algorithm to recognize (and, if possible, 2-process) such graphs in linear time in the number of edges (Subsection 3.1.2). We turn to digraphs in Section 4. We characterize digraphs with process number at most 2 (Proposition 17), and show how to recognize whether a graph D has process number at most 2 (and if yes how to process it) in time $O\big(n^2(n+m)\big)$, where n is the number of vertices of D, and m its number of arcs (Proposition 20). We conclude the paper in Section 5 with some open problems and directions for future works.

Let us give some notations before going further. The *outneighborhood* of X in D is

$$N^+(X) \coloneqq \{v \in V \,|\, \text{there exists } u \in X \text{ such that } (u,v) \in A\},$$

The *strict outneighborhood* of X in D is

$$\mathrm{SN}^+_D(X) \coloneqq N^+_D(X) \setminus X.$$

The (strict) outneighborhood of a vertex $x \in V$ is the (strict) outneighborhood of $\{x\}$, and the (strict) outneighborhood of a subgraph is the (strict) outneighborhood of its vertex set. The *inneighborhood* of X is $N_D^-(X) := N_{D'}^+(X)$ where D' is obtained from D by reversing the direction of every arc. In all these notations, the subscript may be omitted if there is no risk of confusion.

2 General results on the process number

First, notice that the dependency digraph may contain loops. It may occur when the original and final routes of a lightpath use the same wavelength on the same link of the network. In such cases, and depending on the specificities of the router nodes of the network, it is not always possible to establish the new route before switching the lightpath. So a break-before-make might be required.

Observation 1. *Adding loops to the vertices of a digraph D increases the process number by at most* 1.

Proof. Consider a $\mathrm{pn}(D)$-process strategy for D, let L be the order in which the vertices are processed, and let D^* be the digraph obtained from D by adding a loop to each vertex. Since adding a loop to a vertex v forces to cover v by an agent before processing it, we can process D^* following L using at most one extra agent: it suffices to ensure that an agent is placed on a vertex before processing it. So, $\mathrm{pn}(D^*) \leq \mathrm{pn}(D) + 1$. □

Let us note that the bound of Observation 1 cannot be further reduced in general. For instance, the process number of a directed symmetric path (on at least 4 vertices) is 2, and adding a loop on each vertex yields a digraph with process number 2. On the opposite, the process number of the digraph of Figure 2(c) is 1 but adding a loop on w yields a digraph with process number 2.

It is straightforward to construct a loopless digraph D' such that $\mathrm{pn}(D) = \mathrm{pn}(D')$, replacing each loop with a 2-cycle. Hence, unless stated otherwise, we consider in the sequel loopless digraphs. When D is symmetric, we work for convenience on the underlying undirected graph $G = (V, E)$. So each undirected graph of this paper is to be seen as a symmetric digraph.

An important invariant for digraphs and graphs is the notion of *vertex separation*. Let $D = (V, A)$ be a digraph and X a subset of its vertices. A *layout* L of D is an ordering of the vertices, i.e., a one-to-one correspondence between V and $\{1, 2, \cdots, |V|\}$. The *vertex separation of* (D, L) is $\mathrm{vs}_L(D)$, the maximum over all indices $i \in \{1, 2, \cdots, |V|\}$ of the size of the strict outneighborhood of $\{L^{-1}(1), L^{-1}(2), \cdots, L^{-1}(i)\}$. The *vertex separation* $\mathrm{vs}(D)$ *of* D is the minimum, over all orderings L, of $\mathrm{vs}_L(D)$. This notion naturally extends to undirected graphs: the vertex separation of an undirected graph is the vertex separation of the corresponding symmetric digraph. Kinnersley [17] proved that the vertex separation of any undirected graph equals its *pathwidth*, an important invariant of graphs introduced by Robertson and Seymour [23].

The following result establishes a close link between the vertex separation and the process number of a digraph. It was first proved by Coudert *et al.* [10], but we recall the proof here for completeness.

Proposition 2 ([10])**.** *For every digraph D,* $\mathrm{vs}(D) \leqslant \mathrm{pn}(D) \leqslant \mathrm{vs}(D) + 1$.

Proof. Consider a p-process strategy for D, and let L be the order in which the vertices are processed. Observe that if the strategy is stopped just after the i^{th} vertex has been processed, then any non-processed vertex having a processed inneighbor must be covered by an agent. As this is true for every $i \in \{1, 2, \cdots, |V|\}$, this exactly means that the vertex separation of (D, L) is p, so $\mathrm{vs}(D) \leqslant \mathrm{pn}(D)$.

Let L be an ordering of the vertices of D, and let $\mathrm{vs}_L(D)$ be the vertex separation of (D, L). We consider the process strategy for D that consists of processing the vertices in the increasing order induced by L. At any time, let P be the set of processed vertices and let M be the set of vertices covered by an agent. At each step, we ensure that M equals the strict outneighborhood of P in D.

The first vertex can be processed by covering its at most $\mathrm{vs}_L(D)$ neighbors by agents. Suppose that $i \geqslant 1$ vertices have been processed, and let v be the next vertex to be processed. If $v \notin M$, then as the vertex separation of (D, L) is $\mathrm{vs}_L(D)$ we infer that $|M \cup (N^+(v) \setminus P)| \leqslant \mathrm{vs}_L(D)$. So we can put an agent over all the outneighbors of v that are not in $M \cup P$ and process v. This uses at most $\mathrm{vs}_L(D)$ agents simultaneously. If $v \in M$, then $|M \setminus \{v\} \cup (N^+(v) \setminus P)| \leqslant \mathrm{vs}_L(D)$. Thus, putting an agent over all the outneighbors of v not in $M \cup P$ uses at most, and possibly, $\mathrm{vs}_L(D) + 1$ agents simultaneously. Hence, $\mathrm{pn}(D) \leqslant \mathrm{vs}_L(D) + 1$. □

As determining the vertex separation of an arbitrary graph is APX-complete [12], the preceding result shows that the process number problem also is.

The pathwidth of a graph is also its *node-search number*, and is closely related to other graph-searching invariants [5, 27]. Indeed, the process number in undirected graphs can be defined in terms of a *cops-and-robber game*, in which a team of agents aims to catch an invisible and infinitely fast fugitive. The main difference with the node search number is that with the process number, a vertex can be processed if all its neighbors are covered by an agent. Consequently, the fugitive is caught not only when it occupies the same vertex as an agent, but also when it is surrounded by agents. Furthermore, a process strategy is, by the definition, a monotone game. Further study of the links between the process number, the vertex separation and also the search number has been performed recently [10, 26]. We refer the reader to the recent survey of Fomin and Thilikos about graph-searching [13].

The next proposition characterizes the optimal process-strategies for digraphs whose process number is different from their vertex separation.

Proposition 3. *For any digraph D, there exists a $\mathrm{pn}(D)$-process strategy such that each vertex is covered by an agent before being processed if and only if* $\mathrm{pn}(D) = \mathrm{vs}(D) + 1$.

Proof. Suppose that the digraph D has a $\mathrm{pn}(D)$-process strategy such that each vertex is covered by an agent before being processed. Let $v_1, v_2, \cdots, v_n$ be an enumeration of the vertices of D in the order in which they are processed. For each $i \in \{1, 2, \cdots, n\}$, we set $X_i := \{v_1, v_2, \cdots, v_i\}$. Stop the strategy just before the vertex v_i is processed. All the vertices in the strict outneighborhood of X_i must be covered by agents, and so is also v_i. Therefore, $\left|\mathrm{SN}^+(X_i)\right| \leqslant \mathrm{pn}(D) - 1$ for each $i \in \{1, 2, \cdots, n\}$, and hence $\mathrm{vs}(D) \leqslant \mathrm{pn}(D) - 1$. So, $\mathrm{vs}(D) = \mathrm{pn}(D) - 1$ by Proposition 2.

Conversely, suppose that $\mathrm{pn}(D) = \mathrm{vs}(D) + 1$. Let H be the digraph obtained from D by adding a loop to each vertex that does not have one already. Thus, any strategy that processes H must cover each vertex by an agent before processing it. Moreover, $\mathrm{vs}(H) = \mathrm{vs}(D)$ and $\mathrm{pn}(D) \leqslant \mathrm{pn}(H)$. Since $\mathrm{pn}(H) \leqslant \mathrm{vs}(H) + 1$ by Proposition 2, we infer that $\mathrm{pn}(H) = \mathrm{pn}(D)$. Therefore, any $\mathrm{pn}(D)$-strategy for H is a $\mathrm{pn}(D)$-strategy for D that covers each vertex by an agent before processing it, as wanted. □

3 Symmetric digraphs

Recall that for convenience, we work on the underlying undirected graphs of symmetric digraphs. Given an undirected graph $G = (V, E)$, the *neighborhood* $N_G(v)$ of a vertex $v \in V$ is the set of all the vertices adjacent to v in G. We start by characterizing graphs the connectivity of which equals the process number.

Theorem 4. *A p-connected graph G can be p-processed if and only if there exists a vertex v of degree p such that $G - N(v)$ is an independent set.*

Proof. Let G be a p-connected graph. If there is a set of p vertices of G the deletion of which induces an independent set, then G has process number at most p (and hence exactly p since the minimum degree of G is at least p).

Conversely, let G be a p-connected graph with $\text{pn}(G) = p$ and consider a p-process strategy for G. Stop the strategy just before processing the first vertex v. Thus, all the neighbors of v are covered by agents. By the p-connectivity, G has minimum degree p. Consequently, v has degree exactly p. Let X be the set of vertices the neighborhood of which is contained in $N(v)$ — and hence is *exactly* $N(v)$, by the p-connectivity. Without loss of generality, we can assume that the first steps of the strategy consist in processing all the vertices of X. If all the vertices not in $N(v)$ have been processed, then the set $N(v)$ fulfills the desired condition.

Otherwise, there exists a vertex $w \notin X \cup N(v)$. Define z to be the next vertex to be processed. Since the strategy uses p agents and all the vertices of X have already been processed, we deduce that $z \in N(v)$. Moreover, $N(z) \subset X \cup N(v)$. Thus, $N(X \cup \{z\}) \subseteq X \cup N(v)$. Consequently, $N(v) \setminus \{z\}$ is a set of $p-1$ vertices the deletion of which disconnects w from $A \cup \{z\}$; a contradiction. □

The class of graphs with process number at most p is closed under the operation of taking minors. Indeed, assume that there exists a p-process strategy for a given graph G. Let G' be the minor of G obtained by contracting the edge uv into a single vertex w. Without loss of generality, suppose that u is processed before v — hence v is covered by an agent when u is processed. Apply the strategy to G'. The first step concerning the vertices u and v is to cover v by an agent. Instead, put an agent on w. The remaining of the strategy can then be applied, ignoring the processing of u, and processing w instead of v. Thus, G' also has process number at most p. We formalize this as an observation.

Observation 5. *Let G be a graph and H a minor of G. Then* $\text{pn}(G) \geqslant \text{pn}(H)$.

We focus on graphs with small process number. The first interesting case is when p is 2, since only independent sets can be 0-processed and only the stars have process number exactly 1. We note here that Bodlaender proved that every minor-closed class of graphs that does not contain all planar graphs has a linear time recognition algorithm [6]. This result follows from a linear time algorithm that determines whether a graph has treewidth, or pathwidth, at most k, and if so finds a tree decomposition, or a path decomposition, of width at most k, respectively. However, this algorithm is rather impracticable [24].

3.1 Graphs with process number 2

In this section, we characterize graphs with process number at most 2. As pointed out in the introduction, Megiddo *et al.* [21] gave the list of forbidden minors for graphs with search number

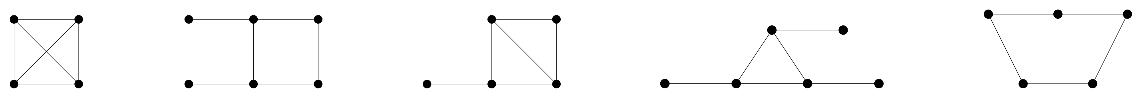

(a) K_4 (b) H_0 (c) H_1 (d) H_2 (e) C_5

Figure 4: Some minor-obstructions for 2-processed graphs.

at most 2. We use similar techniques. Nevertheless, graphs with process number at most 2 can have search number 3, and hence some extra work is needed to find the list of forbidden minors in our case. Next, we derive from the characterization an algorithm to recognize and process such graphs, which is linear (in the number of nodes and edges) in time and space.

3.1.1 Characterization of graphs with process number 2

We start by exhibiting two families M_1 and M_2 of graphs (15 graphs in total) with process number greater than 2. We then prove that a graph has process number at most 2 if and only if none of its minors is in $M_1 \cup M_2$. This is obtained via a structural characterization of those graphs. The next lemma follows from Observations 5 and a straightforward checking.

Lemma 6. *Let J be one of the graphs of Fig. 4. Every graph with a J-minor has process number at least* 3*.*

We now give a technical lemma, which is a direct analogue of a lemma for pathwidth. We do not state it in full generality, since we only need the following particular case.

Lemma 7. *Let G be a graph and v a vertex of G. If $G-v$ has (at least) three connected components with process number p then* $\mathrm{pn}(G) > p$*.*

Proof. Suppose on the contrary that $\mathrm{pn}(G) = p$, and consider a p-process strategy of G. For convenience Let J_1, J_2 and J_3 be three components of $G - v$ each of process number p. Observe that if an agent covers a vertex of $J_i + v$, then there is at least one vertex of $J_i + v$ covered by an agent until all the vertices of $J_i + v$ are processed.

Up to relabeling the components, we may assume that J_i is the i^{th} component among J_1, J_2, J_3 to have p of its vertices covered by agents (each of them must reach such a state since $\mathrm{pn}(J_i) = p$). Stop the strategy when p vertices of J_2 are covered by agents. Then, no vertex of J_1 is covered by an agent, which implies that all the vertices of J_1 are processed (by our choice of the ordering of the components J_i and the remark above). Consequently, either a vertex of $J_3 + v$ is covered by an agent, or all the vertices of J_3 are processed. The former is impossible since we consider a p-process strategy of G, and so is the latter by our ordering of the components J_i. This contradiction concludes the proof. $\square$

Our next lemma exhibits another family of graphs with process number greater than 2. It directly follows from Observation 5 and Lemma 7.

Lemma 8. *Let J consist of three graphs J_1, J_2, J_3 chosen among T_a, T_b, T_c and merged at vertex* □ *(see Fig. 5). Every graph with a J-minor has process number at least* 3*.*

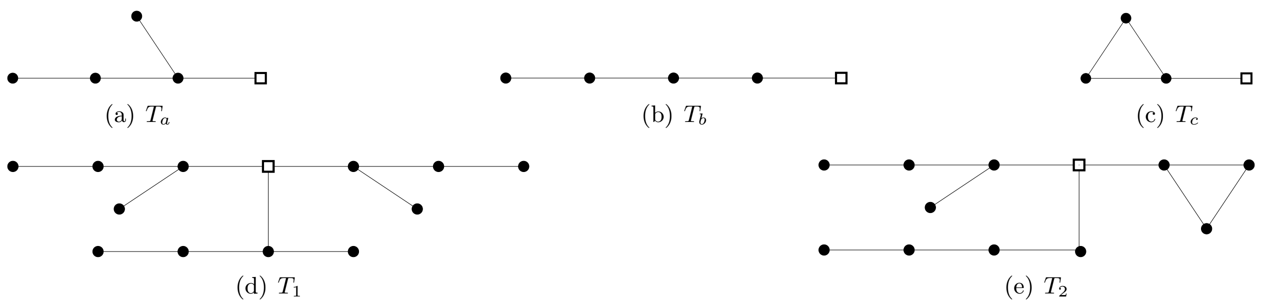


Figure 5: T_1 and T_2 are two of the ten non-isomorphic minor-obstructions for 2-processed graphs obtained using three subgraphs chosen among T_a, T_b and T_c merged at vertex $\square$.

Let M_1 be the collection of graphs depicted in Fig. 4 and let M_2 be the collection of graphs defined in Fig. 5. We observe that if a graph G has a cut-vertex v such that at least three components of $G - v$ are not stars, then G necessarily contains a minor in M_2. This is true because a connected graph that is not a star contains either a cycle, or a path on at least four vertices. This observation is used several times in the sequel.

Given a vertex u of a graph G, a subgraph H of $G - u$ is *attached to* u if the strict outneighborhood of H in G is $\{u\}$. We can now give a complete characterization of graphs that can be 2-processed. Statement (c) of the following theorem is illustrated in Fig. 6.

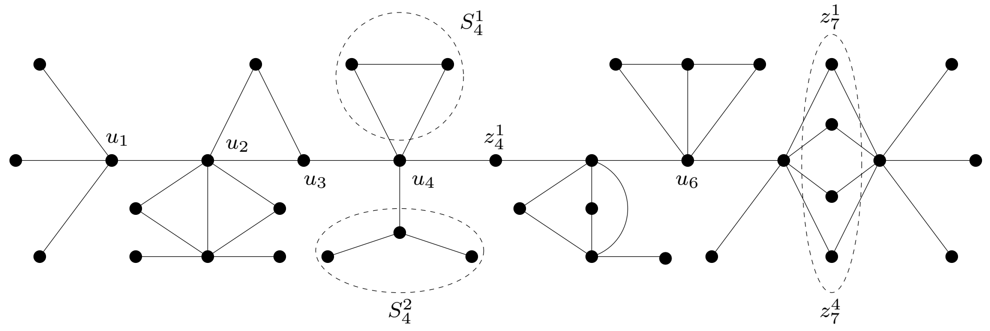


Figure 6: A typical graph with process number 2.

Theorem 9. *For every connected graph $G = (V, E)$, the following assertions are equivalent.*

(a) $\mathrm{pn}(G) \leq 2$*;*

(b) *No minor of G is in $M_1 \cup M_2$;*

(c) *There exists a partition (U, Z, T) of V such that*

- $U = \{u_1, \cdots, u_r\}$ *and* $Z = \left\{ z_i^j \,\middle|\, 1 \leqslant i \leqslant r \text{ and } 1 \leqslant j \leqslant k_i \right\}$ *where $k_1, \cdots, k_r$ are non-negative integers;*

– *each connected component of $G[T]$ is a star S_i^ℓ, for some $\ell \in \mathbb{N}$ and $i \in \{1, 2, \cdots, r\}$, which is attached in G to the vertex u_i of U;*
– *each vertex $z_i^j \in Z$ has degree 2 in G, and its two neighbors are u_i and u_{i+1}; and*
– *$N(u_i) \cap U \subseteq \{u_{i-1}, u_{i+1}\}$, and if $k_i = 0$ then $u_{i+1} \in N(u_i)$.*

Proof. The fact that (a) implies (b) follows from Lemmas 6 and 8. Let us show now that (b) implies (c).

We prove the assertion by induction on the number of vertices of G, the result being true if G has at most 3 vertices.

Suppose first that G is 2-connected. Thus, G contains a cycle of length 3 or 4, because G has no C_5-minor. Assume that G contains a 3-cycle C. Note that condition (c) holds if $G = C$, so let w be a vertex of G not in C. As G is 2-connected, there exists two paths from w to C that intersect only on w. As G has no C_5-minor, these two paths must be edges. Thus, w has two neighbors u and u' in C. Let v be the third vertex of C. Then, v is not adjacent to w since G has no K_4-minor. Moreover, the vertices v and w have degree 2 in G since G contains no H_1-minor. Therefore, G consists of the edge uu' and some vertices of degree 2 adjacent to both u and u', and hence G fulfills condition (c). Assume now that G has no 3-cycle, hence G has an induced 4-cycle C. If G is a 4-cycle, then the conclusion follows, so let us assume that w is a vertex of G not in C. We deduce as before that w has at least 2-neighbors in C because G is 2-connected and has no C_5-minor. Since G has no 3-cycle, w has exactly 2 neighbors in C, which are not adjacent. So w cannot have degree more than 2 in G, for otherwise G would contain an H_0-minor. Consequently, we infer that G consists of a 4-cycle $uvu'v'$ and some vertices of degree 2 adjacent to both u and u', and hence G satisfies condition (c).

We assume now that G has a cut-vertex w. Let $X_1, X_2, \cdots X_b$ be the connected components of $G - w$, and for each index i set $D_i \coloneqq X_i + w$. (Note that $b \geqslant 2$ since w is a cut-vertex.) If each X_i is a star, then setting $r \coloneqq 1$ and $u_1 \coloneqq w$ shows that G satisfies (c). So we assume that X_1 is not a star. As observed earlier, at most two components X_i may not be stars since no minor of G is in $M_1 \cup M_2$.

We assume first that only X_1 is not a star. By the induction hypothesis, D_1 fulfills condition (c), so we let $U' \coloneqq \{u'_1, u'_2, \cdots, u'_s\}$, Z' and T' be as stated in condition (c). Observe that we can moreover assume that each vertex u'_i with $1 < i < s$ is a cut-vertex of D_1 such that exactly two components of $D_1 - u'_i$ are not stars. In particular, for $i \in \{1, s\}$ the vertex u'_i has a neighbor not in $U' \cup Z'$. We now consider several cases regarding whether $w \in U'$, $w \in Z'$ or $w \in T'$.

If $w \in U'$ then the graph G fulfills condition (c), the components $X_2, \cdots, X_n$ being just additional stars attached to w.

Second, suppose that $w \in Z'$, and let u'_i and u'_{i+1} be the two neighbours of w in D_1. Notice that one of u'_i and u'_{i+1} has degree 2 in G, for otherwise G would contain H_1 or H_2 as a minor. By symmetry, we may assume that u'_{i+1} has degree 2. As a result, if $u'_iu'_{i+1}$ is an edge then $i + 1 = s$. Hence, setting $U \coloneqq (U' \setminus \{u'_s\}) \cup \{w\}$ and $Z \coloneqq (Z' \setminus \{w\}) \cup \{u'_s\}$ shows that G fulfills condition (c). On the other hand, if u'_i and u'_{i+1} are not adjacent, then since X_1 is a connected component of $G - w$ there exists a vertex $z'_i \neq w$ of degree 2 that is adjacent to both u'_i and u'_{i+1}. Notice that there is only one such vertex, for otherwise G would have an H_1-minor. Furthermore, since G does not contain H_0 as a minor, we deduce that u'_i has degree 2 in G. Consequently, X_1 is a star on 3 vertices; a contradiction.

Finally, assume that $w \in T'$. So, w belongs to a star S' attached to u'_i for some $i \in \{1, \cdots, s\}$. Suppose that w cannot be considered as the center of S'. Then, by our assumption on U' and because no minor of G is in M_2, we infer that $i \in \{1, s\}$, say $i = 1$. Next, there exists a vertex of $S' - w$ that is adjacent to u_1, since w is not a cut-vertex of D_1. As a consequence, wu'_1 is not an edge of G, for otherwise H_0, H_1 or H_2 would be a minor of G. Let w' be the center of S'. Setting $U \coloneqq U' \cup \{w, w'\}$ yields the desired conclusion. If w is the center of S', then a similar argument (with $w = w'$) applies if $i \in \{1, s\}$. So, suppose that $1 < i < s$. In this case, the subgraph induced by $\cup_{j>2} D_i$ is a star since no minor of G is in M_2. Thus, G has the asserted structure.

It remains to deal with the case where another component X_i, say X_2, is not a star. We assert that there exists a decomposition (U_1, Z_1, T_1) of D_1 as in condition (c), with the extra-condition that w is the vertex u_1 of U_1. This would yield the sought result, since then a similar decomposition (U_2, Z_2, T_2) of D_2 exists (by symmetry), and hence setting $U \coloneqq U_1 \cup U_2$, $Z \coloneqq Z_1 \cup Z_2$ and $T \coloneqq T_1 \cup T_2$ would show that G fulfills condition (c), as wanted.

To see that the assertion holds, we consider a decomposition (U', Z', T') of D_1 as in the previous case, i.e., $U' \coloneqq \{u'_1, \cdots, u'_s\}$, Z' and T' are as given by condition (c) applied to the graph D_1. We make the same assumption on U', i.e., every vertex u'_i with $1 < i < s$ is a cut-vertex of D_1 and exactly two components of $D_1 - u'_i$ are not stars.

It follows that if $w \in U'$ then $w = u'_i$ with $i \in \{1, s\}$, since every vertex u'_j with $1 < j < s$ is a cut-vertex of D_1. So we now assume that $w \notin U'$.

We assert that $w \notin Z'$. To see this, suppose on the contrary that $w \in Z'$ and let u'_i and u'_{i+1} be the two neighbors of w in D_1. First, note that u_i and u_{i+1} are adjacent, for otherwise both of them would have degree 2 (since G has no H_0-minor), and hence X_1 would be a star; a contradiction. As as a result, since G has no H_1- or H_2-minor, one of u_i and u_{i+1} has degree 2. This contradicts our choice of the decomposition (U', Z', T').

Consequently, w belongs to a star S' attached to a vertex u'_i of U'. We infer that $i \in \{1, s\}$ by condition (b), for otherwise u'_i would be a cut-vertex of G such that three connected components of $G - u'_i$ are not stars (recall that X_2 is not a star); a contradiction. By symmetry, assume that $i = 1$. If w is a center of S', then we set $u_0 \coloneqq w$. This yields the sought decomposition of D_1: the vertices of S' adjacent only to w are stars attached to it, and those adjacent to both w and u_1 become vertices z_0^j. And if w cannot be considered as the center of S', then let w' be the center of S'. As previously, we know that there is a vertex of $S - w$ adjacent to u_1. Therefore, w cannot be adjacent to u_1. Thus, setting $u_0 \coloneqq w'$ and $u_{-1} \coloneqq w$ yields the sought decomposition of D_1.

It remains to show that (c) implies (a). The graph G can be 2-processed as follows. (1) Cover u_1 by an agent and set $i \coloneqq 1$; (2) While $i < r$: process all stars S_i^ℓ (this uses a second agent, which is freed at the end), cover u_{i+1} by an agent, process all vertices z_i^j, process the vertex u_i (which frees an agent) and increment i; (3) Process u_r. □

Theorem 9 directly implies the following corollary.

Corollary 10. *Given two graphs H and H' that can be 2-processed and their corresponding vertices $u_1, u_2, \ldots, u_r$ for H and $u'_1, u'_2, \ldots, u'_s$ for H', the graph G built from the union of H and H' and where the vertices u_r and u'_1 are merged can be 2-processed.*

3.1.2 An algorithm to recognize graphs with process number at most 2

We present a linear (in the number of nodes and edges) time and space complexity algorithm for deciding whether a graph can be 2-processed. We use the notations of Theorem 9. The idea of the algorithm is as follows. First, we note that we can decide whether a graph is a star in time

$O(|N(v)| + |N(w)|)$, where v is any vertex of that graph and $w \in N(v)$, since one of them must be a center of the star. Then, if we are given the vertex u_1 of condition (c) of Theorem 9, we can process all attached stars in time linear in their size, next identify the vertex u_2, and so process the whole graph. Also, starting from vertex u_i and thanks to Corollary 10, we can identify in linear time the vertex u_1. Thus, the core of the algorithm is, starting from any vertex v, to identify in linear time a vertex u_i, which is done using a proper analysis of the sizes of the neighborhood at distance 1 and the neighborhood at distance 2 of v.

Before going into details, we need some more ground work. We show that deciding whether a graph can be 2-processed can be done in linear time. To this end, we first note in Proposition 11 that we can decide very efficiently if a graph can be 1-processed, and in Proposition 12 that we can decide in linear time if a 2-connected graph can be 2-processed.

From now on, we assume that a vertex v of G contains the list $N(v)$ of its neighbors, its degree, a Boolean variable v.ACTIVE set to FALSE if the vertex is covered by an agent or if it has been processed, and an integer — or a pointer — v.TAG, which is set to w if the vertex v is visited while processing the vertex w. We also assume that we can access any vertex of G in constant time, and finally that NEI(v, w) is a function that returns in constant time 1 if $v \in N(w)$ and 0 otherwise. More precisely, the function NEI uses an array of size $|V(G)|$ initialized to 0. Neighbors of v are set to 1 at the beginning of the processing phase and set back to 0 at the end of the processing phase which can thus be done in time $O(|N(v)|)$. So, the overall cost due to the management of NEI for all the vertices u_i (see Theorem 9) is linear in the size of G.

Proposition 11. *Given a graph G and a vertex v, we can decide in time $O(|N(v)| + |N(w)|)$ if G can be 1-processed or not, where w is any neighbor of v.*

Proof. Since a star has at most one vertex of degree greater than 1, it is sufficient to check that:

- if $|N(v)| > 1$, then every neighbor of v has degree 1. This can be checked in time $O(|N(v)|)$;
- if $|N(v)| = 1$, then the unique neighbor w of v cannot have neighbors of degree greater than 1, which can be checked in time $O(|N(w)|)$.

So, overall, the time complexity is $O(|N(v)| + |N(w)|)$. □

Proposition 12. *Given a 2-connected graph G, we can decide in linear time if G can be 2-processed.*

Proof. Let $n \geqslant 3$ be the order of G. By Theorem 4, a graph with no-cutvertex can be 2-processed if and only if it is either $K_{2,n-2}$ or $K_{2,n-2}$ plus an edge joining the two vertices of the bipartition of size 2.

Now, checking that G is one of the two aforementioned graphs is done as follows. We choose three arbitrary vertices of G. One of them must have degree 2 and we call u_1 and u_2 its neighbors. Now it remains to check that the neighborhood of each vertex $v \in V \setminus \{u_1, u_2\}$ is exactly $\{u_1, u_2\}$. This procedure is linear in time. □

Proposition 13. *Given a graph G, we can check in linear time if* $\mathrm{pn}(G) \leq 2$.

Proof. The proof consists of three steps. We use the notation of the assertion (c) of Theorem 9.

(1) First, we prove that if we are given a graph G and a vertex w, then we can decide in linear time if G can be 2-processed under the constraint that w is the vertex u_1 of the assertion (c) of Theorem 9. To this end, let us analyze the algorithm described at the end of the proof of

Theorem 9. We set $u_1 := w$, and we suppose that we are at step $i \geqslant 1$ of the loop. Recall that non-active vertices are ignored.

- *Cover the vertex u_i by an agent:* We just have to set the Boolean variable u_i.ACTIVE to FALSE.
- *Remove from G all subgraphs of kind S_i^ℓ:* First, we have to determine which neighbors of u_i belong to the stars S_i^ℓ and which belong to U or Z. According to Proposition 11 we can decide whether a neighbor v of u_i belongs to a star or not in time $O(|N(v)| + |N(u)|)$, where u is a neighbor of v, if any. Note that we consider the degree of v and u minus NEI(u_i, v) and NEI(u_i, u), respectively. Simultaneously, we place a tag on all neighbors of v and u, to avoid double checking. If v belongs to a star, we process it in time $O(|N(v)| + |N(u)|)$, setting the Boolean variables ACTIVE to FALSE. So edges of stars will be visited twice during the processing of u_i. We also visit all edges incident to u_{i+1} once.
- *Determine the vertex u_{i+1} and process all vertices z_i^j:* To determine u_{i+1}, we have to check that all remaining active neighbors of u_i of degree 1 (i.e., vertices z_i^j, if any) have the same neighbor, which should also be the remaining neighbor of degree greater than 1. (Note that there is such a neighbor for otherwise the previous step would have processed all the neighbors of w.) Then it remains to process the vertices z_i^j. To this end, we first cover u_{i+1} by an agent. During this step, we visit all the remaining neighbors of u_i once.
- *Process the vertex u_i, which frees an agent.*

A graph G satisfies the assertion (c) of Theorem 9 with the vertex w being the vertex u_1 if and only if this algorithm processes the whole graph. Note that the algorithm fails if more than one vertex is a candidate to be the vertex u_{i+1}.

Overall, each edge of G is visited twice and a constant number of operations are performed for each vertex. So we can process G in linear time.

(2) According to the previous step and Corollary 10, given a graph G and a vertex u_i, we can check in linear time if G can be 2-processed or not. Indeed, we process all subgraphs of G attached to u_i that can be 1-processed. Now, if $G - u_i$ has more than two components, then G cannot be 2-processed, and the algorithm returns FALSE. If $G - u_i$ has no connected components, then $\mathrm{pn}(G) \leq 2$; while if it has only one connected component H, then we apply step (1) on $H + u_i$ with $w = u_i$. Otherwise, let H and H' be the two components of $G - u_i$. We set $J := H + u_i$ and $J' := H' + u_i$. Let $U^J = \{u_1^J, \cdots, u_t^J\}$ and $U^{J'} = \{u_1^{J'}, \cdots, u_{t'}^{J'}\}$ be the vertices given by Theorem 9(c) applied to J and J', respectively.

As observed in the proof of Theorem 9, the graph G has process number at most 2 if and only if J and J' both satisfy the assertion (c) of Theorem 9 with u_i being considered as the vertex u_1^J and the vertex $u_1^{J'}$. Thus, we apply step (1) on J with $u_1^J = u_i$. If step (1) succeeds then we apply step (1) on J' with $u_1^{J'} = u_i$. Then G can be 2-processed if and only if this last step succeeds.

(3) It remains to find a vertex u_i in G. We explain now a procedure that returns a vertex which can safely be considered as one of the vertices u_i, provided that G fulfills condition (c) of Theorem 9. To this end, choose the vertex w of maximum degree of G. If vertex w has degree 2, then G is either a path or a cycle and step (2) will give a correct answer with $u_i := w$. If $|N(w)| > 2$, then w is either the center of a star or a vertex u_i. Notice that the vertices of degree greater than 2 of G are contained in $U \cup T$. Moreover, they induce a forest, each tree of the forest being a

path of order at least 1 with an arbitrary number of vertices adjacent to precisely one vertex of the path. In particular, w is at distance at most 2 of one of the vertices u_i. For $i \in \{1, 2\}$, let k_i be the number of vertices of degree at least 3 that are at distance i of w. Let x_i be such a vertex, for $i \in \{1, 2\}$, if any. The following follows from our observations.

- If $k_1 + k_2 = 0$, then the procedure can safely return $u_i = w$.
- If $k_1 = 1$, then (at least) one of w or x_1 can safely be returned as being a vertex u_i. So, it is sufficient to check if the subgraph of $G - \{w\}$ containing x_1 is a star. If it is the case, then the procedure returns w and otherwise x_1.
- The case where $k_1 = 0$ and $k_2 = 1$ is similar to the previous one, with x_2 playing the role of x_1.
- If $k_1 \geq 2$, then w can be considered to be a vertex u_i.
- if $k_1 = 0$ and $k_2 \geqslant 2$ then again w can be considered as a vertex u_i.

To sum-up, we can find in linear time a vertex of U, which allows us to check in linear time if G can be 2-processed. This concludes the proof. □

A precise description of an algorithm to recognize graphs with process number 2 (and obtain a 2-process strategy, if any) is given by Algorithms 1, 2, 3, 4 and 5.

Algorithm 1 Function Test-2-process-from

```
Require: a connected graph $G$ and a vertex $u$ covered by an agent.
Ensure: returns SUCCEED if the graph $G$ can be 2-processed with first covering $u$ by an agent.
 1: $w$.ACTIVE $\leftarrow$ FALSE
 2: $CC_2 \leftarrow$ FALSE {$CC_2$ indicates if a connected component that cannot be 1-processed has already
    been found.}
 3: for all $v \in N(u)$ such that $v$.ACTIVE and $v$.TAG $\neq u$ do
 4:   if Is-Star$(G, v, u, u)$ then
 5:     Process-Star$(G, v, u)$
 6:   else if not $CC_2$ then
 7:     $CC_2 \leftarrow$ TRUE
 8:   else
 9:     return FAILED
10: if not $CC_2$ then
11:   return SUCCEED
12: $FC \leftarrow$ FALSE {$FC$ indicates if we have found a candidate for the next vertex to be visited, $u_{i+1}$}
13: for all $v \in N(u)$ such that $v$.ACTIVE do
14:   if not $FC$ then
15:     if $|N(v)| = 2$ then
16:       $u' \leftarrow N(v) - \{u\}$
17:     else
18:       $u' \leftarrow v$
19:     $FC \leftarrow$ TRUE
20:   else if $(|N(v)| = 2$ and $N(v) - \{u\} \neq \{u'\})$ or $(|N(v)| > 2$ and $v \neq u')$ then
21:     return FAILED
22:   $u$.ACTIVE $\leftarrow$ FALSE
23: return Test-2-process-from$(G, u')$
```

Algorithm 2 Function Is-Star

```
Require: a graph $G$, a vertex $w$ that should belong to a star, a vertex $u$ that should not be
    considered in the neighborhoods, and a tag $t$.
Ensure: returns TRUE if $w$ belong to a star and FALSE otherwise. All visited vertices receive tag $t$.
 1: $c \leftarrow w$
 2: if $|N(u)| - \text{NEI}(u, w) = 1$ then
 3:   $c \leftarrow N(w) - \{u\}$
 4: $c$.TAG $\leftarrow u$
 5: $bool \leftarrow$ TRUE
 6: for all $v \in N(c) - \{u\}$ do
 7:   if $|N(v)| - \text{NEI}(u, v) > 1$ then
 8:     $bool \leftarrow$ FALSE
 9:   $v$.TAG $\leftarrow t$
10: return $bool$
```

Algorithm 3 Procedure Process-Star

```
Require: a graph $G$, a vertex $w$ that belongs to a star and a vertex $u$ that should not be considered in the neighborhoods.
Ensure: Inactivate all vertices of the star attached to $w$ except $u$.
 1: $c \leftarrow w$
 2: if $|N(u)| - \textsc{nei}(u, w) = 1$ then
 3:     $c \leftarrow N(w) - \{u\}$
 4: for all $v \in N(c) - \{u\}$ do
 5:     $v.\textsc{active} \leftarrow \textsc{false}$
```

Algorithm 4 Function First-Vertex

```
Require: a connected graph $G$.
Ensure: Returns a vertex of $U$ (i.e., such that there exists a decomposition of $G$ in which the vertex returned belongs to $U$ if $\text{pn}(G) \leqslant 2$). {We first choose the vertex of maximum degree of $G$.}
 1: Let $x_0$ be a vertex of $G$
 2: for all $v \in V(G)$ do
 3:     if $|N(x_0)| < |N(v)|$ then
 4:         $x_0 \leftarrow v$
    {Then we count the number of vertices of degree $\geq 3$ at distance $i$ of $x_0$ for $i \in \{1, 2\}$.}
 5: if $|N(x_0)| \geq 3$ then
 6:     $k_1 \leftarrow 0$, $k_2 \leftarrow 0$
 7:     for all $v \in N(x_0)$ do
 8:         if $|N(v)| \geq 3$ then
 9:             $x_1 \leftarrow v$
10:             $k_1 \leftarrow k_1 + 1$
11:     for all $v \in N(N(x_0)) - \{x_0\}$ do
12:         if $|N(v)| \geq 3$ then
13:             $x_2 \leftarrow v$
14:             $k_2 \leftarrow k_2 + 1$
        {Finally we decide whether $x_0, x_1$ or $x_2$ should be returned.}
15:     if $k_1 = 1$ and Is-Star$(G, x_0, x_1)$ then
16:         $u \leftarrow x_1$
17:     else if $k_1 = 0$ and $k_2 = 1$ and Is-Star$(G, u, x_2)$ then
18:         $u \leftarrow x_2$
19:     else
20:         $u \leftarrow x_0$
21: return $u$
```

Algorithm 5 Main procedure to 2-process graphs

```
Require: a graph $G$.
Ensure: returns TRUE if $G$ can be 2-processed and FALSE otherwise.
1: $w \leftarrow$ First-Vertex($G$)
2: Initialize NEI to 0 and set neighbors of $w$ to 1                $O(n)$
3: $w$.ACTIVE $\leftarrow$ FALSE, $bool \leftarrow$ FALSE, TAG $\leftarrow 0$
4: for all $v \in N(w)$ such that $v$.ACTIVE and $v$.TAG $\neq w$ do
5:     if Is-Star($G, v, w,$ TAG) then
6:         Process-Star($G, v, w$)
7:     else
8:         TAG $\leftarrow$ TAG $+ 1$
9: if TAG $> 0$ and TAG $< 3$ then
10:    for all $v \in N(w)$ such that $v$.TAG $= 1$ do
11:        $v$.ACTIVE $\leftarrow$ FALSE
12:        $x \leftarrow v$
13:    $w$.ACTIVE $\leftarrow$ TRUE
14:    $bool \leftarrow$ Test-2-process-from($G, w$)
15:    if TAG $= 2$ and $bool$ and $x$.TAG $= 1$ then
16:        for all $v \in N(w)$ such that $v$.TAG $= 1$ do
17:            $v$.ACTIVE $\leftarrow$ TRUE
18:        $w$.ACTIVE $\leftarrow$ TRUE
19:        $bool \leftarrow$ Test-2-process-from($G, w$)
20: return $bool$
```

4 Digraphs

In this section we characterize the classes of directed graphs with process number at most 2. We start with some preliminary remarks.

A digraph can be 0-processed if and only if it has no cycles, that is if it is a DAG. In particular a direct path can be 0-processed. Using a Depth First Search [8], one can check in linear time whether a digraph is acyclic.

Lemma 14. *For any digraph D, the process number of D is equal to the maximum of the process numbers of its strongly connected components.*

Proof. Let DAG-$\mathcal{C}$ be the acyclic digraph of the strongly connected components of D, i.e., each vertex of DAG-$\mathcal{C}$ corresponds to a strongly connected component of D, and there is an arc from a vertex u to a vertex v if and only if there is an arc between the corresponding strongly connected components in D. We can process each strongly connected component of D separately, in the order induced by DAG-$\mathcal{C}$. □

4.1 Digraphs with process number 1

First, observe that a strongly connected digraph D can be 1-processed if and only if there exists a vertex u such that $D - \{u\}$ is a DAG. In other words, a strongly connected digraph D can be 1-processed if and only if it has a minimum feedback vertex set of size 1. This can be checked in linear time [28, 25]. From this follows that we can characterize digraphs that can be 1-processed, since a digraph D' obtained from a digraph D by contracting each strongly connected component S_i to a vertex s_i is a DAG. Thus, the following is true.

Lemma 15. *A digraph D can be 1-processed if and only if each of its strongly connected components can be 1-processed.*

Forthwith a simple algorithm that decides in linear time and space complexity if a digraph can be 1-processed. This algorithm is an alternative to the algorithms of Shamir [28] and Rosen [25], which better fits our setting. In particular, it can compute the minimum feedback vertex set of 1-processed digraphs in linear time.

Proposition 16. *Given a digraph D with n vertices and m arcs, deciding whether D can be 1-processed can be done in time and space complexity $O(n + m)$.*

Proof. We assume in this proof that D is a strongly connected digraph. Otherwise, we identify each strongly connected component in time $O(n + m)$ using a Depth First Search (DFS) and next we apply the following algorithm on each strongly connected component without changing the overall complexity.

We use the observation that a strongly connected digraph D has process number 1 if and only if there is a vertex v such that $D - v$ is a DAG. The following shows how to determine whether such a vertex exists, and find one, if any, in time $O(n + m)$.

Since D is strongly connected, it contains a directed cycle $C := x_1 x_2 \dots x_k x_1$ and so the vertex v must be one of the vertices x_i. We maintain a list $\mathcal{L}$ of vertices that are candidates to be the vertex v. To this end, we define $\mathcal{L}$ to be an array of k integers, initialized to 0. A vertex x_i of C is *valid* if $\mathcal{L}[i]$ is 0.

Suppose that there exists a directed path $x_i y_1 y_2 \dots y_\ell x_j$ with $V(C)$ disjoint from $\{y_1, y_2, \dots, y_\ell\}$, for some non-negative integer ℓ. (If $\ell = 0$, then it means that there is an arc from x_i to x_j.) If $i < j$ then none of the vertices $x_{i+1}, x_{i+2}, \dots, x_{j-1}$ can be the sought vertex v. If $j < i$ then none of the vertices $x_{i+1}, \cdots, x_k, x_1, \cdots, x_{j-1}$ can be the sought vertex v.

For i from 1 to k, we run a DFS rooted at x_i, in which we consider neither the outneighbors of the vertices of $V(C) \setminus \{x_i\}$, nor the outneighbors of the vertices already visited during a previous DFS. For each vertex, we record the step in which it is first visited, i.e., we record i if the vertex was first visited during the DFS rooted at x_i. Consider the step $i \in \{1, 2, \cdots, k\}$.

If the DFS reaches a vertex x_j, then we set $\mathcal{L}[\ell] := i$ for each ℓ from $j-1$ down to $i+1$ (modulo k). Note that if $j = i$, then it means that only x_i remains valid, and so we can directly set $v := x_i$ and returns TRUE if and only if $D - x_i$ is a DAG.

If x_i is still valid and the DFS reaches a vertex already visited during some step j with $j < i$, then we set $\mathcal{L}[\ell] := i$ for ℓ from j down to 1 and from k down to $i+1$.

To cope with the complexity requirement, we make the vertices not valid in a backward way and stop when a vertex that has been removed previously is found. So doing, each cell of $\mathcal{L}$ is modified once, and we test in total at most $O(m)$ times if a vertex is still valid. So the cost due to maintaining the list of valid candidates during the algorithm is $O(n + m)$.

Observe that if a vertex $x_i \in V(C)$ is still valid once all the DFS are performed, then all the directed cycles of D that intersect C contain x_i. Thus, if $\mathcal{L}$ is non-empty then it suffices to return TRUE if and only if $D - x_i$ is a DAG. If $\mathcal{L}$ contains no valid vertices, then we can conclude that for each $i \in \{1, 2, \cdots, k\}$, the digraph $D - x_i$ contains a directed cycle, and consequently D cannot be 1-processed.

Overall this algorithm takes time $O(n + m)$. First, we can find a cycle in time $O(n)$, for example by choosing a starting vertex and moving to the first neighbor until we reach a vertex that has already been visited. Second, we visit each arc of D once during the DFS. So this part takes time $O(m)$. Also, the total cost due to $\mathcal{L}$ is $O(n + m)$. Finally, we can check whether a subgraph of D is a DAG in time $O(n + m)$ using a DFS.

The space complexity is linear since except the size of the graph, a DFS needs only a stack, which can be implemented using an integer array of size n, and the list of candidates uses an array of size at most n. □

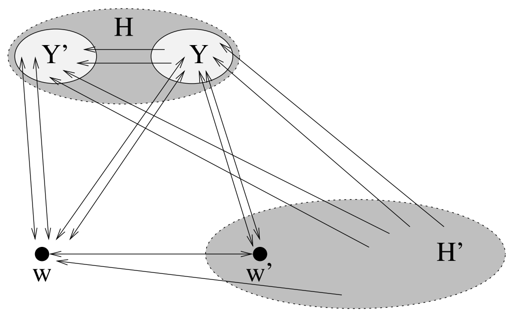
4.2 Digraphs with process number 2

Our aim in this subsection is to present a polynomial-time recognition algorithm for digraphs with process number 2.

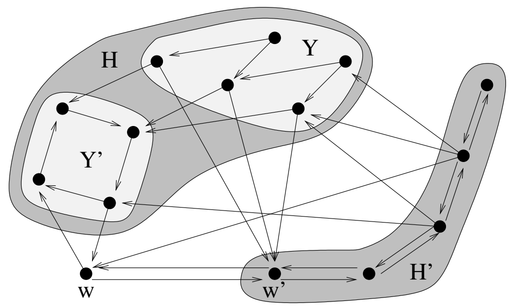
Let D be a digraph and let w be one of its vertices. We say that D is a *$(2, w)$-digraph* if there exists an $(\leqslant 2)$-process strategy to process D, the first step of which is to cover w by an agent. Thus, a digraph can be 2-processed if and only if it is a $(2, w)$-digraph for some vertex w (see Fig. 7(b)). First, we show how to determine efficiently whether D is a $(2, w)$-digraph for a given vertex w.

Proposition 17. *Let D be a (weakly) connected digraph and let w be one of its vertices. The digraph D is a $(2, w)$-digraph if and only if the digraph $D - w$ can be partitioned into two subdigraphs H and H' satisfying the following conditions.*

(i) There exists a vertex w' of H' such that H' is a $(2, w')$-digraph.



(a) General shape of a $(2, w)$-digraph



(b) Example of a $(2, w)$-digraph

Figure 7: The general shape and an example of a $(2, w)$-digraph.

(ii) $\mathrm{SN}_D^+(H + w) \subseteq \{w'\}$.

(iii) Either

- $\mathrm{pn}(H) = 0$*; or*
- $\mathrm{pn}(H) = 1$ *and there exists a (possibly empty) set* $Y \subset V(H)$ *such that* $N_D^-(\{w'\}) \cap V(H) \subseteq Y$*,* $\mathrm{pn}(D[Y]) = 0$*, and* $(Y, V(H) \setminus Y)$ *is a directed cut of* H *from* Y *to* $V(H) \setminus Y$*.*

Proof. If w has no outneighbors, then D is a $(2, w)$-digraph if and only if $\mathrm{pn}(D - w) \leqslant 2$. So the characterization is valid with H being empty and H' being $D - w$. We assume now that w has at least one outneighbor in D.

Suppose that there exist two subdigraphs H and H' as in the statement of the proposition. The following strategy shows that D is a $(2, w)$-digraph. Cover w by an agent. If $\mathrm{pn}(H) = 0$, then cover w' by an agent, process H and then process w by condition (ii). If $\mathrm{pn}(H) = 1$, then set $Y' := V(H) \setminus Y$ and process $D[Y']$. As $\mathrm{SN}_D^+(Y') \subseteq \{w\}$ by by conditions (ii) and (iii), we use at most one more agent during this processing, and only w is still covered by an agent once $D[Y']$ is processed. Now, cover w' by an agent and process the vertices of Y, since $\mathrm{SN}_D^+(Y) \subseteq Y' \cup \{w, w'\}$ by condition (ii) and $\mathrm{pn}(D[Y]) = 0$. Hence, in both cases, we have processed H, and w and w' are covered by agents. By condition (ii), we can now process w and then finish to process D since H' is a $(2, w')$-digraph by condition (i).

Conversely, assume that D is a $(2, w)$-digraph, and consider a corresponding strategy to process it. Note that at most one outneighbor of w is processed after w, since the first step of the strategy consists in covering w by an agent. Stop the strategy just before w is processed. We let H be the subdigraph of D induced by all the vertices processed before w, and H' be the complement of H in $D - w$. By the definition, $\left|N_D^+(H + w) \cap V(H')\right| \leqslant 1$. If $\left|N_D^+(H + w) \cap V(H')\right| = 0$ then we define w' to be the first vertex of H' to be covered by an agent according to the strategy, and otherwise we define w' to be the unique outneighbor of $H + w$ in H'. In this case, the vertex w' must be already covered by an agent. As the strategy uses no more than two agents simultaneously, there are no arcs from $H + w$ to $H' - w'$, and so condition (ii) is fulfilled.

Recall that we consider the strategy only from the first step up to the last step before processing w. Observe that w is covered by agent during all the steps considered, and if w' is covered by an agent at some point, then it stays covered until the last step considered. Consequently, no vertex of

$X \coloneqq N^-_D(\{w'\}) \cap V(H)$ can be covered by an agent at any step considered, and hence $\mathrm{pn}(D[X]) = 0$. It also follows from this observation that $\mathrm{pn}(H) \leqslant 1$. If $\mathrm{pn}(H) = 1$, then we let v_i be the i^{th} vertex of H to be covered by an agent (hence $v_i \notin X$). Let Y'_i be the *outbranching of* v_i *in* H, that is

$$Y'_i \coloneqq \{h \in V(H) \,|\, \text{there exists a directed path from } v_i \text{ to } h \text{ in } H\}\,.$$

We set $Y' \coloneqq \cup_i Y'_i$. By the previous observation, we infer that $Y' \cap X = \emptyset$. Moreover, there are no arcs from Y' to $Y \coloneqq V(H) \setminus Y'$. Finally, $\mathrm{pn}(D[Y]) = 0$. Thus, condition (iii) is fulfilled.

The remaining part of the strategy ensures that H' is a $(2, w')$-digraph, as required by condition (i). □

Before using the preceding characterization to derive a polynomial-time recognition algorithm, we state a useful lemma. Let D be a digraph and let w be a vertex of outdegree at most 1 of D. Let v be the unique outneighbor of w, if any. The *contraction of* w consists of removing w, linking every vertex of $N^-_D(w)$ to v, and removing any parallel arcs created (but not the loops that may appear; we do so since parallel arcs are irrelevant regarding the process number, whereas — as noted in the introduction — loops are not).

Lemma 18. *Let D be a digraph and w a vertex of D with exactly one outneighbor v. Let D' be obtained by contracting w. Then* $\mathrm{pn}(D) = \mathrm{pn}(D')$. *Moreover, for every vertex u of D, it holds that D is a $(2, u)$-digraph if and only if D' is a $(2, u')$ digraph with $u' = u$ if $u \neq w$ and $u' = v$ otherwise.*

Proof. Consider a p-process strategy for D'. Apply it to D with the extra step that w is processed as soon as v is processed or covered by an agent. This shows that $\mathrm{pn}(D) \leqslant p$. Conversely, consider a p-process strategy for D. Note that when w is processed then v is either already processed or covered by an agent. We apply the strategy to D', except that if a step covers w by an agent, we instead cover v (if it is not already covered, or processed). This yields a p-strategy for D', since the remark above ensures that we do not use any extra agent in the strategy for D'.

The 'moreover' part follows from above by a straightforward checking. □

Proposition 19. *Given a strongly-connected digraph D and a vertex $w \in V(D)$, Algorithm 6 decides in time $O(n(n + m))$ if D is a $(2, w)$-digraph.*

Proof. Let us prove that Algorithm 6 is correct. We assume that each time a vertex is processed, the neighborhoods of its predecessors are updated and so is the set V^1_D of vertices of out-degree at most 1.

Suppose first that D is a $(2, w)$-digraph. We consider the partition (H, H') of $D - w$ and the subset $Y \subseteq V(H)$ given by Proposition 17. We set $Y' \coloneqq V(H) \setminus Y$. If $\mathrm{pn}(H) = 0$, then we may assume that $Y = V(H)$, and hence $Y' = \emptyset$.

Since $\mathrm{pn}(H) \leqslant 1$ and $N^+_D(Y') \subseteq Y'$ (arcs to w are removed by line 1), lines 2–5 will remove the whole digraph $D[Y']$, because (Y, Y') is a directed cut of H. Also, $D[Y]$ is a DAG since $\mathrm{pn}(D[Y]) = 0$. Hence, every leaf vertex u of $D[Y]$ without an arc to w' will be removed as $\{u\}$ becomes a strong component once $D[Y']$ is removed. Let $Y^r \subseteq Y$ be the remaining part of Y. The digraph $D[Y^r]$ is a DAG the leaf vertices of which have for unique outneighbor w'. Thus, line 6 — and so Algorithm 7 — will contract every vertex of Y^r into w', starting from the leaf vertices. Note that $H' = D_2 - w$.

Now, Algorithm 6 returns FAILED only if either the vertex w has more than one outneighbor in $H' = D_2 - w$, or it has a unique outneighbor w' in H' but H' is not a $(2, w')$-digraph. None of these cases happens by Proposition 17. Therefore, Algorithm 6 returns SUCCEED, as desired.

Algorithm 6 Function Is-$(2, w)$-digraph

```
Require: a strongly connected digraph $D$ and a vertex $w$
Ensure: returns SUCCEED if $D$ is a $(2, w)$-digraph.
 1: Cover $w$ by an agent and remove $w$ from the neighborhoods of its predecessors
 2: $\mathcal{C} \leftarrow$ set of strongly connected components of $D - w$
 3: Let DAG-$\mathcal{C}$ be the DAG of strongly connected components
 4: while it exists $C \in \mathcal{C}$ such that $C$ is a leaf of DAG-$\mathcal{C}$ and $\text{pn}(C) \leq 1$ do
 5:    Process $C$, and so remove it from $D - w$, $\mathcal{C}$ and DAG-$\mathcal{C}$
    {Let $D_1$ be the remaining digraph}
 6: $D_2 \leftarrow$ Contract-rooted$(D_1, w)$
 7: if $V(D_2) = \{w\}$ then
 8:    return SUCCEED
 9: else if $\left|N^+_{D_2}(w)\right| = 1$ then {we have $N^+_{D_2}(\{w\}) = \{w'\}$}
10:    return Is-$(2, w')$-digraph$(D_2 - w)$
11: else
12:    return FAILED
```

Algorithm 7 Function Contract-rooted

```
Require: a connected digraph $D$ and a vertex $w$ of $D$.
Ensure: returns a reduced digraph, but vertex $w$ being unchanged.
 1: Let $V^1_D$ be the set of vertices of $D$ of outdegree at most 1.
 2: while $V^1_D \setminus \{w\}$ is not empty do
 3:    Let $v$ be any vertex of $V^1_D - \{w\}$, which we remove from $V^1_D$
 4:    if $N^+(v) > 0$ then
 5:       Let $u'$ be the outneighbor of $v$
 6:       for all $v' \in N^-(v)$ do
 7:          $N^+(v') \leftarrow N^+(v') \setminus \{v\} \cup \{u'\}$
 8:          if $|N^+(v')| = 1$ then
 9:             $V^1_D \leftarrow V^1_D \cup \{v'\}$
```

Conversely, suppose now that the algorithm returns SUCCEED for a given digraph D, and let us prove that D is a $(2, w)$-digraph. We start by covering w by an agent. The algorithm starts by removing strongly-connected components that are leaves in DAG-$\mathcal{C}$, and have process number at most 1. We can safely process all these components using at most one other agent, which is freed at the end. Note that after these steps, the remaining digraph may not be strongly connected anymore, but the vertex w has outdegree at least 1. Thanks to Lemma 18, we can ignore the contraction step of line 6. Then, as the algorithm returns SUCCEED, either only w remains, and we just process w to finish, or w has exactly one outneighbor called w', and the digraph $D_2 - w$ is a $(2, w')$-digraph. Thus, we can cover w' by an agent, process w and then finish to process $D_2 - w$ using at most two agents simultaneously. This shows that D is a $(2, w)$-digraph by Lemma 18.

The computation time of Algorithm 6 has two parts. The first part concerns the partition into strongly connected components (line 2) that takes time $O(n + m)$, the construction of DAG-$\mathcal{C}$ (line 3) in time $O(n)$, the application on each strongly-connected component of the algorithm of Proposition 16 for an overall cost in $O(n + m)$ including the update operations of line 5, and finally at most n recursive calls (line 10). Overall this part takes time $O(n(n+m))$.

The second part concerns Algorithm 7 and the maintenance of the corresponding data structures. Since the computation time of line 7 depends on the data structures chosen to store the digraph, we assume that the list of in- (respectively out-) neighbors is stored in an unsorted double linked list plus an array of size n recording for each neighbor its pointer in the list. Thus, we may add or remove a vertex of the in- (respectively out-) neighborhood of a vertex in constant time. Since a vertex may be contracted only once, and since in the worst case it has $O(n)$ predecessors, this part takes an overall time of $O(n^2)$.

Finally, the computation time of Algorithm 6 is in $O(n(n+m))$. □

We note that Algorithm 7 can me modified to decide if a strongly connected digraph D can be 1-processed, since it would then be contracted into a single vertex with a loop.

The process number of a digraph is at most p if and only if the process number of each of its strong components is at most p. Indeed, suppose that each strong component of a digraph D can be p-processed. The digraph D' of the strong components of D is acyclic. It suffices to p-process each strong component of D according to a topological order of D' to p-process D. Thus, we obtain the following result thanks to Proposition 19.

Proposition 20. *Given a digraph D we can decide in time $O(n^2(n+m))$ if it can be 2-processed.*

5 Conclusion

We modeled a rerouting problem in WDM networks using graph theory. To this end, we introduced a new (di)graph invariant, the process number, which turns out to be closely related to other well-studied invariants of (di)graphs. In particular, as Proposition 2 shows, it is a refinement of the vertex-separation (also called pathwidth in the case of undirected graphs). We also characterized the (optimal) process strategies of digraphs the process number of which is different from their vertex separation (Proposition 3).

Our next goal was to characterize and recognize efficiently (di)graphs with small process number. In particular, we gave a linear time algorithm for recognition of graphs with process number at most two (Proposition 13, Algorithm 1), as well as a characterization in terms of excluded minors and a structural description (Theorem 9). For digraphs with process number two, we found a characterization that allows us to recognize (and process) them in time $O(n^2(n+m))$. Finally, we linked the process number to the connectivity, by determining the graphs with process number equal to their connectivity (Theorem 4).

As for the excluded minor characterization, we are currently studying [11] graphs with process number 3. It may be the last case achievable, since we have so far a list of 185 266 forbidden minors, which are highly structured. It is interesting to note that for the pathwidth, such a characterization has been found up to pathwidth three [18] — for which there are 110 forbidden minors. On the other hand, the list for pathwidth 4 is not known, but it contains at least 122 millions forbidden minors and hence is probably out of reach. By Proposition 2, determining the excluded minors for graphs with process number 3 can be viewed as a scaling of this last problem, in the sense that this class contains graphs with pathwidth 3 and graphs with pathwidth 4.

Finally, we point out that heuristics have recently been designed to approximate the process number [9]. It would be interesting to further study this aspect of the question.

Acknowledgments

This work has been partially supported by EA EWIN, ANRs AGAPE and DIMAGREEN and the French *Agence Nationale de la Recherche* under reference ANR 10 JCJC 0204 01.

References

[1] M. Ackroyd. Call repacking in connecting networks. *IEEE Transactions on Communications*, 27(3):589–591, March 1979.

[2] D. Banerjee and B. Mukherjee. Wavelength-routed optical networks: Linear formulation, resource budgeting tradeoffs, and a reconfiguration study. *IEEE/ACM Transactions on Networking*, 8(5):598–607, October 2000.

[3] S. Beker, D. Kofman, and N. Puech. Offline MPLS layout design and reconfiguration: Reducing complexity under dynamic traffic conditions. In *International Network Optimization Conference (INOC)*, pages 61–66, October 2003.

[4] S. Beker, N. Puech, and V. Friderikos. A tabu search heuristic for the off-line MPLS reduced complexity layout design problem. In *IFIP-TC6 Networking Conference (Networking)*, volume 3042 of *Lecture Notes in Computer Science*, pages 514–525, Athens, Greece, May 2004. Springer.

[5] D. Bienstock, N. Robertson, P. Seymour, and R. Thomas. Quickly excluding a forest. *Journal of Combinatorial Theory, Series B*, 52(2):274–283, 1991.

[6] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[7] X. Chu, T. Bu, and X.-Y. Li. A study of lightpath rerouting schemes in wavelength-routed WDM networks. In *IEEE International Conference on Communications (ICC)*, pages 2400–2405, Hong-Kong, June 2007. IEEE.

[8] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[9] D. Coudert, F. Huc, D. Mazauric, N. Nisse, and J-S. Sereni. Reconfiguration of the routing in WDM networks with two classes of services. In *13th Conference on Optical Network Design and Modeling (ONDM)*, Braunschweig, Germany, February 2009. IEEE.

[10] D. Coudert, S. Pérennes, Q.-C. Pham, and J.-S. Sereni. Rerouting requests in WDM networks. In *Septièmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (AlgoTel'05)*, pages 17–20, Presqu'île de Giens, May 2005.

[11] D. Coudert and J.-S. Sereni. Obstruction set for graphs with process number three. in preparation.

[12] N. Deo, S. Krishnamoorthy, and M. A. Langston. Exact and approximate solutions for the gate matrix layout problem. *IEEE Transactions on Computer-Aided Design*, 6:79–84, 1987.

[13] Fedor V. Fomin and Dimitrios M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theoretical Computer Science*, 399(3):236–245, 2008.

[14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the theory of NP-completeness*. Freeman NY, 1979.

[15] N. Jose and A.K. Somani. Connection rerouting/network reconfiguration. In *4th International Workshop on Design of Reliable Communication Networks (DRCN)*, pages 23–30, Banff, Alberta, Canada, October 2003. IEEE.

[16] Balázs Gábor Józsa and Márton Makai. On the solution of reroute sequence planning problem in MPLS networks. *Computer Networks*, 42(2):199–210, 2003.

[17] N. G. Kinnersley. The vertex separation number of a graph equals its pathwidth. *Information Processing Letters*, 42(6):345–350, 1992.

[18] N. G. Kinnersley and M. A. Langston. Obstruction set isolation for the gate matrix layout problem. *Discrete Applied Mathematics*, 54(2-3):169–213, 1994.

[19] Olivier Klopfenstein. Rerouting tunnels for MPLS network resource optimization. *European Journal of Operational Research*, 188(1):293–312, 2008.

[20] K.-C. Lee and V.O.K. Li. A wavelength rerouting algorithm in wide-area all-optical networks. *IEEE/OSA Journal of Lightwave Technology*, 1996.

[21] N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou. The complexity of searching a graph. *J. Assoc. Comput. Mach.*, 35(1):18–44, 1988.

[22] G. Mohan and C.S.R. Murthy. A time optimal wavelength rerouting algorithm for dynamic traffic in WDM networks. *IEEE/OSA Journal of Lightwave Technology*, 17(3):406–417, March 1999.

[23] N. Robertson and P. D. Seymour. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

[24] H. Röhrig. *Tree decomposition: A feasibility study.* Master's thesis, Max-Planck-Institut fur Informatik, Germany, 1998.

[25] B.K. Rosen. Robust linear algorithms for cutsets. *Journal of Algorithms*, 3(3):205–217, September 1982.

[26] J.-S. Sereni. *Colorations de graphes et applications.* PhD thesis, École doctorale STIC, Université de Nice-Sophia Antipolis, July 2006.

[27] P. D. Seymour and R. Thomas. Graph searching and a min-max theorem for tree-width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993.

[28] A. Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM Journal on Computing*, 8(4):645–655, 1979.

[29] E.W.M. Wong, A.K.M. Chan, and T.-S.P. Yum. A taxonomy of rerouting in circuit-switched networks. *IEEE Communications Magazine*, 37(11):116–122, November 1999.

[30] E.W.M. Wong, A.K.M. Chan, and T.-S.P. Yum. Analysis of rerouting in circuit-switched networks. *IEEE/ACM Transactions on Networking*, 8(3):419–427, June 2000.

[31] J. Y. Zhang, O. W. W. Yang, J. Wu, and M. Savoie. Optimization of semi-dynamic lightpath rearrangements in a wdm network. *IEEE Journal on Selected Areas in Communications*, 25(9):3–17, December 2007.