Rerouting requests in WDM networks[†]

D. Coudert[1] and S. Pérennes[1] and Q.-C. Pham[2] and J.-S. Sereni[1]

1-MASCOTTE, I3S-CNRS/INRIA/UNSA, 2004 Route des Lucioles, BP93, F-06902, Sophia-Antipolis Cedex
2-Ecole Normale Supérieure, 45 rue d'Ulm, F-75230 Paris cedex 05

We model a problem related to routing reconfiguration in WDM networks. We establish some similarities and differences with two other known problems: the pathwidth and the pursuit problem. We then present a distributed linear-time algorithm to solve the problem on trees. Last we give the solutions for some classes of graphs, in particular complete d-ary trees and grids.

Keywords: process number, pursuit problem, pebbling, rerouting, WDM

1 Introduction

Usually, when connexion requests are added or removed from a network, for instance a WDM network, the routing of older connexions is not modified. Hence it is likely that after some additions and removings, the overall use of resources is far from optimal. In particular, a new request may be rejected, even if it could be added up to a whole rerouting of older requests. So operators have to reorganise regularly the routing of all requests so as to make better use of the resources. Here we are interested in the problem of going from one routing to another without loss of services.

Given a network, a set of requests I and two different routings for it in the network, R_1 and R_2, we want to switch from routing R_1 to routing R_2. Let u and v be two requests. We denote by $R_i(u)$ (resp. $R_i(v)$) the routing of request u (resp. v) in $R_i, 1 \leq i \leq 2$. If $R_2(u) \cap R_1(v) \neq \emptyset$, i.e. the routing of request u in R_2 uses resources already used by the routing of request v in R_1, then obviously the request v has to be rerouted before we can reroute request u. However, a request might be switched to an intermediate route, that uses available resources. For instance, the operator may reserve a dedicated wavelength in the network for temporary routes. We assume that each request cannot be switched to more than one temporary route, that is the next routing of a request routed on a temporary route has to be its final routing. When a request that was previously switched to a temporary route reaches its final routing, then the freed resources can be used again, for another request. While independent switching of requests can be made simultaneously, we will consider, for matter of exposition, that only one request is switched per unit of time.

The problem is modelled as follows: we construct a directed graph $D = (V, A)$, where each vertex u corresponds to one request, and there is an arc from vertex u to vertex v if and only if $R_2(u) \cap R_1(v) \neq \emptyset$. A vertex is said to be *processed* as soon as its corresponding request has been rerouted. We introduce the notion of Temporary Memory Unit (TMU): routing the request u on an intermediate route corresponds to putting the vertex u in a TMU. So a vertex can be processed if and only if all its outneighbours are either processed or in TMU's. Note that a vertex without any outneighbour can be processed at any time. There are two basic operations: process a vertex according to the preceding rule ; put a vertex in a temporary memory unit. Figure 1 shows the processing steps of a graph using one TMU.

We underline the fact that once placed in a TMU, a vertex cannot recover its original state: it has to be processed. Nevertheless, it can occupy its TMU as long as desired. Processing a vertex which occupies a TMU frees the TMU, so that it can immediately be used by another vertex. The digraph is said to be *processed* when all its vertices have been processed. The problem is hence to find a suitable order to

[†]This work has been partially funded by European project IST FET CRESCCO, the ACI-SI PRESTO and the CRC CORSO with France Telecom

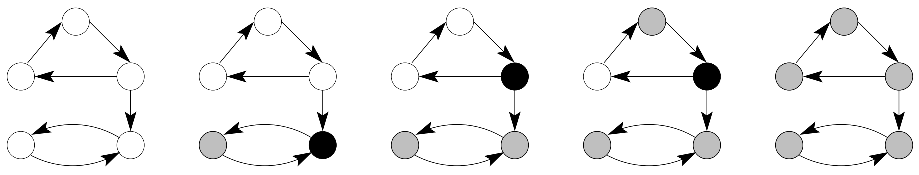

Fig. 1: Processing of a graph: processed vertices are in grey and vertices in TMU are in black.

process all the vertices. If we don't want to use any TMU, then it is clear that 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 TMU's, then we can first put all vertices in TMU's and then process them in any order. We aim at minimising the number of TMU's simultaneously in use. Notice that the problem is upper bounded by the *minimum forward vertex set number*, that is the smallest number of vertices which intersect all directed cycles.

The *process number* of D is the minimum number of temporary memory units for which there exits a process strategy for D. We denote it by $p(D)$. A process strategy which uses p (resp. at most p, resp. at least p) TMU's is called a p-process strategy (resp. $(\leq p)$-process strategy, resp. $(\geq p)$-process strategy). Remark that the digraphs we are dealing with can have loops. This may increase the process number by at most one, and it is straightforward to construct a loopless digraph D' such that $p(D) = p(D')$. When D is symmetric, we will work for convenience on the underlying undirected graph $G = (V,E)$.

This problem recalls the *pursuit problem*, introduced by Breisch [Bre67] and Parsons [Par78a, Par78b]. It has been well studied since, see for instance [MHG+88, BS91, LaP93, KP86, BFFS02]. Given a graph $G = (V,E)$, the goal is to clear the graph using the minimal number of searchers. The basic operations are the following: place a searcher on a vertex ; remove a searcher from a vertex ; and move a searcher along an edge. An edge is said to be *contaminated* if it is capable of harbouring a fugitive. An edge is *cleared* by placing a searcher at one end (as a *guard*) and moving a second searcher along the edge. If all the other edges incident to the guarded endpoint are already clear, then the guard can clear the contaminated edge alone by moving along it. Once cleared, an edge remains clear as long as every path from it to a contaminated edge is blocked by at least one guard. If it is not the case, the edge is *recontaminated*. The graph is cleared as soon as all the edges are simultaneously clear. A *search strategy* is a sequence of pebbling operations that will clear the graph. The *search number* $s(G)$ of a graph is the minimum number of searchers for which a search strategy exists. The following result of Turner (which is proved in [EST94] and was first mentioned as a personal communication of Turner in [KP86]), establishes a link between the pursuit problem and the *vertex separator* (also known as the *pathwidth* problem, see for instance [DPS02]):

Proposition 1 ([EST94]) *For any graph G, $vs(G) \leq s(G) \leq vs(G)+2$, where $vs(G)$ denotes the vertex separator of G.*

To provide some basic examples, let us mention that for any path P, $vs(P) = s(P) = 1$ while $p(P) = 2$ except if P is of length at most 3 in which case $p(P) = 1$. If $\mathcal{S}$ is any star, then $vs(\mathcal{S}) = 1 = p(\mathcal{S})$ while $s(\mathcal{S}) = 2$.

2 Relation to known problems

Proposition 2 *For any digraph D, $vs(D) \leq p(D) \leq vs(D)+1$.*

Proof. Consider a p-process strategy for D, and let L be the order in which vertices are processed. Notice that if we stop the strategy just after the i^{th} vertex has been processed, then any non-processed vertex having a processed neighbour must be in a TMU. As this is true for all $1 \leq i < |V|$, this exactly means that the vertex separator of (D,L) is p, so $vs(D) \leq p(D)$.
Let L be an ordering of the vertices of D, and say the vertex separator of (D,L) is vs. We consider a process strategy for D which consists of processing the vertices in the increasing order induced by L. The first vertex can be processed by putting its at most vs neighbours in temporary memory units by definition. Suppose $i \leq 1$ vertices have been processed. We denote by P the set of processed vertices, M the set of vertices

in TMU's and v the next vertex to be processed. If $v \notin M$, then as the vertex separator of (D,L) is vs, $|M \cup (N^+(v) \setminus P)| \leq vs$, so we can put all the outneighbours of v which are not in $M \cup S$ in TMU's and process v. This will use at most vs TMU's simultaneously. If $v \in M$, then we have $|M \setminus \{v\} \cup (N^+(v) \setminus P)| \leq vs$, so putting all the neighbours of v not in $M \cup S$ in TMU's will use at most (and possibly) $vs+1$ TMU's. $\square$

As the vertex separator problem is APX [DKL87], the preceding result shows that the process number problem also is. The following proposition induces a construction which enforces that each parameter grows by 1.

Proposition 3 *Let G_1, G_2 and G_3 be three connected graphs such that $vs(G_i) = vs, s(G_i) = s$ and $p(G_i) = p, 1 \leq i \leq 3$. We construct the graph G by putting one copy of each of the G_i, and we add one vertex v that has exactly one neighbour in each of the $G_i, 1 \leq i \leq 3$. Then $vs(G) = vs+1, s(G) = s+1$ and $p(G) = p+1$.*

Proof. We only show the proof for the process number. It is evident how to process G with $p+1$ TMU's. Consider a k-process strategy for G. Without loss of generality, we can assume that G_i is the i^{th} graph of G_1, G_2 and G_3 to have p of its vertices simultaneously in temporary memory units. Remark that once a vertex of G_i has been put in a TMU, then there is always at least one vertex of G_i in a TMU until G_i is processed. So the first time p vertices of G_2 occupy TMU's, G_1 must have been totally processed and no vertex of G_3 has been either processed or put in a TMU. Thus the vertex v must be in a TMU. This gives that $k \geq p+1$. $\square$

Using Proposition 3, one can show:

Theorem 1 *Let $vs \geq 3$, $s \in \{vs, vs+1, vs+2\}$ and $p \in \{vs, vs+1\}$. There exists a graph G_{vs} such that $vs(G_{vs}) = vs, s(G_{vs}) = s$ and $p(G_{vs}) = p$.*

So all the cases covered by Propositions 1 and 2 occur. These differences are created by reflecting an initial difference on a small graph. However, if we focus on structural properties of graphs that can be searched or processed with a fixed number of searchers or TMU's, we actually get important differences. In [MHG$^+$88], all graphs with search number at most 3 are characterised. In particular, it is shown that a biconnected graph has search number at most 3 if and only if it is outerplanar and bipolar. On the opposite, the complete bipartite graph $K_{3,3}$ can be 3-processed. Until now, we obtained a minor-excluded characterisation for all graphs that can be 2-processed. We now pay more attention to the case of graphs that can be 3-processed.

3 Optimal algorithm on trees

Both the vertex separator and the search number problem can be solved in linear time on trees [Sko03, BFFS02]. We present here a distributed linear-time algorithm which computes the process number of trees and gives an optimal strategy. Let us first introduce some parameters and make a basic observation on them: let T be a tree, and v a vertex of T. If $u_1, \ldots, u_k$ are neighbours of v, we denote by $T_v(u_1, \ldots, u_k)$ the maximum subtree of T rooted at v such that the neighbours of v are precisely $u_1, \ldots, u_k$. We denote by $p_v^F(T)$ (resp. $p_v^L(T)$) the minimum number of TMU's needed to process the tree T under the constraint that the first (resp. last) step of the strategy must be to put v in a TMU (resp. to process v). It is clear from the definition that for any vertex v of any tree T, we have $p(T) \leq p_v^F(T), p_v^L(T) \leq p(T)+1$. The following lemma plays a key role in our algorithm.

Lemma 1 *Consider a tree T which is not a star. Let v be a vertex of T, and $u_1, \ldots, u_k, k \geq 2$ be neighbours of v in T. For any $i \in \{1, \ldots, k\}$, we denote by p_i (resp. p_i^F, resp. p_i^L) the number $p(T_{u_i}(N(u_i) \setminus \{v\}))$ (resp. $p(T_{u_i}^F(N(u_i) \setminus \{v\}))$, resp. $p(T_{u_i}^L(N(u_i) \setminus \{v\}))$). We assume that the vector $(p_1, p_1^F, p_1^L, \ldots, p_k, p_k^F, p_k^L)$ is maximum for the lexicographic order among all the numberings of the neighbours. If $p_1^F \geq p_1^L$, then $p(T_v(u_1, \ldots, u_k)) = \max(2, p_1^L, p_2^F, p_3+1)$ else $p(T_v(u_1, \ldots, u_k)) = \max(2, p_1^F, p_2^L, p_3+1)$. Furthermore, we have $p_v^F(T_v(u_1, \ldots, u_k)) = \max(2, p_1^L, p_2+1)$ and $p_v^L(T_v(u_1, \ldots, u_k)) = \max(2, p_1^F, p_2+1)$.*

Here is how the algorithm goes. Each vertex v has a list $L(v)$ of 4 values, and is uniquely identified. At the beginning, all the leaves are in the state `active`. All other vertices are in the state `ready`. Every leaf has its values initialised to `[0,0,0,0]`, values of the other vertices being `nil`. Each vertex having all its neighbours but one `active` computes its own values. Let $u_1, \dots, u_k$ be the neighbours of v which are `active`. Then $L(v)[1] = p(T_v(u_1, \dots, u_k)), L(v)[2] = p_v^F(T_v(u_1, \dots, u_k))$ and $L(v)[3] = p_v^L(T_v(u_1, \dots, u_k))$. So we compute them using Lemma 1, and some simple init rules (that may use $L(v)[4]$). So as to record an optimal strategy, the vertex also records the order of its neighbours when computing its values. A vertex which is `ready` stays `ready` until it has received information from all its neighbours but one. Then it computes its values, sends them and becomes `active`. If an `active` vertex receives information from its last neighbour, then if the identifier of the neighbour is lower than its own identifier it just ignores it. Otherwise it computes again its values, using all the information it has collected now, becomes `done` and sends a special message so that any `active` vertex becomes `done`. Eventually, either there will be a step at which exactly one vertex has not computed its values yet, or exactly two vertices compute their values at the last step, and the updating process ensures only one will compute its values using the information from all its neighbours. In any case, if v is the last vertex to compute its values, we will have $L(v)[1] = p(T)$.

Theorem 2 *The preceding algorithm computes all the values and records the strategy in sequential time $O(n)$, and distributively with $O(n)$ messages.*

We note that this result induces an upper bound (along with a corresponding strategy) for processing outerplanar graphs, since the dual graph of an outerplanar graph is a tree.

4 Exact process number for some classes of graphs.

Theorem 3 (i) *The process number of the complete bipartite graph $K_{m,n}$ is* $\min(m,n)$.
(ii) *The process number of the complete binary tree of height h is $\lceil h/2 \rceil$, except if $h = 2$ in which case it is* 2.
(iii) *For any $d \geq 3$, the process number of the complete d-ary tree of height h is h.*
(iv) *The process number of the grid of size $m \times n$ is $\min(m,n)+1$, except if $n = m = 2$ in which case it is* 2.
(v) *The process number of the pyramid of size n is $\lceil \frac{n}{2} \rceil + 1$, except if $n \in \{2,3\}$ in which case it is $n-1$.*
(vi) *Any triangulated outerplanar graph whose dual graph is a caterpillar of maximum degree* 3 *can be* 3*-processed.*

References

[BFFS02] L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Capture of an intruder by mobile agents. In *14th ACM Symp. on Parallel Algorithms and Architectures*, 2002.

[Bre67] R. L. Breisch. An intuitive approach to speleotopology. *Southwestern Cavers*, 6:72–78, 1967.

[BS91] D. Bienstock and P. Seymour. Monotonicity in graph searching. *J. Algorithms*, 12(2):239–245, 1991.

[DKL87] 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.

[DPS02] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34(3):313–356, 2002.

[EST94] J. A. Ellis, I. H. Sudborough, and J. S. Turner. The vertex separation and search number of a graph. *Inform. and Comput.*, 113(1):50–79, 1994.

[KP86] L. M. Kirousis and C. H. Papadimitriou. Searching and pebbling. *Theoret. Comput. Sci.*, 47(2):205–218, 1986.

[LaP93] A. S. LaPaugh. Recontamination does not help to search a graph. *J. Assoc. Comput. Mach.*, 40(2):224–245, 1993.

[MHG$^+$88] 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.

[Par78a] T. D. Parsons. Pursuit-evasion in a graph. In *Theory and applications of graphs (Proc. Internat. Conf., Western Mich. Univ., Kalamazoo, Mich., 1976)*, pages 426–441. Lecture Notes in Math., Vol. 642. Springer, Berlin, 1978.

[Par78b] T. D. Parsons. The search number of a connected graph. In *Proceedings of the Ninth Southeastern Conference on Combinatorics, Graph Theory, and Computing (Florida Atlantic Univ., Boca Raton, Fla., 1978)*, Congress. Numer., XXI, pages 549–554, Winnipeg, Man., 1978. Utilitas Math.

[Sko03] Konstantin Skodinis. Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. *J. Algorithms*, 47(1):40–59, 2003.