Computing shortest, fastest, and foremost journeys in dynamic networks

Binh-Minh Bui-Xuan, Afonso Ferreira, Aubin Jarry

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Computing shortest, fastest, and foremost journeys in dynamic networks.

B. Bui Xuan — A. Ferreira — A. Jarry

N° 4589

Octobre 2002

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4589--FR+ENG

INRIA
SOPHIA ANTIPOLIS

Computing shortest, fastest, and foremost journeys in dynamic networks*.

B. Bui Xuan† , A. Ferreira‡ , A. Jarry§

Thème 1 — Réseaux et systèmes
Projet Mascotte

Rapport de recherche n° 4589 — Octobre 2002 — 21 pages

Abstract: New technologies and the deployment of mobile and nomadic services are driving the emergence of complex communications networks, that have a highly dynamic behavior. This naturally engenders new route-discovery problems under changing conditions over these networks. Unfortunately, the temporal variations in the network topology are hard to be effectively captured in a classical graph model. In this paper, we use and extend a recently proposed graph theoretic model – the evolving graphs –, which helps capture the evolving characteristic of such networks, in order to propose and formally analyze least cost *journeys* (the analog of paths in usual graphs) in a class of dynamic networks. Cost measures investigated here are hop count (*shortest* journeys), arrival date (*foremost* journeys), and time span (*fastest* journeys).

Key-words: dynamic networks, routing, paths, journeys, evolving graphs, graphs, LEO satellite networks, fixed-schedule dynamic networks, graph algorithms

* This work was partially supported by the COLOR action *Dynamic* and the European FET project CRESCCO

† ENS-Lyon, 46, allée d'Italie, 69007 Lyon, France.

‡ CNRS – I3S & INRIA Sophia Antipolis, 2004, Rt. des Lucioles, 06902 Sophia Antipolis Cedex, France. Afonso.Ferreira@sophia.inria.fr.

§ I3S & INRIA Sophia Antipolis, 2004, Rt. des Lucioles, 06902 Sophia Antipolis Cedex, France. Aubin.Jarry@sophia.inria.fr.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

Calcul des trajets les plus courts, les plus rapides, et au plus tôt dans les réseaux dynamiques.

Résumé : Les nouvelles technologies et le déploiement de services nomades et mobiles ont provoqué l'émergence de réseaux de communication complexes et dynamiques. Cela engendre naturellement de nouveaux problèmes de routage en environnement dynamique sur ces réseaux. Malheureusement, les variations temporelles ne peuvent pas être traitées de manière efficace par le modèle de graphes classique. Dans cet article, nous utilisons et développons un modèle de graphes récemment proposé, les graphes évolutifs, qui permettent de mieux maîtriser les propriétés évolutives de ces réseaux, dans le but de calculer et d'analyser les trajets les plus performants (l'équivalent des chemins dans les graphes usuels). Les qualités recherchées et leur mesure correspondante sont le nombre de sauts (trajets les plus courts), la date d'arrivée (trajets au plus tôt) et le temps de parcours (trajets les plus rapides).

Mots-clés : réseaux dynamiques, routage, chemins, trajets, graphes évolutifs, graphes, réseaux satellitaires LEO, réseaux dynamiques à calendrier fixé, algorithmique des graphes

1 Introduction

Infrastructureless mobile communication environments, such as mobile ad-hoc networks and low earth orbiting (LEO) satellite systems, present a paradigm shift from backboned networks, such as cellular telephony, in that data is transfered from node to node via peer-to-peer interactions and not over an underlying backbone of routers. Naturally, this engenders new problems regarding optimal routing of data under various conditions over these dynamic networks. In this setting, the generalized case of mobile network routing using shortest paths or least cost methods are complicated by the arbitrary movement of the mobile agents thereby leading to variations in link costs and connectivity [3]. Unfortunately, temporal dependencies in networks topologies are hard to be effectively captured in a classical graph model. This naturally motivated the study on modeling such dynamics, and designing algorithms that take it into account, for more than four decades [9, 10].

Note, however, that for the case of LEO satellite systems, sensor networks and other mobile networks with predestined trajectories of the mobile agents, the network dynamics are somewhat deterministic. LEO satellite networks [6, 16, 4, 17] in particular, communicate via *inter-satellite links* (ISL's) between satellites that are in range of each other. While ISL's connecting subsequent satellites in the same orbital plane (*intraplanar*) do not vary with time because the satellites move with zero relative angular velocity to each other, *interplanar* ISL's, between satellites in different orbital planes, vary as the satellites move in and out of range of each other. This results in the dynamic topology of the network. However, since the trajectories of the satellites are known ahead of time, it is possible to exploit this determinism in optimizing routing strategies.

Our work deals with communication issues in such networks, henceforth referred to as *fixed schedule dynamic networks* (FSDN's), where the topology dynamics at different time intervals can be predicted (see Figure 1). We can suppose that each node and each edge of a FSDN comes with a list of time intervals, representing the presence schedule over time, plus sets of weights for the edges, representing length, traversal cost, traversal time, etc.

Other works related to time dependent networks can be found in [8, 13, 12], where flow algorithms are studied in static networks with edge traversal times that may depend on the number of flow units traversing it at a given moment. If traversal times are discrete, then the approach proposed in [9], namely of expanding the original graph into layers representing the time steps, may work for computing several path-related problems. The time complexity for computing solutions is increased, but remains reasonable [8, 13, 12]. In the non-discrete case, this approach might also be employed, but the time complexity exploses because of the requirement of many layers (roughly, one per edge) and of the time discretization.

Unfortunately, the time-expanded graph approach does not apply to FSDNs and other dynamic networks, since the expansion factor would be huge, given that there are many networks to expand, and they can have non-discrete traversal times. Therefore, since the classical graph model hardly apply to these networks, recently, *evolving graphs* [5] have been proposed as a formal abstraction for dynamic networks, and can be suited easily to the case of FSDN's. Concisely, an evolving graph is an indexed sequence of subgraphs of a given graph, where the subgraph at a given index point corresponds to the network connectivity

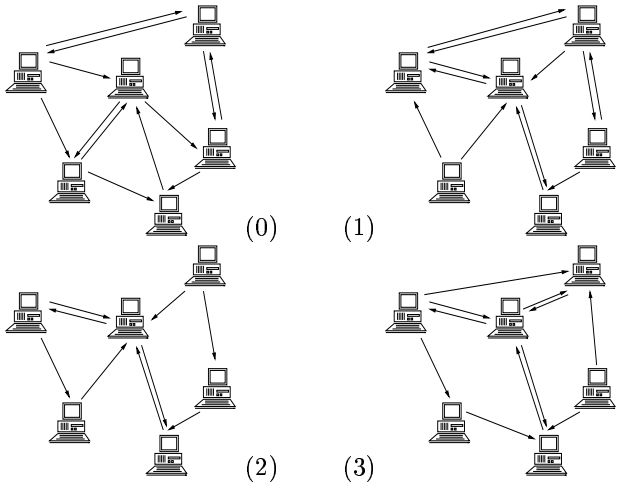

Figure 1: An FSDN represented as an indexed set of networks. The indices correspond to successive time-steps.

at the time interval indicated by the index number. The time domain is further incorporated into the model by restricting *journeys* (i.e., the equivalent of paths in usual graphs) to *never* move into edges which existed only in past subgraphs (cf. Figure 2).

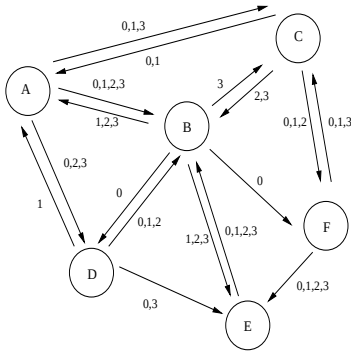

Figure 2: Evolving graph corresponding to FSDN in Figure 1. Arcs are labeled with corresponding time-steps. Observe that CBF is not a valid journey since BF exists only in the past with respect to CB.

Notice that this model allows for arbitrary changes between two subsequent time steps, with the possible creation and/or deletion of any number of vertices and arcs. More interestingly and perhaps surprisingly, previous work showed that, unlike usual graphs, finding connected components in evolving graphs is NP-Complete [1]. In this paper we use and extend evolving graphs in order to propose and formally analyze least cost journeys in FSDNs. Cost measures investigated here are hop count (*shortest* journeys), arrival date (*foremost* journeys), and time span (*fastest* journeys).

Literature on applied routing issues in such networks usually assume limited or no mobility [15, 18] where link-connectivity changes only very gradually and is incorporated by a system of updates to the topology graph with every change [11]. In contrast, we do not presume low mobility nor outright instability of the system. Our approach captures the dynamics of the network, which enables us to be efficient over the full scope of dynamicity : from static to ever-changing networks.

This paper is organized as follows. The formal definitions of evolving graphs and of some of their main parameters are revised in the next section. Then, algorithms for computing shortest (minimum hop count) journeys, foremost (earliest arrival date) journeys, and fastest (minimum time) journeys, are given and analyzed in the subsequent sections. We close the paper with concluding remarks and ways for further research.

2 A model for dynamic networks

A dynamic network can be seen as a – potentially infinite –, sequence $\mathcal{N} = \ldots, \mathcal{N}_{t-1}, \mathcal{N}_t, \mathcal{N}_{t+1}, \ldots$ of networks over time. The dynamic networks considered here are FSDNs, i.e., they have predictable changes in their topologies. We show in this section a graph theoretic model that takes into account such changes.

2.1 Graph theoretic model

Definition 1 (Evolving Graphs) *Let a graph $G(V, E)$ be given, along with an ordered sequence of its subgraphs, $\mathcal{S}_G = G_1, G_2, \ldots, G_{\mathcal{T}}$. Then, the system $\mathcal{G} = (G, \mathcal{S}_G)$ is called an evolving graph.*

Let $E_{\mathcal{G}} = \bigcup E_i$, and $V_{\mathcal{G}} = \bigcup V_i$. It is clear that $M = |E_{\mathcal{G}}| \leq |E| = m$ and that $N = |V_{\mathcal{G}}| \leq |V| = n$. Two vertices are said to be *adjacent in* $\mathcal{G}$ if and only if they are adjacent in some G_i. The degree of a vertex in $\mathcal{G}$ is defined as its degree in $E_{\mathcal{G}}$.

Like usual graphs, evolving graphs (EGs) can be weighted, the weights on the edges representing traversal distance, traversal cost, etc. On the other hand, weights can also belong to the time domain. In this case, we shall speak of *timed* evolving graphs (TEGs) because the weights on the edges will represent their traversal time.

Consider $\mathcal{I} = [t_1, t_{\mathcal{T}}]$ as a *time* interval, where G_i is the subgraph in place during $[t_i, t_i + 1[$. Then $\mathcal{G} = (G, S_G)$ is a simple time-dependent discrete dynamical system, running

during $\mathcal{I}$. In case of untimed evolving graphs, or more generally when the traversal times are discrete, we can make the whole system discrete by taking $\mathcal{I} = [1, \mathcal{T}]$.

Throughout this text we shall consider packet networks. Hence, transmitting one piece of information means transmitting one packet over one arc. The duration of transmitting one packet over a link in a FSDN is given as a function ζ representing the links' traversal times. In order to model a FSDN $\mathcal{N}$ by an evolving graph, it suffices to be given a time window $\mathcal{W}$ of size $\mathcal{T}$, and to work with a *timed evolving graph* $\mathcal{G} = (\bigcup \mathcal{R}_i | i \in \mathcal{W}, \mathrm{FSDN}_{|\mathcal{W}})$, and the function ζ, redefined, from $E_{\mathcal{G}}$ to $\mathbb{R}_+^*$.

2.2 Journeys

We call *route in* $\mathcal{G}$ a path $R(u,v) = \{e_1, e_2, \ldots, e_k\}, e_i \in E_{\mathcal{G}}$ in G. Let σ be a time schedule indicating when each edge of the route $R(u,v)$ is to be traversed. We define a *journey* $\mathcal{J}(u,v,\sigma) = \{R(u,v), \sigma\}$ if and only if σ is in accordance with R, ζ, $\mathcal{G}$, and $\mathcal{I}$, allowing for a traversal from u to v in $\mathcal{G}$. Note, for instance, that journeys cannot go to the past.

A *round journey* is a journey $\mathcal{J}(u,u,\sigma)$ in $\mathcal{G}$. It is the analogous to a usual circuit in a graph, with the difference that once the round journey ends back in $u \in G_k$, for some k, nothing implies the existence of another time schedule allowing to use of the same route again.

In case of untimed evolving graphs, the definition of a journey from u to v can be more simply restated as a sequence $\mathcal{J}(u,v) = P_i, P_{i+1}, \ldots, P_{i+k}$, such that P_j is a (usually defined) path in G_j, with path P_i starting in u, path P_{i+k} ending in v, and each path P_j ending where P_{j+1} starts.

2.3 Distances

These definitions give rise to at least three different quality measures of journeys, two of which are in the time domain, namely, hop-count or length, arrival date, and journey time.

Let $\mathcal{J}(u,v,\sigma)$ be a journey where $R(u,v) = \{e_1, e_2, \ldots, e_k\}, e_i \in E_{\mathcal{G}}$. Then,

- The *hop-count* or *length* of $\mathcal{J}(u,v,\sigma)$ is defined as $|\mathcal{J}(u,v,\sigma)|_h = |R_i| = k$.
- The *arrival date* of $\mathcal{J}(u,v,\sigma)$ is defined as $|\mathcal{J}(u,v,\sigma)|_a = \sigma(e_k) + \zeta(e_k)$, i.e., the scheduled time for the traversal of the last edge in $\mathcal{J}$, plus its traversal time. It is also denoted $a(\mathcal{J})$.
- The *journey time* of $\mathcal{J}(u,v,\sigma)$ is defined as the elapsed time between the departure at u and the arrival at v, i.e. $|\mathcal{J}(u,v,\sigma)|_t = |\mathcal{J}(u,v,\sigma)|_a - \sigma(e_1)$. It is also denoted $t(\mathcal{J})$.

Likewise, there are at least three different ways of defining the notion of "distance" in an evolving graph, as follows.

- The *distance* in $\mathcal{G}$ between two vertices u and v is defined as $d(u,v) = \min\{|\mathcal{J}(u,v,\sigma)|_h\}$, taken over all journeys in $\mathcal{G}$ between u and v. We shall say that one such journey is

the *shortest*. Further, a node w such that $d(u,w)$ is maximum is called the *antipode* of u. In this case, we say that the *eccentricity* of u equals $d(u,w)$.

- The *earliest arrival date* in $\mathcal{G}$ between two vertices u and v is given by the first journey arriving at v from u, denoted $\hat{a}(u,v)$. We shall say that one such journey is the *foremost*. If there is no journey in $\mathcal{G}$ between u and v, we say that $\hat{a}(u,v) = \infty$.
- The *delay* in $\mathcal{G}$ between two vertices u and v is defined as $delay(u,v) = \min\{|\mathcal{J}(u,v,\sigma)|_t\}$, taken over all journeys in $\mathcal{G}$ between u and v. We shall say that one such journey is the *fastest*. If there is no journey in $\mathcal{G}$ between u and v, we say that $delay(u,v) = \infty$. Again, a node w such that $delay(u,w)$ is maximum is called the *time-antipode* of u. In this case, we say that the *time-eccentricity* of u equals $delay(u,w)$.

Summarizing, $distance(u,v)$ gives the minimum number of hops required to go from u to v in $\mathcal{G}$; $delay(u,v)$ gives the minimum time required to go from u to v in $\mathcal{G}$; and the *foremost* journey indicates the *earliest arrival date* at node v from node u. Further, $eccentricity(u)$ gives the maximum number of hops required to go from u to any other node in $\mathcal{G}$; and $time\text{-}eccentricity(u)$ gives the maximum time required to go from u to any other node in $\mathcal{G}$.

Finally, we can define three sorts of "diameter" measures in evolving graphs. One is the usual *hop diameter*, defined as the maximum distance in the system, taken over all pairs of nodes. It can also be defined as the maximum of all eccentricities in the system. The second is a sort of a counterpart in the time domain, which we will denote the *system-lag* of an evolving graph, and it is defined as the maximum of all time-eccentricities. Finally, the *rapidity* of an evolving graph is the maximum of all earliest arrival dates in the system, taken over all pair of nodes.

2.4 Dynamics

Corresponding to each edge e in $E_{\mathcal{G}}$ (respectively, node v in $V_{\mathcal{G}}$) we can define an *edge schedule* $P_E(e)$ (respectively, *node schedule* $P_V(v)$) as a set of intervals indicating the subgraphs in which they are present, and possibly some of its parameters during each interval. Thus, we may alternately define an evolving graph as $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$, where each node and edge has a schedule defined for it.

With the help of these edge and node schedules, we can now introduce ways to measure how much an evolving graph changes its topology during the time interval $\mathcal{I}$. First, we define the *activity of a vertex* v as $\delta_V(v) = |P_V(v)|$, and the *activity of an edge* e as $\delta_E(e) = |P_V(e)|$. We then define the *node activity* of an evolving graph as $\delta_V = \max\{\delta_V(v),\ v \in V_{\mathcal{G}}\}$, and the *edge activity* as $\delta_E = \max\{\delta_E(e),\ e \in E_{\mathcal{G}}\}$. The *activity of an evolving graph* is defined as $\delta = \max(\delta_V, \delta_E)$. And the *dynamics of an evolving graph* is defined as $\frac{(\delta-1)}{\mathcal{I}}$. As a consequence, since usual graphs have $\delta = 1$, they have dynamics zero.

2.5 Coding

In this paper, we assume that the input $\mathcal{G}$ is given as linked adjacency lists, with the sorted edge schedule attached to each neighbor, given as time intervals indicating the time steps where that edge is alive. The traversal time of that edge is also attached to the corresponding neighbor. The head of each list is a vertex with its own sorted node schedule list attached, also given as time intervals (see Figure 3).

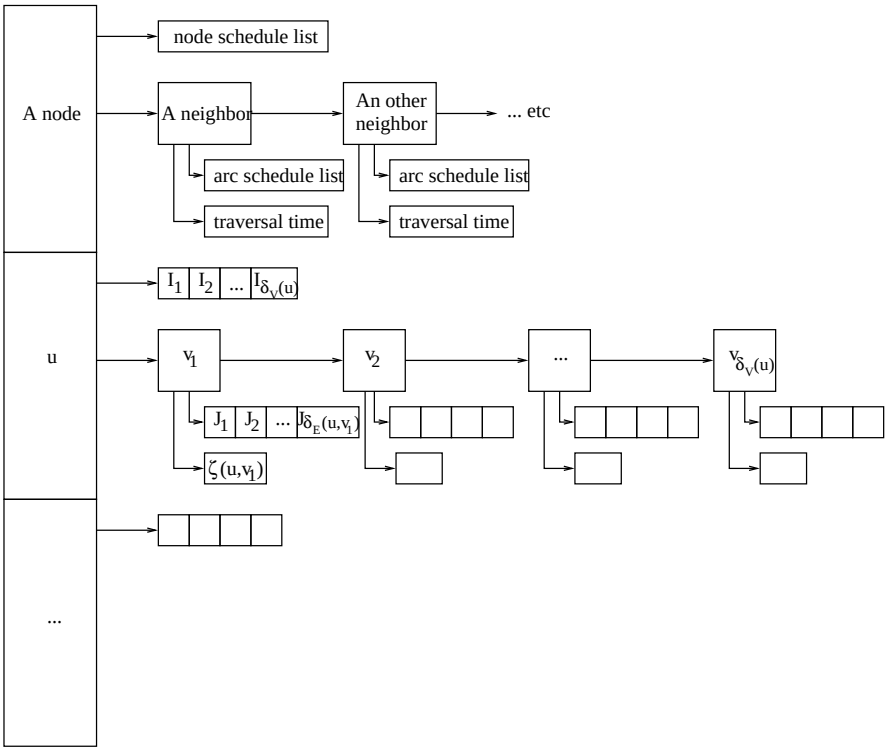

Figure 3: The data structure for coding a fixed-schedule dynamic network modeled by an evolving graph.

Thus, for each vertex v, and each arc e, the corresponding $P_V(v)$ and $P_E(e)$ are defined as sets of time intervals. This data structure is especially useful when considering networks with low *dynamics*, meaning there are few activations/desactivations on nodes and arcs.

The memory space used by the input is proportional to the size of the adjacency linked lists, plus the size of the arcs and nodes schedule lists. Therefore, the total size of the lists is $O(M + M\delta_E + N\delta_V) = O(M\delta)$ in the worst case.

3 Computing shortest journeys in EGs

The algorithm we propose in this section computes all the shortest journeys from a single vertex s to all the other vertices.

Remind that, in order to compute shortest paths, the usual Dijkstra's algorithm [2] proceeds by building a set C of *closed* vertices, for which the shortest paths have already been computed, then choosing a vertex u not in C whose shortest path estimate, $d(u)$, is minimum, and adding u to C, i.e., closing u. At this point, all arcs from u to $V - C$ are *opened*, i.e., they are examined and the respective shortest path estimate, d, is updated for all end-points. In order to have quick access to the best shortest path estimate, the algorithm keeps a min-heap priority queue Q with all vertices in $V - C$, with key d. Note that d is initialized to ∞ for all vertices but for s, which has $d = 0$.

The main observation in Dijkstra's method is that prefix paths of shortest paths are shortest paths themselves. Unfortunately, as Figure 4 shows, prefix journeys of a shortest journey may not be shortest journeys themselves. However, if the last edge, say (v, w), of a shortest journey between vertex u and vertex w starts at time t, then the prefix journey (going from u to v) is shorter than all the journeys from u to v ending *before* t. Therefore, we will consider certain pairs $(u, t) \in V_{\mathcal{G}} \times \mathbb{R}_+^*$ and compute the shortest journeys from s to vertex u arriving *before* time t. In this manner, the prefix property is respected, that is, a prefix of a shortest journey will be shortest, under the condition that it arrives before some time step t'. Using this property, we will build a tree of shortest journeys between $(s, 0)$ and pairs (u, t), in which each vertex u appears at least once.

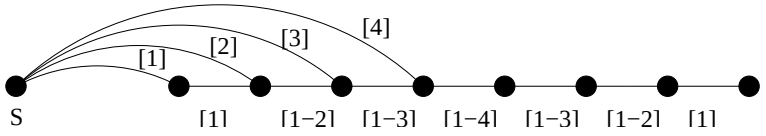

Figure 4: Evolving graph: prefix of shortest journeys are not shortest.

In order to proceed, we need a function, which given a vertex u and a time step t will compute the earliest arrival dates and neigbours of u, after time step t. Computing this function takes $O(|\Gamma(u)| \log(\delta_V))$ steps, where $\Gamma(u)$ is the neighborhoud of u in G.

In our algorithm, the tree T will represent shortest journeys from $(s, 0)$ to pairs (u, t), and the array *location* will tell where one must look for a vertex u in the tree to have the shortest journey from s to u, since a vertex may appear several times (see Figure 5).

An array *earliest* tells when a vertex is reached earliest in the tree. Indeed, if there is a journey between s and some vertex u with hop count h, and if there is a journey taking more than h hops from s to u, this journey is relevant if its arrival date is smaller than the former journey's arrival date.

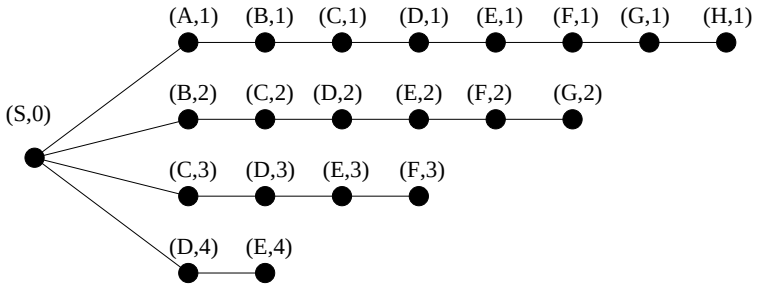

Figure 5: Tree of shortest paths.

Algorithm 1 (shortest)
Input: *An evolving graph $\mathcal{G}$, a vertex $s \in V_{\mathcal{G}}$.*
Ouput: *The shortest path tree T, and an array location : $V_{\mathcal{G}} \to T$.*
Variables: *The tree T of pairs $(u,t) \in V_{\mathcal{G}} \times \mathbb{R}_+^*$, an integer d, an array earliest : $V_{\mathcal{G}} \to \mathbb{R}_+^*$.*

1. *Initialize $T = \{(s,0)\}$; $earliest(s) \leftarrow 0$, and $\forall u \neq s$, $earliest(u) \leftarrow \infty$.*
2. *$d \leftarrow 1$.*
3. *$location(s) \leftarrow (s,0)$.*
4. *While there is $u \in V_{\mathcal{G}}$ such that $location(u)$ is not defined do:*
   (a) *For all pairs (u,t) in the tree at depth d, do:*
      i. *Get all the neighbour pairs (v,t') of (u,t) .*
      ii. *If $location(v)$ is not defined, then $location(v) = (v,t')$. If $earliest(v) > t'$, then $earliest(v) = t'$ and (v,t') is a son of (u,t) in T.*
   (b) *$d \leftarrow d + 1$.*

Once the tree has been properly computed, the retrieval of a shortest journey is straightforward: the *location* of a vertex u gives it corresponding pair (u,t) in the tree. We retrieve the journey by looking for the father of (u,t) in the tree T.

Proposition 1 *The complexity of the algorithm is $O(MD)$, where D is the hop diameter of $\mathcal{G}$.*

Proof: The maximum depth of the tree is the hop diameter D of the evolving graph, so the tree contains at most $N \cdot D$ pairs of $V_{\mathcal{G}} \times \mathbb{R}_+^*$ (N at each depth). Checking the neighbours of a vertex u takes time proportional to the degree of u, so checking the neighbours of all vertices of a certain depth takes at most $2M$ steps. Therefore, the algorithm has complexity $O(MD)$. □

4 Computing foremost journeys in TEGs

In this section we show how to compute foremost journeys from a source node s to all other nodes. To do this, we first present some properties of foremost journeys. We remark that this problem was studied under different settings in [5, 7, 14], and applications in robots networks were reported in [14].

Unfortunately again, it is obvious that a prefix journey of a foremost journey is not necessarily a foremost journey. Notwithstanding, the theorem below shows that there exist foremost journeys with such a property in a timed evolving graph. Further below, Property 1 shows how such journeys help computing earliest journeys in a timed evolving graph.

Theorem 1 (Ubiquitous earliest journey) *Let u and v be two vertices in a given timed evolving graph $\mathcal{G}$. If there is a journey in $\mathcal{G}$ linking u to v, then, among all journeys linking u to v, there exists at least one foremost journey such that all its prefix journeys are themselves foremost journeys. Such a journey is called* ubiquitous foremost journey (UFJ).

Proof: The theorem will be proved by induction on the earliest arrival date from u to v, $\hat{a}(u, v)$.

i. If $\hat{a}(u, v) = 0$, it means that $u = v$ and the empty journey satisfies the hypothesis of an UFJ.

ii. Now suppose that there exists an integer $p > 0$ such that, if $\hat{a}(u', v')$ is strictly smaller than p, then there is an UFJ from u' to v', for any given vertices u' and v'.

Then, let two vertices u and v be such that $\hat{a}(u, v) = p$. We prove that there is an UFJ from u to v as follows.

Since $\hat{a}(u, v)$ is not infinite, there are some journey in $\mathcal{G}$ between u and v. Among them, there is a foremost journey $\mathcal{J}$ on a route $(e_1, e_2, \ldots, e_k)$. As $\hat{a}(u, v)$ is strictly positive, $\mathcal{J}$ is not empty and $k \geq 1$. Let now w be the vertex which immediately precedes v in $\mathcal{J}$. Since ζ is strictly positive, the earliest arrival date between u and w is strictly smaller than p. Hence, there is an UFJ between u and w, denoted K.

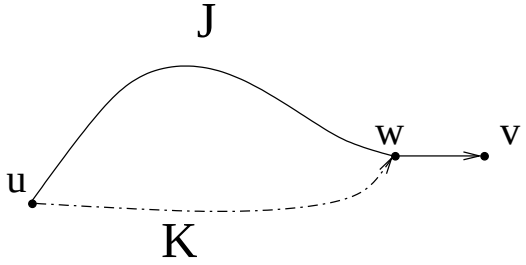

Figure 6: Ubiquitous foremost journey.

Besides, since $|K|_h$ is obviously smaller than $|\mathcal{J}_{|u\to w}|_h$, where $\mathcal{J}_{|u\to w}$ stands for the prefix journey of $\mathcal{J}$ restricted from u to w, we deduce that $K' = K \cup \{(w,v)$ traversed at time $\sigma((w,v))\}$ is a journey in $\mathcal{G}$. Furthermore, K' is a ubiquitous foremost journey between u and v, because it is a foremost journey itself (same arrival date as $\mathcal{J}$), and K is an ubiquitous foremost one.

iii. We deduce that for all couple of vertices u and v in $\mathcal{G}$, if u and v are connected, which is equivalent to $\hat{a}(u,v) < \infty$, then there is an UFJ between u and v.

□

We now point out how earliest arrival dates in an UFJ can be easily computed thanks to the function $f(e,i)$, which gives, for each arc $e = (u,v)$, and each time instant i, the earliest moment after i where node u can transmit the message to its neighbor v:

$$\forall e \in E_{\mathcal{G}}\ \forall i \in \mathbb{R}, \quad f(e,i) = \min\{t \geq i \mid \exists I \in P_E(e) \textit{ such that } [t, t+\zeta(e)] \subset I\}.$$

Property 1 (Computing earliest arrival dates in an UFJ) *Let s and v be two distinct vertices in $\mathcal{G}$, and $\mathcal{J}$ be an UFJ from s to v, on a route $(e_1, e_2, \ldots, e_k)$, and $k > 1$. Let u be the vertex which immediately precedes v in $\mathcal{J}$. Then $\hat{a}(s,v) = f(e_k, \hat{a}(s,u)) + \zeta(e_k)$.*

Proof: We prove that $\hat{a}(s,v) \leq f(e_k, \hat{a}(s,u)) + \zeta(e_k)$ as follows.
As $\mathcal{J}$ is an UFJ, we have $|\mathcal{J}_{|s\to u}|_a = \hat{a}(s,u)$.
Therefore,

$$\mathcal{J}' = \mathcal{J}_{|s\to u} \cup \{e_k \text{ traversed at time } f(e_k, \hat{a}(s,u))\}$$

is a journey between s and v.
Considering the arrival date of the journey $\mathcal{J}'$, we deduce that

$$\hat{a}(s,v) \leq f(e_k, \hat{a}(s,u)) + \zeta(e_k).$$

Since $\mathcal{J}'$ cannot leave node u before arriving at it, the property follows. □

4.1 Computing UFJs

Below, we give an efficient algorithm to compute the single-source UFJs in FSDNs. In opposition to Dijkstra's algorithm, we may have to open arcs from arbitrary closed nodes in case they appear in an upcoming G_i.

Algorithm 2

1. *Make all $d[v] = \infty$, but for $d[s] = 0$. Make all father$[v] = NIL$. Initialize a min-heap Q, sorted by d, with only s in the root.*
2. *While $Q \neq \emptyset$ do*

(a) *Extract x, the vertex at root(Q), and close it.*

(b) *Delete root(Q).*

(c) *Traverse the adjacency list of x, and for each open neighbor v do:*

i. *Compute $f(x, v, d[x])$.*

ii. *RELAX(x, v): If $f(x, v, d[x])+\zeta(x, v) < d[v]$ then update $d[v]$ with $f(x, v, d[x])+\zeta(x, v)$, update father$[v]$ with (x at $f(x, v, d[x])$), and insert v in Q if it was not there already.*

(d) *Update Q.*

Notice that our data structure allows a quick computation of the variable f. Indeed, with a binary search, this computation can be done in $O(\log \delta_E)$. As we have seen before, the variable $f(x, v, d[x])$ indicates the earliest that node x can transmit the message to its neighbor v. If $f(x, v, d[x])$ is early enough, then the earliest arrival date from the source s to v becomes the time instant when the transmission over (x, v) finishes, i.e. $f(x, v, d[x])+\zeta(x, v)$.

The foremost journey is found by backtracking the variable father. In case two successive time-labels differ by more than the corresponding ζ, this implies that the foremost journey yields a forced stay of the information in that vertex for a number of steps, until the connection is established to its successor.

We can see that, starting from s, the algorithm examines all its neighbors ($\Gamma(s)$), and for each one there is one table look-up to find the valid edge schedule times, plus a heap update. Therefore, for each closed vertex, the algorithm performs $O(\log \delta_E + \log N)$ operations. Hence, the total number of operations is at most $O(\sum_{v \in V_\mathcal{G}} [|\Gamma(v)|(\log \delta_E + \log N)])$ $= O(M(\log \delta_E + \log N))$.

The algorithm termination is clear: in each step of the loop 2, one vertex is closed and we never re-insert a closed vertex into the heap Q. Thus the loop is repeated at most N times, and the algorithm ends. The validity of the algorithm will be proved through the following lemmas.

Lemma 1 *For all vertices u in $V_\mathcal{G}$, $d[u] = \hat{a}(s, u)$ when u is closed.*

Proof: By induction on the set $C = \{closed\ vertices\}$.

i. At the beginning, $C = \{s\}$ and $d[s] = 0 = \hat{a}(s, s)$. The property holds.

ii. Suppose that at some moment the algorithm has correctly computed C, and a vertex u is to be closed, i.e., it is at the moment just before closing u.

If s and u are not connected, u can never be inserted in the heap Q. Consequently, it cannot be closed.

Thus, s and u are connected, and there is a journey between s and u. Let $\mathcal{J}$ be an UFJ from s to u. This journey links the vertex s inside of C to the vertex u outside of C. Let now y be the first vertex in $\mathcal{J}$ which is not in C, and x be the vertex which precedes immediately y in $\mathcal{J}$.

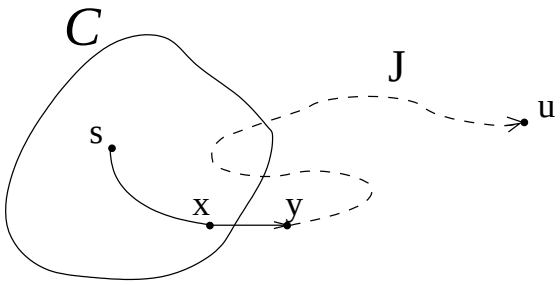

Figure 7: Validity of Algorithm 2: earliest arrival dates.

Since C has been correctly computed, then $d[x] = \hat{a}(s, x)$.

After $RELAX(x, y)$, which was performed when x was closed, we had

$$\begin{aligned} d[y] &\leq f(x, y, d[x]) + \zeta(x, y) \\ &\leq f(x, y, \hat{a}(s, x)) + \zeta(x, y) \\ &\leq \hat{a}(s, y) \text{ (} see\ Property1 \text{).} \end{aligned}$$

But $d[y]$ is larger or equal to $\hat{a}(s, y)$ by definition. Thus, we deduce that $d[y] = \hat{a}(s, y)$ at the considered moment.

Besides, it takes a positive time to go from y to u, implying $\hat{a}(s, y) \leq \hat{a}(s, u)$.

Hence,

$$\begin{aligned} d[y] &= \hat{a}(s, y) \\ &\leq \hat{a}(s, u) \\ &\leq d[u]. \end{aligned}$$

But y belongs to the heap Q −at least since $RELAX(x, y)$ was executed− and u has just been at the root of Q,

$$d[u] \leq d[y].$$

Whence, $d[y] = \hat{a}(s, y) = \hat{a}(s, u) = d[u]$, and C remains correctly computed after u is closed.

Notice as well that $\hat{a}(s, y) = \hat{a}(s, u)$, implying $y = u$ for y and u belonging to $\mathcal{J}$.

This result can be seen as follows. When we close a vertex x, if y is an out-neighbor of x, $RELAX(x, y)$ computes correctly $d[y]$ *if and only if* there is an UFJ between s and

y which uses the arc (x, y) to reach y. If such a journey does not exist, $RELAX(x, y)$ gives a (provisory) estimated value to $d[y]$, which will subsequently be modified and corrected before y is closed.

iii. Therefore, when the algorithm ends, all vertices u verify $d[u] = \hat{a}(s, u)$.

□

In order to prove that Algorithm 2 effectively computes UFJs between s and all other vertices, we first formally define how to recover UFJs from the variable father used in the procedure $RELAX$.

Definition 2 (UFJs induced by father)
The journey $(\mathcal{J}_v^{father}, \sigma_v^{father})$, induced by father$[v]$ between s and v, is defined recursively as follows.

i. If father$[v] = NIL$, $(\mathcal{J}_v^{father}, \sigma_v^{father})$ is defined as the empty journey.

ii. Let now a vertex u be such that $(\mathcal{J}_u^{father}, \sigma_u^{father})$ has been defined.
For all v with vertex(father$[v]) = u$, we first notice it is easy to see that
$I_{(u,v)} = [f(u, v, d[u]), f(u, v, d[u]) + \zeta(u, v)]$ belongs to some element in $P_E(u, v)$.
Then we define $(\mathcal{J}_v^{father}, \sigma_v^{father}) = (\mathcal{J}_u^{father}, \sigma_u^{father}) \cup \{(u, v)$ traversed at time$f(u, v, d[u])\}$.

Below, Lemma 2 shows that when Algorithm 2 ends, if a vertex v is connected to s, then $(\mathcal{J}_v^{\text{father}}, \sigma_v^{\text{father}})$ is well defined. Furthermore, $(\mathcal{J}_v^{\text{father}}, \sigma_v^{\text{father}})$ is an UFJ from s to v.

Lemma 2 *For all vertices v in $V_{\mathcal{G}}$, when v is closed, $(\mathcal{J}_v^{father}, \sigma_v^{father})$ is well defined and is a UFJ from s to v.*

Proof: The result will be proved by induction on the set C of closed vertices.

i. At the beginning $C = \{s\}$ and the property holds.

ii. Suppose that at some moment C has been correctly computed, and a vertex v is to be closed. As it was done in Lemma 1, we consider the instant just before closing v.

Since $d[v]$ was in the root of the heap Q, $d[v]$ is not infinite, implying we have previously made some relaxing over some arcs with destination v. Let $RELAX(u, v)$ be the last one. When this operation was executed, father$[v]$ was updated with $(u \text{ at } f(u, v, d[u]))$. As it is the last relaxing, father$[v]$ still keeps this value.

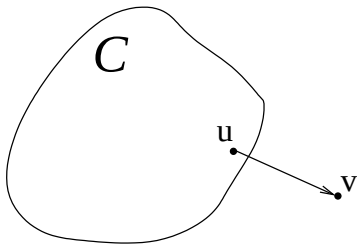

Figure 8: Validity of Algorithm 2: UFJs.

Now notice that $RELAX(u,v)$ can only be executed right after closing u. Therefore, $u \in C$, and $(\mathcal{J}_u^{\text{father}}, \sigma_u^{\text{father}})$ is well defined, and it is an UFJ from s to u. Hence, $(\mathcal{J}_v^{\text{father}}, \sigma_v^{\text{father}})$ is well defined. Furthermore, by Definition 2 $I_{(u,v)}$ is such that $(\mathcal{J}_v^{\text{father}}, \sigma_v^{\text{father}})$ can be easily proved as being a journey in $\mathcal{G}$ between s and v. Lastly, checking the arrival date, with

$$f(u,v,d[u]) + \zeta(u,v) = d[v] = \hat{a}(s,v),$$

we deduce that $(\mathcal{J}_v^{\text{father}}, \sigma_v^{\text{father}})$ is a foremost journey.

It is an UFJ because $(\mathcal{J}_u^{\text{father}}, \sigma_u^{\text{father}})$ is an ubiquitous foremost one, and the lemma follows.

□

A consequence of the above results is the following theorem.

Theorem 2 *Algorithm 2 correctly computes* UFJs *from a source node s to all others nodes in $O(M(\log \delta_E + \log N))$ time.*

5 Computing fastest journeys in EGs

In order to compute a fastest journey between any two vertices u and v, we could use the algorithm above to compute, for each time step i, the foremost journey between u and v in the sub-evolving graphs $(G, (G_t, G_{t+1} \dots G_{\mathcal{T}}))$, and then compute the minimum over all $t \leq T$. This would yield a time complexity of $O(M\mathcal{T}(\log \delta_E + \log N))$. We show in this section how to improve this time bound in the case of untimed evolving graphs.

Observe again that a prefix of a fastest journey is not a fastest journey. Unfortunately, this remains true, even if we use the notion of *ubiquitous* fastest journey, as we did in the previous section.

In untimed evolving graphs, journey times represent the number of subgraphs utilized, and a fastest journey between two vertices will be one that requires the minimum number of consecutive subgraphs to connect them. Thus, we can say that all nodes belonging to a same connected component in a given subgraph can reach each other in minimum time (defined as 0 in this case). Therefore, the algorithms given in this section are based on the computation of connected components in each and all subgraphs.

Computing connected components in a graph $G = (V, E)$ takes $O(|V| + |E|)$ steps [2], so our algorithms will perform at least $M\mathcal{T}+\Sigma_t|E_t|$ operations. However, since $delay(u, v) = 0$ if and only if they are connected in one of the subgraphs, at least for the computation of all pairs fastest journeys described in Section 5.2 below, such a computation represents a lower bound on the time complexity.

5.1 Single-source fastest journeys

The following algorithm first create a digraph where the vertices are the equivalent classes defined by the connected components. A connected component at time step t has an arc into a connected component at time step $t + 1$, if they share a common vertex.

Algorithm 3 (shrink)
Input : *An evolving graph $\mathcal{G}$.*
Ouput : *The digraph G_{CC}.*

1. *At each time step $1 \leq t \leq \mathcal{T}$ compute all the connected components CC of G_t, which gives the set $V_{CC}(t)$.*
2. *If a vertex u is in the connected component $CC \in V_{CC}(t)$ and in the connected component $CC' \in V_{CC}(t + 1)$, then add (CC, CC') to $E_{CC}(t)$.*
3. $G_{\mathrm{CC}} = (\bigcup V_{CC}(t), \bigcup E_{CC}(t))$.

Time complexity : $O(N\mathcal{T}+\Sigma_t|E_t|)$.

In order to compute minimum delays in this new represention, we need to connect the actual vertices of $\mathcal{G}$ to their connected components. This is done in the algorithm below.

Algorithm 4 (expand)
Input : G_{CC} *and* $\mathcal{G}$.
Output : *Another representation of* $\mathcal{G}$: $G_{exp} = (V_{exp}, E_{exp})$.

1. *Create* $V_{exp} = V \bigcup V_{CC}$.
2. $E_{exp} \leftarrow E_{CC}$.
3. *For each vertex $u \in V$, if u is in a connected component $CC \in V_{CC}$, then add (u, CC) to E_{exp}.*

Time complexity : $O(N\mathcal{T})$.

Now we can compute the single-source fastest journeys with the help of this digraph.

Lemma 3 *A shortest path between any two vertices u, v of V in G_{exp} is a fastest journey between u and v in $\mathcal{G}$.*

Proof: Let $\mathcal{J}$ be a fastest journey between u and v in $\mathcal{G}$. $\mathcal{J}$ goes through $|\mathcal{J}|_t + 1$ subgraphs G_i. Therefore, $\mathcal{J}$ goes through $|\mathcal{J}|_t + 1$ connected components, and the length of the corresponding path in G_{exp} is $|\mathcal{J}|_t + 2$. As a consequence, the length of paths in G_{exp} is two plus the journey time of the corresponding journey in $\mathcal{G}$. Hence, a shortest path between any two vertices u, v of V in G_{exp} is a fastest journey between u and v in $\mathcal{G}$. □

Corollary 1 *Given an evolving graph $\mathcal{G}(V_\mathcal{G}, E_\mathcal{G})$, the single-source fastest journeys can be computed in time $O((N\mathcal{T})\log(N\mathcal{T}))$.*

Proof: Once the expanded graph G_{exp} has been computed, computing single-source fastest journeys in $\mathcal{G}$ corresponds to computing single-source shortest paths in G_{exp}. The computation of these paths takes $O((|E_{exp}|)\log(|V_{exp}|))$ steps, which is in the worst case $O((N\mathcal{T})\log(N\mathcal{T}))$.
□

5.2 All-pairs fastest journeys

In this section, we show how to compute all the foremost journeys and all the fastest journeys between every pair (u, v) of vertices in one run. Although its time complexity is slightly worse than the previous algorithms, this computation is done only once.

This algorithm computes everything backwards, starting from time step $\mathcal{T}$, based on the following observation. At a given time step $t \leq \mathcal{T}$ and for a given couple (u, w) of vertices, a foremost journey between u and w starting from time step t is either a path from u to w at time step t if u and w are in the same connected component in G_t, or it is a path from u to v at time step t plus a foremost journey from v to w starting from time step t+1. Hence, a fastest journey between u and w starting from t is either a fastest journey between u and w starting from $t + 1$, or a foremost journey from u to w starting from time step t.

Algorithm 5 (all-pairs)
Input : *An evolving graph $\mathcal{G}$.*
Output : *All-pairs foremost journeys and all-pairs fastest journeys.*
Variables : *An array $E[N \times N \times (\mathcal{T}+1)]$ of earliest arrival dates, and an array $F[N \times N \times (\mathcal{T}+1)]$ of minimum delays.*

1. *For all pairs of vertices (u, v), let $E[u, v, \mathcal{T}+1] = \infty$ and let $F[u, v, \mathcal{T}+1] = \infty$.*
2. *For t from T down to 1 do:*
   (a) *Compute the connected components of G_t.*

(b) For all connected components CC of G_t do:

i. For all vertex $u \in CC$, set $D(CC, u) = 0$.

ii. For all vertex $v \notin CC$, set $D(CC, v) = 1 + \min_{u \in CC} E[u, v, t+1]$.

iii. For all vertex $u \in CC$, for all vertex v, let $E[u, v, t] = D(CC, v)$.

(c) For all pairs of vertices (u, v), let $F[u, v, t] = \min(E[u, v, t], F[u, v, t+1])$.

Space complexity : $O(N^2)$.
Time complexity : $O(\mathcal{T}N^2)$.

The arrays E and F span from time step 1 to time step $\mathcal{T}+1$, but this is only for convenience of notations. In fact, at each iteration, we need only the data of time step $t+1$. Therefore, we do not need to keep track of the matrices from time step $t+2$ to $\mathcal{T}+1$, so the space complexity is only N^2.

As a consequence, we have the following result.

Proposition 2 *Given an evolving graph $\mathcal{G}(V_\mathcal{G}, E_\mathcal{G})$, the all-pairs foremost journeys and the all-pairs fastest journeys can be computed in time $O(\mathcal{T}N^2)$.*

6 Conclusion and perspectives

Our contribution in this paper rests in the formalizing of a FSDN thanks to evolving graphs, and in exploiting the model to decrease the complexity of standard algorithms. In this paper, we focussed more precisely on journeys, the extension of paths over time, which embodies the traversal of the network from one node to another, associating each link to a time schedule. The most important results lie in the computation of shortest, foremost, and fastest journeys depending on what one wants to minimize: the hop-count, the arrival date or the time of a journey.

We are currently working on other route discovery problems in timed evolving graphs. Flows and queues over evolving graphs could prove to be another interesting area of study to analyze buffering issues in dynamic networks. Moreover, the framework of evolving graphs could be suited to the case of random variations of link costs and connectivity for a formal analysis of problems in dynamic networks with unpredictable topology changes.

Acknowledgments

The authors are grateful to Frédéric Havet, Karina Marcus, Hervé Rivano, and Laurent Viennot, for very fruitful discussions, which in particular led to preliminary forms of the algorithm in Section 4.

References

[1] S. Bhadra and A. Ferreira. Computing multicast trees in dynamic networks using evolving graphs. Research Report 4531, INRIA, 2002.

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

[3] C.Scheideler. Models and techniques for communication in dynamic networks. In In H. Alt and A. Ferreira, editors, *Proceedings of the 19th International Symposium on Theoretical Aspects of Computer Science*, volume 2285, pages 27–49. Springer-Verlag, March 2002.

[4] E. Ekici, I. F. Akyildiz, and M. D. Bender. Datagram routing algorithm for LEO satellite networks. In *IEEE Infocom*, pages 500–508, 2000.

[5] A. Ferreira. On models and algorithms for dynamic communication networks: The case for evolving graphs. In *Proceedings of* 4^e *rencontres francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL'2002)*, pages 155–161, Mèze, France, May 2002. INRIA Press.

[6] A. Ferreira, J. Galtier, and P. Penna. Topological design, routing and handover in satellite networks. In I. Stojmenovic, editor, *Handbook of Wireless Networks and Mobile Computing*, pages 473–493. John Wiley and Sons, 2002.

[7] A. Ferreira and L. Viennot. A note on models, algorithms, and data structures for dynamic communication networks. Technical Report 4403, INRIA, 2002.

[8] L. Fleisher and Martin Skutella. The quickest multicommodity flow problem. In *Proc. of IPCO'02*, 2002.

[9] L.R. Ford and D.R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6:419–433, 1958.

[10] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[11] J. Holm, K. De Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

[12] E. Köhler, K. Langkau, and M. Skutella. Time-expanded graphs for flow-dependent transit times. In *proc. ESA'02*, 2002.

[13] E. Köhler and M.Skutella. Flows over time with load-dependent transit times. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 174–183, 2002.

[14] L. Viennot. Routage entre robots dont les déplacements sont connus – Un exemple de graphe dynamique. Réunion TAROT, ENST, Paris, Novembre 2001.

[15] P.-J. Wan, G. Calinescu, X. Li, and O. Frieder. Minimum-energy broadcast routing in static ad hoc wireless networks. In *Proc. IEEE Infocom*, pages 1162–1171, Anchorage, Alaska, 2001.

[16] M. Werner and G. Maral. Traffic flows and dynamic routing in leo intersatellite link networks. In *In Proceedings 5th International Mobile Satellite Conference (IMSC '97)*, Pasadena, California, USA, June 1997.

[17] M. Werner and F. Wauquiez. Capacity dimensioning of ISL networks in broadband LEO satellite systems. In *Sixth International Mobile Satellite Conference : IMSC 99*, pages 334–341, Ottawa, Canada, June 1999.

[18] J. Wieselthier, G. Nguyen, and A. Ephremides. On the construction of energy-efficient broadcast and multicast trees in wireless networks. In *Proc. IEEE Infocom*, pages 585–594, Tel Aviv, 2000.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399