International Journal of Foundations of Computer Science

COMPUTING SHORTEST, FASTEST, AND FOREMOST JOURNEYS IN DYNAMIC NETWORKS*

B. BUI XUAN
École Normale Supérieure de Lyon, 46 allée d'Italie
69007 Lyon, France
Binh.Minh.Bui.Xuan@ens-lyon.fr

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

and

A. JARRY
I3S & INRIA Sophia Antipolis, 2004 Route des Lucioles
06902 Sophia Antipolis Cedex, France
Aubin.Jarry@sophia.inria.fr

Received (received date)
Revised (revised date)
Communicated by Editor's name

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, 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, where the changes in the topology can be predicted in advance. Cost measures investigated here are hop count (*shortest* journeys), arrival date (*foremost* journeys), and time span (*fastest* journeys).

Keywords: dynamic networks, routing, evolving graphs, graph theoretical models, LEO satellite networks, fixed-schedule dynamic networks

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

1. Introduction

Infrastructure-less mobile communication environments, such as mobile ad-hoc networks and low earth orbiting (LEO) satellite systems, present a paradigm shift from back-boned 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 [16]. In this setting, the generalized case of 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. This naturally motivates studying the modeling of such dynamics, and designing algorithms that take it into account [17].

Note, however, that for the case of sensor networks, LEO satellite systems and other mobile networks with predestined trajectories of the mobile agents, the network dynamics are somewhat deterministic. LEO satellite networks [4, 6, 19, 20] 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 (*intra-planar*) do not vary with time because the satellites move with zero relative angular velocity to each other, *inter-planar* 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 in advance, it is possible to exploit this determinism in optimizing routing strategies [6].

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.

Literature related to route discovery issues in dynamic networks started more than four decades ago, with papers dealing with operations of transport networks (e.g., [3, 8, 9, 10, 11, 15]). Recent work on time-dependent networks can be found in [7, 13, 14], 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 [8], namely of expanding the original graph into layers representing the time steps, may work for computing several path-related problems (see [7, 13, 14] and references therein).

In the non-discrete case, this approach might also be employed, but the time complexity explodes because of the requirement of many layers (roughly, one per edge) and of the time discretization. This precludes the use of the time-expanded graph approach, since the expansion factor would be huge, given that there are many networks to expand, and they can have non-discrete traversal times. Therefore, different techniques were developed in the literature in order to cope with the dynamics of networks as well as with their time dependency. For instance, in [3, 10] shortest time paths were first addressed and in [15], the continuous flow problem

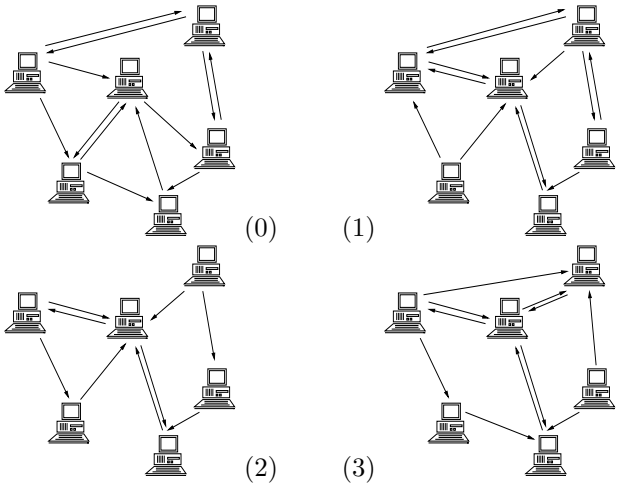

Figure 1: An FSDN represented as an indexed set of networks. The indices correspond to successive time-steps.

was discussed.

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 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 below, and Section 2.2).

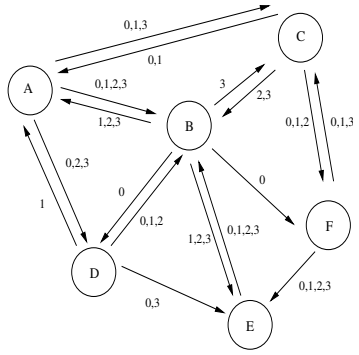

Figure 2: Evolving graph corresponding to FSDN in Figure 1. Edges are labeled with corresponding time-steps. Observe that CBF is not a valid journey since BF exists only in the past with respect to CB.

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 edges. 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 arrival date (*foremost* journeys), hop-count (*shortest* journeys), and time span (*fastest journeys*).

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 foremost (earliest arrival date) journeys, shortest (minimum hop count) journeys and fastest (minimum time) journeys, are given and analyzed in the subsequent sections. Since these questions in untimed evolving graphs can be solved through the time-expansion approach, our algorithms are all designed for timed evolving graphs. 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 be given a graph $G(V, E)$ and an ordered sequence of its subgraphs, $\mathcal{S}_G = G_1, G_2, \ldots, G_{\mathcal{T}}$ such that $\bigcup_{i=1}^{\mathcal{T}} G_i = G$. Then, the system $\mathcal{G} = (G, \mathcal{S}_G)$ is called an evolving graph.*

Let $E_{\mathcal{G}} = \bigcup E_i$, and $V_{\mathcal{G}} = \bigcup V_i$. We denote $M = |E_{\mathcal{G}}|$ and $N = |V_{\mathcal{G}}|$. 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 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 paper, we denote it $\mathbb{T} = \mathbb{R}_+ \bigcup \{\infty\}$). In this case, we shall speak of *timed* evolving graphs because the weights on the edges will represent their traversal time.

Consider $\mathcal{I} = [t_1, t_{\mathcal{T}+1}[\subset \mathbb{T}$ 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 edge. 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 $\mathcal{G} = (\bigcup \mathcal{N}_i | i \in \mathcal{W}, \mathrm{FSDN}_{|\mathcal{W}})$, and the function ζ, redefined, from $E_\mathcal{G}$ to $\mathbb{T}$.

2.2. Journeys

We call *route in* $\mathcal{G}$ a path $R = e_1, e_2, \ldots, e_k$ with $e_i \in E_\mathcal{G}$ in G. Let $R_\sigma = \sigma_1, \sigma_2, \ldots, \sigma_k$ with $\sigma_i \in \mathbb{T}$ be a time schedule indicating when each edge of the route R is to be traversed. We define a *journey* $\mathcal{J} = (R, R_\sigma)$ if and only if R_σ is in accordance with R, ζ, $\mathcal{G}$ and $\mathcal{I}$, i.e., $\mathcal{J}$ allows 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 $\mathcal{J} = (R, R_\sigma)$ in $\mathcal{G}$, where R is a cycle in 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 the same route again.

2.3. Distances

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

Let $\mathcal{J} = (R, R_\sigma)$ be a journey where $R = e_1, e_2, \ldots, e_k$ and $R_\sigma = \sigma_1, \sigma_2, \ldots, \sigma_k$. Then,

- The *hop-count* or *length* of $\mathcal{J}$ is defined as $|\mathcal{J}|_h = |R| = k$. It is also denoted $h(\mathcal{J})$.
- The *arrival date* of $\mathcal{J}$ is defined as $|\mathcal{J}|_a = \sigma_k + \zeta(e_k)$, 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 $\mathcal{J}$ is defined as the elapsed time between the departure and the arrival, i.e. $|\mathcal{J}|_t = |\mathcal{J}|_a - \sigma_1$. 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 $d(u, v) = \min\{|\mathcal{J}|_h\}$, 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 $a(u, v)$. 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 $a(u, v) = \infty$.
- The *delay* in $\mathcal{G}$ between two vertices u and v is defined as $delay(u, v) = \min\{|\mathcal{J}|_t\}$, 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, $d(u, v)$ 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 $a(u, v)$ 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 presence schedule* $P_E(e)$ (respectively, *node presence 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 alternatively define an evolving graph as $\mathcal{G} = (V_{\mathcal{G}}, E_{\mathcal{G}})$, where each node and edge has a schedule defined for it. An edge e and a schedule σ are valid in a journey, if and only if e is present at least in the interval $[\sigma; \sigma + \zeta(e)]$. To simplify our computations, it is safe to assume that the intervals of presence of an edge e are closed and that they are longer than $\zeta(e)$.

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 when 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).

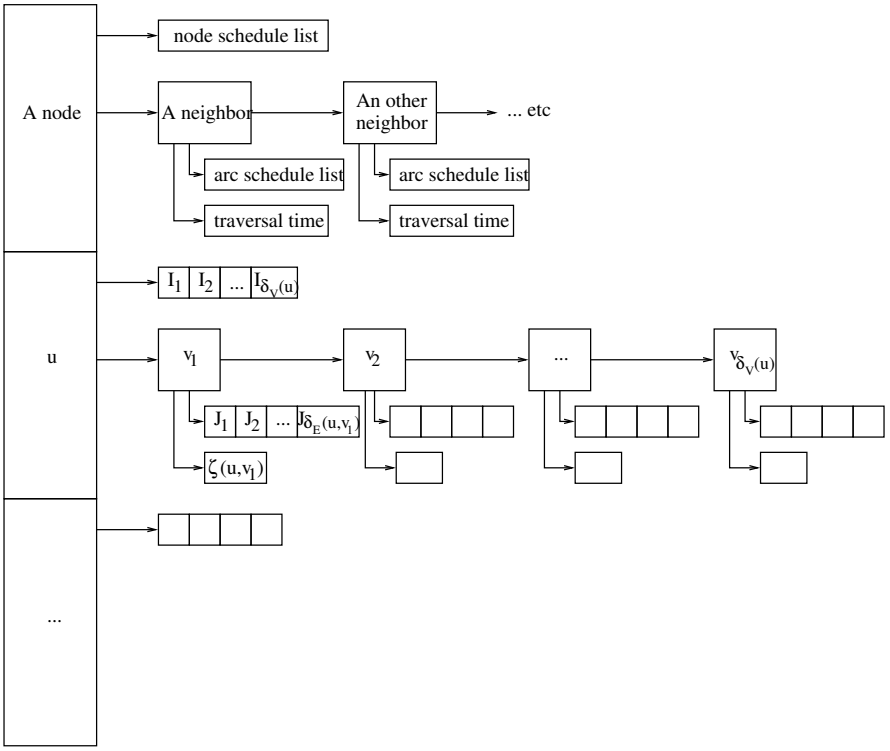

Figure 3: The data structure for coding a fixed-schedule dynamic network modeled by an evolving graph.

Thus, for each vertex v, and each edge 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 edges.

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

2.6. A Very Useful Function

Below we give a standard function on timed evolving graphs which will be used in the remainder of this paper. Let consider an edge $(u, v) \in E_{\mathcal{G}}$ and a time instant t. We call $f((u, v), t)$ the function which gives, for each edge (u, v), and each time instant t, the earliest moment after t where node u can transmit a message to v. If such a moment does not exist, f returns ∞.

Notice that our data structure allows for a quick computation of the function f. Indeed, with a binary search, this computation can be done in $O(\log \delta_E)$.

3. Computing Foremost Journeys

In this section we show how to compute foremost journeys from a source node s to all other nodes, problem already studied several times in the literature, as shortest (time) path in time-dependent (transport) networks, e.g. in [3, 9, 11, 18]. To our knowledge, ours is the first algorithm to be presented with a formal and detailed complexity analysis for the non-discrete case. 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, 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.

Proposition 1 (Ubiquitous earliest journey) *Let s and v be two vertices in a given timed evolving graph $\mathcal{G}$. If there is a journey in $\mathcal{G}$ linking s to v, then, among all journeys linking s 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. Let $\mathcal{J} = (e_1, \ldots, e_k, \sigma_1, \ldots, \sigma_k)$ be a journey from s to v. If the hop-count $|\mathcal{J}|_h = k$ of $\mathcal{J}$ is greater than $N \times \delta_V + 1$, then there are two integers $i < j$, a vertex u and a time interval $[t_1, t_2]$ such that both e_i and e_j start from u, and both σ_i and σ_j are in $[t_1, t_2]$. In this case, we can produce $R' = e_1, \ldots, e_{i-1}, e_j, \ldots, e_k$ and $R'_\sigma = \sigma_1, \ldots, \sigma_{i-1}, \sigma_j, \ldots, \sigma_k$, so that $\mathcal{J}' = (R', R'_\sigma)$ is a journey from s to v with $|\mathcal{J}'|_h < |\mathcal{J}|_h$ and $|\mathcal{J}'|_a = |\mathcal{J}|_a$. Observe that if $\mathcal{J}$ was an UFJ, then so is $\mathcal{J}'$. This means that the only relevant journeys for our problem contain at most $N \times \delta_V$ edges, and that if there is a UFJ, then there is an UFJ with less than $N \times \delta_V$ edges.

Now, let $\mathcal{J} = (e_1, \ldots, e_k, \sigma_1, \ldots, \sigma_k)$ and $\mathcal{J}' = (e'_1, \ldots, e'_{k'}, \sigma'_1, \ldots, \sigma'_{k'})$ be two journeys from s to v. We say that $\mathcal{J} \leq_{ubiquituous} \mathcal{J}'$ if and only if:

1. $|\mathcal{J}|_a < |\mathcal{J}'|_a$ or
2. $|\mathcal{J}|_a = |\mathcal{J}'|_a$ and $\sigma_k < \sigma'_k$ or
3. $|\mathcal{J}|_a = |\mathcal{J}'|_a$, there is i such that $\forall j > i$, $\sigma_j = \sigma'_j$ and $\sigma_i < \sigma'_i$.
4. $|\mathcal{J}|_a = |\mathcal{J}'|_a$, $\forall j$, $\sigma_j = \sigma'_j$.

Notice that $\leq_{ubiquituous}$ defines a total pre order relation[a] over all the journeys from

[a] I.e., transitive and reflexive.

s to v. The space of relevant journeys from s to v is bounded in time and in space ($N \times \delta_V$) and closed (we consider closed intervals of presence), so it has a minimum. Observe that such a minimum is an UFJ, so this proves the existence of an UFJ from s to v. □

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

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, with $k = |\mathcal{J}|_h \geq 1$. Let u be the vertex which immediately precedes v in $\mathcal{J} = (R(u), (u,v), R_\sigma(u), \sigma_k)$. Then $a(s,v) = f((u,v), a(s,u)) + \zeta(u,v)$.*

Proof. As $\mathcal{J}$ is an UFJ, we have $|(R(u), R_\sigma(u))|_a = a(s,u)$. Also, considering the arrival date of the journey $\mathcal{J}$, we deduce that $a(s,v) \leq f((u,v), a(s,u)) + \zeta(u,v)$. Since $\mathcal{J}$ cannot leave node u before arriving at it, the property follows. □

3.1. Computing UFJs

Below, we give an efficient algorithm to compute the single-source UFJs in evolving graphs.

Algorithm 1 (UFJs)
Input : *An evolving graph $\mathcal{G}$, a vertex $s \in V_\mathcal{G}$*
Output : *An array $t_{EAD}[v] \in \mathbb{T}$ which gives for each vertex $v \in V_\mathcal{G}$ the Earliest Arrival Date from s; and an array $father[v] \in V_\mathcal{G}$ which gives for each vertex $v \neq s \in V_\mathcal{G}$ its father in the ubiquitous foremost journey tree.*
Variables : *A min-heap Q of vertices, sorted by the array t_{EAD}. The array t_{EAD} will be updated.*

1. *Initialize $t_{EAD}[s] \leftarrow 0$; and for all $v \neq s \in V_\mathcal{G}$, $t_{EAD}[v] \leftarrow \infty$. Initialize Q with only s in the root.*
2. *While $Q \neq \emptyset$ do:*
   (a) *Extract u, the vertex at $root(Q)$, and close it.*
   (b) *Delete $root(Q)$.*
   (c) *Traverse the adjacency list of u, and for each open neighbor v do:*
      i. *Let $t = f((u,v), t_{EAD}[u])$.*
      ii. *If $t + \zeta(u,v) < t_{EAD}[v]$ then*
      Update $t_{EAD}[v] \leftarrow t + \zeta(u,v)$,
      Update $father[v] \leftarrow u$ and
      insert v in the Q if it was not there already.
   (d) *Update Q.*

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.

The algorithm termination is clear. In each step of 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 lemma.

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

Proof. By induction on the set C of closed vertices. At the beginning, $C = \{s\}$ and $t_{EAD}[s] = 0 = a(s,s)$. The property holds.

Suppose that at some moment the algorithm has correctly computed C, and a vertex v is to be closed, i.e., the algorithm is at the moment just before closing v. Thus v has been inserted in the heap Q, so s and v are connected. Let $\mathcal{J}$ be an UFJ from s to v. This journey links the vertex s inside of C to the vertex v outside of C. Let now v' be the first vertex in $\mathcal{J}$ which is not in C, and u be the vertex which precedes immediately v' in $\mathcal{J}$ (see Figure 3.1).

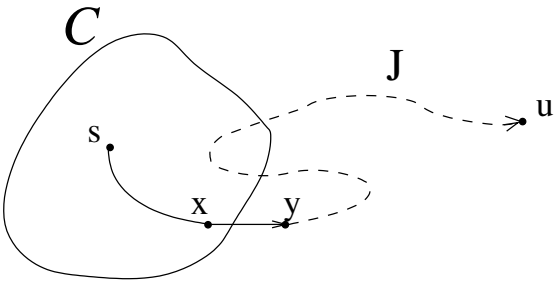

Figure 4: Validity of Algorithm 1: earliest arrival dates.

Since C has been correctly computed, then $t_{EAD}[u] = a(s,u)$. When u was closed, v' was inserted in the Q, and since v' is before v in the journey $\mathcal{J}$, $t_{EAD}[v'] \leq t_{EAD}[v]$ and clearly $v' = v$. Furthermore, Property 1 states that $a(s,v) = f((u,v), a(s,u))$, hence $t_{EAD}[v] = a(s,v)$. □

We can see that, starting from s, the algorithm examines all its neighbors, 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 per neighbor. Hence, the total number of operations is at most $O(\sum_{v \in V_{\mathcal{G}}} [|\Gamma(v)|(\log \delta_E + \log N)]) = O(M(\log \delta_E + \log N))$.

A consequence of the above results is the following theorem.

Theorem 1 *Algorithm 1 correctly computes* UFJ*s from a source node s to all others nodes in $O(M(\log \delta_E + \log N))$ time.*

4. Computing Shortest Journeys

In this section, we focus on the hop-count of journeys, which we want to minimize. We will again use an approach close to Dijkstra's algorithm [2], computing all the shortest journeys from a single vertex s to all the other vertices.

The difficulty stems from the edge traversal times, which again make that prefix of shortest journeys are not necessarily shortest (see Figure 5).

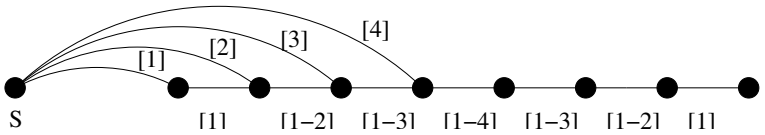

Figure 5: The shortest journey from s to its antipode takes 8 hops at time step 1, whereas there is a shortcut a time step 4 to the fourth point.

Nevertheless, we note that if the last edge, say (u, v), of a shortest journey between vertex s and vertex v arrives at time t, then the prefix journey (going from s to u) is shorter than all the journeys from s to u ending *before* t. Therefore, we will consider certain pairs $(u, t) \in V_{\mathcal{G}} \times \mathbb{T}$ 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 journeys between s and pairs (u, t), in which each vertex u appears at least once (see Figure 6).

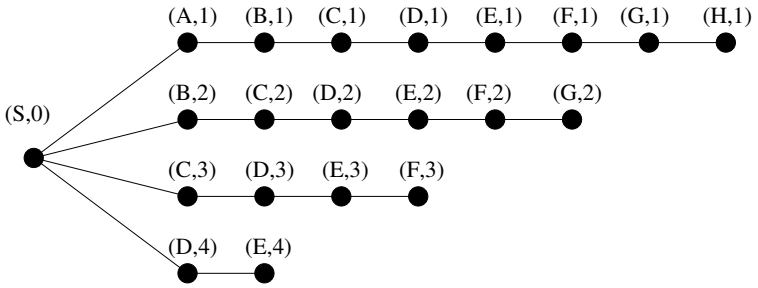

Figure 6: Tree of shortest paths.

In order to proceed, we introduce Algorithm 2, below. Given an array t_{LBD} (indexed on $u \in V_{\mathcal{G}}$) of Lower Bound on Departure times $t_{LBD}[u] \in \mathbb{T}$, it computes an array e_{min} and an array t_{min} (indexed on $v \in V_{\mathcal{G}}$) of edges $e_{min}[v] \in E_{\mathcal{G}}$ and schedules $t_{min}[v] \in \mathbb{T}$ such that $e[v] = (u, v)$ exists during the whole $[t_{min}[v], t_{min}[v] + \zeta(e_{min}[v])]$ time interval; and such that $t_{LBD}[u] \leq t_{min}[v]$. Moreover, the couple $(e_{min}[v], t_{min}[v])$ is chosen so that $t_{min}[v] + \zeta(e_{min}[v])$ is minimal over all possible couples, and we add the condition that $t_{min}[v] + \zeta(e_{min}[v]) \leq t_{LBD}[v]$. If no such couple exist, the default one is (nil, ∞).

Algorithm 2 (Edge and schedule selection)

Input: *A timed evolving graph $\mathcal{G}$ and an array $t_{LBD}[u] \in \mathbb{T}$ which gives for each $u \in V_{\mathcal{G}}$ a Lower Bound on Departure time.*

Output : *Two arrays $e_{min}[v] \in E_{\mathcal{G}}$ and $t_{min}[v] \in \mathbb{T}$ which give for each $v \in V_{\mathcal{G}}$ an edge $e_{min}[v] = (u, v)$ along with a schedule $t_{min}[v]$ for this edge.*

Variable : *An array $t_{arrival}[v] \in \mathbb{T}$ which gives the arrival date of the former edge($t_{arrival} = t_{min} + \zeta(e_{min})$). The output array will vary.*

1. *For all $v \in V_{\mathcal{G}}$ initialize $e_{min}[v] \leftarrow nil$; $t_{min}[v] \leftarrow \infty$ and $t_{arrival}[v] \leftarrow t_{LBD}[v]$.*
2. *For all (u, v) in $E_{\mathcal{G}}$ do:*
   (a) *Let $t = f((u, v), t_{LBD}[u])$.*
   (b) *If $(t + \zeta(u, v)) < t_{arrival}[v]$ then*
      i. *Let $e_{min}[v] \leftarrow (u, v)$.*
      ii. *Let $t_{min}[v] \leftarrow t$.*
      iii. *Let $t_{arrival}[v] \leftarrow t + \zeta(u, v)$.*

Now, observe that given a shortest journey with hop-count k, all its prefixes have a hop-count smaller than $k - 1$. Our algorithm will compute all the arrival dates with hop-count $k-1$, and then proceed for hop-count k. The algorithm stops when all vertices have an arrival date smaller than ∞, and keeps track of the first time when a vertex was encountered and its shortest path from s. The number of iterations in our algorithm is of course the eccentricity of $\mathcal{G}$.

Algorithm 3 (Shortest journeys)
Input: *A timed evolving graph $\mathcal{G}$, a vertex $s \in V_{\mathcal{G}}$.*
Output: *An array $\mathcal{J}_{shortest}[v] \in$ {set of all journeys} which gives for each $v \in V_{\mathcal{G}}$ a shortest journey from s to v.*
Variables: *An array $\mathcal{J}[v] \in$ {set of all journeys} which gives for each $v \in V_{\mathcal{G}}$ a journey from s to v; an array $t_{LBD}[u] \in \mathbb{T}$ which gives for each $u \in V_{\mathcal{G}}$ a Lower Bound on Departure time; two arrays $e_{min}[v] \in E_{\mathcal{G}}$ and $t_{min}[v] \in \mathbb{T}$ which give for each $v \in V_{\mathcal{G}}$ an edge $e_{min}[v] = (u, v)$ along with a schedule $t_{min}[v]$ for this edge; and a number of hops $k \in \mathbb{N}$.*

1. *Initialize $t_{LBD}[s] \leftarrow 0$, $\mathcal{J}[s] \leftarrow ()$ and define $\mathcal{J}_{shortest}[s] = ()$; for all $v \neq s$ $t_{LBD}[v] \leftarrow \infty$ and $\mathcal{J}[v] \leftarrow ()$; $k \leftarrow 0$.*
2. *While there is $v \in V_{\mathcal{G}}$ such that $t_{LBD}(v) = \infty$ and While $k < N$ do:*
   (a) *$k \leftarrow k + 1$*
   (b) *Call Algorithm 2 with input $(\mathcal{G}, t_{LBD})$, and store the results in the arrays e_{min} and t_{min}.*
   (c) *For each vertex $v \in V_{\mathcal{G}}$ do:*
      If $e_{min}[v] \neq nil$ then
      i. *Let $e_{min}[v] = (u, v)$.*
      ii. *Let $(R, R_\sigma) = \mathcal{J}[u]$.*
      iii. *Update $\mathcal{J}[v] \leftarrow (R, e_{min}[v], R_\sigma, t_{min}[v])$.*
      iv. *If $t_{LBD}[v] = \infty$ then define $\mathcal{J}_{shortest}[v] = \mathcal{J}[v]$.*

v. Update $t_{LBD}[v] \leftarrow t_{min}[v] + \zeta(e_{min}[v])$.

Proposition 2 *The algorithm above computes shortest journeys from a single source s to all the vertices in $\mathcal{G}$, if such journeys exist. If $\mathcal{G}$ is connected, then the complexity of the algorithm is $O(Md\ log\delta_E)$, where d is the eccentricity of s. If $\mathcal{G}$ is not connected the complexity of the algorithm is $O(NM\ \log\delta_E)$*

Proof. For each k, all the edges are processed, and the retrieval of arrival dates takes $O(log\delta_E)$, so the overall complexity for a value of k is $O(M\ log\delta_E)$. We will prove by induction that for each k, the earliest arrival date with k hops is computed for the couple $(s, v), \forall v \in V_{\mathcal{G}}$. It is immediately true for $k = 0$. As in Section 3, observe that a journey that gives such an arrival date can be computed via its prefix, which has $k-1$ hops. Thus, the shortest journeys of k hops are reached after step k. So the overall complexity is $O(Md\ log\delta_E)$, where d is the eccentricity of s, if $\mathcal{G}$ is connected. In case $\mathcal{G}$ is not connected, the algorithm will stop because $k > N$ at step 2. In this case, the overall complexity is $O(NM\ \log\delta_E)$. □

5. Computing Fastest Journeys

In this section, we are interested in the *journey time* measure, and we will compute fastest journeys from s to all the other vertices. This problem is much more complex than the two former ones, because a faster journey may appear well ahead in time, or can be really long compared to the shortest journeys. Moreover, the speed of a journey prefix is almost irrelevant regarding the speed of the whole journey. Indeed, a fast prefix may well imply a long waiting time, offsetting the apparent gain in speed. On the other hand, some prefix journeys are too slow and hence useless to our computations. The remaining prefixes will be grouped within classes of *relevant* journeys.

As in Section 4, we will proceed hop by hop, since the number of hops in a fastest journey is also bounded by N. For each k, we will build a list of relevant journey classes of length k starting in s, by taking the list for hop-count $k-1$ and extending its relevant journey classes to k hops. This is done by examining each edge of the graph, building the journeys classes that can go through this edge, and then eliminating irrelevant journey classes. After N hops, we know that the fastest journeys are included in the final relevant journey class list. Therefore it suffices to search for the minimum journeys in this list to obtain the requested fastest journeys. The number of relevant classes is bounded by the size of $\mathcal{G}$, and thus the complexity of our algorithm remains reasonable.

For each journey, the measure of quality (the journey time) is different from the hop-count, and from the arrival date. We will not keep track of the hop-count, which is implicit in our algorithm. We will however use it to stop the algorithm (after hop-count $N-1$). Regarding the two other parameters, we will keep track at each step of several possible journeys from the source s to some vertex v, along with the arrival date $t_{arrival}$ of each journey and the journey time t_{speed} of each journey. Given two journeys from s to v with departure date $t_{departure_1}, t_{departure_2}$ and arrival dates $t_{arrival_1}, t_{arrival_2}$, observe that if we have both $t_{departure_1} \geq t_{departure_2}$ and

$t_{arrival_1} \leq t_{arrival_2}$, then not only journey 1 starts after journey 2, but journey 1 arrives before journey 2. In this case journey 2 is useless for our problem, so we will keep track only of journey 1.

Definition 2 (Relevance) *Let $\leq_{route}$ be an arbitrary order on the routes in G. Let $\leq_{rel}$ be an order on the journeys of $\mathcal{G}$ defined as follows.*

1. *(on speed) $\mathcal{J}_1 <_{rel} \mathcal{J}_2$ if $|\mathcal{J}_1|_t < |\mathcal{J}_2|_t$ or*
2. *(on hop-count) $\mathcal{J}_1 <_{rel} \mathcal{J}_2$ if $|\mathcal{J}_1|_t = |\mathcal{J}_2|_t$ and $|\mathcal{J}_1|_h < |\mathcal{J}_2|_h$ or*
3. *(on $\leq_{route}$) $\mathcal{J}_1 = (R_1, R_{\sigma_1}) \leq_{rel} \mathcal{J}_2 = (R_2, R_{\sigma_2})$ if $|\mathcal{J}_1|_t = |\mathcal{J}_2|_t$ and $|\mathcal{J}_1|_h = |\mathcal{J}_2|_h$ and $R_1 \leq_{route} R_2$.*

Definition 3 (Relevant journeys, Irrelevant journeys) *Let $\mathcal{G}$ be a timed evolving graph, and $u, v \in V_{\mathcal{G}}$. Let $\mathcal{J}$ be a journey from u to v, with departure time $t_{departure}$ and arrival time $t_{arrival}$. If there is another journey $\mathcal{J}'$ from u to v which starts at $t'_{departure}$ and arrives at $t'_{arrival}$ such that $\mathcal{J}' \leq_{rel} \mathcal{J}$ and $[t_{departure}, t_{arrival}] \subset [t'_{departure}, t'_{arrival}]$, then $\mathcal{J}$ is called* irrelevant journey*. Otherwise, $\mathcal{J}$ is called* relevant journey*.*

Another observation is that journeys with no *waiting time* in any vertex may yield a whole class of journeys with the same speed. To see this, consider two edges (u, v) and (v, w). Let edge (u, v) have traversal time 3 and be valid at interval $[1, 8]$. Let edge (v, w) have traversal time 4 and be valid at interval $[5, 13]$. The journey $((u, v), (v, w), 3, 6)$ arrives takes 7 time steps to complete. But so do all journeys $((u, v), (v, w), [2, 5], [5, 8])$, as illustrated in Figure 7.

In order to handle this, given a journey $\mathcal{J} = (R, R_\sigma)$ with $R = e_1, e_2, \ldots, e_k$ and $R_\sigma = \sigma_1, \sigma_2, \ldots, \sigma_k$, if each edge e_i is valid during interval $[\sigma_i, \sigma_i + \delta + \zeta(e_i)]$ for some $\delta \in \mathbb{T}$, then we introduce schedule intervals $I(R_\sigma, \delta) = [\sigma_1, \sigma_1 + \delta], [\sigma_2, \sigma_2 + \delta], \ldots, [\sigma_k, \sigma_k + \delta]$ and we say that $(R, I(R_\sigma, \delta))$ is a *class of journeys*. Observe that for all $\epsilon \in [0, \delta]$, $(R, R_\sigma + \epsilon)$ is a journey. A journey class will be called *relevant journey class* if it contains only relevant journeys. A relevant journey class that is contained in no other relevant journey class will be called *maximal relevant journey class*. We are interested in these maximal relevant journey classes.

Lemma 2 (Maximal Relevant Journey Classes) *Given an evolving graph $\mathcal{G}$, two vertices $u, v \in V_{\mathcal{G}}$ and $k \in \mathbb{N}$, the number of maximal relevant journey classes of k hops (or less) from u to v is bounded by the edges of $\mathcal{G}$ and their activity, that is $2M\delta_E$.*

Proof. Let us consider a maximal relevant journey class $(R, I(R_\sigma, \delta))$ of k hops (or less) from u to v. Let δ_{max} be the biggest integer, such that $(R, I(R_\sigma, \delta_{max}))$ is a journey class. If $\delta = \delta_{max}$, then there is an edge $e_i \in R$ with schedule $\sigma_i \in R_\sigma$, such that $\sigma_i + \delta$ is the upper extremity of a presence interval of e_i. Clearly, no other relevant journey may use e_i during $[\sigma_i, \sigma_i + \delta]$. On the other hand, if $\delta_{max} > \delta$, then there is a maximal relevant journey class $(R', I(R'_\sigma, \delta'))$ which is considered better than $(R, I(R_\sigma, \delta))$, that is $(R', R'_\sigma) <_{rel} (R, R_\sigma)$ and $|(R', R'_\sigma)|_a = |(R, R_\sigma + \delta)|_a$. In this case, there is an edge $e'_i \in R'$ and a schedule $\sigma'_i \in R'_\sigma$ such that σ'_i is the lower limit of a presence interval of e'_i. Clearly, no other relevant journey may arrive

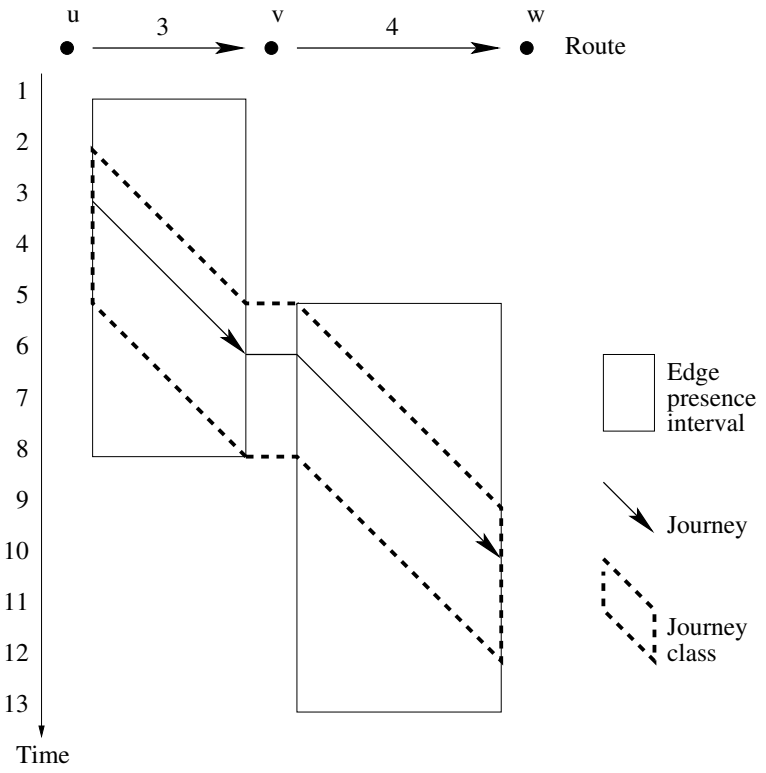

Figure 7: A journey and a class of similar journeys.

at $|(R, R_\sigma + \delta)|_a$. Thus, we can assign an extremity of an edge presence interval to each different maximal relevant journey class so that they are pairwise disjoint. Therefore, the number of maximal relevant classes is bounded by twice the number of edge presence intervals in $E_{\mathcal{G}}$. □

Given an edge (u, v) and a list of journey classes arriving on vertex u, the first algorithm we give below computes a list of journey classes arriving on vertex v, such that each journey in the new classes is formed by an old journey plus the edge (u, v) scheduled as early as possible.

Algorithm 4 (Augmenting journey classes)

Input: *A timed evolving graph $\mathcal{G}$; an edge $(u, v) \in E_{\mathcal{G}}$; and a list L_u of journey classes from s to u.*

We suppose that the list is ordered by increasing departure and arrival dates. Let $first(L_u)$ be the first element of the L_u. We will add and remove elements at the beginning of the list L_u.

Output: *A list L_v of journey classes from s to v.*

We add elements one by one to an empty list. Elements are added to the end of the list L_v.

Variables: *A departure time for (u, v): $\sigma(u, v) \in \mathbb{T}$; a maximum delay for the departure $\delta(u, v) \in \mathbb{T}$ so that a message can enter (u, v) during $[\sigma(u, v), \sigma(u, v) +$*

$\delta(u,v)]$; and a class of journeys $(R, I(R_\sigma, \delta))$. L_u will be emptied step by step, while L_v will be filled.

1. *Initialize $L_v \leftarrow \emptyset$.*
2. *While L_u is not empty, do:*
   (a) *Let $(R, I(R_\sigma, \delta)) \leftarrow first(L_u)$ and*
   Remove $(R, I(R_\sigma, \delta))$ from L_u.
   (b) *Let $\sigma(u,v) \leftarrow f((u,v), |(R, R_\sigma)|_a)$ and*
   If $\sigma = \infty$, then end the algorithm.
   (c) *Let $[t_1, t_2]$ be the presence interval of (u,v)*
   such that $\{t_1 \leq \sigma(u,v)$ and $\sigma(u,v) \leq t_2\}$ and
   Let $\delta(u,v) \leftarrow (t_2 - \zeta(u,v))$.
   (d) *If $(|(R, R_\sigma)|_a + \delta) \leq \sigma(u,v)$ then*
      i. *$R_\sigma \leftarrow R_\sigma + \delta$.*
      ii. *$\delta \leftarrow 0$.*
      iii. *Add $(R, I(R_\sigma, \delta))$ to the end of the list L_v.*
      iv. *Go to 2.*
   (e) *If $|(R, R_\sigma)|_a < \sigma(u,v)$ then*
      i. *Let $\delta_{split} = \sigma(u,v) - |(R, R_\sigma)|_a$.*
      ii. *$R_\sigma \leftarrow R_\sigma + \delta_{split}$.*
      iii. *$\delta \leftarrow \delta - \delta_{split}$.*
   (f) *If $(|(R, R_\sigma)|_a + \delta) > (\sigma(u,v) + \delta(u,v))$ then*
      i. *Let $\delta_{split} = (\sigma(u,v) + \delta(u,v)) - (|(R, R_\sigma)|_a + \delta)$.*
      ii. *Add the class $(R, I(R_\sigma + \delta_{split}, \delta - \delta split))$ to the beginning of L_u.*
      iii. *$\delta \leftarrow \delta_{split}$.*
   (g) *Add $(R, I(R_\sigma, \delta))$ to the end of the list L_v.*

In order to proceed, we will need an algorithm that merges arriving journey class lists arriving on the same vertex v, while removing irrelevant journeys. This is a classical merge of ordered lists with a removal functionality, which we will denote Merge.

In the following, we give the main algorithm for fastest journeys.

Algorithm 5 (Fastest journeys)
Input: *A timed evolving graph $\mathcal{G}$, a vertex $s \in V_\mathcal{G}$.*
Output: *An array $\mathcal{J}_{fastest}[v] \in \{$set of journeys$\}$ which gives for each vertex $v \in V_\mathcal{G}$ a fastest journey from s to v.*
Variables: *Two vertices u and $v \in V_\mathcal{G}$; a hop-count k; and two arrays $L[v], L'[v] \in \{$set of all journey class lists$\}$ which give for all $v \in V_\mathcal{G}$ a list of journey classes from s to v.*

1. *Initialize* $L(s) \leftarrow ((nil, I(nil, \infty)))$ *(A list with one element, the empty journey class with possible delay* ∞*). And for all* $v \neq s$*, let* $L(v) \leftarrow \emptyset$*.*

2. *For* k *from 1 to* N *do:*

   (a) *For each vertex* $v \in V_{\mathcal{G}}$ *do* $L'[v] \leftarrow L[v]$*.*

   (b) *For each edge* $(u, v) \in E_{\mathcal{G}}$ *do*

      i. *Call Algorithm 4 with input* $(\mathcal{G}, (u, v), L[u])$ *and output* L_v*.*

      ii. *Merge* $L'[v]$ *with* L_v*, the output of the call to Algorithm 4, and place the resulting list in* $L'[v]$ *.*

   (c) *For each vertex* $u \in V_{\mathcal{G}}$ *do* $L[u] \leftarrow L'[u]$*.*

3. *For each vertex* v*, let* $fastest(v) \leftarrow (R, R_\sigma)$*, where* $(R, I(R_\sigma, \delta))$ *is the minimum for* $\leq_{rel}$ *of* $L(v)$*.*

Proposition 3 *Algorithm 5 completes in* $O(NM^2\delta_E)$ *steps.*

Proof. According to Lemma 2 the number of maximal relevant journey classes is bounded by $2M\delta_E$. The calls for Algorithm 4 on an edge (u, v) cost the activity of (u, v) plus the size of the lists, that is $O(M\delta_E)$. The calls to the Merge algorithm cost each the size of the lists, that is $O(M\delta_E)$. Thus, for each vertex v, the computation of the lists and the merge procedures cost $O(\Gamma(V)M\delta_E)$, where $\Gamma(V)$ is the number of neighbors of v. Therefore, at each hop count, the cost is $O(M^2\delta_E)$. Since there are N hop counts, the final cost is $O(NM^2\delta_E)$ time steps. □

6. The Case of Fading Messages

With respect to communication networks, one of the hypothesis used in this paper is that during the routing procedure, if a node which have already received the message disappears and then reappears, then the message remains valid in its memory and the node can again participate in the routing (this hypothesis is somewhat analogous to *parking allowed* in [15]).

However, in networks where nodes have small batteries, if a node disappears and then reappears, it may have lost the received messages. In this case, observe that if the activity of a vertex $v \in V_{\mathcal{G}}$ is more than one, it is possible to duplicate the vertex $\delta_V(v) - 1$ times. Each duplicate is assigned one of the time intervals in the old node schedule list. After this operation is done, all new node schedule lists contain one single interval. Edges are duplicated accordingly.

We remark that the total number of intervals in edge or node schedules does not change with this transformation. However, it is useful for each old vertex $v \in V_{\mathcal{G}}$ to keep track of its corresponding vertices in the node schedule list. Given an evolving graph $\mathcal{G}$, we can then build an evolving graph $\mathcal{G}'$ with single presence interval vertices, along with a node schedule P'_V for $V_{\mathcal{G}}$ which states which are the copies of the old nodes. If we call $N' = |V_{\mathcal{G}'}|$ and $M' = |E_{\mathcal{G}'}|$, notice that whereas the total number of intervals remains the same, $N' \leq \delta_V \times N$ and $M' \leq 2M \times \delta_V$. The complexities of Algorithms 1 and 3 for foremost journeys and shortest journeys

increase by a factor proportional to the node activity and become $O(M\delta_V\ (log\delta_E + logN))$ and $O(Md\delta_V log\delta_E)$, respectively, whereas the complexity of Algorithm 5 for fastest journeys, which is based on the number of edge presence intervals, increases to $O(NM^2\ \delta_V^2\delta_E)$.

7. Conclusion and Perspectives

In this paper, we studied route-discovery problems in fixed-schedule wireless dynamic networks modeled by timed evolving graphs. We focused 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 spent on a journey.

Remarkably, the fact that the evolving graphs are timed and, further, that the edge traversal times are continuous, turns the computation of shortest journeys more difficult than in usual graphs, and also makes the computation of fastest journeys quite intricate. Only the computation of foremost journeys can be inspired by usual shortest paths computations, and that is probably the explanation why this problem had received considerable attention in the literature, while the former ones had not been thoroughly investigated so far.

Finally, we note that recently the same modeling of dynamic networks was proposed in order to tackle wavelength on-line assignment for optical networks [12]. Therefore, evolving graphs may trigger several new insights in the study of wireless, optical, mobile, or fixed dynamic networks, opening wide new ways for further research.

Acknowledgments

We are grateful to Balazs Kotnyek for bringing several references to our attention. We also thank Hervé Rivano for continuous help and motivation.

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. S.E. Dreyfus. An appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17:269–271, 1969.
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. L. Fleisher and Martin Skutella. The quickest multicommodity flow problem. In *Proc. of IPCO'02*, 2002.
8. L.R. Ford and D.R. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6:419–433, 1958.
9. L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
10. J. Halpern. Shortest route with time dependent length of edges and limited delay possibilities in nodes. *Zeitschrift für Operations Research*, 21:117–124, 1977.
11. J. Halpern and I. Priess. Shortest path with time constraints on movement and parking. *Networks*, 4:241–253, 1974.
12. P. Haxell, A. Rasala, G. Wilfong, and P. Winkler. Wide-sense nonblocking WDM cross-connects. In R. Möhring and R. Raman, editors, *Proceedings of ESA 2002*, volume 2461 of *LNCS*, pages 538–550, Rome, Italy, September 2002. Springer-Verlag.
13. E. Köhler, K. Langkau, and M. Skutella. Time-expanded graphs for flow-dependent transit times. In *proc. ESA'02*, 2002.
14. 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.
15. A.B. Philpott. Continuous-time flows in networks. *Mathematics of Operations Research*, 4(15):640–661, 1990.
16. 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.
17. I. Stojmenovic, editor. *Handbook of Wireless Networks and Mobile Computing*. John Wiley & Sons, February 2002.
18. L. Viennot. Routage entre robots dont les déplacements sont connus – Un exemple de graphe dynamique. Réunion TAROT, ENST, Paris, November 2001.
19. 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.
20. 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.