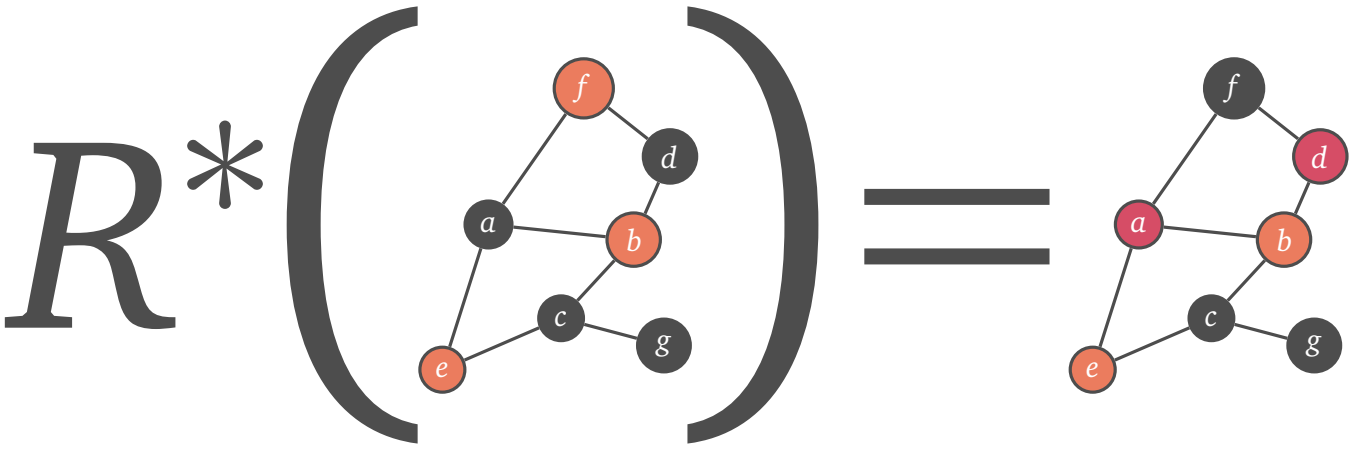
Relabeling Algorithms on Dynamic Graphs

Florent Marchand de Kerchove

Relabeling Algorithms On Dynamic Graphs

Author:
Florent MARCHAND DE KERCHOVE

Supervisor:
Frédéric GUINAND



Research report
March – June 2012
University of Le Havre

Acknowledgments

Invaluable thanks go to my supervisor, Frédéric Guinand, for his guidance and insight stemming from expertise. Your honest advice was always the most helpful; I'll be sure to take heed of it.

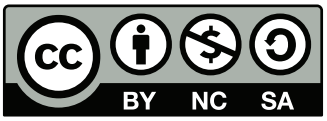
Then I would like to sincerely thank my lab mates. Alexis, for our many philosophical, somewhat metaphysical, and mostly fruitful discussions. Guillaume, for bringing your distinctive opinions to the table. Thibaut, for coping with my frequent, ludicrous teasing. Merwan, for your exquisite tastes in herbal tea, and for sharing a desk with me for so long, without complaining once.

Lastly, my dearest thanks go to Marie, for her understanding and patience during the long hours I spent on this report.

Revision 1.0.2 on 7 August 2012.

Colophon

This report was drafted with Org mode 7.8.11 under GNU Emacs 24.1.1. It is typeset in Bitstream Charter using LaTeX. All diagrams were made using the TikZ package. No mustaches were harmed in the making of this report.



Abstract

Distributed algorithms executed over dynamic graphs are difficult to study due to the limited knowledge of decentralized algorithms, the uncertainty dynamism in graphs entails, and the myriad of competing and immature models. One particular framework combining relabeling systems and evolving graphs struck us as both elegant and sound [CCF12]. We use this framework to ground our contributions to the domain of distributed algorithms, which are threefold: (1) we provide a sufficient condition for the decentralized counting algorithm by Casteigts [Cas07], (2) we define conditions with stronger guarantees, (3) we give a novel counting algorithm which has the property of local detection of termination. In addition, we outline algorithms for the naming problem using the same framework.

Contents

1 Context

1.1 Dynamic graphs

Dynamic graphs arose from a need to include time into graph-based models. Graphs are powerful mathematical structures, suited to numerous theoretical and real-world applications. Indeed, if one wants to study a set of *things* and the *relationships* between these things, a graph is probably the most suited abstraction for that end. As far as we know, the concept of graph was first used by Euler to solve the "Seven Bridges of Könisberg" problem, where one must find a circuit crossing each of the seven bridges once and only once [Eul41]. Although the first recorded use of the term "graph" appeared 150 years later in [Syl78].

However, for all their usefulness in a static context, graphs are not sufficient when it comes to modeling dynamic relationships between things. Consider, and we will refer to this example countless times in this report, the relationships in an ad-hoc network of mobile devices. A mobile device is capable of establishing a communication link with another device in the network when they are within range of each other's sensors. We consider networks of homogeneous devices of identical sensors, thus communication between mobile devices is assumed to be bidirectional [1]. We can use a graph to abstract this *mobile network*: the mobile devices will be our graph's nodes, and the communication links between the devices will be our graph's edges. This abstraction is perfectly valid when we look at only a snapshot of the mobile network, an instantaneous capture of the devices and their communication links. However, these devices are mobile, i.e. they move freely in the environment in which they are situated. When two mobile devices move out of range of each other, they are suddenly unable to communicate, breaking the link between them. If we want to fully study this mobile network using graphs, we need to account for the disappearance (and appearance) of edges. In other words, we now must work with a *dynamic* set of edges.

That is precisely the notion introduced by Harary and Gupta [HG97]. In this paper, the authors define four basic kinds of dynamic graphs, depending on where the dynamic lies:

- If the set of nodes is dynamic, i.e. nodes may be added or removed, we have a *node-dynamic* graph. When a node disappears, its incident edges are removed from the graph as well.
- If the set of edges is dynamic, we have an *edge-dynamic* graph.
- If the weights on nodes are dynamic, i.e. they change with time, we have a *node weighted dynamic graph*.
- If the weights on edges are dynamic, we have an *edge weighted dynamic graph*.

The authors note that all combinations of the above basic types can occur. In our mobile network example, if we assume that mobile devices can appear and disappear, then we need a graph that has a dynamic set of nodes as well as a dynamic set of edges.

Once we have properly defined the dynamic of our network, we still need a framework with which to study this dynamic. Harary and Gupta, still in the same article, suggest two approaches to study dynamic graphs. The first one is to model dynamic graphs as a logic program, arguing that this is a natural model, on the basis that logic programming is a relational programming paradigm, and graphs are relations. In addition, the

[1] In practice, environmental hazards will affect the communicating range of the mobile devices (walls and other obstacles, weather, etc.), such that communication may become unidirectional.

ability of logic programs to handle nondeterminism is ideal to deal with combinatorial searches and algorithms. To our knowledge, this model did not garner a strong following, despite its legitimacy.

The second model, however, had more impact in the field. Straightforwardly, a dynamic graph can be decomposed as a discrete sequence of static graphs. Each static graph is a snapshot of the dynamic graph at a given time. These static graphs can then be studied by all the mathematical tools already at our disposal. This is precisely the view adopted by Ferreira and his "evolving graph" model [Fer04].

Born in the context of mobile ad-hoc networks, the formalism of evolving graphs is a mathematical tool to analyze a sequence of static graphs captured from the same dynamic network. It introduces new concepts required by the incorporation of time, like the one of a path over time, called a journey. Our contribution is deeply rooted in this formalism, and as a matter of fact we will make use of the definitions presented in this seminal article [Fer04].

1.2 Distributed algorithms and local computation

Armed with a sound mathematical basis for modeling mobile networks, we must now turn to characterizing distributed algorithms on these networks.

Distributed algorithm are designed to work on a heterogeneous set of hardware devices, without requiring a central entity to control them. The algorithm is shared by all nodes in a network, but each will execute it independently and concurrently. Typically, there is no synchronization between nodes, and information is only shared via messages. Since nodes can only send messages directly to adjacent nodes, communication with distant nodes is always indirect. Thus, nodes can never be sure of the algorithmic state of a non-adjacent node. This is a first cause of difficulty in designing distributed algorithms, in contrast to algorithms driven by a central entity. A centralized algorithm has information on the whole graph, and can act on any node without delay. A second difficulty arises from the fact that nodes do not have access to information on the network as a whole, such as topological information (network size, diameter, ...). Distributed algorithms are thus constrained by both *local computation* and *local knowledge*.

Fundamental problems of distributed programming include:

Counting Finding out the number of nodes of a network. We will focus mainly on this problem in this report.

Leader election It is often desirable in a network to elect a leading node, in order to reestablish a central authority. The network leader can then command the other nodes to execute some computation, and to return their results back to the leader. In turn, the leader is polled for results by a user, abstracting the distributed aspect of the computation. By using a leader, one can take advantage of the wealth of knowledge concerning algorithms for centralized networks.

Spanning tree Spanning trees are a practical structure for distributed networks. Communication can be achieved without redundancy, and some problems, like leader election, can be reduced to finding a spanning tree (the root of the tree can be chosen as the leader).

Termination detection Discovering when all nodes have completed their execution of an algorithm, without the help of a central authority. Detecting termination is essential in order to exploit the computation results, and to start another algorithm.

While these problems have been studied thoroughly for static networks, the combination of distributed algorithms and dynamic graphs is still a nascent research area. We

focused our efforts on these problems because they are well-defined, abstract, though practically relevant, all the while being still open for improvement when applied to dynamic networks.

1.3 Related work

Distributed algorithms have been extensively studied in the past [CR79; Ang80; GHS83; NS93], and still are an active research area [OW05; KLO10]. This interest resulted in a large number of models to characterize distributed algorithms, and the communication networks they are usually executed on. These models are seldom compatible in their assumptions on *synchronizations protocols* and *mobility models* for example. As a consequence, algorithms that are elaborated for distinct, specific models are not easily comparable to each other. This diversity called for higher-order theoretical frameworks, where the intrinsic properties of distributed algorithms can be analyzed without getting dragged into implementation details.

We focus on the model introduced by Casteigts [Cas07], where graph relabeling systems [LMS99] are coupled with evolving graphs [Fer04]. Though alternatives do exist [Ang+06], we felt they were still too restrictive in their approach, whereas the model we consider is more general. This framework was later extended by Casteigts et al. [CCF09; CCF12]. While these works are fairly recent, there have already been a couple follow-ups [Pig+10; FGA11]; this report can be inscribed along those.

2 Preliminaries

Our contributions build upon the work established by Casteigts et al. [CCF12]. Therefore, the concepts of *local computation*, *graph relabelings* and *evolving graphs* are required to fully understand the next section.

For the sake of clarity and completeness, we reproduce here the definitions by Casteigts et al. [CCF12] needed to express our results. Readers already familiar with these concepts may skip this section and refer to it when the need arises.

2.1 Graph relabelings

Graph relabelings [LMS99] are a formalism where distributed algorithms are represented as a set of local interaction rules. These rules are independent from any communication protocol. Abstracting the effective communication allows us to specify and reason about important properties of distributed algorithms, such as correctness and termination, without limiting these results to a specific implementation.

As the name implies, a graph relabeling system is first and foremost a graph with labels on its vertices and edges. These labels are used by the interaction rules of the *relabeling algorithm*. An interaction rule is defined as a transition from one pattern of vertices and edges and their associated labels (*preconditions*), to another such pattern (*actions*). Since graph relabeling systems were introduced to characterize properties of local computation, interaction rules are local: they involve a limited set of connected vertices and edges. In this paper, we only consider interaction rules between pairs of vertices; see Figure 1 for the commonly-used scopes of computation.

We now give a formal definition of graph relabeling systems in the context of mobile networks. Let $G = (V_G, E_G)$ be a finite undirected loopless graph, with V_G as the set of nodes in our network, and E_G representing the set of communication links between them. Two vertices u and v are *neighbors* if and only if they share a common edge (u, v) in E_G.

Figure 1: Scopes of local computation. Nodes in the scope of preconditions are filled in white, while nodes affected by the actions are filled in black (reproduced from Casteigts et al. [CCF12]).

Let $\lambda : V_G \cup E_G \to L^*$ be a mapping that associates every vertex and edge from G with one or several labels from an alphabet L. The label of a given vertex or edge u, at a given time t is denoted by $\lambda_t(u)$. The pair (G, λ) is the *labeled graph*, written $\boldsymbol{G}$.

A complete algorithm is defined by the triplet $\{L, I, P\}$, where I is the set of initial states, and P is a set of *relabeling rules*. Since we are interested in distributed algorithms, where every node in the network will execute the same identical algorithm, the set of rules P is the same for all vertices.

Algorithm 1 (or A_1 for short) is an example of a complete algorithm expressed in the formalism of graph relabelings. Each vertex can be in one of two states: I and N, standing for *informed* and *non-informed* respectively. Initially, only the emitter vertex has the I state. Then, repeated application of the rule diffuses the information in the network. This is a simple and general information propagation algorithm.

Algorithm 1: Information propagation

Initial states: I for the emitter, N for every other node

The algorithm works in the following way. Since rules are patterns, each node looks for another node in order to match the rule's preconditions (the left-hand side of the rule). When a pair matches, both nodes modify their labels in the way described by the right-hand side of the rule. Figure 2 gives a step-by-step example of executing Algorithm 1 on a static graph.

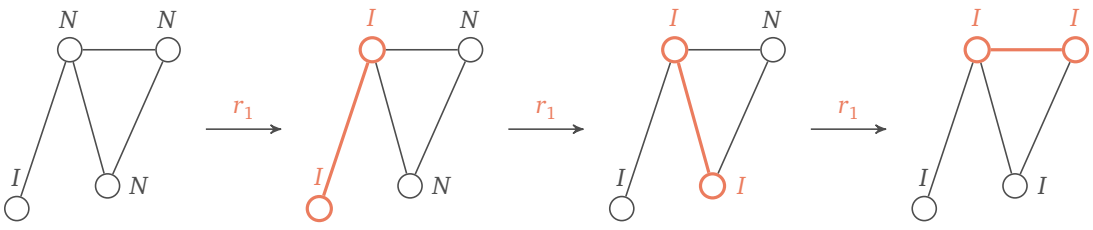


Figure 2: Example execution of the information propagation relabeling algorithm on a static graph. Here, all nodes are informed at the end. Note that this is only one possible execution sequence of the algorithm for this graph; others appear when matching pairs of nodes are selected in a different order.

Note that, from a node's perspective, when two or more neighbor nodes match a rule's preconditions, only one rule can be applied at a time. Depending on the order the

rules are applied to these matching pairs, the algorithm may have different outcomes.

2.2 Evolving graphs

2.2.1 Definition

Evolving graphs were introduced by Ferreira [Fer04] as a model for dynamic networks. In this model, the evolution of the network topology is simply recorded as a sequence of static graphs, where each static graph can be seen as a snapshot of the network at a given time.

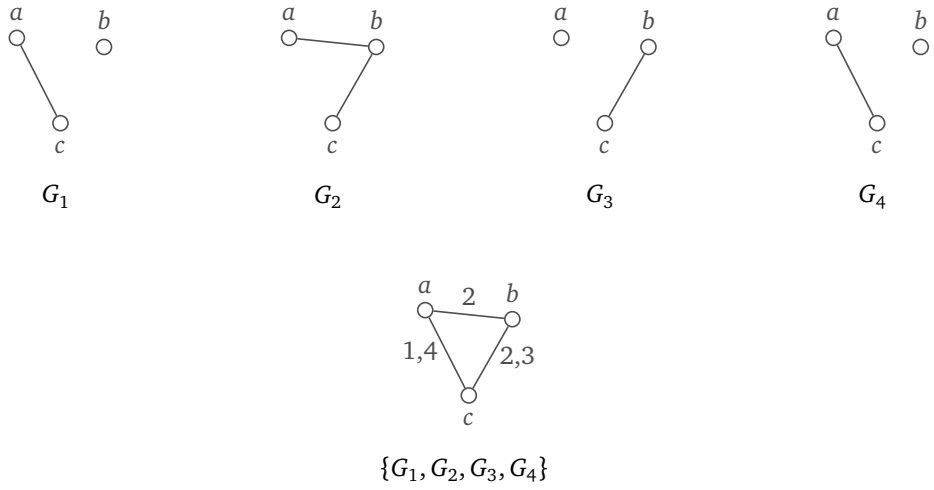


Figure 3: An evolving graph is a sequence of static graphs. On the lower hand-side is a compact representation of an evolving graph. Edges are labeled with the date of their presence.

Formally, an evolving graph is a triplet $(G, S_G, S_\mathbb{T}) = \mathcal{G}$, where:

- $S_\mathbb{T}$ is the sequence of dates used to capture the static graphs. $\mathbb{T}$ can be anything meaningful to the network: discrete ($\mathbb{T} \subset \mathbb{N}$) and continuous ($\mathbb{T} \subset \mathbb{R}$) time systems are common.
- S_G is the sequence of undirected static graphs $G_i = (V_i, E_i)$, where G_i is a snapshot of the network topology during an interval $[t_i, t_{i+1})$.
- G is the union of all G_i in S_G, called the *underlying graph* of $\mathcal{G}$.

We will use the simple notations V and E to denote the sets of vertices and edges of the underlying graph G. A vertex (resp. edge) u is in V (resp. E) if and only if it belongs to at least one static graph in S_G. In addition, we will use the notation $\mathcal{G}_{[t_a,t_b)}$ when taking a *temporal subgraph* $\mathcal{G}' = (G', S'_G, S'_\mathbb{T})$ of $\mathcal{G} = (G, S_G, S_\mathbb{T})$, where $G' \subseteq G$, $S'_G = \{G_i \in S_G : t_i \in [t_a, t_b)\}$, and $S'_\mathbb{T} = \{t_i \in S_\mathbb{T} \cap [t_a, t_b)\}$.

2.2.2 Basic concepts

First, we consider a *presence function* $\rho : E \times \mathbb{T} \to \{0, 1\}$ that indicates whether a given edge is present at a given date. For $e \in E$ and $t \in [t_i, t_{i+1})$, with t_i and t_{i+1} being two consecutive dates in $S_\mathbb{T}$, $\rho(e, t) = 1 \iff e \in E_i$.

A *journey* is a path *over time* between two vertices. Formally, a journey in $\mathcal{G}$ is a sequence of couples $J = \{(e_1, \sigma_1), (e_2, \sigma_2), \ldots, (e_k, \sigma_k)\}$ where $\{e_1, e_2, \ldots, e_k\}$ is a walk in G, $\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ is a non-decreasing sequence of dates from $\mathbb{T}$, and $\rho(e_i, \sigma_i) = 1$ for all $i \leq k$. A *strict journey* only contains couples (e_i, σ_i) taken from distinct graphs of the sequence S_G.

For any u, v in V, if a journey from u to v exists in $\mathcal{G}$, we write $u \rightsquigarrow v$, or $u \overset{\text{st}}{\rightsquigarrow} v$ in the case of a strict journey. We assume that $u \rightsquigarrow u$ for all u in V. Note that a journey is not necessarily symmetrical, even if edges are undirected, because time intervals create a new level of direction. The *horizon* of a node u is the set $\{v \in V : u \rightsquigarrow v\}$, thus u is included in its own horizon.

A network is *connected over time* if and only if $\forall u, v \in V$, $u \rightsquigarrow v \wedge v \rightsquigarrow u$. In other words, there is a round-trip journey between any two nodes. This property is especially useful for algorithms where acknowledgments of sent data are needed.

2.3 Relabelings over evolving graphs

Now we combine graph relabelings and evolving graphs to create an analysis framework for distributed algorithms on dynamic networks.

Let $\mathcal{G} = (G, S_G, S_{\mathbb{T}})$ be an evolving graph. The static graph in S_G that covers the time interval $[t_i, t_{i+1})$ is written G_i; we have $G_i \in S_G$ and $t_i, t_{i+1} \in S_{\mathbb{T}}$. The labeled graph $(G_i, \lambda_{t_i+\epsilon})$, denoted $\boldsymbol{G}_i$, represents the state of the network *just after* the topological event of date t_i, and $\boldsymbol{G}_{i[}$ denotes the labeled graph $(G_{i-1}, \lambda_{t_i-\epsilon})$ representing the network state *just before* that event. Thus, $\boldsymbol{G}_i = \text{Event}_{t_i}(\boldsymbol{G}_{i[})$.

Between two consecutive topological events, any number of relabelings may take place. For a given algorithm A and two consecutive dates $t_i, t_{i+1} \in S_{\mathbb{T}}$, we denote by $R_{A_{[t_i, t_{i+1})}}$ one of the possible relabeling sequence induced by A on the graph G_i during the period $[t_i, t_{i+1})$. We have $\boldsymbol{G}_{i+1[} = R_{A_{[t_i, t_{i+1})}}(\boldsymbol{G}_i)$.

Let us call t_k the date of the last static graph in S_G. A complete execution sequence from t_0 to t_k is then given by an alternated sequence of relabeling steps and topological events:

$$X = R_{A_{[t_{k-1}, t_k)}} \circ \text{Event}_{t_{k-1}} \circ \cdots \circ \text{Event}_{t_i} \circ R_{A_{[t_{i-1}, t_i)}} \circ \cdots \circ \text{Event}_{t_1} \circ R_{A_{[t_0, t_1)}}(\boldsymbol{G}_0)$$

We already mentioned that the order of execution of the rules is not deterministic, since it depends on implementation details concerning the selection of pairs of nodes matching a rule's preconditions. We denote by $\chi_{A/\mathcal{G}}$ the set of all possible execution sequences of an algorithm A over an evolving graph $\mathcal{G}$. An example execution of a relabeling algorithm over an evolving graph is given in Figure 4.

2.4 Distributed algorithm analysis

A distributed algorithm can have multiple outcomes for the same graph. We usually want the algorithm to be complete: it should achieve its goal in all cases. For example, the propagation algorithm (Algorithm 1) should inform all nodes in the network. Unfortunately, all networks will not necessarily allow the algorithm to complete. If the network contains at least two connected components, then only one of them will have all of its nodes informed, since there is only one informed node initially, and information is propagated along edges.

Hence, we find a first way of analyzing distributed algorithm: by characterizing graphs on which they are complete, and graphs on which they can not reach their goal

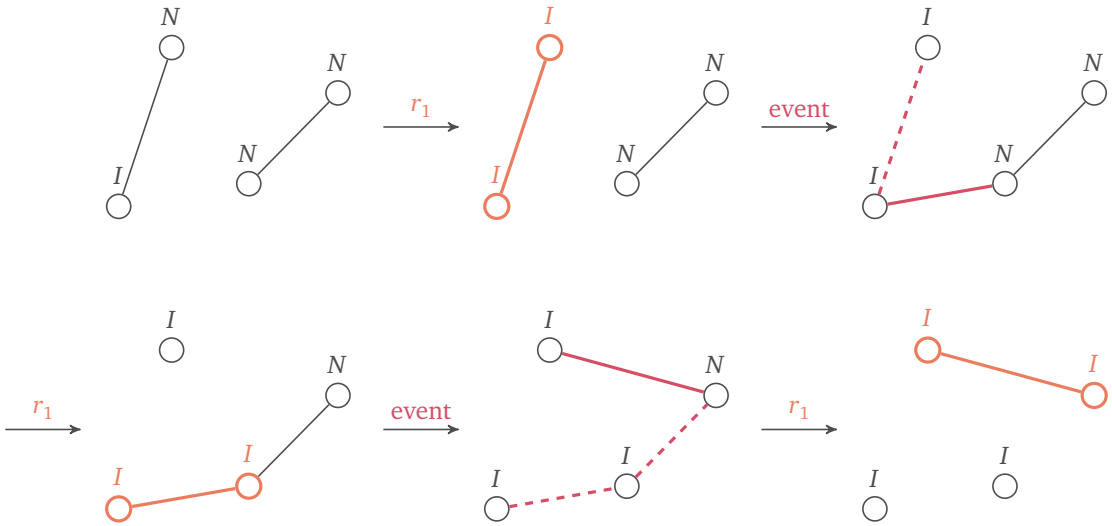


Figure 4: Execution of the information propagation relabeling algorithm on an evolving graph. This time, the execution is a sequence of intertwined relabeling steps and topological events. In future diagrams, we will combine relabeling steps and topological events for the sake of brevity; thus we will only show the sequence of labeled graphs.

at all. A condition on graphs ensuring the completeness of algorithm A will be called a sufficient condition for A. Conversely, a necessary condition for A defines the class of graphs on which A will always fail.

Formal definitions of these concepts follow.

2.4.1 Objectives of an algorithm

Given an algorithm A and a labeled graph $\boldsymbol{G}$, the state one wishes to reach can be given by a logic formula P on the labels of vertices (and edges, if appropriate). In the case of the propagation algorithm, such a terminal state could be that all nodes are informed,

$$P_1(\boldsymbol{G}) = \forall v \in V, \lambda(v) = I$$

The objective O_A is then defined as the fact of verifying the desired property by the end of the execution, that is, on the final labeled graph, after the last relabeling step.

$$O_{A_1} = P_1(\boldsymbol{G}_{k[})$$

2.4.2 Necessary conditions

Given an algorithm A, its objective O_A and an evolving graph property C_N, the property C_N is a *(topology-related) necessary condition* for O_A if and only if:

$$\forall \mathcal{G}, \neg C_N(\mathcal{G}) \implies \neg O_A$$

Proving this result comes to prove that $\forall \mathcal{G}, \neg C_N(\mathcal{G}) \implies \nexists X \in \chi_{A/\mathcal{G}} | P(\boldsymbol{G}_{k[})$. (The desired state is not reachable by the end of the execution (time k), unless the condition is verified).

2.4.3 Sufficient conditions

Symmetrically, an evolving graph property C_S is a *(topology-related) sufficient condition* for A if and only if:

$$\forall \mathcal{G}, C_S(\mathcal{G}) \implies O_A$$

Proving this result comes to prove that $\forall \mathcal{G}, C_S(\mathcal{G}) \implies \forall X \in \chi_{A/\mathcal{G}} | P(\boldsymbol{G}_{k[})$.

Because we have not made any assumptions on the synchronization between nodes, we have no way of ensuring that a rule will effectively be applied. Therefore, we must formulate a progression hypothesis that will allow us to characterize sufficient conditions.

Progression Hypothesis 1. *In every time interval $[t_i, t_{i+1})$, with t_i in $S_{\mathbb{T}}$, each vertex is able to apply at least one relabeling rule with each of its neighbors, provided the rule preconditions are* already *satisfied at time t_i (and still satisfied at the time the rule is applied).*

2.4.4 Analysis of the propagation algorithm

Casteigts et al. show [CCF12] that the information propagation algorithm has the following necessary and sufficient conditions.

Condition 1. $\exists u \in V : \forall v \in V, u \rightsquigarrow v$.
(There is a node that can reach all the others by a journey.)

Condition 2. $\exists u \in V : \forall v \in V, u \overset{st}{\rightsquigarrow} v$.

2.5 Scope of analysis

When dealing with mobile networks, it is often desirable to *maintain* a property or structure over time, instead of merely reaching it. For example, maintaining a spanning tree, or an elected leader, or an accurate estimation of the graph size. Although analyzing such maintenance algorithms with the framework of relabelings over evolving graph is possible [Pig+10], we will not do so in this report.

Another aspect of mobile networks we will not consider explicitly is the appearance and disappearance of nodes during algorithm execution. Actually, this aspect is implicitly handled by the formalism of evolving graphs. Recall that evolving graphs are a *post-mortem* view of the mobile network; in a given evolving graph, all the nodes that can appear or disappear belong to V. As such, even if a node can be seen as "appearing" from another node's viewpoint, for us it was present all along.

3 Contributions

In this section, we present our core contributions to the analysis of distributed algorithms on dynamic graphs.

First, we provide a sufficient condition for the decentralized counting algorithm from Casteigts [Cas07]. A necessary condition for this algorithm was formulated in Casteigts et al. [CCF12], but a sufficient condition was left open. We then introduce the concept of *tight* conditions, to strengthen the guarantees offered by necessary and sufficient conditions. We review the conditions and algorithms introduced so far with respect to this tightness criterion. Then, we combine the counting and information propagation algorithms to obtain a distributed counting algorithm where all nodes are informed of the final count. Finally, we propose another counting algorithm for dynamic networks where nodes have unique identifiers and can locally infer termination.

3.1 Sufficient condition for the decentralized counting algorithm

3.1.1 Decentralized counting algorithm

This decentralized counting algorithm was first proposed in Casteigts' thesis [Cas07], along with other variants.

Algorithm 2: Decentralized counting

Initial states: $(C, 1)$ for every node

This is an example of a truly distributed algorithm: all nodes have the same initial state, and all nodes execute the same algorithm. Each node has two labels: a state indicator (C of F), and a counter. Initially, a node starts in the "counting" state (C), with a counter of value 1. The counter indicates the number of counted nodes so far by its holder, hence they all begin at 1. When two nodes in the counting state meet and apply rule 1, one of them transitions to the "counted" state (F), and the other updates the value of its counter by summing the counters of both nodes. Ultimately, the goal is to have the last node remaining in the counting state to hold the total number of nodes in the network as the value of its counter. Figure 5 gives an example execution sequence.

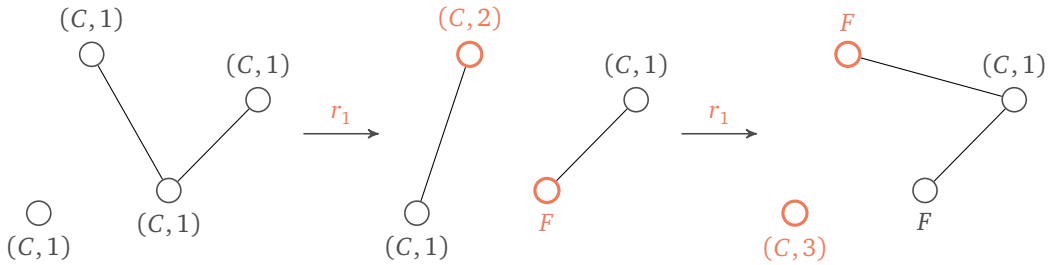


Figure 5: Example execution of the decentralized counting algorithm. If we consider this to be the whole execution sequence, then the counting was not successful: two nodes remain in the counting state. For these two nodes to apply rule 1 and merge, there must be an edge between them; this is the intuition behind a sufficient condition for this algorithm.

Casteigts showed [Cas07] that this algorithm has the following invariant. Let $\mathscr{C}$ (resp. $\mathscr{F}$) be the set of nodes in state "C" (resp. "F"), and V be the set of all nodes. Then $|\mathscr{C}| + |\mathscr{F}| = |V|$ holds at any time of the computation. It follows that if $\mathscr{C} = \{u\}$ (only one counting node remains), then u's counter value is equal to $|V|$.

We can express the objective of this algorithm as the property that only one counting node remains at the end of the computation. Formally:

$$P_2 = \exists u \in V : \forall v \in V \setminus \{u\}, \lambda(u) = C \wedge \lambda(v) = F$$
$$O_{A_2} = P_2(\boldsymbol{G}_{k[})$$

The necessary condition to this algorithm was then shown to be:

Condition 3. $\exists v \in V : \forall u \in V, u \rightsquigarrow v$.
(There is a node reachable by all the others.)

3.1.2 Sufficient condition

Let us now prove that Condition 4 is a sufficient condition for Algorithm 2.

Condition 4. *The underlying graph G is complete. Precisely:* $\forall u, v \in V,\ u \neq v \implies (u, v) \in E$.

First, note the following properties of Algorithm 2.

Lemma 1. $\forall t_i, t_j \in S_{\mathbb{T}},\ j \leq i,\ \forall u \in V,\ \lambda_{t_i}(u) = C \implies \lambda_{t_j}(u) = C$.
("C" labels never change until they disappear.)

Proof. Counters can only disappear from a vertex by application of r_1. Any counter still present at $t_i \in S_{\mathbb{T}} \setminus \{t_0\}$ must have been there from the beginning. □

Lemma 2. $\forall t_i, t_j \in S_{\mathbb{T}},\ j \geq i,\ \forall u \in V,\ \lambda_{t_i}(u) = F \implies \lambda_{t_j}(u) = F$.
("F" labels never change once they appear.)

Proof. No rule can apply to a vertex with a "F" label, thus its label can never change once it becomes "F". □

Lemma 3. *Under Progression Hypothesis 1,* $\forall t_i \in S_{\mathbb{T}} \setminus \{t_k\},\ \forall (u, v) \in E_i,\ \lambda_{t_i}(u) = \lambda_{t_i}(v) = C \implies F \in \lambda_{t_{i+1}[}(\{u, v\})$.
(If two counters share an edge, at least one of them will disappear.)

Proof. During the relabeling step $R_{[t_i, t_{i+1})}$, either r_1 is applied to (u, v), or r_1 can not be applied because preconditions on the labels are not met anymore after an intermediary relabeling. In the first case, one counter disappears from one vertex of $\{u, v\}$; in the second case, one vertex of $\{u, v\}$ already lost its counter. In both cases, $F \in \lambda_{t_{i+1}[}(\{u, v\})$. □

We can now show that Condition 4 (C_2) is sufficient for Algorithm 2 to fulfill its objective.

Proposition 1. *Condition 4 is sufficient on G to guarantee that Algorithm 2 will reach its objective O_{A_2}.*

Proof. (By contradiction). Assume O_{A_2} is not satisfied: for some execution sequence $X \in \chi_{A_2/\mathcal{G}}$, there are at least two final counters in $\boldsymbol{G}_k$. Let u and v be two nodes with such counters: $u, v \in V,\ u \neq v,\ \lambda_{t_k}(u) = \lambda_{t_k}(v) = C$. Since G is complete (by Condition 4), $(u, v) \in E_i$, for some $t_i \in S_{\mathbb{T}}$. By Lemma 1, $\lambda_{t_i}(u) = \lambda_{t_i}(v) = C$. Then, by Lemma 3, either $\lambda_{t_{i+1}[}(u) = F$ or $\lambda_{t_{i+1}[}(v) = F$, and in both cases we have $F \in \lambda_{t_k}(\{u, v\})$ by Lemma 2, leading to a contradiction. Hence, $C_2 \implies O_{A_2}$. □

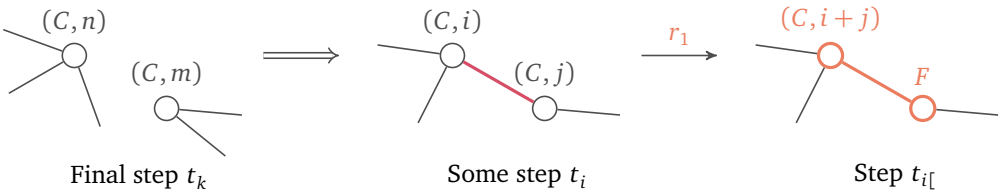


Figure 6: Illustration of the sufficient condition proof. If the final step t_k has two nodes in the counting state, and since the underlying graph is complete, there must have been an edge between these two nodes at some step t_i. If two nodes share an edge at some step t_i, then they inevitably apply rule 1. Thus only one counter remains, even though we assumed there were two.

$F_1 : \exists u \in V : \forall v \in V, u \rightsquigarrow v$
$F_2 : \forall u, v \in V, u \rightsquigarrow v$
$F_3 : \exists u \in V : \forall v \in V, u \overset{\text{st}}{\rightsquigarrow} v$
$F_4 : \forall u, v \in V, u \overset{\text{st}}{\rightsquigarrow} v$
$F_5 : \exists u \in V : \forall v \in V \setminus \{u\}, (u, v) \in E$
$F_6 : \forall u, v \in V, u \neq v, (u, v) \in E$
$F_7 : \exists u \in V : \forall v \in V, v \rightsquigarrow u$

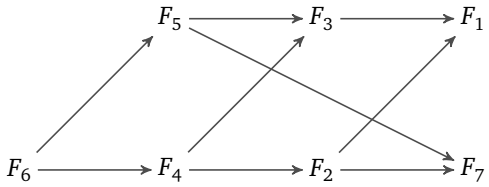


Figure 7: Classification of graphs (reproduced from Casteigts et al. [CCF12]).

3.1.3 Comparison of counting algorithms

In Casteigts et al. [CCF12], a classification of graphs is given. Each condition (necessary or sufficient) defines a class of graphs: all the graphs satisfying the condition are part of this class. Classes of graphs can then be compared to each other, via the partial order induced by inclusion. Figure 7 illustrates the relationships between seven conditions.

Under this classification, Condition 4 gives class F_6, one the of narrowest, whereas the necessary condition to the decentralized counting algorithm is represented by F_7, one of the largest. Theses classes allow us to compare the decentralized counting to other counting algorithms, on the basis of their requirements. An algorithm with a looser sufficient condition (inducing a larger class of graphs) is less specific, i.e. applicable to more graphs.

We can thus compare the decentralized counting algorithm with its centralized variant, Algorithm 3.

Algorithm 3: Centralized counting

Initial states: $(C, 1)$ for the counting node, N for every other node

In this algorithm, counting is handled by a single node in the network. This node starts in the "counting" state (label C) and keeps a counter initialized to 1, whereas all

the others start in the "neutral" state (label N). When the counting node meets a neutral node, the counter is increased by one, and the neutral node becomes "counted" (label F). This is not a distributed algorithm in the purest sense, since the initial states are not uniform: one node begins in the unique state of counting.

This algorithm can be shown to have Condition 5 as a necessary condition.

Condition 5. $\exists u \in V, \forall v \in V \setminus \{u\}, (u,v) \in E$.
(A node shares an edge with all the others.)

Condition 5 gives class F_5, which is larger than F_6. We can thus conclude that the centralized counting algorithm is less specific than the decentralized version.

However, Condition 5 is sufficient for the centralized counting algorithm only if we pick the node u that shares an edge with all the other nodes as the initial node with a counter. If we want to be able to pick any node arbitrarily at random to be the counting node, then all nodes must share an edge with all the others. This condition is none other than Condition 4. In that case, the decentralized counting algorithm is the less specific, because its necessary condition has a larger class of graph; it also holds the advantage of requiring no centralization caused by a node with a special initial state.

3.2 Combination of algorithms

In the decentralized counting algorithm, only one node has knowledge of the total number of nodes. We would like to let all nodes know of the final count. Specifically, we want an algorithm capable of reaching the following desired state.

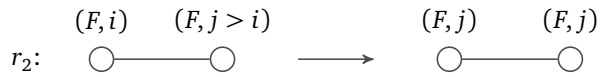
$$P_3 = \exists u \in V : \forall v \in V \setminus \{u\}, \lambda(u) = (C, |V|) \wedge \lambda(v) = (F, |V|)$$

The straightforward way to reach this state is to execute the information propagation algorithm (Algorithm 1) just after the counting algorithm is over. Unfortunately, nodes have no way of knowing that the algorithm they are currently executing is over. This is a known limit of computation in anonymous network [Tel00]. So, unless we give nodes a way of breaking symmetry (by assigning unique identifiers for example), we will not be able to chain both algorithms this way.

One solution is to execute both algorithms in *parallel* instead of chaining them. We get the following relabeling algorithm:

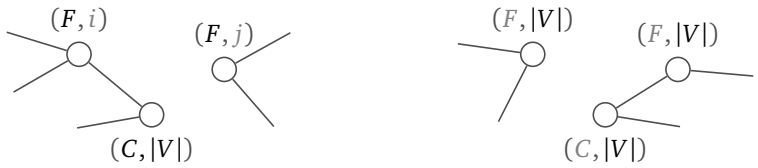
Algorithm 4: Counting combined to propagation

Initial states: $(C, 1)$ for every node



In this new algorithm, a counted node also updates its counter when rule 1 is applied. And because of rule 2, counted nodes diffuse the value of the highest counter. Clearly, once we arrive at a sole counting node, repeated application of r_2 will propagate the final tally to all counted nodes. Until the correct total count is reached however, propagation

still occurs but is only of practical interest; it has the side effect of updating a lower bound on the census. Figure 8 illustrates this scheme.



Counting nodes Propagating the result

Figure 8: Overview of the algorithm combining counting and information propagation. The first phase is concerned with getting the total number of nodes; propagation can be seen as a side effect. In the second phase, only rule 2 can match nodes. It broadcasts the final count to every node.

Once the algorithm is defined, we want a necessary and a sufficient condition. We focus on the sufficient condition first. Intuitively, the sufficient conditions for the two algorithms from which Algorithm 4 is created will take part in elaborating the sufficient condition for the hybrid algorithm. The two rules correspond to the two phases of counting and propagation. The latter phase is only of interest once the former is over, even though the rules are applied concurrently; propagation of the *final count* is what matters for the algorithm. Hence, the counting phase needs to end, and this requires a complete underlying graph (Condition 4). For the second phase, a strict journey from the final counting node to all the other nodes should suffice, but since we can not know in advance which node will fill this role, we must ensure such a journey can begin from any node (Condition 6).

Condition 6. $\forall u, v \in V : u \overset{st}{\rightsquigarrow} v$
(There is a strict journey between any two nodes.)

However, these journeys are unnecessary until the counting phase is over. Since the requirements for both phases are separate, the sufficient condition can be expressed from Condition 4 and Condition 6 using temporal subgraphs (see subsubsection 2.2.1), giving Condition 7.

Condition 7. *$\exists t_p \in S_{\mathbb{T}}$ such that $\mathcal{G}_{[0,t_p)}$ has a complete underlying graph, and $\mathcal{G}_{[t_p,t_k)}$ satisfies Condition 6.*

Let us now turn over to the necessary condition. Symmetrically, the two necessary conditions from the counting algorithm and the propagation algorithm will be used to express the necessary condition for the combined algorithm. Let us call them respectively N_1 (Condition 3) and N_2 (Condition 1). Clearly, if N_1 does not hold, then the counting phase can not succeed, leading to the failure of the combined algorithm. Likewise for N_2, regardless of whether N_1 is satisfied. Consequently, any of the two can serve as a necessary condition for Algorithm 4.

It is interesting to note that, even if N_1 holds for an evolving graph $\mathcal{G}$, we have no guarantee that any relabeling sequence will reach the desired goal state. The next subsection elaborates on this issue.

3.3 Tight necessary and sufficient conditions

A necessary condition might be broader than required. We know that any graph not satisfying this condition will never fulfill the algorithm's objective, but it gives us no additional clue about graphs *satisfying* the condition. Indeed, for any graph satisfying the condition, some may never fulfill the objective, some may fulfill it in some cases, and some may fulfill it in all cases; we can not know without a stronger definition.

Besides, it is not difficult to find trivial necessary conditions that are overly broad. For any relabeling algorithm, any condition which encompass all graphs is a necessary condition. Take the condition that the graph should have nodes: $|V| > 0$. Then, let S be the set of graphs not satisfying this condition; trivially, $S = \emptyset$. It follows that any graph in S will never fulfill the objective, since S is empty, hence $(|V| > 0)$ is a necessary condition. It is trivial in the sense that it does not enlarge the set of graphs for which the algorithm will fail.

When finding necessary conditions, we preferably want them to be as tight as possible. A necessary condition should partition the set of dynamic graphs into two subsets: the set of all graphs for which the algorithm will systematically fail, and the set of all graphs for which the algorithm will succeed at least once. To this end, we define *tight necessary conditions* that generates such partitioning; Figure 9 illustrates their advantage.

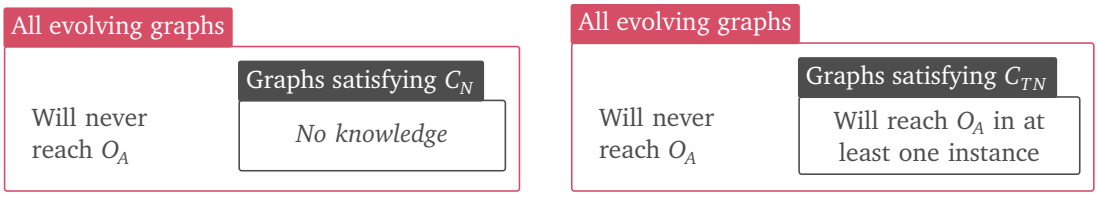


Figure 9: Partitioning the set of all evolving graphs with a necessary condition C_N (left-hand side), and a tight necessary condition C_{TN} (right-hand side).

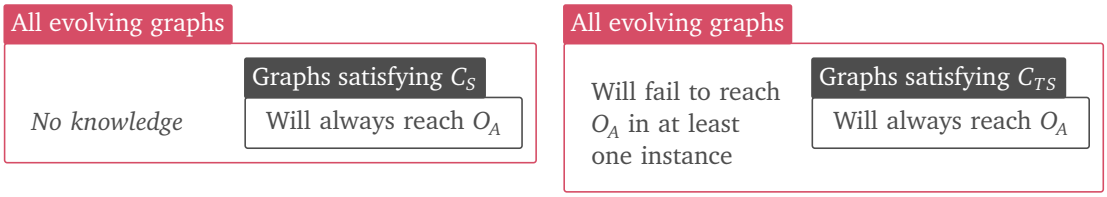


Figure 10: Partitioning the set of all evolving graphs with a sufficient condition C_S (left-hand side), and a tight sufficient condition C_{TS} (right-hand side).

Definition 1. *Let A be an algorithm, O_A be its objective, and C_N be a necessary condition. C_N is a tight necessary condition if and only if*

$$\forall \mathcal{G}, C_N(\mathcal{G}) \Longrightarrow \exists X \in \chi_{A/\mathcal{G}} | P(\boldsymbol{G}_k)$$

In other words, if C_N holds for an evolving graph $\mathcal{G}$, then at least one execution sequence of A over $\mathcal{G}$ will reach the desired state.

Symmetrically, we define tight sufficient conditions.

Definition 2. *Let A be an algorithm, O_A be its objective, and C_S be a sufficient condition. C_S is a tight sufficient condition if and only if*

$$\forall \mathcal{G}, \neg C_S(\mathcal{G}) \Longrightarrow \exists X \in \chi_{A/\mathcal{G}} | \neg P(\boldsymbol{G}_k)$$

In other words, if C_S does not hold for a graph $\mathcal{G}$, then at least one execution sequence of A over $\mathcal{G}$ will fail to reach the desired state.

Note that a condition that is both necessary and sufficient is also a tight necessary and tight sufficient condition (like Condition 5 for the centralized counting algorithm).

3.3.1 Tightening known conditions

We now show that Condition 4 is a tight sufficient condition for Algorithm 2. We have two ways to do so:

- Show that an evolving graph lacking completeness of its underlying graph will fail to fulfill its objective in at least *one* outcome.
- Show that all graphs for which all outcomes succeed in fulfilling the objective have a complete underlying graph.

The following proof uses the former path.

Proof. Let $\mathcal{G} = (G, S_G, S_{\mathbb{T}})$ be an evolving graph. By hypothesis, G is not complete; i.e. there are two distinct nodes $u, v \in V$ such that $(u, v) \notin E$. Since u and v are never in direct contact, they can not apply rule r_1. If u and v are the only two counting nodes left, then it follows that $\neg P(\boldsymbol{G}_k)$, thus the objective O_{A_2} can not be fulfilled and Condition 4 is tight. We are left with exposing a relabeling sequence which leaves u and v with two C labels.

When applying r_1, any of the two nodes can keep the C label, creating two possible outcomes. Every time r_1 is applied to u (resp. v) with another node, we choose the outcome where u (resp. v) keeps the C label. Ultimately, there are at least two nodes in the counting state at time t_k: u and v. There may be more, but in all cases the algorithm fails to fulfill the objective. □

It can also be shown that all necessary and sufficient conditions given for the algorithms in Casteigts et al. [CCF12] are tight. This is expected, because mathematical proofs have a tendency to follow Occam's razor; the smallest set of hypotheses needed by the proof is kept. Nonetheless, tight conditions ensure we narrowed down the right property required by the algorithm to fail (or succeed).

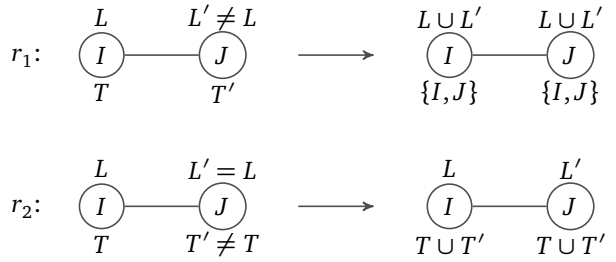
3.4 Counting with identifiers

Let us now present a novel distributed algorithm for counting in networks where nodes have identifiers. We already mentioned that local detection of global termination is not feasible in anonymous networks (as shown by Tel [Tel00]). Local detection of global termination is invaluable in practice: each node can determine on its own when to stop executing the algorithm, and when to exploit its result. That is the reason this counting algorithm will make use of unique identifiers for nodes.

The algorithm can be expressed using two relabeling rules (Algorithm 5). Each node has three labels:

Algorithm 5: Counting with identifiers

Initial states: $L = \{i_u\}$, $T = \{i_u\}$ for each node u



- A unique identifier. In the diagram, this identifier is inscribed in the node's inside. Note that identifiers can be in any domain. When we write I and J, we do not intend that only the nodes with the respective identifiers I and J will match the rule. These are variables intended to capture a node's identifier and use its value later. We write i_u the identifier of node u.
- A set L of identifiers of known nodes. Here again, one should not confuse L for a static label, for it represents a *dynamic set* of labels. During the algorithm execution, if we expand L we may get $L = \{a, b, c, \ldots, z\}$, or $L = \{1, 2, 3, \ldots, n\}$, or anything, depending on the domain from which identifiers are taken. We denote by L_u the set L of a node u.
- A set T of identifiers of nodes who shared the same list L at some point in time. The previous remark holds for T as well; here again we write T_u to denote the set T of a node u.

Initially, each node u has a unique identifier i_u, and $L_u = T_u = \{i_u\}$. The first rule will match two nodes, u and v, of distinct sets L_u and L_v, who proceed to merge them together. The set L is thus used to make a census of known nodes, by sharing knowledge with adjacent nodes. Conversely, the second rule matches two nodes with identical sets L, but with distinct sets T. This time, the sets L are left alone, and the sets T are merged together. Intuitively, the set T is a set of acknowledgments: two nodes agree on their set of known nodes L, and they assume that this set is stable. Hence, by transitivity, for any node v whose identifier is in the set T_u of node u, we have $L_u \subset L_v$. If L_u is updated after that, then the acknowledgment is obsolete, and that is why the set T is reset when rule 1 is applied. Figure 11 gives an example of a couple relabeling steps for this algorithm.

Ultimately, if the set L is complete ($L = \{i_x | \forall x \in V\}$) and all nodes agree ($T = L$), then counting is over. Every node has the result, and also the identifiers of each node in the network. However, our interest here is to give nodes the power to know when the algorithm is over. In other words, we want the property that, for any node u, $L_u = T_u \implies L_u = \{i_x | \forall x \in V\}$. Unfortunately, that is not always the case; see the counter-example in Figure 12.

Nevertheless, there is a way for the implication to be true: when the graph is connected initially, i.e. in G_0 there is only one connected component containing all nodes of V. Under Progression Hypothesis 1, every node u will fill its set L_u with the identifiers of its neighbors at this initial time t_0. This is enough to guarantee that local detection of global termination will raise no false positive, as the following proof demonstrates.

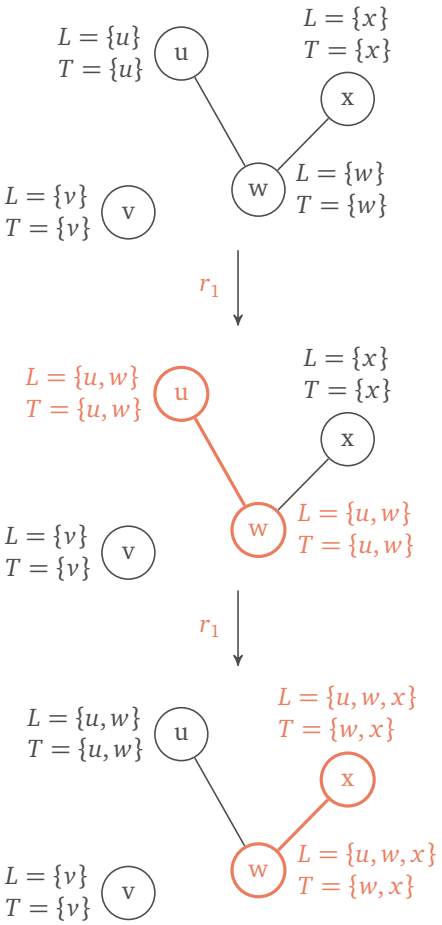


Figure 11: Counting with identifiers. The identifiers are set in each node's circle. Each set L grows with each application of r_1, listing all known nodes. Two nodes acknowledge their identical lists by inscribing each other in the set T.

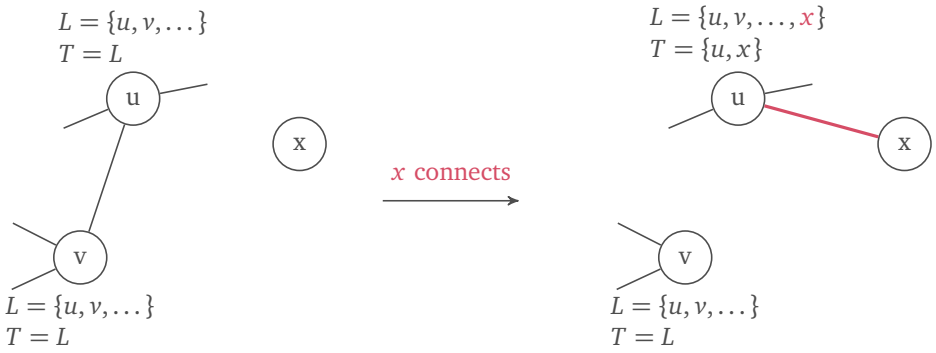


Figure 12: Node u and v have never shared an edge with x until then. They both end up with $L = T$ after dutifully executing the algorithm. When x finally connects, T_u is reset. $L_u = T_u \implies L_u = V$ is thus false in this scenario.

Proof. (By contradiction). Let $\mathcal{G} = (G, S_G, S_{\mathbb{T}})$ be an evolving graph. At t_0, this first date in $S_{\mathbb{T}}$, the labeled graph $\boldsymbol{G}_0$ is connected. Furthermore, assume there is a node $u \in V$ such that $L_u = T_u$ but L_u is not complete. That is, $\exists x \in V : i_x \notin L_u$.

Since $\boldsymbol{G}_0$ was connected, x had at least one neighbor in $\boldsymbol{G}_0$, call that neighbor node y. Now, since $(x, y) \in E_0$, $i_x \in L_y$ because x and y would have applied r_1 under Progression Hypothesis 1 and because identifiers never disappear from a set L. Then, either:

- $i_y \in T_u$, which means that both y and u acknowledged their L sets. Since $i_x \in L_y$ it follows that $i_x \in L_u$, leading to a contradiction;
- $i_y \notin T_u$, and since $L_u = T_u$ we have $i_y \notin L_u$. We found another node, y, which is not in L_u. By the same argument, we can find a neighbor node of y, call it z, and show that i_z must not be in L_u, and so on and so forth until we reach u, because the graph is finite, and show that $i_u \notin L_u$, which is absurd.

Thus, if $L_u = T_u$ then $L_u = \{i_x | \forall x \in V\}$. □

We have not yet characterized necessary and sufficient conditions for this algorithm. The interactions between the two rules make the analysis more complex, but we think these conditions can be found nonetheless.

4 Algorithms for the naming problem

In this section we discuss another distributed computing problem we have worked on but not to the point of giving formal proofs. We share these preliminary findings in the hope they can serve as a basis for further research.

The naming problem is another classical problem of distributed computing. Consider a mobile network of anonymous nodes able to communicate with their neighbors. The naming problem consists in attributing a unique identifier to each node in the network.

4.1 Local resolution

A first, naive way of trying to solve the naming problem is to locally resolve collisions between identifiers. When two nodes with the same identifier meet, one of them must change to another identifier.

Algorithm 6: Local naming

Initial states: 0 for every node

r_1: 

This collision may be resolved in any number of ways. For example, if identifiers are in $\mathbb{N}$ and two nodes have the same identifier value $i = j$, for $i, j \in \mathbb{N}$, then increasing i to $i + 1$ is enough to have $i \neq j$. Generally, if identifiers are taken from a set S, and if any two elements from S are either equal or distinct, then a collision can be resolved by randomly picking an element from S until the identifiers differ.

Clearly, S needs to contain at least two elements otherwise this procedure for resolving collisions would never end. Furthermore, S should be at least as large as $|V|$ if

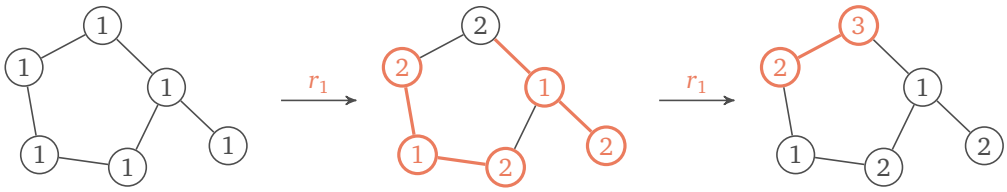


Figure 13: Solving the naming problem with local resolution.

we hope to solve the naming problem. In practice, we can easily take an infinite set of distinct elements, like $\mathbb{N}$, or words constructed from an alphabet.

By resolving all local collisions between pairs of nodes, we ensure each node will not have the same identifier as any of its neighbors. This is in fact a coloring of the nodes in the network. However, we can not guarantee that a node will not share its identifier with another non-adjacent node, unless the underlying graph is complete.

4.2 Centralized naming

A second method of solving the naming problem is to use a central counter. Initially, one node starts in a differentiated state: as a counter which holds the next identifier to hand out to neighbor nodes; we will call it the *source node*. Initially, the source node gives itself the identifier 0, and its counter is set to 1. When the source node encounters another node u, u receives the current value of the counter as its identifier, and the counter is increased. If all nodes in the network meet with the source node, then all will have distinct identifiers. The algorithm thus solves the naming problem.

Algorithm 7: Centralized naming

Initial states: $(0,1)$ for the naming node, N for every other node

r_1: $(0,i)$ N $\longrightarrow$ $(0,i+1)$ i

The algorithm is actually quite similar to the centralized counting algorithm from Casteigts [Cas07]. Although in this case, in addition to counting, the differentiated node identifies neighbors as well. Nevertheless, it follows that the necessary and sufficient conditions for both algorithms are the same, i.e. the source node has degree $n-1$ in the underlying graph (Condition 5).

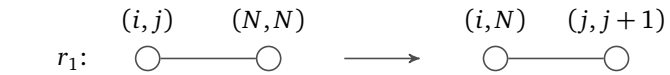
4.3 Random walking counter

Instead of using a centralized counter, we could pass the counter from node to node, like in a relay race. The counter is initialized to zero once again, and each time it is relayed to an unnamed node, that node takes the counter's current value as its identifier, before increasing it.

Depending on the topology of the network, all nodes may receive the counter, and thus have a unique identifier. Like in centralized naming, this counter can obviously be used for counting the nodes in the network, as well as naming them.

Algorithm 8: Token naming

Initial states: $(0,1)$ for the node holding the token, (N,N) otherwise



However, and contrarily to centralized naming, the circulation of the token induces a random walk in the network. Hence, the necessary and sufficient conditions which held for the centralized naming algorithm are inadequate in this case.

Note that the token can only pass from a named node to an unnamed one; it can never end up on a previously visited node. Since, to solve the naming problem, we need to name all nodes in the network, what we need then is a journey which starts from the source node and goes through every other node, until no node is left unnamed. We will call such a journey an *Hamiltonian journey*.

It can be shown that the token will only progress along an Hamiltonian journey. Consequently, we can express the necessary condition for this naming algorithm: the evolving graph needs to contain at least one Hamiltonian journey starting from the source node. In the absence of such a journey, the token can not name all nodes, and thus the algorithm fails.

The sufficient condition is harder to find. It is easy to see that a chain of nodes can always be named, as well as ring networks of any size. In these graphs, the token has always only one possible node with which to interact, until all nodes are named. Adding only one edge may create situations where the token can be trapped in a dead-end, but not necessarily. Hamiltonian journeys may be a piece of the puzzle, but as of now, a sufficient condition for this algorithm still eludes us.

4.4 Tree method

The token method is sequential, since there is only one token traversing the graph at all times. The random walk of a lone token in large graph can be impractical. If we want the algorithm to finish quickly, we should make use of parallel computation. The problem then is to ensure that two nodes never receive the same identifier, even though naming is done concurrently and nodes have no direct way to know which identifiers may have been already assigned.

One solution is to use a naming scheme that will prevent duplicate identifiers from appearing. Consider a tree where the root has ID 0. Now, any children of the root can be uniquely identified by its child index, starting at 0. But if we use this index as an identifier number, the root will share its identifier with its first child. However, any node in the tree, save the root, can be uniquely described by the ID of its parent and its child index. Thus, if the root has ID 0, its first child has ID (0,0), its second child (0,1), and its second child's first child has ID (0,1,0). Identifiers are defined recursively, using the identifier of the parent node.

First, consider a static network. The algorithm can be described as follows:

- Assign identifier 0 to a source node;
- A node which already has an ID send its own ID suffixed with i (ID:i) to its i-th neighbor, for every neighbor that is not its parent (its parent is the node that gave

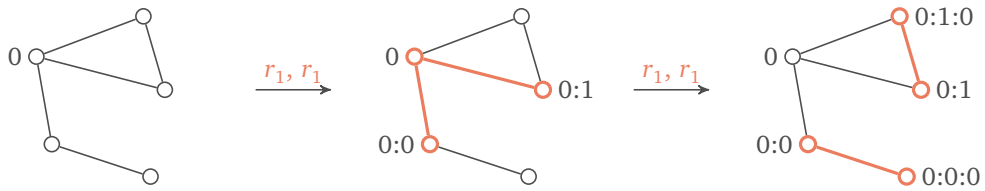


Figure 14: Using a tree to solve the naming problem.

it its identifier). Alternatively, the node can broadcast to all its neighbors, and the parent will just ignore the new ID as it already has one.

For static graphs, this process is bounded in the number of vertices, and we have global termination, but at the cost of having to choose a source node. We can also notify the source node when all nodes have a name by using the Dijkstra-Scholten algorithm.

The algorithm needs some modifications to accommodate an evolving graph:

- Each node keeps an internal counter C, initially zero.
- When node a with an ID meets unnamed node b, b gets assigned an ID constructed from a's ID suffixed with the value of a's counter C, and this counter is increased by one.

Algorithm 9: Tree naming

Initial states: $(0,0)$ for the root node, $(N,0)$ otherwise

This tree method is more similar to flooding algorithms: all nodes with an ID will assign IDs to encountered nodes; the naming of nodes is made parallel. The expected time to terminate the algorithm should be lower than for the single token method.

This method has one downside though: identifiers grow in length with each level in the tree. On the one hand, identifiers can be used to trace a path back to the root, but this path is only valid at the time the identifier is constructed; in a dynamic network, this information may quickly become obsolete. On the other hand, the memory required to hold an identifier grows with the tree.

We have not yet characterized necessary or sufficient conditions for this algorithm.

5 Perspectives

We did not provide a necessary or a sufficient condition for the counting algorithm with identifiers. Instead, we focused on finding a sufficient condition for ensuring local detection of termination. Similarly, the four methods we proposed to solve the naming problem would also benefit from further analysis, e.g. necessary and sufficient conditions, and formal proofs for these conditions when we provided them.

5.1 Analysis of maintenance algorithms

We mentioned in subsection 2.5 that there are two kinds of objectives for distributed algorithms: reaching a desired state, or maintaining it. This is an important distinction to make. Reaching a state where the graph contains a spanning tree is certainly not as useful in practice as maintaining the spanning tree over time.

While there has been preliminary studies using simulations for maintenance algorithms expressed as relabeling systems [Pig+10], they restricted their analysis to static graphs. Simulations over dynamic graphs are often tied to a particular mobility model (or several), each model bringing its bias to the results. The framework provided by relabeling algorithms over evolving graphs is free of these biases, by its higher level of concern. It would be interesting to see if this framework would still be as potent when analyzing maintenance algorithms.

5.2 Algorithmic complexity

In the literature, distributed algorithms are most often characterized by their algorithmic complexity. Whole articles are devoted to improving the worst-case complexity of one algorithm and bring it closer to the theoretical lower bound [Awe87].

While this is an area we barely touched in this report, the framework we used can help with the complexity analysis of relabeling algorithms. We could find, for example, how many relabeling steps it would take for the information propagation algorithm to inform all n nodes for an evolving graph satisfying the sufficient condition. Or we could find additional sufficient conditions under which the algorithm terminates in fewer relabeling steps. More guarantees can be given, while still basing the analysis on the same framework.

5.3 Mechanization

Finally, Casteigts et al. [CCF12] already alluded to the mechanization of their work. Precisely, they envision a software platform which would be used to define distributed relabeling algorithms and run a library of tests on them. This tool would help its user find necessary and sufficient conditions for a given algorithm, by testing known conditions and by giving him clues via exhaustive combinatorial search. Towards this end, this software could make an extensive use of formal proof systems, such as Coq (a goal shared with Castéran et al. [CFM09]). Another possibility is to follow-up on the idea from Harary and Gupta [HG97] of using logic programming to model dynamic graphs and relabeling algorithms. The backtracking backbone of logic languages such as Prolog is particularly suited to handling combinatorial searches.

Even a modest simulation tool for looking at execution traces of relabeling algorithms over dynamic graphs could be useful for the analyst. Using a variety of mobility models mimicking the real world, simulations would validate or invalidate distributed algorithms for practical applications. This prospect has already been tackled [Pig08], but much remain to be uncovered still.

Conclusion

Throughout this report, we built upon the foundations provided by Casteigts et al. [CCF12] in the domain of distributed algorithms over dynamic graphs.

First, we showed that a complete underlying graph was sufficient for the decentralized counting algorithm to succeed. Then we created a hybrid counting algorithm which

propagate its result to all nodes in the network, by combining two preexisting algorithms. We also presented a novel algorithm for counting in a network of named nodes; it gives nodes the power to detect the end of the execution by merely looking at their neighborhood.

To ensure the generality of necessary and sufficient conditions, we defined a tightness criterion, and demonstrated the tightness of the sufficient condition we gave for the decentralized counting algorithm.

Finally, while we had focused on counting algorithms so far, we still enunciated four methods of solving the naming problem. However, we presented these methods less formally, and did not provide a necessary or sufficient condition for all of them.

References

[Ang+06] D. Angluin et al. "Computation in Networks of Passively Mobile Finite-State Sensors". In: *Distributed Computing* 18 (4 2006), pp. 235–253. DOI: 10.1007/s00446-005-0138-3.

[Ang80] D. Angluin. "Local and global properties in networks of processors (Extended Abstract)". In: *Proceedings of the 12th annual ACM symposium on Theory of Computing*. STOC '80. ACM, 1980, pp. 82–93. DOI: 10.1145/800141.804655.

[Awe87] B. Awerbuch. "Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election, and related problems". In: *Proceedings of the 19th annual ACM Symposium on Theory of Computing*. STOC '87. ACM, 1987, pp. 230–240. DOI: 10.1145/28395.28421.

[Cas07] A. Casteigts. "Contribution à l'algorithmique distribuée dans les réseaux mobiles ad hoc - Calculs locaux et réétiquetages de graphes dynamiques". PhD thesis. University of Bordeaux, 2007.

[CCF09] A. Casteigts, S. Chaumette, and A. Ferreira. "Characterizing Topological Assumptions of Distributed Algorithms in Dynamic Networks". In: *Proc. of 16th Intl. Conference on Structural Information and Communication Complexity*. SIROCCO'09 (Piran, Slovenia). Vol. 5869. Lecture Notes in Computer Science. Springer-Verlag, May 2009, pp. 126–140.

[CCF12] A. Casteigts, S. Chaumette, and A. Ferreira. "Distributed Computing in Dynamic Networks: Towards a Framework for Automated Analysis of Algorithms". In: *CoRR* (2012). URL: http://arxiv.org/abs/1102.5529.

[CFM09] P. Castéran, V. Filou, and M. Mosbah. "Certifying distributed algorithms by embedding local computation systems in the Coq proof assistant". In: *Proceedings of Symbolic Computation in Software Science*. SCSS 2009. Sept. 2009. URL: http://hal.archives-ouvertes.fr/hal-00407990.

[CR79] E. Chang and R. Roberts. "An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes". In: *Communications of the ACM* 22 (5 May 1979), pp. 281–283. DOI: 10.1145/359104.359108.

[Eul41] L. Euler. "Solutio problematis ad geometriam situs pertinentis". In: *Commentarii academiae scientarum Petropolitinae* 8 (1741), pp. 128–140. URL: http://www.math.dartmouth.edu/~euler/pages/E053.html.

[Fer04] A. Ferreira. "Building a Reference Combinatorial Model for MANETs". In: *Network, IEEE* 18.5 (2004), pp. 24–29.

[FGA11] P. Floriano, A. Goldman, and L. Arantes. “Formalization of the necessary and sufficient connectivity conditions to the distributed mutual exclusion problem in dynamic networks”. In: *10th IEEE International Symposium on Network Computing and Applications*. 2011, pp. 203–210. DOI: 10.1109/NCA.2011.35.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. “A Distributed Algorithm for Minimum-Weight Spanning Trees”. In: *ACM Transactions on Programming Languages and Systems* 5 (1 Jan. 1983), pp. 66–77. DOI: 10.1145/357195.357200.

[HG97] F. Harary and G. Gupta. “Dynamic Graph Models”. In: *Mathematical and Computer Modelling* 25.7 (1997), pp. 79–87.

[KLO10] F. Kuhn, N. Lynch, and R. Oshman. “Distributed Computation in Dynamic Networks”. In: *Proceedings of the 42nd ACM Symposium on Theory of computing*. STOC ’10. ACM, 2010, pp. 513–522. DOI: 10.1145/1806689.1806760.

[LMS99] I. Litovsky, Y. Métivier, and É. Sopena. “Graph relabelling systems and distributed algorithms”. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. 3. 1999. Chap. 1, pp. 1–56.

[NS93] M. Naor and L. Stockmeyer. “What can be computed locally?” In: *Proceedings of the 25th annual ACM Symposium on Theory of Computing*. STOC ’93. ACM, 1993, pp. 184–193. DOI: 10.1145/167088.167149.

[OW05] R. O’Dell and R. Wattenhofer. “Information Dissemination in Highly Dynamic Graphs”. In: *Proceedings of the 2005 joint workshop on Foundations of Mobile Computing*. DIALM-POMC ’05. ACM, 2005, pp. 104–110. DOI: 10.1145/1080810.1080828.

[Pig+10] Y. Pigné et al. “Construction et maintien d’une forêt couvrante dans un réseau dynamique”. In: *12e Rencontres francophones sur les aspects algorithmiques de télécommunications*. ALGOTEL’10 (Belle Dune, France). June 2010.

[Pig08] Y. Pigné. “Modélisation et traitement décentralisé des graphes dynamiques - Application aux réseaux mobiles ad hoc”. French. PhD thesis. Université du Havre, Dec. 2008.

[Syl78] J. Sylvester. “Chemistry and algebra”. In: *Nature* 17.432 (1878), pp. 284–284.

[Tel00] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000, pp. 268–334.