A Self-Stabilizing K-clustering algorithm using an arbitrary metric

E. Caron, A.K. Datta, B. Depardon, L.L. Larmore

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

A Self-Stabilizing K-clustering algorithm using an arbitrary metric

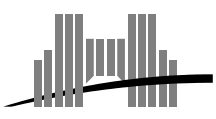
E. Caron[1,2], A. K. Datta[3], B. Depardon[1,2]
and L. L. Larmore[3]

[1] University of Lyon. LIP Laboratory. UMR CNRS - ENS Lyon[2] - INRIA - UCB Lyon 5668
[3] University of Nevada, Las Vegas, USA

September 2008

Research Report N° RR2008-31

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr



A Self-Stabilizing K-clustering algorithm using an arbitrary metric

E. Caron[1,2], A. K. Datta[3], B. Depardon[1,2] and L. L. Larmore[3]

[1] University of Lyon. LIP Laboratory. UMR CNRS - ENS Lyon[2] - INRIA - UCB Lyon 5668
[3] University of Nevada, Las Vegas, USA

September 2008

Abstract

A self-stabilizing asynchronous distributed algorithm is given for constructing a k-dominating set, and hence a k-clustering, of a connected network of processes with unique IDs and weighted edges. The algorithm is comparison-based, takes $O(nk)$ time, and uses $O(\log n + \log k)$ space per process, where n is the size of the network.
It is known that finding a minimal k-dominating set is $\mathcal{NP}$-hard [1]. This is the first distributed solution to the k-clustering problem on weighted graphs.

Keywords: k-clustering, self-stabilization, weighted graph.

Résumé

Nous présentons un algorithme asynchrone, distribué et auto-stabilisant pour construire un ensemble k-dominant, donnant lieu à un k-regroupement de nœuds ayant des identifiants uniques sur un graphe pondéré. L'algorithme se base sur les comparaisons des identifiants, il s'exécute en $O(nk)$, et requiert $O(\log n + \log k)$ d'espace mémoire, où n est la taille du réseau.
Trouver un ensemble k-dominant minimal est un problème $\mathcal{NP}$-difficile [1]. Nous présentons la première solution au problème du k-regroupement sur des graphes pondérés.

Mots-clés: k-regroupement, auto-stabilisation, graphe pondéré.

1 Introduction

Scalability is a critical issue in the design of large networks, such as mobile ad hoc Networks (MANET) and grid computing networks. In a large network, control overhead, such as routing packets, may be inefficient if the distance between the nodes is not taken into account. One approach to improve this situation is clustering the network. A cluster is a subset of the nodes of the underlying network that satisfies a certain property which permits for example, creation of sets of homogeneous resources, or sets nodes no farther apart than a certain distance. A cluster structure facilitates the spatial *reuse of resources* to increase system capacity. Clustering also helps routing; it can be used to design a low-hop backbone network in MANET, or can improve the efficiency of parallel software if it runs on a cluster of well connected resources. It is also of importance when deploying grid middleware. Another advantage of clustering is that many changes in the network can be made locally, *i.e.*, restricted to particular clusters.

The “hop" distance, *i.e.*, the number of links in the path between to processes, is used as a metric in some applications, but it is not relevant in many platforms, such as in a grid. Using an arbitrary metric is a reasonable option in such heterogeneous distributed systems. Distributed grid middleware, like DIET [2], GridSolve [11] also needs this more accurate measure of distance to do accurate job scheduling.

Self-stabilization [5] is a desirable property of fault-tolerant systems. A self-stabilizing system, regardless of the initial states of the processes and initial messages in the links, is guaranteed to converge to the intended behavior in finite time.

1.1 The k-Clustering Problem

We now formally define the problem solved in this paper. Let $G = (V, E)$ a connected graph (network) consisting of n nodes (processes), with positively weighted edges. For any $x, y \in V$, let $w(x, y)$ be the *distance* from x to y, defined to be the least weight of any path from x to y. We will assume that the edge weights are integers.

Given a non-negative integer k, we define a *k-cluster* of G to be a non-empty connected subgraph of G of radius at most k. If C is a k-cluster of G, we say that $x \in C$ is a *clusterhead* of C if, for any $y \in C$, there is a path of length at most k in C from x to y.

We define a *k-clustering* of G to be a partitioning of V into k-clusters. The *k-clustering problem* is then the problem of finding a k-clustering of a given graph[1]. In this paper, we require that a k-clustering specify one node, which we call the *clusterhead* within each cluster, which is within k of all nodes of the cluster, and a *shortest path tree* rooted at the clusterhead which spans all the nodes of the cluster.

A set of nodes $D \subseteq V$ is a *k-dominating set* of G if, for every $x \in V$, there exists $y \in D$ such that $w(x, y) \leq k$. k-dominating set determines a k-clustering in a simply way. For each $x \in V$, let *Clusterhead*$(x) \in D$ be the member of D that is closest to x. Ties can be broken by any method, such as by using IDs. For each $y \in D$, $C_y = \{x : \textit{Clusterhead}(x) = y\}$ is a k-cluster, and $\{C_y\}\_y \in D$ is a k-clustering of G.

We say that a k-dominating set D is *optimal* if no k-dominating set of G has fewer elements than D. The problem of finding an optimal k-dominating set is known to be $\mathcal{NP}$-hard [1].

[1] There are several alternative definitions of k-clustering, or the k-clustering problem, in the literature.

1.2 Related Work

To the best of our knowledge, there exist only three asynchronous distributed solutions to the k-clustering problem in mobile *ad hoc* networks (MANETs), in the comparison based model, *i.e.*, where the only operation allowed on IDs is comparison. Amis *et al.* [1] give the first distributed solution to this problem. The time and space complexities of their solution are $O(k)$ and $O(k \log n)$, respectively. Spohn and Garcia-Luna-Aceves [10] give a distributed solution to a more generalized version of the k-clustering problem. In this version, a parameter m is given, and each process must be a member of m different k-clusters. The k-clustering problem discussed in this paper is then the case $m = 1$. The time and space complexities of the distributed algorithm in [10] are not given. Fernandess and Malkhi [8] presented an algorithm for the k-clustering problem that uses $O(\log n)$ memory per process, takes $O(n)$ steps, provided a BFS tree for the network is already given.
The first self-stabilizing solution to the k-clustering problem was given in [4]; this solution takes $O(k)$ time and $O(k \log n)$ space.

1.3 Contributions and Outline

Our solution, Algorithm Weighted-Clustering, given in Section 3, is partially inspired by that of Amis *et al.* [1], who use simply the hop distance instead of arbitrary edge weights. Weighted-Clustering uses $O(\log n + \log k)$ bits per process. It finds a k-dominating set in a network of processes, assuming that each process has a unique ID and that each edge has a positive weight. It is also self-stabilizing and converges in $O(nk)$ rounds. When Algorithm Weighted-Clustering stabilizes, the network is divided into a set of k-clusters, and inside each cluster, the processes form a shortest path tree rooted at the clusterhead.
In Section 2, we describe the model of computation used in the paper, and give some additional needed definitions. In Section 3, we define the algorithm Weighted-Clustering, and give its time and space complexity. We also show an example execution of Weighted-Clustering in Section 4. Due to space constraints, we do not give the full proof of the algorithm, but only present the sketch of proof in Section 5. Finally, Section 6 concludes the paper.

2 Model and Self-Stabilization

We are given a connected undirected network, $G = (V, E)$ of $|V| = n$ processes with no self-loops, where $n \geq 2$, and a distributed algorithm $\mathcal{A}$ on that network. Each process P has a unique ID, $P.id$, which we assume can be written with $O(\log n)$ bits. We use the *shared memory model* of computation introduced in [5]. In this model, process P maintains registers. P can read its own registers and those of its neighbors, but can write only to its own registers.
The *state* of a process is defined by the values of its registers. A *configuration* of the network is a function from processes to states; if γ is the current configuration, then $\gamma(P)$ is the current state of each process P. An *execution* of $\mathcal{A}$ is a sequence of states $e = \gamma_0 \mapsto \gamma_1 \mapsto \ldots \mapsto \gamma_i \ldots$, where $\gamma_i \mapsto \gamma_{i+1}$ means that it is possible for the network to change from configuration γ_i to configuration γ_{i+1} in one step. We say that an execution is *maximal* if it is infinite, or if it ends at a *sink*, *i.e.*, a configuration from which no execution is possible.
The *program* of each process consists of a set of registers and a finite set of actions of the following form: $< label >:: < guard > \longrightarrow < statement >$. The *guard* of an action in the program of a process P is a Boolean expression involving the variables of P and its neighbors. The *statement* of an action of P updates one or more variables of P. An action can be executed only if it is *enabled*, *i.e.*,

its guard evaluates to true. A process is said to be *enabled* if at least one of its actions is enabled. A step $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more *enabled* processes executing an *action*. We use the *composite atomicity* [6] model of computation; the evaluations of all guards and executions of all statements of those actions are presumed to take place in one atomic step.
We assume that each transition from a configuration to another is driven by a *scheduler*, also called a *daemon*. If one or more processes are enabled, the daemon selects at least one of these enabled processes to execute an action. We assume that the daemon is also *unfair*, meaning that, even if a process P is continuously enabled, P might never be selected by the daemon unless P is the only enabled process.
We say that a process P is *neutralized* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if P is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations. The neutralization of a process represents the following situation: at least one neighbor of P changes its state between γ_i and γ_{i+1}, and this change effectively makes the guard of all actions of P false.
We use the notion of *round* [7], which captures the speed of the slowest process in an execution. We say that a finite execution $\varrho = \gamma_i \mapsto \gamma_{i+1} \mapsto \ldots \mapsto \gamma_j$ is a *round* if the following two conditions hold:

1. Every process P that is enabled at γ_i either executes or becomes neutralized during some step of ϱ.
2. The execution $\gamma_i \mapsto \ldots \mapsto \gamma_{j-1}$ does not satisfy condition 1.

We define the *round complexity* of an execution to be the number of disjoint rounds in the execution, possibly plus 1 if there are some steps left over.
The concept of *self-stabilization* was introduced by Dijkstra [5]. Informally, we say that $\mathcal{A}$ is *self-stabilizing* if, starting from a completely arbitrary configuration, the network will eventually reach a legitimate configuration.
More formally, we assume that we are given a *legitimacy predicate* $\mathcal{L}_{\mathcal{A}}$ on configurations. Let $\mathbb{L}_{\mathcal{A}}$ be the set of all *legitimate* configurations, *i.e.,* configurations which satisfy $\mathcal{L}_{\mathcal{A}}$. Then we define $\mathcal{A}$ to be *self-stabilizing* if the following two conditions hold:

1. (Convergence) Every maximal execution contains some member of $\mathbb{L}_{\mathcal{A}}$.
2. (Closure) If an execution e begins at a member of $\mathbb{L}_{\mathcal{A}}$, then all configurations of e are members of $\mathbb{L}_{\mathcal{A}}$.

We say that $\mathcal{A}$ is *silent* if every execution is finite. In other words, starting from an arbitrary configuration, the network will eventually reach a *sink, i.e.,* a configuration where no process is enabled.

3 **The Algorithm** Weighted-Clustering

In this section, we present Weighted-Clustering, a self-stabilizing algorithm that computes a k-clustering of a weighted network of size n, stabilizes within $O(nk)$ rounds, and uses $O(\log n + \log k)$ space per process.

Overview of Weighted-Clustering**.** The basic idea is that a process P is chosen to be a *clusterhead* if and only if, for some process Q, P has the smallest ID of any process within a distance k of Q. The set of clusterheads so chosen is a k-dominating set, and a clustering of the network is then obtained by every process joining a shortest path tree rooted at the nearest clusterhead; the processes of each tree

form one cluster. Every process is within a distance k of some clusterhead, and thus, our clustering is a k-clustering.
Throughout, we write $\mathcal{N}_P$ for the set of all neighbors of P, and $\mathcal{U}_P = \mathcal{N}_P \cup \{P\}$.

For each process P, define the following values:

$$
\begin{aligned}
\mathit{MinHop}(P) &= \min\{w(P,Q) : Q \in \mathcal{N}_P\} \\
\mathit{MinId}(P,d) &= \min\{Q.\mathit{id} : w(P,Q) \le d\} \\
\mathit{MaxMinId}(P,d) &= \max\{\mathit{MinId}(Q,k) : w(P,Q) \le d\} \\
\mathit{Clusterhead\_Set} &= \{P : \mathit{MaxMinId}(P,k) = P.\mathit{id}\} \\
\mathit{Dist}(P) &= \min\{w(P,Q) : Q \in \mathit{Clusterhead\_Set}\} \\
\mathit{Parent}(P) &= \begin{cases} P.\mathit{id} & \text{if } P \in \mathit{Clusterhead\_Set} \\ \min\{Q.\mathit{id} : (Q \in \mathcal{N}_P) \wedge & \\ \quad (\mathit{Dist}(Q) + w(P,Q) = \mathit{Dist}(P))\} & \text{otherwise} \end{cases} \\
\mathit{Clusterhead}(P) &= \begin{cases} P.\mathit{id} & \text{if } P \in \mathit{Clusterhead\_Set} \\ \mathit{Clusterhead}(\mathit{Parent}(P)) & \text{otherwise} \end{cases}
\end{aligned}
$$

The *output* of Weighted-Clustering consists of shared variables $P.\mathit{parent}$ and $P.\mathit{clusterhead}$ for each process P. The output is *correct* if $P.\mathit{parent} = \mathit{Parent}(P)$ and $P.\mathit{clusterhead} = \mathit{Clusterhead}(P)$ for each P. Weighted-Clustering is self-stabilizing. Although it can compute incorrect output, eventually the output shared variables will stabilize to their correct values.
Weighted-Clustering requires, as a module, a silent self-stabilizing algorithm for finding a breadth-first-search (BFS) spanning tree. We use the algorithm SSLE defined in [3] for this purpose. The BFS tree created by SSLE is used to implement an efficient broadcast and convergecast mechanism, which we call *color waves*, used in the other modules of Weighted-Clustering.

Structure of Weighted-Clustering **: combining algorithms.** Weighted-Clustering consists of the following four phases.

1. Self-Stabilizing Leader Election (SSLE). Given a connected network, SSLE yields a BFS spanning tree for the network, rooted at the process of lowest ID, which we call *Root*_BFS. This algorithm is self-stabilizing and silent. We do not give any details of SSLE here, but instead refer the reader to [3]. The BFS tree is defined by pointers $P.\mathit{parent}$_BFS for all P. This tree is used to synchronize the second and third phases.

2. A non-silent self-stabilizing algorithm Interval. Given a positively weighted connected network with a rooted spanning tree, a number $k > 0$, and a function f on processes, Interval computes $\min\{f(Q) : w(P,Q) \le k\}$ for each process P in the network, where $w(P,Q)$ is the minimum weight of any path through the network from P to Q. We do not give an abstract definition of the algorithm Interval, but we use it in the second and third phases of Weighted-Clustering as follows:

   - Minid, which computes, using Interval, for each process P, $\mathit{MinId}(P,k)$, the smallest ID of any process which is within distance k of P. The color waves, *i.e.*, the Broadcast-convergecast waves on the BFS tree computed by SSLE, are used to ensure that Minid begins from a clean state, and also to detect its termination. Minid is not silent; after computing all $\mathit{MinId}(P,k)$, it resets and starts over.

- Maxminid, which computes, using Interval, for each process P, *MaxMinId*(P, k), the largest value of *MinId*(Q, k) of any process Q which is within distance k of P.

  The color waves are timed so that the computations of Minid and Maxminid alternate. Minid will produce the correct values of *MinId*(P, k) during its first complete execution after SSLE finishes, and Maxminid will produce the correct values of *MaxMinId*(P, k) during its first complete execution after that.

3. *Clustering*, a silent self-stabilizing algorithm which computes the clusters given *Clusterhead_Set*, which is the set of processes P for which *MaxMinId*$(P, k) = P.id$. *Clustering* runs concurrently with Minid and Maxminid, but until those have both finished their first correct computations, *Clustering* may produce incorrect values. *Clusterhead_Set* eventually stabilizes (despite the fact that Minid and Maxminid continue running forever), after which Clustering computes the correct values of $P.clusterhead$ and $P.parent$ for each P, and then becomes silent.

BFS Spanning Tree Module SSLE. We will not give a description of the algorithm SSLE here. It is only necessary to know certain conditions that will hold when SSLE converges. In that list of conditions, given below, we affix the suffix BFS to each variable of SSLE to avoid confusion with the variables of Weighted-Clustering.

- There is one *root* process, which we call *Root*_BFS, which SSLE chooses to be the process of smallest ID in the network.
- $P.dist\_\text{BFS} =$ the length (number of hops) of the shortest path from P to *Root*_BFS.
- $$P.parent\_\text{BFS} = \begin{cases} P & \text{if } P = Root\_\text{BFS} \\ \min\{Q.id : (Q \in \mathcal{N}_P) \wedge \\ \quad (Q.dist\_\text{BFS} + 1) = P.dist\_\text{BFS}\} & \text{otherwise} \end{cases}$$

We also note that SSLE converges in $O(n)$ rounds from an arbitrary configuration.

3.1 Formal Definition of Weighted-Clustering

We now give a formal description of Weighted-Clustering, and present the variables, functions and actions of the algorithm.

Variables. Each process P has the variables listed in Table 1.

Functions. Each process P can evaluate the following functions by reading its variables and those of its neighbors. The functions for the Minid phase, and those for the Maxminid phase are similar.

- $Color\_Error(P) \equiv$
  $\big((P.color \in \{1, 3\}) \wedge (P.parent\_\text{BFS}.color \neq P.color)\big) \vee$
  $\big((P.color = 2) \wedge (P.parent\_\text{BFS}.color = 0)\big) \vee$
  $\big((P.color = 2) \wedge (\exists Q \in Chldrn\_\text{BFS}(P) : Q.color \neq 2)\big)$
  If P evaluates this function to false, then it means that there is a problem in the color waves.
- $Min\_Nbrs(P) = \{Q \in \mathcal{N}_P : (Q.minid < P.minid) \wedge (Q.minlevel + w(P, Q) \leq k)\}$

Variable	Description
All the variables of SSLE	we affix $\_BFS$ to the name of those variables.
P.*color*	in $\{0, 1, 2, 3\}$.
P.*minid*	of ID type
P.*minlevel*	an integer in the range 0 to k
P.*minhilevel*	an integer in the range 1 to $k+1$. Its purpose is to define a search interval for the Minid phase.
$P.minkey$	$= (P.minid, P.minlevel)$, which does not require additional space.
P.*maxminid*	of ID type
P.*maxminlevel*	an number in the range 0 to k
P.*maxminhilevel*	an integer in the range 1 to $k+1$. Its purpose is to define a search interval for the Maxminid phase.
$P.maxminkey$	$= (P.maxminid, P.maxminlevel)$, which does not require additional space.
P.*isclusterhead*	Boolean. After the algorithm has stabilized, this predicate holds if and only if P is a clusterhead.
P.*dist*	an integer in the range 0 to $k+1$. This variable never changes after stabilization.
P.*parent*	ID type. This variable never changes after stabilization, and is the parent in the local spanning tree of the cluster that P is a member of.
P.*clusterhead*	ID type. This variable never changes after stabilization, and is the clusterhead of the cluster that P is a member of.

Table 1: Constants and variables attached to each process P.

- $\mathit{MinLevel\_F}(P) = \begin{cases} \min\{Q.\mathit{minlevel} + w(Q,P) : Q \in \mathit{Min\_Nbrs}(P)\} & \text{if } \mathit{Min\_Nbrs}(P) \neq \emptyset \\ P.\mathit{minlevel} & \text{otherwise} \end{cases}$
- $\mathit{MinId\_F}(P) = \min\{Q.\mathit{minid} : (Q \in \mathcal{U}_P) \wedge (Q.\mathit{minlevel} + w(Q,P) = \mathit{MinLevel\_F}(P))\}$
- $\mathit{MinKey\_F}(P) = (\mathit{MinId\_F}(P), \mathit{MinLevel\_F}(P))$
- $\mathit{MinHiLevel\_F}(P,Q) = \begin{cases} \min\{k+1, Q.\mathit{minlevel} + w(P,Q)\} & \text{if } Q.\mathit{minid} < P.\mathit{minid} \\ \min\{k+1, Q.\mathit{minhilevel} + w(P,Q)\} & \text{otherwise} \end{cases}$
- $\mathit{MinHiLevel\_F}(P) = \min\{\mathit{MinHiLevel\_F}(P,Q) : Q \in \mathcal{N}_P\}$

  Defines the upper bound on the search interval to find the minimum ID.
- $P \xrightarrow{\min} Q \equiv \big((Q \in \mathcal{N}_P) \wedge (P.\mathit{minid} < Q.\mathit{minid}) \wedge (P.\mathit{minlevel} + w(P,Q) \leq k)\big) \vee$
  $\big((Q.\mathit{minid} = P.\mathit{minid}) \wedge (Q.\mathit{minhilevel} + w(P,Q) = P.\mathit{minhilevel})\big)$

  The meaning of the predicate $P \xrightarrow{\min} Q$ is that Q prevents P from executing Action A7, the Minid update action.
- $\mathit{MinLevel\_Valid}(P) \equiv (P.\mathit{minlevel} < P.\mathit{minhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : P.\mathit{minhilevel} + w(P,Q) \geq Q.\mathit{minhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : Q.\mathit{minid} > P.\mathit{minid} \implies P.\mathit{minlevel} + w(P,Q) \geq Q.\mathit{minhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : Q.\mathit{minid} = P.\mathit{minid} \implies P.\mathit{minlevel} + w(P,Q) \geq Q.\mathit{minlevel})$

- *MinLevel_Error*$(P) \equiv$
  $\big((P.\textit{color} = 1) \wedge \neg \textit{MinLevel}\,Valid(P)\big) \vee \big((P.\textit{color} = 2) \wedge (P.\textit{minhilevel} \neq k+1)\big)$
- *MinInit_Error*$(P) \equiv (P.\textit{color} = 0) \wedge$
  $\big((P.\textit{minid} \neq P.\textit{id}) \vee (P.\textit{minlevel} \neq 0) \vee (P.\textit{minhilevel} \neq \textit{MinHop}(P))\big)$
- *MaxMin_Nbrs*$(P) =$
  $\{Q \in \mathcal{N}_P : (Q.\textit{maxminid} > P.\textit{maxminid}) \wedge (Q.\textit{maxminlevel} + w(P,Q) \leq k)\}$
- *MaxMinLevel_F*$(P) =$
  $\begin{cases} \min\{Q.\textit{maxminlevel} + w(Q,P) : Q \in \textit{MaxMin\_Nbrs}(P)\} & \text{if } \textit{MaxMin\_Nbrs}(P) \neq \emptyset \\ P.\textit{maxminlevel} & \text{otherwise} \end{cases}$
- *MaxMinId_F*$(P) =$
  $\max\{Q.\textit{maxminid} : (Q \in \mathcal{U}_P) \wedge (Q.\textit{maxminlevel} + w(Q,P) = \textit{MaxMinLevel\_F}(P))\}$
- *MaxMinKey_F*$(P) = (\textit{MaxMinId\_F}(P), \textit{MaxMinLevel\_F}(P))$
- *MaxMinHiLevel_F*$(P,Q) = \begin{cases} \min\{k+1, Q.\textit{maxminlevel} + w(P,Q)\} & \text{if } Q.\textit{maxminid} > P.\textit{maxminid} \\ \min\{k+1, Q.\textit{maxminhilevel} + w(P,Q)\} & \text{otherwise} \end{cases}$
- *MaxMinHiLevel_F*$(P) = \min\{\textit{MaxMinHiLevel\_F}(P,Q) : Q \in \mathcal{N}_P\}$
  Defines the upper bound on the research interval to find the maximum ID.
- *MaxMinLevel_Valid*$(P) \equiv (P.\textit{maxminlevel} < P.\textit{maxminhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : P.\textit{maxminhilevel} + w(P,Q) \geq Q.\textit{maxminhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : Q.\textit{maxminid} < P.\textit{maxminid} \implies P.\textit{maxminlevel} + w(P,Q) \geq Q.\textit{maxminhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : Q.\textit{maxminid} = P.\textit{maxminid} \implies P.\textit{maxminlevel} + w(P,Q) \geq Q.\textit{maxminlevel})$
- *MaxMinLevel_Error*$(P) \equiv$
  $\big((P.\textit{color} = 3) \wedge \neg \textit{MaxMinLevel}\,Valid(P)\big) \vee \big((P.\textit{color} = 0) \wedge (P.\textit{maxminhilevel} \neq k+1)\big)$
- *MaxMinInit_Error*$(P) \equiv (P.\textit{color} = 2) \wedge$
  $\big((P.\textit{maxminid} \neq P.\textit{minid}) \vee (P.\textit{maxminlevel} \neq 0) \vee (P.\textit{maxminhilevel} \neq \textit{MinHop}(P))\big)$
- $P \overset{\max\min}{\longrightarrow} Q \equiv$
  $\big((Q \in \mathcal{N}_P) \wedge (P.\textit{maxminid} > Q.\textit{maxminid}) \wedge (Q.\textit{maxminlevel} < P.\textit{maxminlevel} + w(P,Q) \leq k)\big) \vee$
  $\big((Q.\textit{maxminid} = P.\textit{maxminid}) \wedge (Q.\textit{maxminhilevel} + w(P,Q) = P.\textit{maxminhilevel})\big)$
  The meaning of the predicate $P \overset{\max\min}{\longrightarrow} Q$ is that Q prevents P from executing Action A9, the Maxminid update action.
- *IsClusterhead_F*$(P) \equiv P.\textit{maxminid} = P.\textit{id}$, of Boolean type.
- *Dist_F*$(P) = \begin{cases} 0 & \text{if } P.\textit{isclusterhead} \\ \min\{k+1, \min\{Q.\textit{dist} + w(P,Q) : Q \in \mathcal{N}_P\}\} & \text{otherwise} \end{cases}$

- $\mathit{Parent\_F}(P) = \begin{cases} P.\mathit{id} & \text{if } P.\mathit{isclusterhead} \\ \min\{Q.\mathit{id} : (Q \in \mathcal{N}_P) \wedge & \\ \quad (Q.\mathit{dist} + w(P,Q) = \mathit{Dist\_F}(P))\} & \text{otherwise} \end{cases}$
- $\mathit{Clusterhead\_F}(P) = \begin{cases} P.\mathit{id} & \text{if } P.\mathit{isclusterhead} \\ P.\mathit{parent}.\mathit{clusterhead} & \text{otherwise} \end{cases}$

We also define the following macro, which implements the clustering phase. It executes during every round after errors are eliminated, endlessly checking for local correctness of the clustering module variables.

Cluster(P):
if $(P.\mathit{dist} \neq \mathit{Dist\_F}(P))) \vee (P.\mathit{parent} \neq \mathit{Parent\_F}(P)) \vee (P.\mathit{clusterhead} \neq \mathit{Clusterhead\_F}(P))$
  $P.\mathit{dist} \leftarrow \mathit{Dist\_F}(P)$
  $P.\mathit{parent} \leftarrow \mathit{Parent\_F}(P)$
  $P.\mathit{clusterhead} \leftarrow \mathit{Clusterhead\_F}(P)$
end if

Actions. We give the actions of Weighted-Clustering in Table 2. The short name of each action is listed in the first column, along with its priority number. The second column gives the full name. The guard of each action is the conjunction of each condition listed in the third column. In order for an action to be enabled, its guard must be true, and no action with a lower priority number may be enabled.

Action A1 builds a BFS tree. Actions A2 to A6 correct the errors on the color waves, and the Minid and Maxminid variables; once these actions have executed the process is in a *clean state*. Actions A7 to A8 compute the minimum ID at a distance no greater than k. Actions A9 to A10 retrieve the maximum $\mathit{MinId}(P,k)$. The color waves that synchronize the different phases are implemented by actions A11 to A14.

Note that $\mathit{MinLevel\_F}(P) = \mathit{MinHiLevel\_F}(P)$ if the guard of Action A7 holds, and that $\mathit{MaxMinLevel\_F}(P) = \mathit{MaxMinHiLevel\_F}(P)$ if the guard of Action A9 holds,

Time and Space Complexity. The algorithm uses all the variables of SSLE [3] and 11 internal variables. SSLE uses $O(\log n)$ space. The internal ID variables can be encoded on $O(\log n)$ space, and the distance variables on $O(\log k)$ space. Hence Weighted-Clustering requires on the whole $O(\log n + \log k)$ memory per process.

SSLE converges in $O(n)$ rounds, while the clustering module requires $O(n)$ rounds once *Clusterhead_Set* has been correctly computed. It is the algorithm $\mathrm{Interval}$ that is the most time-consuming, requiring $O(nk)$ rounds to converge. The total time complexity of the Weighted-Clustering is thus $O(nk)$.

Priority	**Name**	**Guard**		**Action**
A1 priority 1	SSLE Action	P is enabled to execute an action of SSLE	$\longrightarrow$	Execute an enabled action of SSLE
A2 priority 2	Color Error	*Color_Error*(P)	$\longrightarrow$	P.*color* $\leftarrow 0$
A3 priority 3	Min Level Error	*MinLevel_Error*(P)	$\longrightarrow$	P.*minlevel*$(P) \leftarrow k$ P.*minhilevel*$(P) \leftarrow k+1$
A4 priority 3	Maxmin Level Error	*MaxMinLevel_Error*(P)	$\longrightarrow$	P.*maxminlevel*$(P) \leftarrow k$ P.*maxminhilevel*$(P) \leftarrow k+1$
A5 priority 3	Min Init Error	*MinInit_Error*(P)	$\longrightarrow$	P.*minid* $\leftarrow P$.*id* P.*minlevel*$(P) \leftarrow 0$ P.*minhilevel*$(P) \leftarrow$ *MinHop*(P)
A6 priority 3	Maxmin Init Error	*MaxMinInit_Error*(P)	$\longrightarrow$	P.*maxminid* $\leftarrow P$.*minid* P.*maxminlevel*$(P) \leftarrow 0$ P.*maxminhilevel*$(P) \leftarrow$ *MinHop*(P)
A7 priority 4	Minid Update	P.*color* $= 1$ *MinHiLevel_F*$(P) = P$.*minhilevel* $\leq k$ $\neg\exists Q \in \mathcal{N}_P : P \xrightarrow{\min} Q$	$\longrightarrow$	*Cluster*(P) P.*minkey* $\leftarrow$ *MinKey_F*(P) P.*minhilevel* $\leftarrow$ *MinHiLevel_F*(P)
A8 priority 4	Min Hi Level	P.*color* $= 1$ P.*minhilevel* $<$ *MinHiLevel_F*(P)	$\longrightarrow$	*Cluster*(P) P.*minhilevel* $\leftarrow$ *MinHiLevel_F*(P)
A9 priority 4	Maxminid Update	P.*color* $= 3$ *MaxMinHiLevel_F*$(P) = P$.*maxminhilevel* $\leq k$ $\neg\exists Q \in \mathcal{N}_P : P \xrightarrow{\max\min} Q$	$\longrightarrow$	*Cluster*(P) P.*maxminkey* $\leftarrow$ *MaxMinKey_F*(P) P.*maxminhilevel* $\leftarrow$ *MaxMinHiLevel_F*(P)
A10 priority 4	Maxmin Hi Level	P.*color* $= 3$ P.*minhilevel* $<$ *MaxMinHiLevel_F*(P)	$\longrightarrow$	*Cluster*(P) P.*minhilevel* $\leftarrow$ *MaxMinHiLevel_F*(P)
A11 priority 5	Color 1	P.*color* $= 0$ $(P =$ *Root*_BFS$) \vee (P$.*parent*_BFS.*color* $= 1)$	$\longrightarrow$	*Cluster*(P) P.*color* $\leftarrow 1$
A12 priority 5	Color 2	P.*color* $= 1$ P.*minhilevel* $= k+1$ $\forall Q \in$ *Chldrn*_BFS$(P) : P$.*color* $= 2$	$\longrightarrow$	*Cluster*(P) P.*color* $\leftarrow 2$ P.*maxminid* $\leftarrow P$.*minid* P.*maxminlevel* $\leftarrow 0$ P.*maxminhilevel* $\leftarrow$ *MinHop*(P)
A13 priority 5	Color 3	P.*color* $= 2$ $(P =$ *Root*_BFS$) \vee (P$.*parent*_BFS.*color* $= 3)$	$\longrightarrow$	*Cluster*(P) P.*color* $\leftarrow 3$
A14 priority 5	Color 0	P.*color* $= 3$ P.*maxminhilevel* $= k+1$ $\forall Q \in$ *Chldrn*_BFS$(P) : P$.*color* $= 0$	$\longrightarrow$	*Cluster*(P) P.*color* $\leftarrow 0$ P.*minid* $\leftarrow P$.*id* P.*minlevel* $\leftarrow 0$ P.*minhilevel* $\leftarrow$ *MinHop*(P) P.*isclusterhead* $\leftarrow$ *IsClusterhead_F*(P)
A15 priority 6	Clustering		$\longrightarrow$	*Cluster*(P)

Table 2: Actions of Weighted-Clustering.

4 An Example Computation

4.1 A toy example

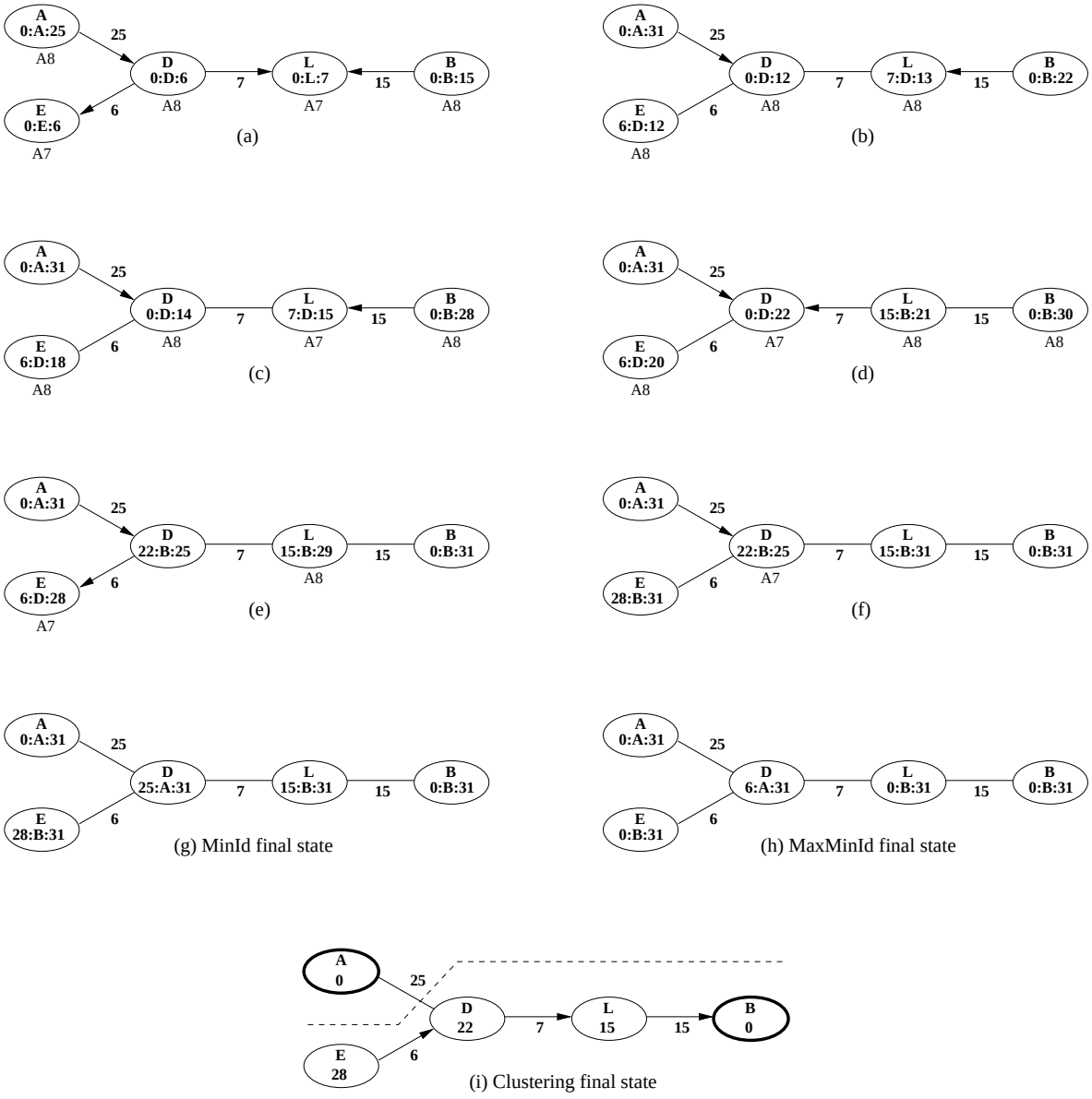


Figure 1: Example computation of Weighted-Clustering for $k = 30$.

In Figure 1, each oval represents a process P. The top letter in the oval is $P.id$. Below that, for subfigures (a) to (g) we show $P.minlevel$, followed by a colon, followed by $P.minid$, followed by a colon, followed by $P.minhilevel$; an arrow from P to Q indicates that $P \xrightarrow{\min} Q$. Below each oval is shown the action the process is enabled to execute (none if the process is disabled). In subfigure (h) we show the final values of $P.maxminlevel$, followed by a colon, followed by $P.maxminid$, followed by a colon, followed by $P.maxminhilevel$. In subfigure (i) we show the final value of $P.dist$; an arrow from P to Q indicates that $P.parent = Q.id$, a bold oval means that the process is a clusterhead. The

dashed line represents the separation between the two final k-clusters.
In Figure 1(a) to (g), we show a synchronous execution of the *MinId* phase. The result would have been the same with an asynchronous execution, the synchronicity just makes the example easier to understand. We want to find a k-clustering for $k = 30$.
In each step, if an arrow leaves a process, then this process cannot execute Action A7, but can possibly execute Action A8 to update its *minhilevel* variable. Note, that at each step, two neighbors cannot execute simultaneously Action A7 due to the $\stackrel{\min}{\longrightarrow}$ function in the guard. This prevents miscalculations of *minid*.
Consider the process L. Initially it is enabled to execute Action A7 (subfigure (a)). It will, after the first execution (subfigure (b)), find the value of the smallest ID within a distance of $L.minhilevel = 7$, which is D, and will at the same time update its *minhilevel* value to $D.minhilevel + w(D, L) = 6 + 7 = 13$. As during this step, D and B have updated their *minhilevel* value, $L.minhilevel$ is an underestimate of the real *minhilevel*, thus L is now enabled to execute Action A8 to correct this value.
The idea behind the *minhilevel* variable, is to prevent the process from searching a minimum ID at a distance greater than *minhilevel*. Thus a process will not look at the closest minimum ID in terms of number of hops (as could do process D at the beginning by choosing process A), but will compute the minimum ID within a radius equal to *minhilevel* around itself (hence process D is only able to choose process A in the final step, even if A is closer than B in terms of number of hops). The Minid phase halts when $P.minhilevel = k+1$ for all P (subfigure (g)). In the final step every P knows the process of minimum ID at a distance no greater than k, and $P.minlevel$ holds the distance to this process.
Sometimes, a process P can be elected clusterhead by another process Q without having elected itself clusterhead (this case do not appear in our example); P could have the smallest ID of any process within k of Q, but not the smallest ID of any node within k of itself. The *MaxMinId* phase corrects this; it allows the information to flow back to the processes which are elected clusterheads.

4.2 Detailed explanation of the example

We give in this section a few more details to better understand the example given in Figure 1.
We first define for any two processes P and Q the following:

- $\mathit{minlo}_{P,Q} = \min\{d : \mathit{MinId}(P, d) \leq Q.\mathit{id}\}$. If $\mathit{MinId}(P, k) > Q.\mathit{id}$, we assign the default value $\mathit{minlo}(P, Q) = k + 1$.
- $\mathit{minhi}_{P,Q} = \min\{d : \mathit{MinId}(P, d) < Q.\mathit{id}\}$. If $\mathit{MinId}(P, k) \geq Q.\mathit{id}$, we assign the default value $\mathit{minhi}(P, Q) = k + 1$.
- $\mathcal{I}_{P,Q} = \{0 \leq d \leq k : \mathit{MinId}(P, d) = Q.\mathit{id}\} = \{\mathit{minlo}_{P,Q} \leq d < \mathit{minhi}_{P,Q}\}$

Remark 1 *Let P and Q be any processes. Then:*

1. *$\mathcal{I}_{P,Q}$ is either empty or is the half-open interval: $[\mathit{minlo}_{P,Q}, \mathit{minhi}_{P,Q})$.*
2. *If $P.\mathit{id} < Q.\mathit{id}$, then $\mathit{minlo}_{P,Q} = \mathit{minhi}_{P,Q} = k + 1$, and $\mathcal{I}_{P,Q} = \emptyset$.*
3. *For any $0 \leq d \leq k$, $MinId(P, d) = Q.\mathit{id}$ if and only if $d \in \mathcal{I}_{P,Q}$.*

As an example, consider the network shown in Figure 1, where the names of the processes are A, B, D, E, L. We then have the following intervals:

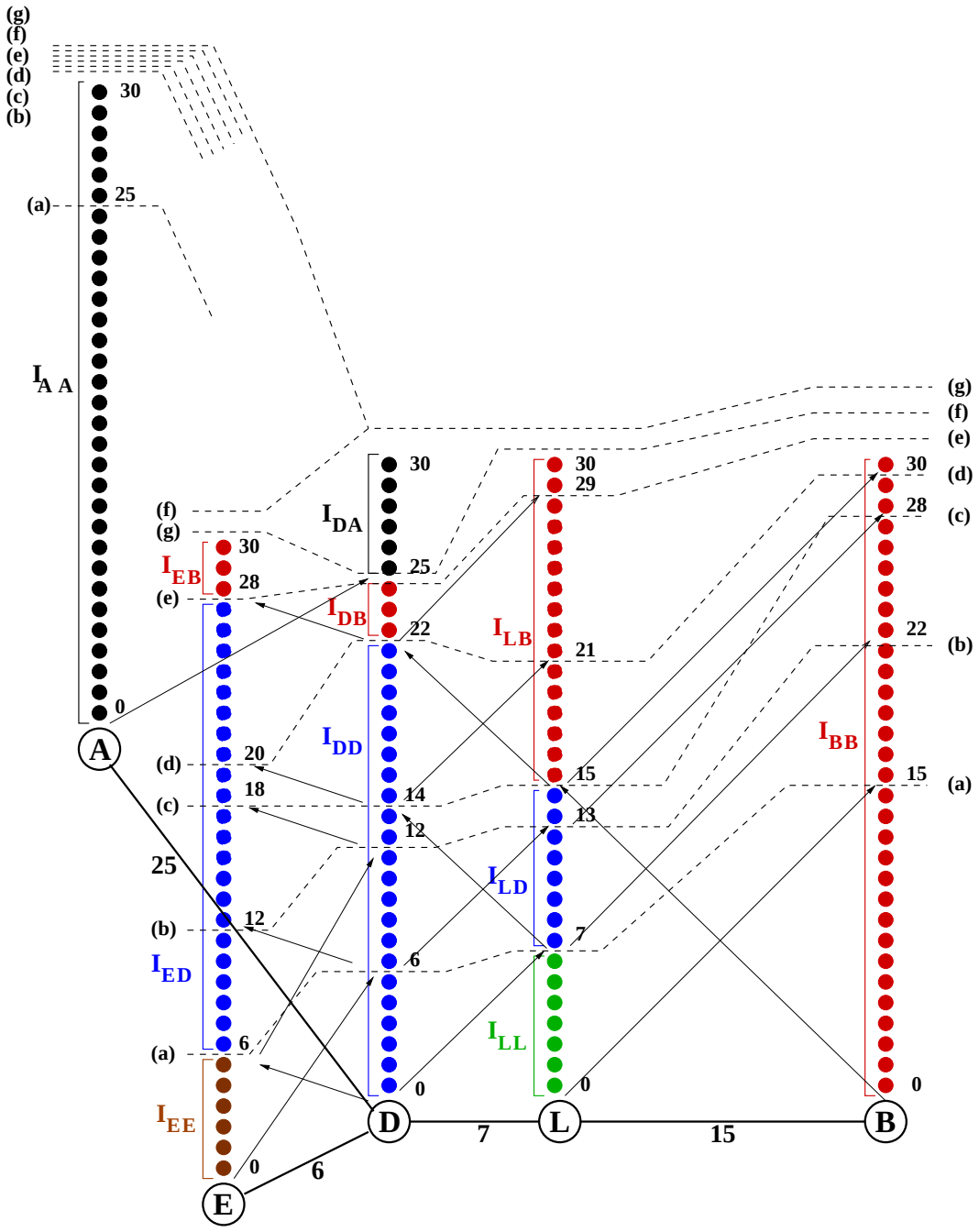


Figure 2: Growth of the Function *minhilevel*.

$$\mathcal{I}_{A,A} = [0, 31) \quad \mathcal{I}_{B,B} = [0, 31) \quad \mathcal{I}_{D,D} = [0, 22) \quad \mathcal{I}_{D,B} = [22, 25)$$
$$\mathcal{I}_{D,A} = [25, 31) \quad \mathcal{I}_{E,E} = [0, 6) \quad \mathcal{I}_{E,D} = [6, 28) \quad \mathcal{I}_{E,B} = [28, 31)$$
$$\mathcal{I}_{L,L} = [0, 7) \quad \mathcal{I}_{L,D} = [7, 15) \quad \mathcal{I}_{L,B} = [15, 31)$$

Explanation of Figure 2. The intervals $\mathcal{I}_{P,Q}$ are shown as columns of colored dots over the processes P. The colors depend on Q: black for A, red for B, blue for D, brown for E, and green for L.

There is one dot over each process for each level in the range 0 to 30, but levels are only shown in the figure for dots of particular interest.
The dashed paths show the values of *minhilevel* at any step in the computation shown in Figure 1. The dashed paths above A should be connected to the dashed paths between E and D, but to avoid clutter, only the top one, labeled (h), is connected.
There is one dashed path from E to B for each step in the computation. At each step, the values of *MinId*(P, d) have been computed for all d which are below the dashed path above P in Figure 2.
At any given point in the computation, P.*minlevel* is the left (low) end of the interval $\mathcal{I}_{P,Q}$, namely *minhi*$_{P,Q}$, where Q.*id* $=$ P.*minid*. At the same time, the value of P.*minhilevel* is an estimate of *minhi*$_{P,Q}$, the right (high) end of $\mathcal{I}_{P,Q}$; never an overestimate, but sometimes an underestimate.
If P executes Action A8 (Min Hi Level), it raises the value of P.*minhilevel*, improving its estimate of the high end of the interval, *minhi*$_{P,Q}$. If P executes Action A7 (Minid), it changes to the next interval, setting the new value of P.*minlevel* to the old value of P.*minhilevel*, and computing a new, higher value of P.*minhilevel*.
At every point in the computation, P raises P.*minhilevel* to the largest possible value it can based on its current knowledge, *i.e.*, the values of its variables and those of its neighbors, subject to the condition that it may not overestimate *minhi*$_{P,Q}$.

Example Consider the process L in the example. Initially, as shown in Figure 1(a), L.*minid* $= L$, L.*minlevel* $= 0$, and L.*minhilevel* $= 7$, since the nearest neighbor is at distance 7.
In Figure 1(b), L.*minid* $\leftarrow D$, L.*minlevel* $\leftarrow 7$, and L.*minhilevel* $\leftarrow 13$. At the end of this step L.*minhilevel* is underestimated, thus in the next step (c), L is enabled to execute A8 and correct its value L.*minhilevel* to 15.
At step (d), L.*minid* $\leftarrow B$, L.*minlevel* $\leftarrow 15$, and L.*minhilevel* $\leftarrow 21$.
At step (e), L.*minhilevel* $\leftarrow 21$. 21 is an underestimate of *minhi*$_{L,B} = 29$.
At step (f), L.*minhilevel* obtains its final value of $k + 1 = 31$. Finally nothing changes at step (g), and at this point, L has stabilized.

Thin arrows. Thin arrows show influence. For example, the thin arrow from 12 at D to 18 at E shows that E.*minhilevel* $\leftarrow D$.*minhilevel* $+ w(E, D) = 12 + 6$ at step (c).

5 Proofs

5.1 Overview of Proof

We will now give an overview of the sketch of the proof of the algorithm.
We use the *convergence stair* method of proof [9]. We define a sequence of *benchmarks*, each of which is closed. The first benchmark is that the first phase, which uses SSLE, has converged. SSLE gives us a leader and a BFS spanning tree, which we use to synchronize the second and third phases of Weighted-Clustering.
Computation of the second and third phases alternates, using two convergecast-broadcast waves of the BFS tree. During the first wave the second phase, Minid, calculates, for each process P, the minimum ID of any process within distance k of P. This ID we call *MinId*(P, k); in general, *MinId*(P, d) is the minimum ID of any neighbor within distance d of P.

During the second wave the third phase, Maxminid, calculates, for each process P, the maximum value of any *MinId*(Q, k) for any process Q within distance k of P. This ID is stored as P.*maxminid*. P is then designated to be a *clusterhead* if P.*maxminid* $= P$.*id*.
At any point in the second phase, each process P has explored its neighbors of distance less than some $d \leq k$. Each P keeps track of an interval of values of d for which *MinId*$(P, d) = P$.*minid*. The value of d increases during the computation, and is stored as P.*minhilevel*, while the lowest ID within that radius is stored as P.*minid*.
The phase ends when P.*minhilevel* $= k + 1$ for all P. The third phase uses the same algorithm, substituting max for min when appropriate.
The sequence of benchmarks are progressively stronger conditions, each stating that the calculation up to a certain point has been done correctly. The final benchmark is that a legitimate configuration has been achieved.

5.2 Properties of the Abstract Functions *MinId* and *MaxMinId*

We will now prove that Weighted-Clustering correctly computes *MinId*(P, k) for all P. The following properties either hold by definition, or follow immediately from the definitions.

1. *MinId*$(P, d_1) \leq$ *MinId*(P, d_2) if $d_1 > d_2$.
2. *MinId*$(P, w(P, Q)) \leq Q$.*id*
3. If $d \geq 0$ and $d + w(P, Q) \leq k$, then *MinId*$(P, d + w(P, Q)) \leq$ *MinId*(Q, d)
4. *MaxMinId*$(P, d_1) \leq$ *MaxMinId*(P, d_2) if $d_1 > d_2$.
5. *MaxMinId*$(P, w(P, Q)) \geq Q$.*minid*
6. If $d \geq 0$ and $d + w(P, Q) \leq k$, then *MaxMinId*$(P, d + w(P, Q)) \geq$ *MaxMinId*(Q, d)

5.3 Benchmarks

We use the *convergence stair* method of proof. We define a sequence of seven benchmarks. We prove that each benchmark is closed, and that once a given benchmark has been achieved, the next benchmark will EVENTUALLY hold. The last benchmark is then the legitimacy condition.
We define $diam$ to be the diameter of the network in number of hops.

Benchmark B1: Action A1 is not enabled for any process.

Benchmark B2: Benchmark B1 holds, and there is no color error.

Benchmark B3: Benchmark B2 holds, and for every process P:

1. If P.*color* $= 0$, then
   (a) P.*maxminhilevel* $= k + 1$
   (b) P.*minid* $= P$.*id*
   (c) P.*minlevel* $= 0$
   (d) P.*minhilevel* $= \min\{w(P, Q) : Q \in \mathcal{N}_P\}$

2. If $P.color = 2$, then
   (a) $P.minhilevel = k + 1$
   (b) $P.maxminid = P.minid$
   (c) $P.maxminlevel = 0$
   (d) $P.maxminhilevel = \min\{w(P, Q) : Q \in \mathcal{N}_P\}$

Benchmark B4: Benchmark B3 holds, and either *Root*_BFS.*color* $= 3$ or the Minid Invariants hold.

Benchmark B5: Benchmark B4 holds, and either *Root*_BFS.*color* $= 1$ or the Maxminid Invariants hold.

Benchmark B6: Benchmark B5 holds, and for all P, P.*isclusterhead* if and only if $P \in$ *Clusterhead_Set*.

Benchmark B7: Benchmark B6 holds, and for all P, the following hold:

1. $P.dist = Dist(P)$
2. $P.parent = Parent(P)$
3. $P.clusterhead = Clusterhead(P)$

Lemma 1 *Benchmark B1 is closed, and will hold within $O(n)$ rounds of initialization.*

Proof: From [3], SSLE is silent, and converges in $O(n)$ rounds of arbitrary initialization. □

Lemma 2 *Benchmark B2 is closed, and if Benchmark B1 holds, B2 will hold within $O(diam)$ additional rounds.*

Proof: Let h be the height of the BFS tree. Define a *color* string to be the string of *color* values of processes of a path in the BFS tree starting from *Root*_BFS and ending at a leaf. Benchmark B2 holds if and only if every *color* string lies in the language W described by the regular expression $(1^* + 3^*)(0^* + 2^*)$. For any string $w \in \{0, 1, 2, 3\}^*$, we define integers $\theta(w), \psi(w) \geq 0$ as follows.

$$\theta(w) = \psi(w) = 0 \text{ if } w \in W$$

Otherwise, write $w = uav$ where $a \in \{0, 1, 2, 3\}$, $u \in W$, and $ua \notin W$. Then let

$$\theta(w) = \begin{cases} 0 & \text{if } u \text{ ends with } 0 \\ |u| & \text{otherwise} \end{cases}$$

$$\psi(w) = \begin{cases} |v| & \text{if } u \text{ ends with } 0 \\ 0 & \text{otherwise} \end{cases}$$

Now define

$$\Phi_{color} = \begin{cases} \max\{\theta(w)\} + h & \text{if } \exists w : \theta(w) > 0 \\ \max\{\psi(w)\} & \text{otherwise} \end{cases}$$

where the maximum is taken over all *color* strings.

We now make the following three claims.
Claim A: If w is a *color* string and $\theta(w) > 0$, then $\theta(w)$ decreases during the next round.
Proof: The last symbol of u will change to 0 in the next round, by execution of Action A2. □
Claim B: If w is a *color* string, $\psi(w) > 0$, and $\theta(w') = 0$ for all color strings w', then $\psi(w)$ decreases during the next round.
Proof: In the next round, a will change to 0 by execution of Action A2. □
Claim C: If $\Phi_{color} > 0$, then Φ_{color} decreases during the next round.
Proof: If $\theta(w) = 0$ for all w, we are done by Claim B. Otherwise, we are done by Claim A, since $\psi(w) \leq h$ for all w. □

Returning to the proof of the Lemma, we note that $\Phi_{color} \leq 2h = O(diam)$. By Claim C, we are done. □

Lemma 3 *Benchmark B3 is closed, and if Benchmark B2 holds, B3 will hold within two rounds.*

Proof: No action can cause Benchmark B3 to change from true to false.

Since Actions A1 and A2 are not enabled, Actions A3 A4, A5, and A6 will execute for every process which violates Benchmark B3. Since a process could have up to two errors, two rounds may be necessary. □

Lemma 4 *If Benchmark B3 holds, Root_*BFS*.color* $= 1$, $P.color \in \{1, 2\}$ *for all processes* P*, then at least one process is enabled to execute Action A3, A7, or A8.*

Proof: Since Benchmark B2 holds, Actions A1 and A2 cannot be enabled for any process P. Action A3 cannot be enabled since $P.color = 1$ for all P. Thus, the priority conditions on the guards of Actions A3, A7, and A8 hold for all processes.
Pick a process P such that

1. $P.minhilevel \leq k$,
2. $P.minid$ is maximum subject to 1., and
3. $P.minhilevel$ is minimum subject to 1. and 2.

Case I: $\neg MinLevel\,Valid(P)$. Then P is enabled to execute Action A3.

Case II: $MinLevel\,Valid(P) \wedge (P.minhilevel < MinHiLevel(P))$. Then P is enabled to execute Action A8.

Case III: $P \xrightarrow{\min} Q$, for some $Q \in \mathcal{N}_P$ such that $Q.minid > P.minid$. By the definition of P, $Q.minhilevel = k + 1$. By definition of $P \xrightarrow{\min} Q$,

$$P.minlevel + w(P, Q) < k + 1 = Q.minhilevel$$

Thus $\neg$*Min*$\,Valid(P)$, contradiction.

Case IV: $P \xrightarrow{\min} Q$, for some $Q \in \mathcal{N}_P$ such that $Q.$*minid* $= P.$*minid*. Then $Q.$*minhilevel* $<$ $P.$*minhilevel*, which contradicts the definition of P.

Case V: $\left(\neg\exists Q : P \xrightarrow{\min} Q\right) \wedge$ *MinLevel*$\,Valid(P) \wedge (P.$*minhilevel* $=$ *MinHiLevel*$(P))$. Then P can execute Action A7. □

Lemma 5 *If Benchmark B3 holds, then*

(a) *If Root*_BFS$.color = 0$*, then within one round, Root*_BFS*.color* $= 1$.

(b) *If Root*_BFS$.color = 1$*, then within* $nk + 2$*diam rounds, Root*_BFS*.color* $= 2$.

(c) *If Root*_BFS$.color = 2$*, then within one round, Root*_BFS*.color* $= 3$.

(d) *If Root*_BFS$.color = 3$*, then within* $nk + 2$*diam rounds, Root*_BFS*.color* $= 0$.

Proof: (a): If *Root*_BFS*.color* $= 0$, then *Root*_BFS is enabled to execute Action A11, and will do so within one round.

(b): If there is any process of color 0 whose parent has color 1, that process is only able to execute Action A11. Thus, at within *diam* steps, all processes will have color either 1 or 2.
At this time, $\sum P.$*minhilevel* $\geq n$. By Lemma 4, during each round, $\sum P.$*minhilevel* will increase by at least 1 during each round, until it reaches its maximum value of $n(k+1)$. Within at most *diam* additional rounds, all processes of color 1 will execute Action A12, changing their color to 2.

(c): If *Root*_BFS*.color* $= 2$, then *Root*_BFS is enabled to execute Action A13, and will do so within one round.

(d) is similar to (b). □

To prove the remaining benchmarks, we now define the following invariants for the Minid and Maxminid phases:

Minid Invariants. For each process P:

1. *MinLevel*$\_Valid(P)$. This consists of four parts:
   (a) $P.$*minlevel* $< P.$*minhilevel*
   (b) For any $Q \in \mathcal{N}_P$, $P.$*minhilevel* $+ w(P,Q) \geq Q.$*minhilevel*.
   (c) For any $Q \in \mathcal{N}_P$, $Q.$*minid* $> P.$*minid* $\implies$ $P.$*minlevel* $+ w(P,Q) \geq Q.$*minhilevel*.
   (d) For any $Q \in \mathcal{N}_P$, $Q.$*minid* $= P.$*minid* $\implies$ $P.$*minlevel* $+ w(P,Q) \geq Q.$*minlevel*.
2. There is some process Q such that $P.$*minid* $= Q.$*id*.
3. If $d \geq P.$*minlevel* then *MinId*$(P,d) \leq P.$*minid*.

4. If $\mathit{MinId}(P, w(P,Q)) = Q.\mathit{id}$, $R \in \mathcal{N}_P$, and $w(P,Q) = w(P,R) + w(R,Q)$, then either $P.\mathit{minid} \leq Q.id$ or $R.\mathit{minid} \geq Q.id$.
5. If $P.\mathit{minlevel} \leq d < P.\mathit{minhilevel}$ then $\mathit{MinId}(P,d) = P.\mathit{minid}$.

Maxminid Invariants. For each process P:

6. $\mathit{MaxMinLevel\_Valid}(P)$. This consists of four parts:
   (a) $P.\mathit{maxminlevel} < P.\mathit{maxminhilevel}$
   (b) For any $Q \in \mathcal{N}_P$, $P.\mathit{maxminhilevel} + w(P,Q) \geq Q.\mathit{maxminhilevel}$.
   (c) For any $Q \in \mathcal{N}_P$, $Q.\mathit{maxminid} < P.\mathit{maxminid} \implies P.\mathit{maxminlevel} + w(P,Q) \geq Q.\mathit{maxminhilevel}$.
   (d) For any $Q \in \mathcal{N}_P$, $Q.\mathit{maxminid} = P.\mathit{maxminid} \implies P.\mathit{maxminlevel} + w(P,Q) \geq Q.\mathit{maxminlevel}$.
7. There is some process Q such that $P.\mathit{maxminid} = Q.\mathit{id}$.
8. If $d \geq P.\mathit{maxminlevel}$ then $\mathit{MaxMinId}(P,d) \geq P.\mathit{maxminid}$.
9. If $\mathit{MaxMinId}(P, w(P,Q)) = Q.\mathit{minid}$, $R \in \mathcal{N}_P$, $w(P,Q) = w(P,R) + w(R,Q)$, and $\mathit{MaxMinId}(R, w(R,Q)) = Q.\mathit{minid}$, then either $P.\mathit{maxminid} \geq Q.\mathit{minid}$ or $R.\mathit{maxminid} \leq Q.itminid$.
10. If $P.\mathit{maxminlevel} \leq d < P.\mathit{maxminhilevel}$ then $\mathit{MaxMinId}(P,d) = P.\mathit{maxminid}$.

Lemma 6 *Benchmark B4 is closed, and if Benchmark B3 holds, then Benchmark B4 will hold within $O(nk)$ additional rounds.*

Proof: Suppose Benchmark B3 holds. By Lemma 5, $\mathit{Root\_BFS.color} = 3$ within $O(nk)$ rounds, and thus Benchmark B4 holds.

We now show that B4 is closed. Consider consecutive steps $\gamma \mapsto \gamma'$ in an execution of Weighted-Clustering, and assume that Benchmark B4 holds at configuration γ. We need to prove that B4 holds at γ'.

Case I: $\mathit{Root\_BFS.color} = 3$ at γ'. This case is trivial.

Case II: $\mathit{Root\_BFS.color} = 0$ at γ'. Then $P.\mathit{minid} = P.\mathit{id}$, $P.\mathit{minlevel} = 0$, and $P.\mathit{minhilevel} = \mathit{MinHop}(P)$ for all P. It is a routine exercise to verify that the Minid invariants hold, and thus that Benchmark B4 holds.

Case III: Not Case I or Case II.

Suppose all the invariants hold at a configuration γ. We need to show that all invariants hold at configuration γ'. Since Invariant 1 holds before the step, the only actions that could effect the invariants that can occur during the step are A7 and A8.

Pick a process P.

Invariant 1:

1a holds for P, since no action can make it false.

Consider 1b. Suppose $Q \in \mathcal{N}_P$. If Q does not execute during the step, the inequality cannot change from true to false, since $P.minhilevel$ cannot decrease. If Q executes Action A7 or A8, then $Q.minhilevel \leftarrow MinHiLevel\_F(Q) \leq P.minhilevel + w(P,Q)$.

Consider 1c. Suppose $Q \in \mathcal{N}_P$, and $Q.minid > P.minid$ at γ'. If P does not execute, $MinHiLevel\_F(Q) \leq P.minlevel(P) + w(P,Q)$ is an upper bound on the value of $Q.minhilevel$ after the step. If P executes, then the new value of $P.minlevel + w(P,Q)$ is equal to the old value of $P.minhilevel + w(P,Q)$, which is also an upper bound on the value of $Q.minhilevel$ after the step.

Consider 1d. Suppose $Q \in \mathcal{N}_P$, and $Q.minid = P.minid$ at γ'. Suppose P does not execute Action A7. If Q does not execute Action A7, and the invariant holds because by the inductive hypothesis, since the inequality does not change. If Q executes A7, then $Q.minid > P.minid$ before the step, Since Invariant 1c holds before the step, $P.minlevel + w(P,Q)$ is at least as great as the old value of $Q.minhilevel$ which equals the new value of $Q.minlevel$.

On the other hand, suppose P executes A7. We are done since Invariant 1b holds before the step.

Invariant 2:

The set of all values of *minid* over all process cannot gain a member during the step, since any new value of $P.minid$ is copied from $R.minid$ for some neighbor process R.

Invariant 3:

If $P.minid = P.id$, we are done. Otherwise, by Invariant 2, there is some process Q such that $P.minid = Q.id$. If P does not execute Action A7 during the step, we are done, since the invariant holds at γ. Otherwise Pick $R.minid = Q.id$ and $P.minlevel = R.minlevel + w(P,R)$. Since the invariant holds at γ, $MinId(R, d - w(P,R)) \leq Q.id$, and thus, by Property 3, $MinId(P,d) \leq Q.id$.

Invariant 4:

By Properties 2 and 3, $MinId(R, w(R,Q)) = Q.id$. We prove the invariant by contradiction. Suppose $R.minid < Q.id < P.minid$. By Invariant 3, $R.minlevel > w(R,Q)$. It follows, by Invariant 1c, that $P.minhilevel > w(P,Q)$.

Since Invariant 4 holds at configuration γ, R must execute Action A7 during the step, and $R.minid = S.id$ at configuration γ for some process S. $MinLevel(R) = R.minhilevel$ at γ because R is enabled to execute A7. That value is equal to the value of $R.minlevel$ at γ', which is greater than $w(P,R)$. If $S.id < P.minid$, then $R \xrightarrow{\min} P$ at γ, preventing A7 from executing during the step, contradiction.

Suppose $S.id > P.minid > Q.id$. Let $d = P.minhilevel - 1 \geq w(P,Q)$. By Invariant 5 at γ, $MinId(R,d) = S.id$, while $MinId(R,d) \leq Q.id$ by definition of *MinId*, contradiction.

Invariant 5:

Pick P such that $P.minlevel \leq d < P.minhilevel$, and $MinId(P,d) = Q.id \neq P.minid$ for some Q. If there is more than one such choice, we insist that d be minimized.

By Invariant 3, $Q.id < P.minid$. Pick $R \in \mathcal{N}_P$ such that $w(P,Q) = w(P,R) + w(R,Q)$ and $MinId(R, d - w(P,R)) = Q.id$. Since d was chosen to be minimum, Invariant 5 holds for R.

If $R.minid < Q.id$, then Invariant 4 fails.

If $R.minid > Q.id$ and $R.minhilevel \leq w(R,Q)$, then Invariant 1b fails.

Suppose $R.minid > Q.id$ and $R.minhilevel > w(R,Q)$. Then, by Property 1, and since Invariant 5 holds at R, $Q.id \geq MinId(R, w(R,Q) \geq MinId(R, minhilevel) - 1) = R.minid$, contradiction. □

Lemma 7 *Benchmark B5 is closed, and if Benchmark B4 holds, then Benchmark B5 will hold within $O(nk)$ additional rounds.*

Proof: We omit the proof of Lemma 7 since it is similar to the proof of Lemma 6. □

Lemma 8 *Benchmark B6 is closed, and will hold within $O(nk)$ rounds after Benchmark B5 holds.*

Proof: $P.isclusterhead$ is only changed when P executes Action A14. Once Benchmark B5 holds, by Lemma 5, P will execute A14 within $O(nk)$ rounds. At each such execution, $P.isclusterhead$ will be set to the correct value, since Maxminid Invariant 10 holds with $d = k$. □

Lemma 9 *Benchmark B7 is closed, and will hold within $k+1$ rounds after Benchmark B6 holds.*

Proof: We first note that, after B6 holds, $Cluster(P)$ will execute at least once during every round.

Claim: For any $0 \leq d \leq k$, within $d+1$ rounds after Benchmark B6 holds:

(a) $P.dist \geq \min\{Dist(P), d+1\}$, and

(b) if $Dist(P) \leq d$, then $P.dist = Dist(P)$.

We prove the claim by induction on d. First, note that $P.dist \geq 0$ by definition, which implies that $Dist\_F(P) > 0$ if $\neg P.isclusterhead$ If $d = 0$, then $Cluster(P)$ has executed at least once, which implies the claim.

Suppose $d > 0$. If $Dist(P) \leq d$, then after d rounds, by the inductive hypothesis, either P is a clusterhead or $Dist(P) = \min\{Q.dist + w(P,Q) : Q \in \mathcal{N}_P\}$, and we are done. If $Dist(P) > d$, then after d rounds, by the inductive hypothesis, $Q.dist + w(P,Q) > d$ for all $Q \in \mathcal{N}_P$, and we are done.

The lemma follows from the claim, by letting $d = k$. □

Theorem 1 *Starting from an arbitrary configuration,* Weighted-Clustering *stabilizes within $O(nk)$ rounds.*

6 Conclusion

In this article, we present a self-stabilizing asynchronous distributed algorithm for construction of a k-dominating set, and hence a k-clustering, for a given k, for any weighted network. In contrast with previous work, our algorithm deals with an arbitrary metric on the network. The algorithm executes in $O(nk)$ round, and requires $O(\log n + \log k)$ space per process.
In future work, we will attempt to improve the time complexity of the algorithm. We also plan to extend the results to the case of multiple metrics. For example, the case where there are weights on the links and also on the processes. We also intend to explore the possibility of using k-clustering to design efficient deployment algorithms for applications on a grid infrastructure.

7 Acknowledgment

This work was developed with financial support from the ANR (Agence Nationale de la Recherche) through the LEGO project referenced ANR-05-CIGC-11.

References

[1] A. D. Amis, R. Prakash, T. H.P. Vuong, and D. T. Huynh. Max-min d-cluster formation in wireless ad hoc networks. In *IEEE INFOCOM*, pages 32–41, 2000.

[2] Eddy Caron and Frédéric Desprez. Diet: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[3] A. K. Datta, L. L. Larmore, and P. Vemula. Self-stabilizing leader election in optimal space. In *10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Detroit, MI, nov. 2008. Currently available on `http://www.egr.unlv.edu/~larmore/Research/datLarVemSSLE.pdf`.

[4] A. K. Datta, L. L. Larmore, and P. Vemula. A self-stabilizing O(k)-time k-clustering algorithm. *To appear in Computer Journal*, 2008. Currently available on `http://www.egr.unlv.edu/~larmore/Research/datLarVemStCl.pdf`.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.

[6] S. Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

[7] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 1997.

[8] Y. Fernandess and D. Malkhi. K-clustering in wireless ad hoc networks. In *ACM Workshop on Principles of Mobile Computing POMC 2002*, pages 31–37, 2002.

[9] M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *IEEE Trans. Comput.*, 40(4):448–458, 1991.

[10] M.A. Spohn and J.J. Garcia-Luna-Aceves. Bounded-distance multi-clusterhead formation in wireless ad hoc networks. *Ad Hoc Networks*, 5:504–530, 2004.

[11] A. YarKhan, J. Dongarra, and K. Seymour. GridSolve: The Evolution of Network Enabled Solver. In James C. T. Pool Patrick Gaffney, editor, *Grid-Based Problem Solving Environments: IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments (Prescott, AZ, July 2006)*, pages 215–226. Springer, 2007.