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 (Revised Version)

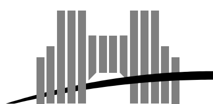
E. Caron[1,2], A. K. Datta[3], B. Depardon[1,2] and L. L. Larmore[3]

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

December 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
(Revised Version)

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

December 2008

Abstract

Mobile *ad hoc* networks as well as grid platforms are distributed, changing and error prone environments. Communication costs within such infrastructure can be improved, or at least bounded, by using *k-clustering*. A k-clustering of a graph, is a partition of the nodes into disjoint sets, called clusters, in which every node is distance at most k from a designated node in its cluster, called the *clusterhead*. A self-stabilizing asynchronous distributed algorithm is given for constructing 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. This is the first distributed solution to the k-clustering problem on weighted graphs.

Keywords: K-Clustering, Self-Stabilization, Weighted Graph.

Résumé

Les réseaux mobiles *ad hoc* ainsi que les plate-formes de grille sont des environnements distribués et sujets à de nombreuses erreurs. Les coûts de communication au sein de ses infrastructures peuvent être améliorés, ou tout au moins bornés par l'utilisation d'un *k-regroupement*. Un k-regroupement d'un graphe, est une partition des nœuds en ensembles disjoints, nommés grappes ou clusters, dans lesquels chaque nœud est à une distance au plus k d'un nœud élu au sein du cluster, appelé *clusterhead*. Nous présentons un algorithme asynchrone, distribué et auto-stabilisant pour construire un ensemble k-regroupement d'un réseau de nœuds ayant des identifiants uniques, et connectés par des arêtes pondérées. 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 par processus, où n est la taille du réseau. Nous présentons la première solution au problème du k-regroupement sur des graphes pondérés.

Mots-clés: K-Regroupement, Auto-Stabilisation, Graphe Pondéré.

1 Introduction

Nowadays distributed systems are built over a large number of resources. Overlay structures require taking into account locality among the entities they manage. For example, communication time between resources is the main performance metric in many systems. A cluster structure facilitates the spatial *reuse of resources* to increase system capacity. Clustering also helps routing and can improve the efficiency of a parallel software if it runs on a cluster of well connected resources. Another advantage of clustering is that many changes in the network can be made locally, *i.e.,* restricted to particular clusters.

Many applications require that entities are grouped into clusters according to a certain distance which measures proximity with respect to some relevant criterion; the clustering will result in clusters with similar or bounded readings. We are interested in two particular fields of research which can make use of resource clustering: mobile *ad hoc* networks (MANET) and application deployment on grid environments.

In MANET, scalability of large networks is a critical issue. Clustering can be used to design a low-hop backbone network in MANET with routing facilities provided by clustering. However, using only hops, *i.e.,* the number of links in the path between two processes, may hide the true communication time between two nodes.

A major aspect of grid computing is the deployment of grid middleware. The hop distance is used as a metric in some applications, but it may not be relevant in many platforms, such as a grid. Using an arbitrary metric (*i.e.,* a weighted metric) is a reasonable option in such heterogeneous distributed systems. Distributed grid middleware, like DIET [2] and GridSolve [13] can make use of accurate distance measurements to do efficient job scheduling.

Another important aspect is that both MANET and grid environments are highly dynamic systems: nodes can join and leave the platform anytime, and may be subject to errors. Thus, designing an efficient fault-tolerant algorithm which clusters the nodes according to a given distance k, and which can dynamically adapt to any change, is a necessity for many applications, including MANET and grid platforms.

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. Self-stabilization has been shown to be a powerful tool to design dynamic systems. As MANET and grid platforms are dynamic and error prone infrastructures, self-stabilization is a good approach to design efficient algorithms.

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. We also define the radius of a graph G as follows:

$$radius(G) = \min_{x \in V} \max_{y \in V} \{w(x, y)\}$$

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 specifies 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*[2] of G if, for every $x \in V$, there exists $y \in D$ such that $w(x,y) \leq k$. A k-dominating set determines a k-clustering in a simple 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 $y \in D$, $C_y = \{x : \mathit{Clusterhead}(x) = y\}$ 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].

1.2 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, 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 [12] 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 [12] are not given. Fernandess and Malkhi [8] give 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. However, this algorithm only deal with the hop metric, and is thus unable to deal with more general weighted graphs.

Other self-stabilizing clustering algorithms deal with “weighted graphs” [11], however the weights are placed on the vertices, not on the edges. Moreover, the algorithm does not compute a k-dominating set, but a dominating set (nodes are direct neighbors of the clusterheads).

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 first present a broad and intuitive explanation of the algorithm Weighted-Clustering before defining it more formally, and give its time and space complexity. We also show an

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

[2] Note that this definition of the k-*dominating set* is different than another well known problem consisting in finding a subset $V' \subseteq V$ such that $|V'| \leq k$, and such that $\forall v \in V - V', \exists y \in V' : (x,y) \in E$. [9]

example execution of Weighted-Clustering in Section 4. We also give the proofs of its correctness and complexity in Section 5. Finally, we present some simulation results in Section 6, before concluding the paper in Section 7.

2 Model and Self-Stabilization

We are given a connected undirected network of size $n \geq 2$, 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. 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 *shared memory/composite atomicity model* of computation [5, 6]. Each process can read its own registers and those of its neighbors, but can write only to its own registers; the evaluations of the guard and executions of the statement of any action is 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*. At a given step, if one or more processes are enabled, the daemon selects an arbitrary non-empty set of enabled processes to execute an action. The daemon is thus *unfair*: even if a process P is continuously enabled, P might never be selected by the daemon, unless, at some step, P is the only enabled process.
We say that a process P is *neutralized* during a step, if P is enabled before the step but not after the step, and does not execute any action during that step. This situation could occur if some neighbors of P change some of their registers in such a way as to cause the guards of all actions of P to become 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 one more 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 to $\mathbb{L}_{\mathcal{A}}$, or simply *self-stabilizing* if $\mathbb{L}_{\mathcal{A}}$ is understood, 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.

3.1 Overview of Weighted-Clustering.

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 nodes of each such tree form one k-cluster.
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:

$$
\begin{aligned}
\mathcal{N}_P &= \text{the set of all neighbors of } P \\
\mathcal{U}_P &= \mathcal{N}_P \cup \{P\} \\
\mathit{MinHop}(P) &= \min \begin{cases} \min\{w(P,Q) : Q \in \mathcal{N}_P\} \\ k+1 \end{cases} \\
\mathit{MinId}(P,d) &= \min\{Q.\mathit{id} : w(P,Q) \le d\} \\
\mathit{MaxMinId}(P,d) &= \max\{\mathit{MinId}(Q,k) : w(P,Q) \le d\} \\
\mathit{Clusterhead_Set} &= \{P : \mathit{MaxMinId}(P,k) = P.\mathit{id}\} \\
\mathit{Dist}(P) &= \min\{w(P,Q) : Q \in \mathit{Clusterhead_Set}\} \\
\mathit{Parent}(P) &= \begin{cases} P.\mathit{id} & \text{if } P \in \mathit{Clusterhead_Set} \\ \min\{Q.\mathit{id} : (Q \in \mathcal{N}_P) \wedge \\ \quad (\mathit{Dist}(Q) + w(P,Q) = \mathit{Dist}(P))\} & \text{otherwise} \end{cases} \\
\mathit{Clusterhead}(P) &= \begin{cases} P.\mathit{id} & \text{if } P \in \mathit{Clusterhead_Set} \\ \mathit{Clusterhead}(\mathit{Parent}(P)) & \text{otherwise} \end{cases}
\end{aligned}
$$

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, the output shared variables will eventually 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.

3.2 Structure of Weighted-Clustering : Combining Algorithms.

The formal definition of Weighted-Clustering requires 26 functions and 15 actions, and thus it is difficult to grasp the intuitive principles that guide it. We will first present a broad and intuitive explanation of how the algorithm works, before defining it more formally.
Weighted-Clustering consists of the following four phases.

Phase 1, Self-Stabilizing Leader Election (SSLE) We make use of an algorithm SSLE, defined in [3], which constructs a breadth-first-search (BFS) spanning tree rooted at the process of lowest ID, which we call *Root*_BFS. The BFS tree is defined by pointers *P.parent*_BFS for all P, and is used to synchronize the second and third phases of Weighted-Clustering. SSLE is self-stabilizing and silent. We do not give its details here, but instead refer the reader to [3].
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 phases of Weighted-Clustering.

Phase 2 and 3, 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 $\min\{f(Q) : w(P,Q) \leq k\}$ 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.

- ***Phase 2, MinId***, which computes, for each process P, *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 (after perhaps one unclean start) MinId begins from a clean state, and also to detect its termination. MinId is not silent; after computing all *MinId*(P,k), it resets and starts over.

- ***Phase 3, 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.

Phase 4, 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 has computed the correct values of *P.clusterhead* and *P.parent* for each P.

3.3 The BFS Spanning Tree Module SSLE.

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 $Root_{\text{BFS}}$, 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 $Root_{\text{BFS}}$.

- $P.parent_{\text{BFS}} = \begin{cases} P.id & \text{if } P = Root_{\text{BFS}} \\ \min\{Q.id : (Q \in \mathcal{N}_P) \wedge \\ \quad (Q.dist_{\text{BFS}} + 1) = P.dist_{\text{BFS}}\} & \text{otherwise} \end{cases}$

SSLE converges in $O(n)$ rounds from an arbitrary configuration, and remains silent, thus throughout the remainder of the execution of Weighted-Clustering, the BFS tree will not change.

3.4 Error Detection and Correction.

There are four colors, 0, 1, 2, and 3. The BFS tree supports the color waves; these scan the tree up and down, starting from the bottom of the tree for colors 0 and 2, and from the top for 1 and 3. The color waves have two purposes: they help the detection of inconsistencies within the variables, and allow synchronization of the different phases of the algorithm. Before the computation of clusterheads can be guaranteed correct, all possible errors in the processes must be corrected. Five kinds of errors are detected and corrected:

- Color errors: $P.color$ is 1 or 3, and the color of its parent in the BFS tree is not the same, then it is an error. Similarly, if the process' color is 2 and if its parent in the BFS tree has color 0, or if one of its children in the BFS tree has a color different than 2, then it is an error. Upon a color error detection, the color of the process is set to 0.

- Level errors: throughout the algorithm, P makes use of four variables which define a search interval, two for the MinId phase: $P.minlevel$ and $P.minhilevel$, and two for the MaxMinId phase: $P.maxminlevel$ and $P.maxminhilevel$. Depending on the color of the process, the values of $P.minlevel$ and $P.minhilevel$ must fulfill certain conditions, if they do not they are set to k and $k+1$, respectively; the variables $P.maxminlevel$ and $P.maxminhilevel$ are treated similarly. We do not give the details of the actions which accomplish these tasks; they are quite complex and rely on the color, the minimum (maximum) ID, and the levels of a process and of its neighbors.

- Initialization errors: P also makes use of the variables $P.minid$ and $P.maxminid$ to store the minimum ID found in the MinId phase, and the largest $MinId(Q, k)$ found in the MaxMinId phase. If the process has color 0, in order to correctly start the computation of the MinId phase, the values of $P.minid$, $P.minlevel$ and $P.minhilevel$ must be set respectively to $P.id$, 0 and $MinHop(P)$; if they do not have these values, they are corrected. The variables $P.maxminlevel$ and $P.maxminhilevel$ are treated similarly when $P.color = 2$, except that $P.maxminid$ is set to $P.minid$, in order to ensure that the MaxMinId phase starts correctly.

When all errors have been corrected, no process will ever return to an error state for as long as the algorithm runs without external interference.

3.5 Building Clusters

The heart of the algorithm is identification of the clusterheads, which consists of two phases, MinId and MaxMinId. We will describe only MinId in detail, as MaxMinId is similar.

MinId computes, for each process P, the smallest ID of any process which is within distance k of P. This phase starts when $P.color = 0$, and ends when $P.minhilevel = k + 1$. Three steps constitute this phase:

First step: Synchronization: a color wave starting from the root of the BFS tree sets the color of all processes to 1.

Second step: At each substep, process P defines a search interval, it gives lower and upper bounds on the distance d up to which P has looked to find the lowest ID: $P.minlevel \leq d < P.minhilevel$. The bounds can never decrease in each substep of the algorithm. Initially each process is only able to look no further than itself. Then, a process is able to update its *P.minid*, *P.minlevel* and *P.minhilevel* only when no neighbor prevents it from executing, *i.e.,* when P is included in the search interval of one of its neighbors Q, and P has a lower *minid* value than Q: a neighbor has not finished updating its variables according to its search interval. The levels are increased in accordance with the current levels and the *minid* of the neighbors: *P.minlevel* is set to the minimum $Q.minlevel + w(P, Q) \leq k$ such that $Q.minid < P.minid$, and *P.minhilevel* is set to the minimum of all $\min\{Q.minlevel + w(P, Q), k+1\}$ if $Q.minid < P.minid$, or $\min\{Q.minhilevel + w(P, Q), k + 1\}$.

Of course, a process cannot directly look at processes which are not its direct neighbors, but the evolution of the search intervals gives time for the information to gradually travel from process to process, thus by reading its neighbors variables, the process will eventually receive the values.

An example of the evolution of the search intervals in given Figure 1. For example, process E starts by looking at itself, then it looks at D, then for the next three steps it is able to look at D and L, and finally looks at B. It never looks at A, as the distance between E and A is greater than $k = 30$.

Third step: Once $P.minhilevel = k + 1$, another color wave starts at the bottom of the tree; this wave sets the color of all processes to 2. The processes are now ready to for the MaxMinId phase.

The MaxMinId phase is similar to the MinId phase, except that the colors used are 3 and 0. At the end of the MaxMinId phase, if $P.maxminid = P.id$, P is a clusterhead.

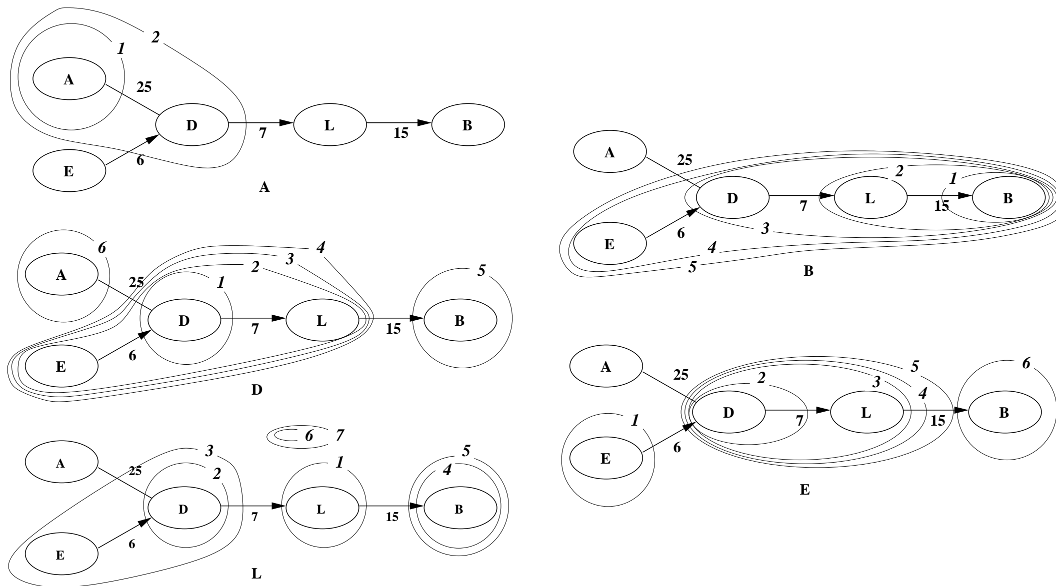


Figure 1: Evolution of the search interval $P.minlevel \leq d < P.minhilevel$ for the example computation Figure 2.

3.6 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.

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.

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 true, then it means that there is a problem in the color waves.
- $Min_Nbrs(P) = \{Q \in \mathcal{N}_P : (Q.minid < P.minid) \wedge (Q.minlevel + w(P,Q) \leq k)\}$
- $MinLevel_F(P) = \begin{cases} \min\{Q.minlevel + w(Q,P) : Q \in Min_Nbrs(P)\} & \text{if } Min_Nbrs(P) \neq \emptyset \\ P.minlevel & \text{otherwise} \end{cases}$
- $MinId_F(P) = \min\{Q.minid : (Q \in \mathcal{U}_P) \wedge (Q.minlevel + w(Q,P) = MinLevel_F(P))\}$

- $\mathit{MinKey_F}(P) = (\mathit{MinId_F}(P), \mathit{MinLevel_F}(P))$
- $\mathit{MinHiLevel_F}(P,Q) = \begin{cases} \min\{k+1, Q.\mathit{minlevel} + w(P,Q)\} & \text{if } Q.\mathit{minid} < P.\mathit{minid} \\ \min\{k+1, Q.\mathit{minhilevel} + w(P,Q)\} & \text{otherwise} \end{cases}$
- $\mathit{MinHiLevel_F}(P) = \min\{\mathit{MinHiLevel_F}(P,Q) : Q \in \mathcal{N}_P\}$

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

  The meaning of the predicate $P \xrightarrow{\min} Q$ is that Q prevents P from executing Action A7, the Minid update action.
- $\mathit{MinLevel_Valid}(P) \equiv (P.\mathit{minlevel} < P.\mathit{minhilevel} \leq k+1) \wedge$
  $(\forall Q \in \mathcal{N}_P : P.\mathit{minhilevel} + w(P,Q) \geq Q.\mathit{minhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : Q.\mathit{minid} > P.\mathit{minid} \implies P.\mathit{minlevel} + w(P,Q) \geq Q.\mathit{minhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : Q.\mathit{minid} = P.\mathit{minid} \implies P.\mathit{minlevel} + w(P,Q) \geq Q.\mathit{minlevel})$
- $\mathit{MinLevel_Error}(P) \equiv$
  $\big((P.\mathit{color} = 1) \wedge \neg \mathit{MinLevelValid}(P)\big) \vee \big((P.\mathit{color} = 2) \wedge (P.\mathit{minhilevel} \neq k+1)\big)$
- $\mathit{MinInit_Error}(P) \equiv (P.\mathit{color} = 0) \wedge$
  $\big((P.\mathit{minid} \neq P.\mathit{id}) \vee (P.\mathit{minlevel} \neq 0) \vee (P.\mathit{minhilevel} \neq \mathit{MinHop}(P))\big)$
- $\mathit{MaxMin_Nbrs}(P) =$
  $\{Q \in \mathcal{N}_P : (Q.\mathit{maxminid} > P.\mathit{maxminid}) \wedge (Q.\mathit{maxminlevel} + w(P,Q) \leq k)\}$
- $\mathit{MaxMinLevel_F}(P) =$
  $\begin{cases} \min\{Q.\mathit{maxminlevel} + w(Q,P) : Q \in \mathit{MaxMin_Nbrs}(P)\} & \text{if } \mathit{MaxMin_Nbrs}(P) \neq \emptyset \\ P.\mathit{maxminlevel} & \text{otherwise} \end{cases}$
- $\mathit{MaxMinId_F}(P) =$
  $\max\{Q.\mathit{maxminid} : (Q \in \mathcal{U}_P) \wedge (Q.\mathit{maxminlevel} + w(Q,P) = \mathit{MaxMinLevel_F}(P))\}$
- $\mathit{MaxMinKey_F}(P) = (\mathit{MaxMinId_F}(P), \mathit{MaxMinLevel_F}(P))$
- $\mathit{MaxMinHiLevel_F}(P,Q) = \begin{cases} \min\{k+1, Q.\mathit{maxminlevel} + w(P,Q)\} & \text{if } Q.\mathit{maxminid} > P.\mathit{maxminid} \\ \min\{k+1, Q.\mathit{maxminhilevel} + w(P,Q)\} & \text{otherwise} \end{cases}$
- $\mathit{MaxMinHiLevel_F}(P) = \min\{\mathit{MaxMinHiLevel_F}(P,Q) : Q \in \mathcal{N}_P\}$

  Defines the upper bound on the research interval to find the maximum ID.
- $\mathit{MaxMinLevel_Valid}(P) \equiv (P.\mathit{maxminlevel} < P.\mathit{maxminhilevel} \leq k+1) \wedge$
  $(\forall Q \in \mathcal{N}_P : P.\mathit{maxminhilevel} + w(P,Q) \geq Q.\mathit{maxminhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : Q.\mathit{maxminid} < P.\mathit{maxminid} \implies P.\mathit{maxminlevel} + w(P,Q) \geq Q.\mathit{maxminhilevel}) \wedge$
  $(\forall Q \in \mathcal{N}_P : Q.\mathit{maxminid} = P.\mathit{maxminid} \implies P.\mathit{maxminlevel} + w(P,Q) \geq Q.\mathit{maxminlevel})$

- $\mathit{MaxMinLevel_Error}(P) \equiv$
  $\big((P.\mathit{color} = 3) \land \neg \mathit{MaxMinLevelValid}(P)\big) \lor \big((P.\mathit{color} = 0) \land (P.\mathit{maxminhilevel} \neq k+1)\big)$
- $\mathit{MaxMinInit_Error}(P) \equiv (P.\mathit{color} = 2) \land$
  $\big((P.\mathit{maxminid} \neq P.\mathit{minid}) \lor (P.\mathit{maxminlevel} \neq 0) \lor (P.\mathit{maxminhilevel} \neq \mathit{MinHop}(P))\big)$
- $P \stackrel{\max\min}{\longrightarrow} Q \equiv$
  $\big((Q \in \mathcal{N}_P) \land (P.\mathit{maxminid} > Q.\mathit{maxminid}) \land (Q.\mathit{maxminlevel} < P.\mathit{maxminlevel} + w(P,Q) \leq k)\big) \lor$
  $\big((Q.\mathit{maxminhilevel} + w(P,Q) = P.\mathit{maxminhilevel})\big)$

  The meaning of the predicate $P \stackrel{\max\min}{\longrightarrow} Q$ is that Q prevents P from executing Action A9, the Maxminid update action.
- $\mathit{IsClusterhead_F}(P) \equiv P.\mathit{maxminid} = P.\mathit{id}$, of Boolean type.
- $\mathit{Dist_F}(P) = \begin{cases} 0 & \text{if } P.\mathit{isclusterhead} \\ \min\{k+1, \min\{Q.\mathit{dist} + w(P,Q) : Q \in \mathcal{N}_P\}\} & \text{otherwise} \end{cases}$
- $\mathit{Parent_F}(P) = \begin{cases} P.\mathit{id} & \text{if } P.\mathit{isclusterhead} \\ \min\{Q.\mathit{id} : (Q \in \mathcal{N}_P) \land & \text{if } \exists (Q \in \mathcal{N}_P) \\ \quad (Q.\mathit{dist} + w(P,Q) = \mathit{Dist_F}(P))\} & \quad : (Q.\mathit{dist} + w(P,Q) = \mathit{Dist_F}(P)) \\ P.\mathit{id} & \text{otherwise} \end{cases}$
- $\mathit{Clusterhead_F}(P) = \begin{cases} P.\mathit{id} & \text{if } P.\mathit{isclusterhead} \\ P.\mathit{parent.clusterhead} & \text{otherwise} \end{cases}$

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.
$\mathit{Cluster}(P)$:
if $(P.\mathit{dist} \neq \mathit{Dist_F}(P))) \lor (P.\mathit{parent} \neq \mathit{Parent_F}(P)) \lor (P.\mathit{clusterhead} \neq \mathit{Clusterhead_F}(P))$
  $P.\mathit{dist} \leftarrow \mathit{Dist_F}(P)$
  $P.\mathit{parent} \leftarrow \mathit{Parent_F}(P)$
  $P.\mathit{clusterhead} \leftarrow \mathit{Clusterhead_F}(P)$
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,

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	$MinLevel_Error(P)$	$\longrightarrow$	$P.minlevel \leftarrow k$ $P.minhilevel \leftarrow k+1$
A4 priority 3	Maxmin Level Error	$MaxMinLevel_Error(P)$	$\longrightarrow$	$P.maxminlevel \leftarrow k$ $P.maxminhilevel \leftarrow k+1$
A5 priority 3	Min Init Error	$MinInit_Error(P)$	$\longrightarrow$	$P.minid \leftarrow P.id$ $P.minlevel \leftarrow 0$ $P.minhilevel \leftarrow MinHop(P)$
A6 priority 3	Maxmin Init Error	$MaxMinInit_Error(P)$	$\longrightarrow$	$P.maxminid \leftarrow P.minid$ $P.maxminlevel \leftarrow 0$ $P.maxminhilevel \leftarrow MinHop(P)$
A7 priority 4	Minid Update	$P.color = 1$ $MinHiLevel_F(P) = P.minhilevel \leq k$ $MinLevel_F(P) = P.minhilevel$ $\neg\exists Q \in \mathcal{N}_P : P \overset{\min}{\longrightarrow} Q$	$\longrightarrow$	$Cluster(P)$ $P.minkey \leftarrow MinKey_F(P)$ $P.minhilevel \leftarrow MinHiLevel_F(P)$
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	$P.color = 3$ $MaxMinHiLevel_F(P) = P.maxminhilevel \leq k$ $MaxMinLevel_F(P) = P.maxminhilevel$ $\neg\exists Q \in \mathcal{N}_P : P \overset{\max\min}{\longrightarrow} Q$	$\longrightarrow$	$Cluster(P)$ $P.maxminkey \leftarrow MaxMinKey_F(P)$ $P.maxminhilevel \leftarrow MaxMinHiLevel_F(P)$
A10 priority 4	Maxmin Hi Level	$P.color = 3$ $P.maxminhilevel < MaxMinHiLevel_F(P)$	$\longrightarrow$	$Cluster(P)$ $P.maxminhilevel \leftarrow MaxMinHiLevel_F(P)$
A11 priority 5	Color 1	$P.color = 0$ $(P = Root_\mathrm{BFS}) \vee (P.parent_\mathrm{BFS}.color = 1)$	$\longrightarrow$	$Cluster(P)$ $P.color \leftarrow 1$
A12 priority 5	Color 2	$P.color = 1$ $P.minhilevel = k+1$ $\forall Q \in Chldrn_\mathrm{BFS}(P) : P.color = 2$ $\forall Q \in \mathcal{N}_P : Q.minhilevel = k+1$	$\longrightarrow$	$Cluster(P)$ $P.color \leftarrow 2$ $P.maxminid \leftarrow P.minid$ $P.maxminlevel \leftarrow 0$ $P.maxminhilevel \leftarrow MinHop(P)$
A13 priority 5	Color 3	$P.color = 2$ $(P = Root_\mathrm{BFS}) \vee (P.parent_\mathrm{BFS}.color = 3)$	$\longrightarrow$	$Cluster(P)$ $P.color \leftarrow 3$
A14 priority 5	Color 0	$P.color = 3$ $P.maxminhilevel = k+1$ $\forall Q \in Chldrn_\mathrm{BFS}(P) : P.color = 0$ $\forall Q \in \mathcal{N}_P : Q.maxminhilevel = k+1$	$\longrightarrow$	$Cluster(P)$ $P.color \leftarrow 0$ $P.minid \leftarrow P.id$ $P.minlevel \leftarrow 0$ $P.minhilevel \leftarrow MinHop(P)$ $P.isclusterhead \leftarrow IsClusterhead_F(P)$
A15 priority 6	Clustering		$\longrightarrow$	$Cluster(P)$

Table 2: Actions of Weighted-Clustering.

3.7 Time and Space Complexity.

The algorithm uses all the variables of SSLE [3] and 11 other variables in each process. SSLE uses $O(\log n)$ space. The internal ID variables can be encoded on $O(\log n)$ space, distance in $O(\log k)$ space, and colors in only 2 bits. Hence, Weighted-Clustering requires $O(\log n + \log k)$ memory per process.
SSLE converges in $O(n)$ rounds, while the clustering module requires $O(n)$ rounds once *Clusterhead_Set* has been correctly computed. MinId and MaxMinId are the most time-consuming, requiring $O(nk)$ rounds each to converge. The total time complexity of the Weighted-Clustering is thus $O(nk)$.

4 An Example Computation

4.1 A toy example

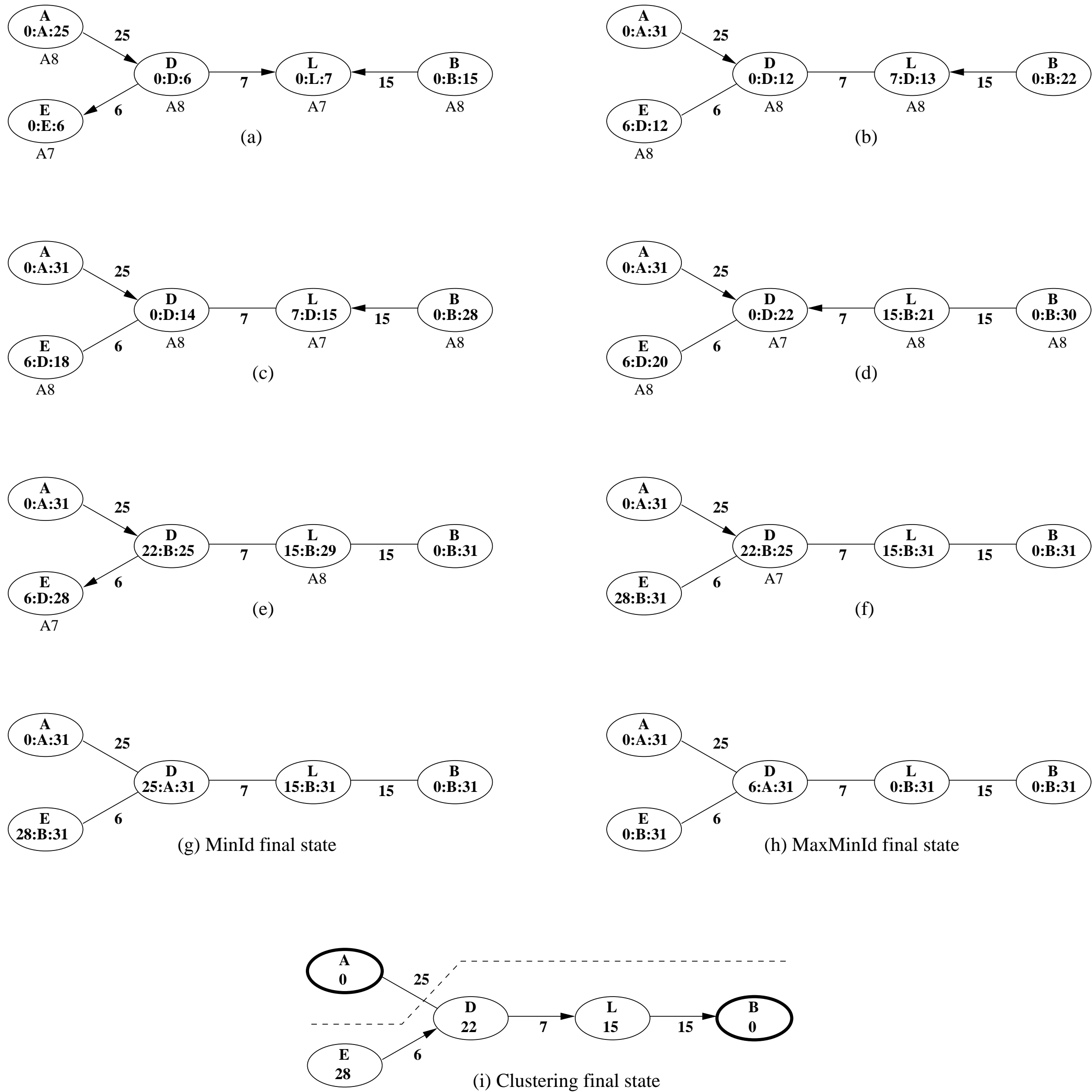


Figure 2: Example computation of Weighted-Clustering for $k = 30$.

In Figure 2, we show an example where $k = 30$. In that figure, each oval represents a process P and the numbers on the lines between the ovals represent the weights of the links. To help distinguish IDs from distances, we use letters for IDs. The top letter in the oval representing a process P 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*. Below each oval is shown the action the process is enabled to execute (none if the process is disabled). An arrow from P to Q indicates that $P \xrightarrow{\min} Q$, thus Q prevents P from executing Action A7. 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*, and a bold oval means that the process is a clusterhead. The dashed line represents the separation between the two final k-clusters.

In Figure 2(a) to (g), we show synchronous execution of the *MinId* phase. The result would have been the same with an asynchronous execution, but using synchrony makes the example easier to understand.

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. At any step, two neighbors cannot both execute Action A7 due to the $\xrightarrow{\min}$ 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 have done 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 that a process P was elected a clusterhead to flow back to P.

4.2 Detailed explanation of the example

We give in this section a few more details to better understand the example given in Figure 2.

We first define for any two processes P and Q the following:

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

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

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

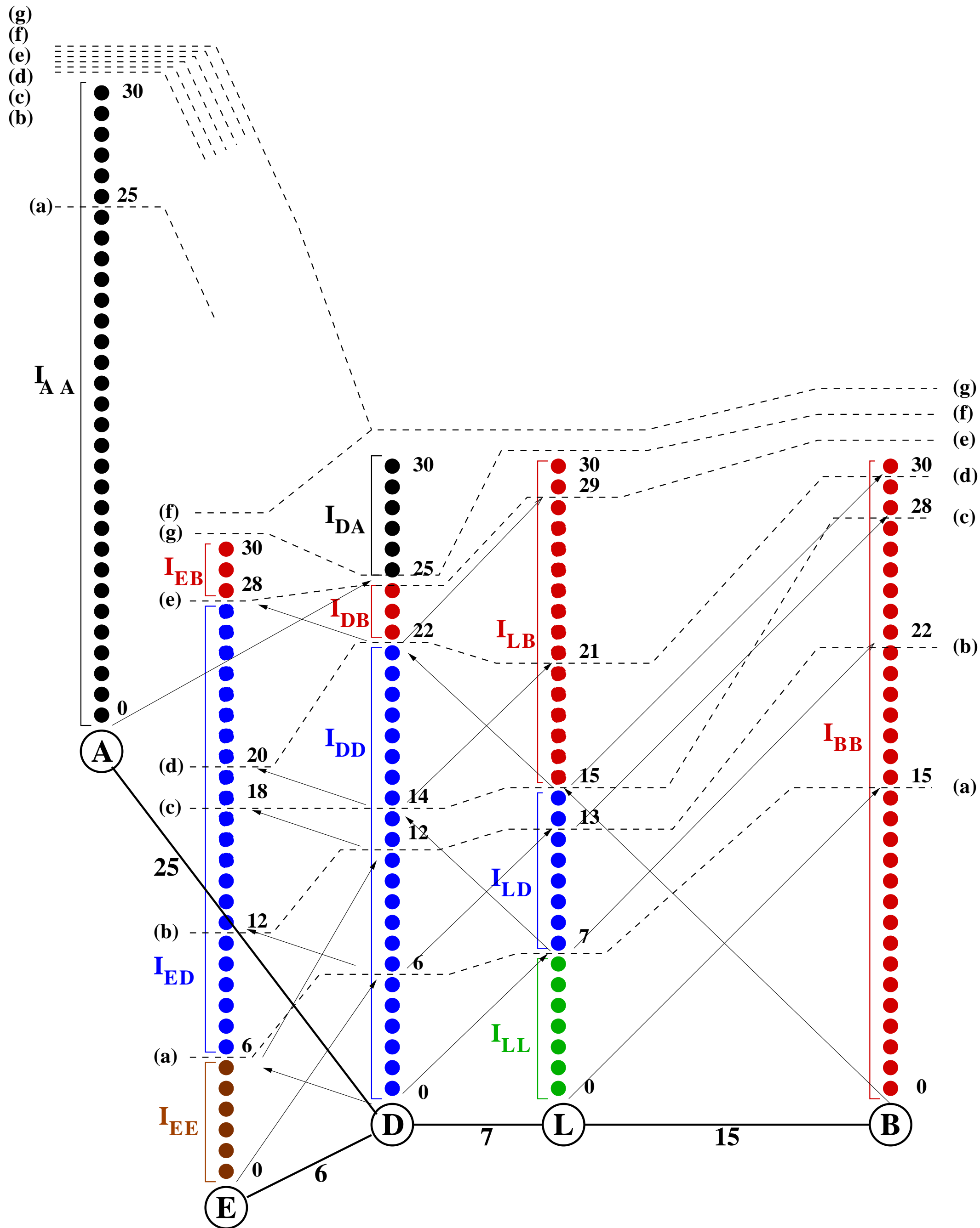


Figure 3: Growth of the Function *minhilevel*.

As an example, consider the network shown in Figure 2, where the names of the processes are A, B, D, E, L. We then have the following intervals:

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

Explanation of Figure 3. 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 2. 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 $\mathit{MinId}(P, d)$ have been computed for all d which are below the dashed path above P in Figure 3.

At any given point in the computation, $P.\mathit{minlevel}$ is the left (low) end of the interval $\mathcal{I}_{P,Q}$, namely $\mathit{minlo}_{P,Q}$, where $Q.\mathit{id} = P.\mathit{minid}$. At the same time, the value of $P.\mathit{minhilevel}$ is an estimate of $\mathit{minhi}_{P,Q}$, 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 $P.\mathit{minhilevel}$, improving its estimate of the high end of the interval, $\mathit{minhi}_{P,Q}$. If P executes Action A7 (MinId), it changes to the next interval, setting the new value of $P.\mathit{minlevel}$ to the old value of $P.\mathit{minhilevel}$, and computing a new, higher value of $P.\mathit{minhilevel}$.

At every point in the computation, P raises $P.\mathit{minhilevel}$ 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 $\mathit{minhi}_{P,Q}$.

Example Consider the process L in the example. Initially, as shown in Figure 2(a), $L.\mathit{minid} = L$, $L.\mathit{minlevel} = 0$, and $L.\mathit{minhilevel} = 7$, since the nearest neighbor is at distance 7.

In Figure 2(b), $L.\mathit{minid} \leftarrow D$, $L.\mathit{minlevel} \leftarrow 7$, and $L.\mathit{minhilevel} \leftarrow 13$. At the end of this step $L.\mathit{minhilevel}$ is underestimated, thus in the next step (c), L is enabled to execute A8 and correct its value $L.\mathit{minhilevel}$ to 15.

At step (d), $L.\mathit{minid} \leftarrow B$, $L.\mathit{minlevel} \leftarrow 15$, and $L.\mathit{minhilevel} \leftarrow 21$.

At step (e), $L.\mathit{minhilevel} \leftarrow 21$. 21 is an underestimate of $\mathit{minhi}_{L,B} = 29$.

At step (f), $L.\mathit{minhilevel}$ 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 $E.\mathit{minhilevel} \leftarrow D.\mathit{minhilevel} + w(E, D) = 12 + 6$ at step (c).

5 Proofs

5.1 Overview of Proof

We will now give an overview of the sketch of the proof of the correctness and time complexity of the algorithm.

We use the *convergence stair* method of proof [10]. 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*$(P, d_1) \geq$ *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 MinId and MaxMinId Invariants

To prove the correctness of Weighted-Clustering, 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 $\mathit{MinId}(P, w(P,Q)) = Q.id$, $R \in \mathcal{N}_P$, 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 $P.\mathit{maxminid} = Q.id$.
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 $R.\mathit{maxminid} \leq Q.\mathit{minid}$.
10. If $P.\mathit{maxminlevel} \leq d < P.\mathit{maxminhilevel}$ then $\mathit{MaxMinId}(P,d) = P.\mathit{maxminid}$.

5.4 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 $P.\mathit{color} = 0$, then
   (a) $P.\mathit{maxminhilevel} = k + 1$
   (b) $P.\mathit{minid} = P.\mathit{id}$
   (c) $P.\mathit{minlevel} = 0$
   (d) $P.\mathit{minhilevel} = \mathit{MinHop}(P)$
2. If $P.\mathit{color} = 2$, then
   (a) $P.\mathit{minhilevel} = k + 1$

(b) $P.maxminid = P.minid$

(c) $P.maxminlevel = 0$

(d) $P.maxminhilevel = MinHop(P)$

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

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

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, $Root_{\text{BFS}}.color = 1$, $P.color \in \{1, 2\}$ 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: $MinLevel_Valid(P) \wedge (P.minhilevel < MinHiLevel_F(P))$. 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 $\neg Minlevel_Valid(P)$, 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: $\left(\neg\exists Q : P \stackrel{\min}{\longrightarrow} Q\right) \wedge \mathit{MinLevel_Valid}(P) \wedge (P.\mathit{minhilevel} = \mathit{MinHiLevel_F}(P))$. Then P can execute Action A7. □

Lemma 5 *If Benchmark B3 holds, then*

(a) *If* $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 0$*, then within one round,* $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 1$.

(b) *If* $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 1$*, then within* $nk + 2\mathit{diam}$ *rounds,* $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 2$.

(c) *If* $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 2$*, then within one round,* $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 3$.

(d) *If* $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 3$*, then within* $nk + 2\mathit{diam}$ *rounds,* $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 0$.

Proof: (a): If $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 0$, then $\mathit{Root}_\mathrm{BFS}$ 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, within *diam* steps, all processes will have color either 1 or 2.
At this time, $\sum P.\mathit{minhilevel} \geq n$. By Lemma 4, during each round, $\sum P.\mathit{minhilevel}$ 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 $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 2$, then $\mathit{Root}_\mathrm{BFS}$ is enabled to execute Action A13, and will do so within one round.

(d) is similar to (b). □

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, $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 3$ 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: $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 3$ at γ'. This case is trivial.

Case II: $\mathit{Root}_\mathrm{BFS}.\mathit{color} = 0$ 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 $P.\mathit{minhilevel}$ cannot decrease. If Q executes Action A7 or A8, then $Q.\mathit{minhilevel} \leftarrow \mathit{MinHiLevel_F}(Q) \leq P.\mathit{minhilevel} + w(P,Q)$.

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

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

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 $P.\mathit{minid}$ is copied from $R.\mathit{minid}$ for some neighbor process R.

Invariant 3:

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

Invariant 4:

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

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

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

Invariant 5:

Pick P such that $P.\mathit{minlevel} \leq d < P.\mathit{minhilevel}$, and $\mathit{MinId}(P,d) = Q.\mathit{id} \neq P.\mathit{minid}$ 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 $d = k$. □

5.5 Complexity proofs

5.5.1 Upper Bounds

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

Proof: During one phase, the value of $P.minhilevel$ cannot decrease for any P. Thus, the number of times A7 and A8 together execute cannot exceed k for any P, and thus A7 and A8 together execute during a total of $O(nk)$ rounds.
We now need to show that at least one of the actions A7 and A8 must execute during every round. (More details are needed.) □

5.5.2 Lower Bounds

Figure 4: The Graph $G_{n,k}$

In Figure 4, we assume that n is even. The network is a chain, *i.e.*,, there are edges between P_i and P_j if and only if $|i - j| = 1$. Edge weights are given as follows:

$$||P_i, P_{i+1}|| = \begin{cases} 1 & \text{if } i \text{ is odd} \\ k & \text{if } i \text{ is even} \end{cases}$$

Lemma 10 *If the algorithm runs on graph $G_{n,k}$, the convergence time is $\Theta(nk)$ rounds in the worst case.*

Proof:
For sake of simplicity, we suppose that the processes start in a clean state, *i.e.,* all possible errors have been corrected, and $P.minid = P.id$, $P.minlevel = 0$ and $P.minhilevel = MinHop(P)$, note that in this graph for all Process P, $MinHop(P) = 1$.
Initially, only Process n is able to execute Action A7 due to the $P \overset{\min}{\longrightarrow} Q$ function present in the guard. Concurrently, all *odd* processes are enabled to execute Action A8 to update their $P.minhilevel$ variable to 2. Thus, after one round, $n.minid = n - 1$, $n.minlevel = 0$ and $n.minhilevel = 2$; and for all odd processes P, $P.minhilevel = 2$.
During the next $k-1$ steps, only Processes n and $n-1$ are enabled to alternatively execute Action A8 to update their *minhilevel* variable. $n.minhilevel$ and $(n-1).minhilevel$ are only able to increase by 2 at each step.
Once $(n-1).minhilevel = k+1$, $n-1$ is enabled to execute Action A7, and set $(n-1).minid = n-2$. Then, Process $n-2$ is enabled to execute Action A7, which starts a new cycle of $k-1$ rounds between $n-2$ and $n-3$ to update their *minhilevel*.
These update cycles are repeated until a cycle reaches process 2, which is the last cycle between 1 and 2: the processes can only be updated following a descending order on their IDs.
Overall, it requires $(1 + (k - 1)) \times n/2 = kn/2$ rounds to complete the MinId phase. Hence the $\Theta(nk)$ bound.

□

The Random Graph $R_{n,k}$. Assuming n is even, we construct the graph $R_{n,k}$ as follows.

- The nodes of $R_{n,k}$ are the integers $\{1, \dots n\}$.
- Randomly partition the processes into pairs, which we call *special pairs*, in such a manner that all such partitions are equally likely. If $\{i, j\}$ is a special pair, we write *partner*$(i) = j$ and *partner*$(j) = i$. We say that i is *superior* if *partner*$(i) < i$; otherwise we say that i is *inferior*.

- $R_{n,k}$ is complete.
- For any nodes i and $j \neq i$, the weight of the edge between i and j is 1 if $j = partner(i)$, and k otherwise.

Figure 5 presents an example of such a graph.

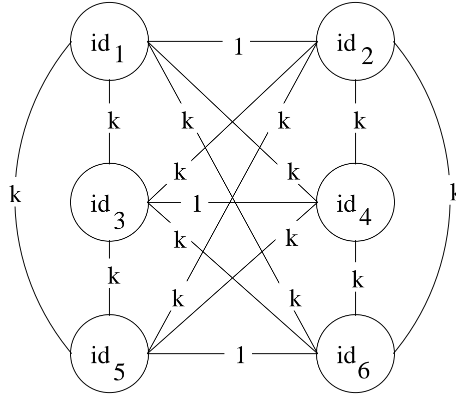


Figure 5: The Graph $R_{n,k}$.

Lemma 11 *Let a and b be integers such that $1 \leq a < b \leq n$, then the probability that there exists a special pair in the interval $\{a \dots b\}$ is at least*

$$1 - e^{-\frac{d(d-1)}{2n}}$$

where $d = b - a + 1$, the cardinality of that interval.

Proof: If $d > n/2$, then there must be a special pair in the interval, and we are done.

Suppose $d \leq n/2$. Without loss of generality, $a = 1$ and $b = d$. For any $1 \leq i \leq d$, let E_i be the event that *partner*$(i) > d$. Then $E_1 \cap \dots \cap E_d$ is the event that there is no special pair in the interval. We now compute $Pr(E_i \mid E_1 \cap \dots \cap E_{i-1})$, the conditional probability that *partner*$(i) > d$, given that *partner*$(j) > d$ for all $1 \leq j < i$. By the uniformity of the choice of partitions, *partner*(i) is equally likely to be any of the available nodes. Since i is unavailable, and both j and *partner*(j) are unavailable for all $1 \leq j < i - 1$, there are $n - (2(i-1)+1)$ nodes that are available. Since *partner*$(j) > d$ for all $1 \leq j < i - 1$, there are $n - d - (i-1)$ available nodes which are not in the interval. Thus

$$Pr(E_i \mid E_1 \cap \dots \cap E_{i-1}) = \frac{n - d - i + 1}{n - 2i + 1}$$

By the usual formula for the probability of the intersection of events, as stated in Remark A.1, the probability that no special pair exists in the interval is

$$\begin{aligned}\prod_{i=1}^{d}\frac{n-d-i+1}{n-2i+1} &\leq \prod_{i=1}^{d}\left(1-\frac{d-i}{n}\right)\\ &\leq \prod_{i=1}^{d} e^{-\frac{d-i}{n}}\\ &= e^{-\frac{d(d-1)}{2n}}\end{aligned}$$

We use Lemmas A.2 and A.3 for the first and second inequalities above.

□

We say special pairs $\{i,j\}$ and $\{k,\ell\}$ are *disjoint* if $\max\{i,j\} < \min\{k,\ell\}$ or $\max\{k,\ell\} < \min\{i,j\}$.
Let $M_{r,k}$ be the largest cardinality of any set of mutually disjoint special pairs of $R_{n,k}$. Note that $M_{r,k}$ is a random variable.

Lemma 12 *If $k \geq 2$, the expected value of $M_{r,k}$ is $\Omega(\sqrt{n})$.*

Proof: Let $d = \lceil 2\sqrt{n}\rceil + 1$. Partition $\{1,\ldots n\}$ into $\left\lfloor \frac{n}{d}\right\rfloor$. intervals each of length at least d, where $q = \left\lfloor \frac{n}{d}\right\rfloor$. By Lemma 11, the probability that a given interval contains a special pair is at least $1 - e^{-2}$. The expected number of those intervals that contain at least one special pair is at least $\left\lfloor \frac{n}{d}\right\rfloor (1-e^{-2}) = \Theta(\sqrt{n})$. Thus $m = \Omega(\sqrt{n})$. □

Lemma 13 *If $k \geq 2$, The expected time complexity of the algorithm on $R_{n,k}$ is $\Omega(\sqrt{n}k)$.*

Proof: Assume that we have picked, at random, a specific instance R of $R_{n,k}$. Let m be the greatest cardinality of any set of mutually disjoint special pairs in R.

Overview. We shall define a partial execution

$$\gamma_0 \mapsto \gamma_1 \mapsto \ldots \mapsto \gamma_T$$

for $T \geq mk$, such that each round takes just one step, and the algorithm has not yet stabilized at γ_T. Since, from Lemma 12, we know that the expected value of m is $\Omega(\sqrt{n})$, we will be done.

Definition of γ_0. The configuration γ_0 is defined as follows. The variables of SSLE have reached their stable values, and for all $1 \leq i \leq n$:

- $i.\mathit{color} = 1$.
- $i.\mathit{minid} = i$.
- $i.\mathit{minlevel} = 0$.
- $i.\mathit{minhilevel} = \begin{cases} 2 & \text{if } i \text{ is inferior} \\ 1 & \text{if } i \text{ is superior}\end{cases}$

The remaining variables actually play no role in the proof, but we must make sure that they cause no error. We can set them as follows:

- $i.maxminid = i$.
- $i.maxminlevel = 0$.
- $i.maxminhilevel = 1$.
- $i.dist = 0$.
- $i.parent = i$.

The remaining configurations are defined inductively: for $t > 0$, γ_t is the configuration obtained if every process enabled at γ_{t-1} is selected. We then define γ_T to be the first configuration in the sequence where $i.minhilevel = k+1$ for all i. Observe that the only actions that can take place during this sequence are A7, A8, and A15 (although we are not concerned with A15) and thus $i.minlevel$ and $i.minhilevel$ cannot decrease.

For any $1 \leq i \leq n$ and $1 \leq \ell \leq k+1$, define

$$t_{i,\ell} = \min \{0 \leq t \leq T : i.minhilevel \geq \ell \text{ at } \gamma_t\}$$

Claim I: If $j = partner(i)$ and $2 \leq \ell \leq k$, then $t_{i,\ell} < t_{j,\ell+1}$.

Proof of Claim I: Since $w(i,j) = 1$ and the Minid invariants hold during the sequence, $j.minhilevel$ cannot exceed ℓ until j sees that $i.minhilevel$ has exceeded $\ell - 1$.

Claim II: If j is inferior, $j < i$, and $j < partner(i)$, then $t_{i,k+1} < t_{j,2}$.

Proof of Claim II: Note that $j.minid \leftarrow partner(j)$ at step $t_{j,2}$, by executing Action A7, and this is the first execution of A7 by j during the sequence.

If $t < t_{i,k+1}$, then $i.minlevel < k$, and hence $i.minid \in \{i, partner(i)\}$ at γ_t. Thus $j \xrightarrow{\min} i$ at γ_t. It follows that j is not enabled to execute Action A7 during the first $t_{i,k+1}$ steps.

Completing the proof, we pick $i_1 < j_1 < i_2 < j_2 < i_3 < \ldots < j_{m-1} < i_m < j_m$, where each $\{i_s, j_s\}$ is a special pair.

Applying Claim I (k-1) times, we have $\max\{t_{i_s,k+1}, t_{j_s,k+1}\} \geq t_{j_s,2}$ for all s.

By Claim II, $t_{j_s,2} > \max\left\{t_{i_{s+1},k+1}, t_{j_{s+1},k+1}\right\}$ if $s < m$.

Since $t_{j_m,2} \geq 1$, by induction on s, we have $T \geq \max\{t_{i_1,k+1}, t_{i_1,k+1}\} \geq mk$, and we are done. □

6 Simulations

We designed a simulator to evaluate the performance of our algorithm. In order to verify the results, a sequential version of the algorithm was run, and all simulation results compared to the sequential version results. Thus, we made sure that the returned clustered graph was the correct one. In order to detect when the algorithm becomes stable and has computed the correct clustering, we compared, at each step, the current graph with the previous one; the result was then output only if there was a difference. The stable result is the last graph output once the algorithm has reached an upper bound on the number of rounds (we set this number at least two orders of magnitude higher than the convergence time of the algorithm).

6.1 Effect of the k value

We ran the simulator on the weighted graph illustrated in Figure 6. For each value of k, we ran 10 simulations starting from an arbitrary initial state where the value of each variable of each process was randomly chosen. Each process had a specific computing power so that they could not execute all at the same speed; we set the ratio between the slowest and the fastest process to $1/100$.
Figure 7 shows the number of clusterheads found for each run and each value of k. As the algorithm returns exactly the same set of clusterheads whatever the initial condition, the results for a given k are all the same. Note that the number of clusterheads decreases as k increases, and even if the algorithm may not find the optimal solution, it gives a clustering far better than a naive $O(1)$ self-stabilizing algorithm which would consists in electing each process a clusterhead. The figure shows that the number of clusterheads quickly decreases as k increases.
Figure 8 shows the number of rounds required to converge. This figure shows two kinds of runs: with an unfair daemon and different computing speed, and with a fair daemon and identical computing speed for all processes. As can be seen, the number of rounds is far lower than the theoretical bound $O(nk)$, even with an unfair daemon.

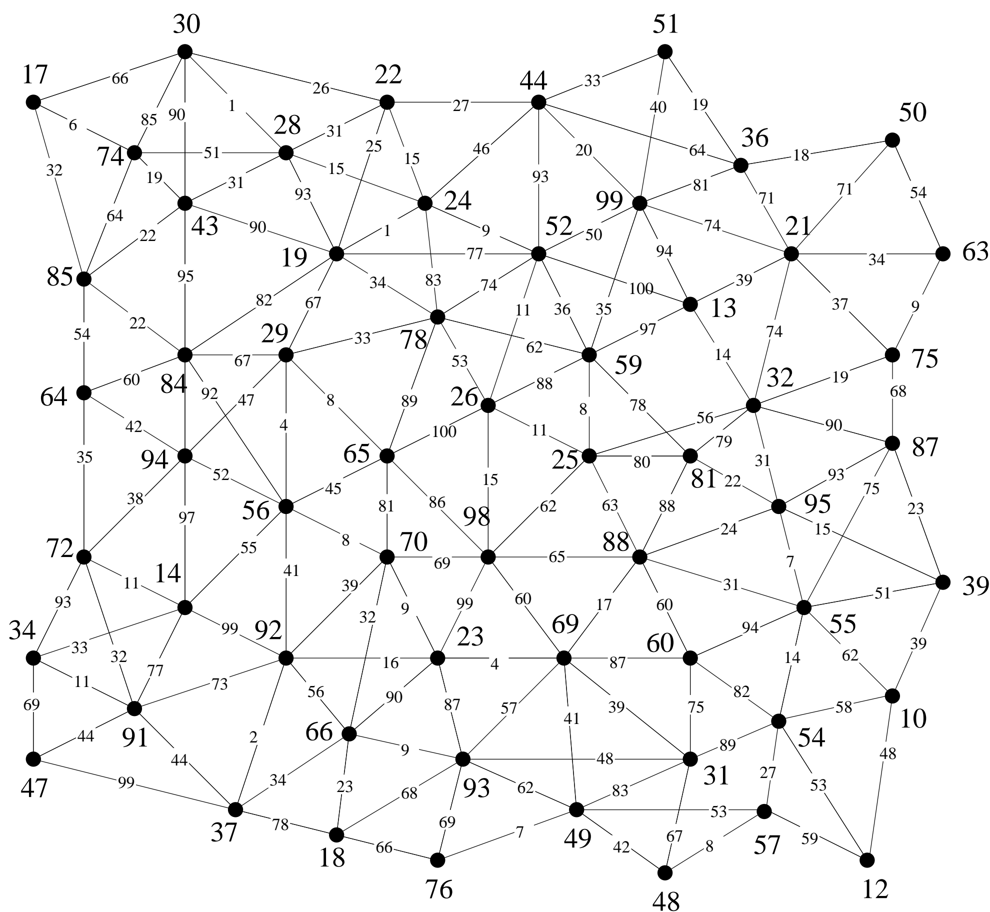


Figure 6: Example graph: $number\ of\ nodes = 59$, $diameter = 282$ (9 hops), weights between 1 and 100.

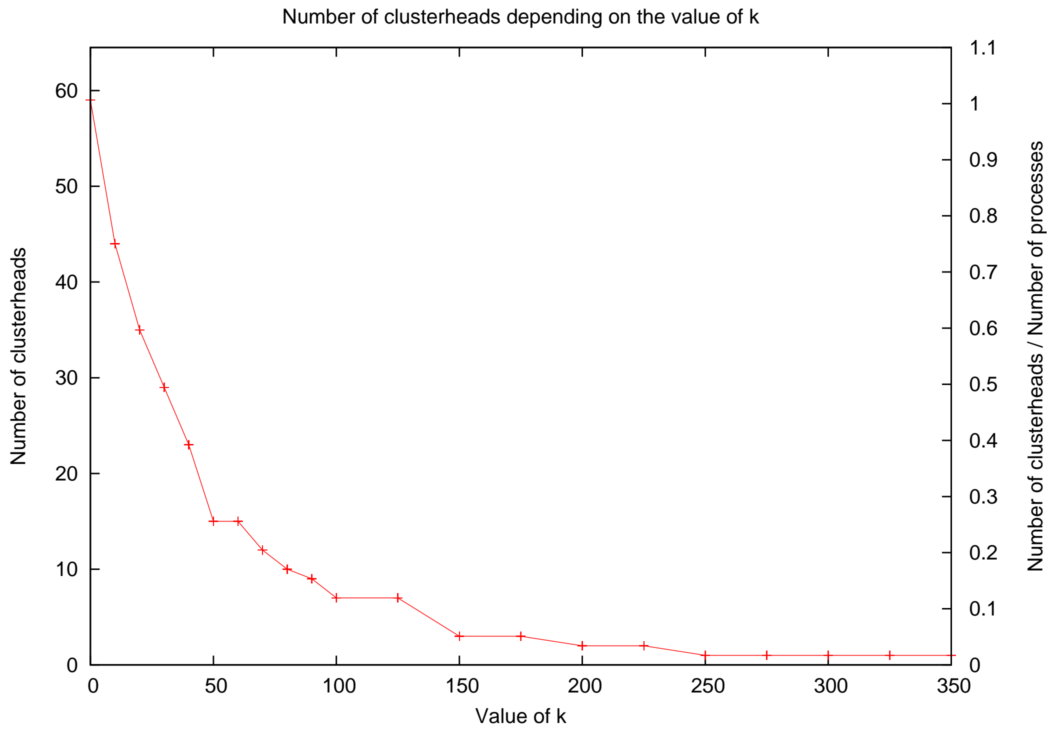


Figure 7: Number of clusterheads for graph Figure 6.

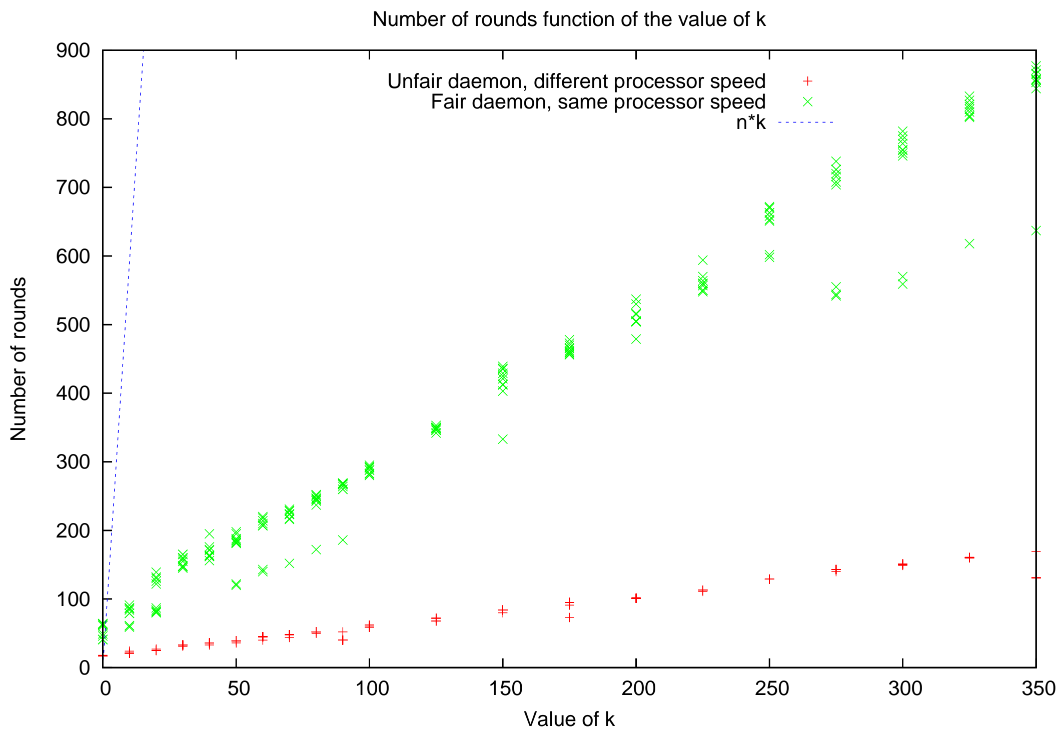


Figure 8: Number of rounds with unfairness and different computation speeds, and without unfairness and the same computation speed, for graph Figure 6.

6.2 Complexity bounds

Graph $G_{n,k}$ The number of clusterheads obtained for each instance of the graph is $n-1$: every node is elected clusterhead, appart from node n which connects itself to node $n-1$.
Figure 9 presents the number of rounds obtained with and without unfairness for different values of n, for $k=100$. It can be observed that the number of rounds follows the theoretical bound $O(nk)$.

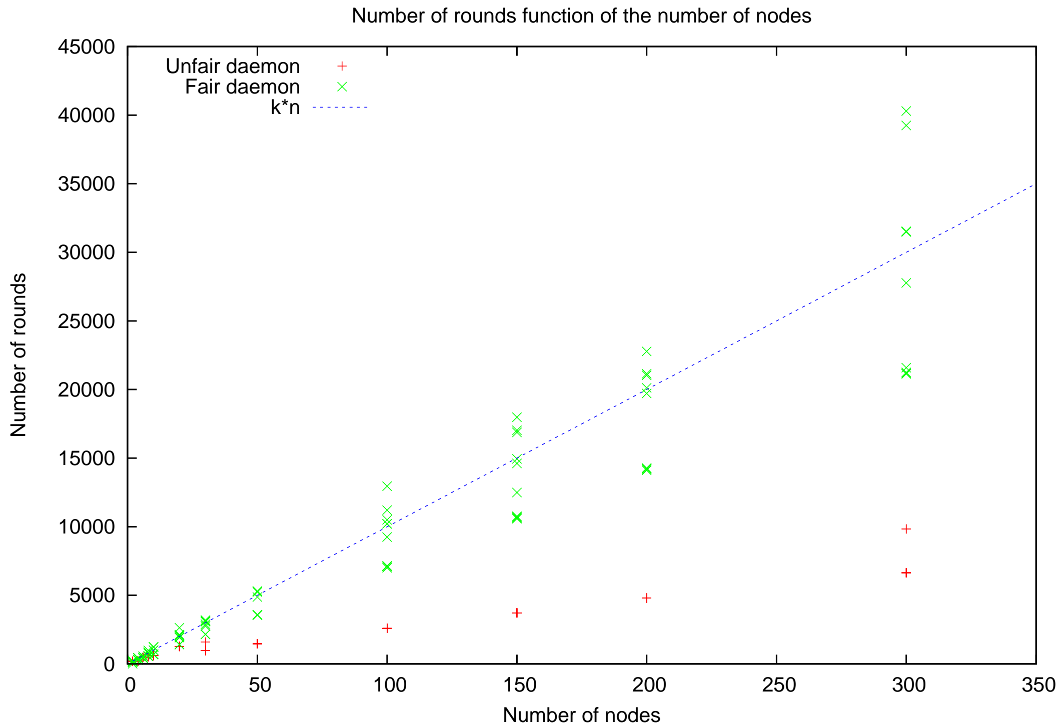


Figure 9: Number of rounds with unfairness and without unfairness for graph $G_{n,k}$, represented Figure 4. $k=100$ for all runs.

Graph $H_{n,k}$ The number of clusterheads obtained for each instance of the graph is 1: every node connects itself to the node of lowest ID: 1.
Figure 10 presents the number of rounds obtained with and without unfairness for different values of n, for $k=100$. It can be observed that the number of rounds follows the theoretical bound $\Omega(\sqrt{n}k)$.

6.3 Random graphs

The following results were obtained on graphs generated randomly. Each graph contains 100 nodes, their IDs are randomly taken between 1 and 500, the edges have weights between 1 and 100, the computing power of each node is taken between 1 and 10.

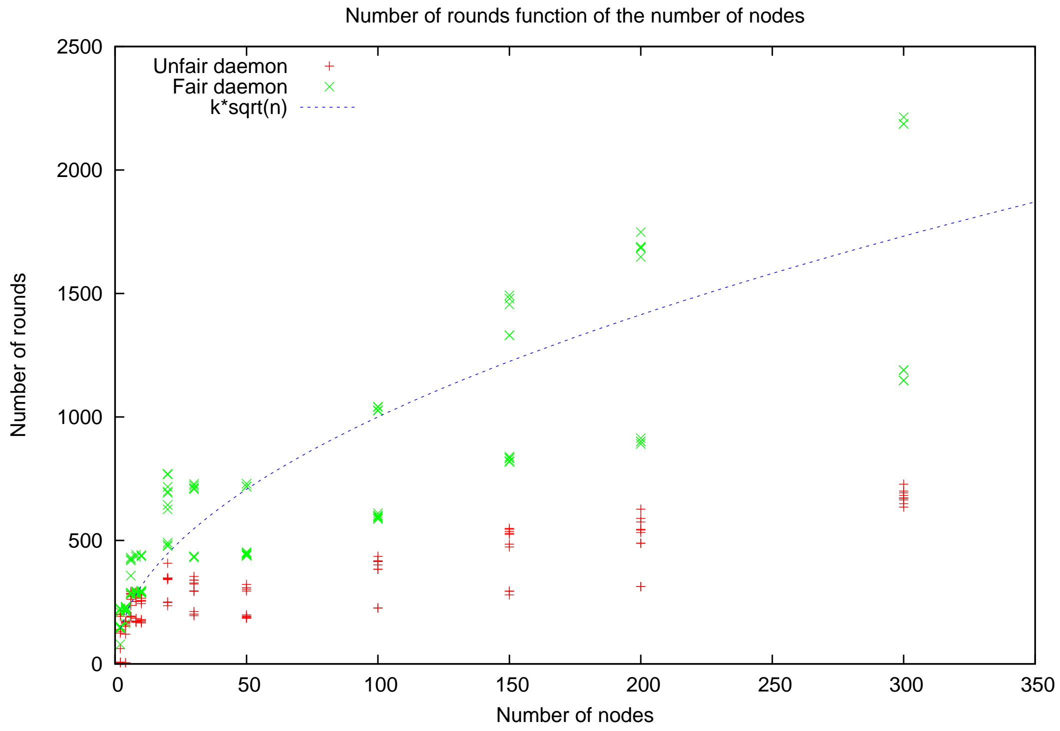


Figure 10: Number of rounds with and without unfairness for graph Figure 5.

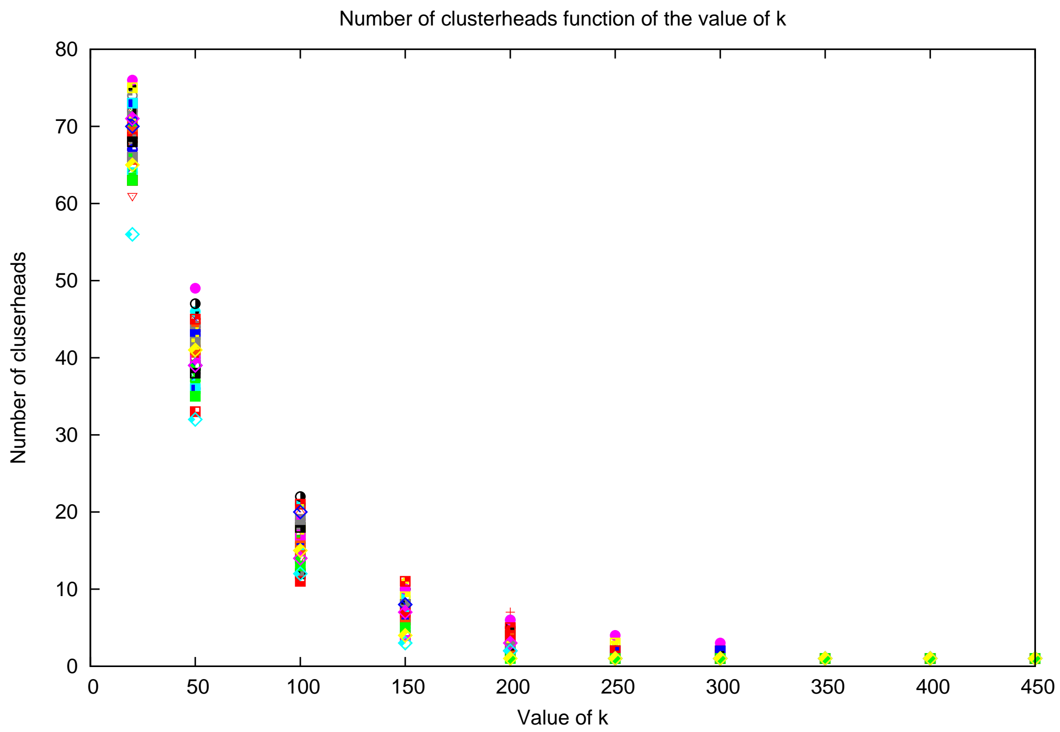


Figure 11: Number of clusterheads for random graphs.

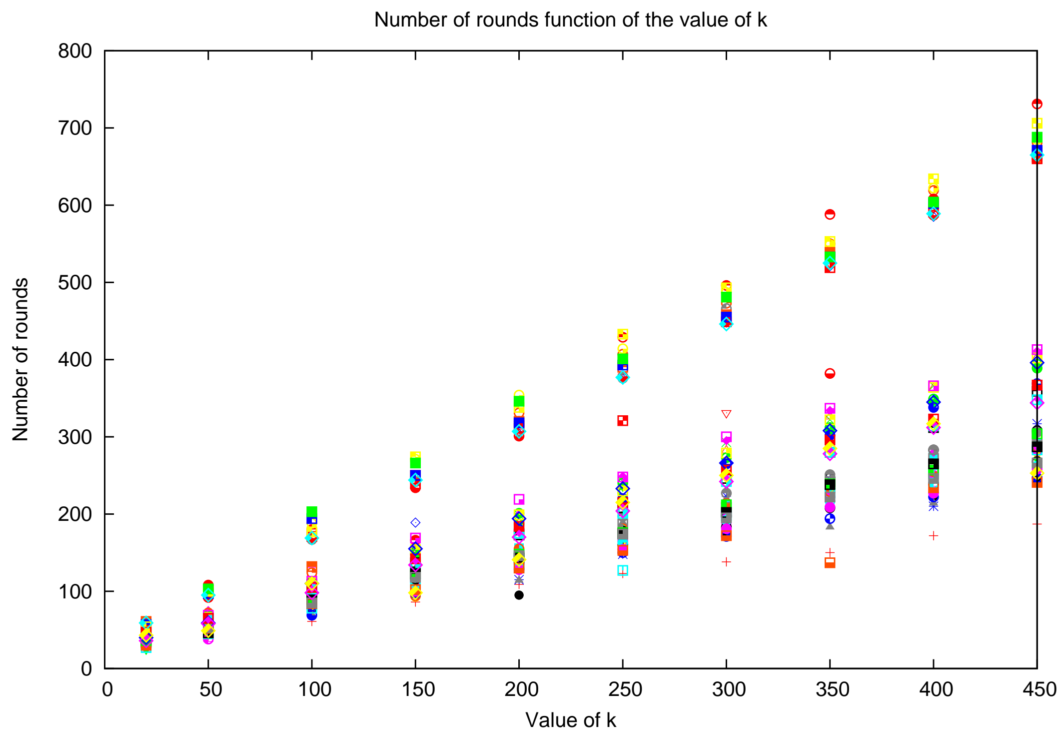


Figure 12: Number of rounds for random graphs.

7 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 $O(nk)$ rounds, and requires only $O(\log n + \log k)$ space per process.
In future work, we will attempt to improve the time complexity of the algorithm. We also intend to explore the possibility of using k-clustering to design efficient deployment algorithms for applications on a grid infrastructure.

8 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] E. Caron and F. 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.

[4] A. K. Datta, L. L. Larmore, and P. Vemula. A self-stabilizing O(k)-time k-clustering algorithm. *Computer Journal*, 2008.

[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] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

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

[11] C. Johnen and L. H. Nguyen. Self-stabilizing weight-based clustering algorithm for ad hoc sensor networks. In Sotiris E. Nikoletseas and José D. P. Rolim, editors, *ALGOSENSORS*, volume 4240 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2006.

[12] 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.

[13] 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.

Appendix

A Probability

Remark A.1 *Suppose $E_1, E_2, \ldots E_k$ are events. Then*

$$Pr(E_1 \cap \ldots E_k) = \prod_{i=1}^{k} Pr(E_i \mid E_1 \cap \ldots \cap E_{i-1})$$

Lemma A.2 *If $1 \leq i \leq d \leq n/2$, then $\frac{n-d-i+1}{n-2i+1} \leq 1 - \frac{d-i}{n}$.*

Proof: Since n and $n - 2i + 1$ are both positive, the statement of the lemma is equivalent to the following inequality, obtained by cross-multiplication:

$$n(n-d-i+1) \quad \leq \quad (n-d+i)(n-2i+1) \tag{1}$$

By routine calculation, we can verify that the right hand side of (1) minus the left hand side is $(d - i)(2i - 1) \geq 0$. □

The following lemma is well-known.

Lemma A.3 $1 - x \leq e^{-x}$ *for any real number x.*