A Self-stabilizing K-Clustering Algorithm Using an Arbitrary Metric

Eddy Caron[1,2], Ajoy K. Datta[3], Benjamin Depardon[1,2], and Lawrence L. Larmore[3]

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

Abstract. Mobile *ad hoc* networks as well as grid platforms are distributed, changing and error prone environments. Communication costs within such infrastructures 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. To the best of our knowledge, this is the first distributed solution to the k-clustering problem on weighted graphs.

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

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.

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 602–614, 2009.

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 [1] and GridSolve [2] 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 [3] 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.

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

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 $y \in D$, $C_y = \{x : Clusterhead(x) = y\}$ 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 [5].

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.* [5] 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 [6] 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 [6] are not given. Fernandess and Malkhi [7] 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 [8]; it 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.

1.3 Contributions and Outline

Our solution, Algorithm Weighted-Clustering, given in Sect. 3, is partially inspired by that of Amis *et al.* [5], 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 Sect. 2, we describe the model of computation used in the paper, and give some additional needed definitions. In Sect. 3, we define the algorithm Weighted-Clustering, and give its time and space complexity. We also show an example execution of Weighted-Clustering in Sect. 4. Finally, we present some simulation results in Sect. 5 and conclude the paper in Sect. 6.

[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 $\forall v \in V - V', \exists y \in V' : (x, y) \in E$. [4].

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 [3,9]. 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* [10], 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: (i) Every process P that is enabled at γ_i either executes or becomes neutralized during some step of ϱ, (ii) The execution $\gamma_i \mapsto \ldots \mapsto \gamma_{j-1}$ does not satisfy condition (i). 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 [3]. 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: (i) (Convergence) Every maximal execution contains some member of $\mathbb{L}_{\mathcal{A}}$, (ii) (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 $\mathcal{U}_P = \mathcal{N}_P \cup \{P\}$ the closed neighborhood of P. For each process P, we define the following values:

$$
\begin{aligned}
\mathit{MinHop}(P) &= \min\{\min\{w(P,Q) : Q \in \mathcal{N}_P\}, k+1\} \\
\mathit{MinId}(P,d) &= \min\{Q.id : w(P,Q) \leq d\} \\
\mathit{MaxMinId}(P,d) &= \max\{\mathit{MinId}(Q,k) : w(P,Q) \leq d\} \\
\mathit{Clusterhead\_Set} &= \{P : \mathit{MaxMinId}(P,k) = P.id\} \\
\mathit{Dist}(P) &= \min\{w(P,Q) : Q \in \mathit{Clusterhead\_Set}\} \\
\mathit{Parent}(P) &= \begin{cases} P.id & \text{if } P \in \mathit{Clusterhead\_Set} \\ \min\left\{Q.id : \begin{array}{l}(Q \in \mathcal{N}_P) \quad \wedge \\ (\mathit{Dist}(Q) + w(P,Q) = \mathit{Dist}(P))\end{array}\right\} & \text{otherwise} \end{cases} \\
\mathit{Clusterhead}(P) &= \begin{cases} P.id & \text{if } P \in \mathit{Clusterhead\_Set} \\ \mathit{Clusterhead}(\mathit{Parent}(P)) & \text{otherwise} \end{cases}
\end{aligned}
$$

The *output* of Weighted-Clustering consists of shared variables $P.parent$ and $P.clusterhead$ for each process P. The output is *correct* if $P.parent = Parent(P)$ and $P.clusterhead = Clusterhead(P)$ for each P. Hence, the previous values define the sequential version of our algorithm. Weighted-Clustering is self-stabilizing. Although it can compute incorrect output, the output shared variables will eventually stabilize to their correct values.

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. In this conference paper, we present a broad and intuitive explanation of how the

algorithm works. Technical details of Weighted-Clustering can be found in our research report [11] which also contains proofs of its correctness and complexity.

Weighted-Clustering consists of the following four phases.

Phase 1, Self-Stabilizing Leader Election (SSLE***).*** We make use of an algorithm SSLE, defined in [12], 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 [12].

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.

***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.

- 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 $Root\_\text{BFS}$.
- $P.parent\_\text{BFS} = \begin{cases} P.id & \text{if } P = Root\_\text{BFS} \\ \min \left\{ Q.id : \begin{array}{l} (Q \in \mathcal{N}_P) \wedge \\ (Q.dist\_\text{BFS} + 1) = P.dist\_\text{BFS} \end{array} \right\} & \text{otherwise} \end{cases}$

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. Three 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.
- 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 $P.minid$, so 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.

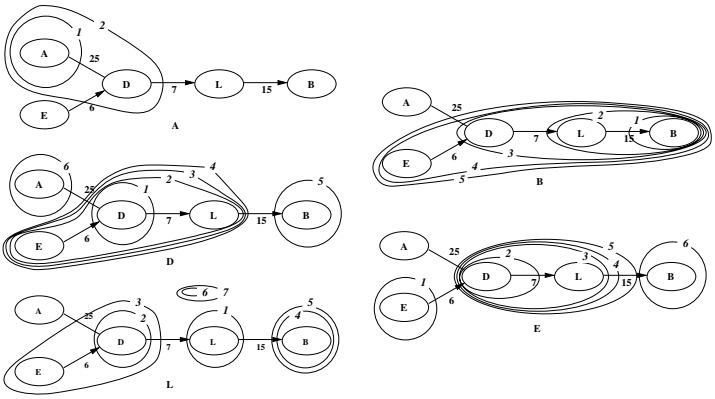


Fig. 1. Evolution of the search interval $P.minlevel \leq d < P.minhilevel$ for the example computation Fig. 2

3.5 Building Clusters

The heart of the algorithm is identification of the clusterheads, which consists of two phases, MinId and MaxMinId. We describe only MinId in detail, as MaxMinId is similar. When it ends, if $P.maxminid = P.id$, P is a clusterhead.

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 is given Fig. 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.

3.6 Time and Space Complexity

The algorithm uses all the variables of SSLE [12] 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

In Fig. 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). We name A_{minid} the action consisting in updating $P.minlevel$, $P.minid$ and $P.minhilevel$, and $A_{hilevel}$ the action consisting in updating only $P.minhilevel$. An arrow in the figure from a process P to a process Q indicates that Q prevents P from executing Action A_{minid}. 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 Fig. 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 A_{minid}, but can possibly execute $A_{hilevel}$ to update its $minhilevel$ variable. At any step, two neighbors cannot both execute Action A_{minid} due to a special condition present in the guard. This prevents miscalculations of $minid$.

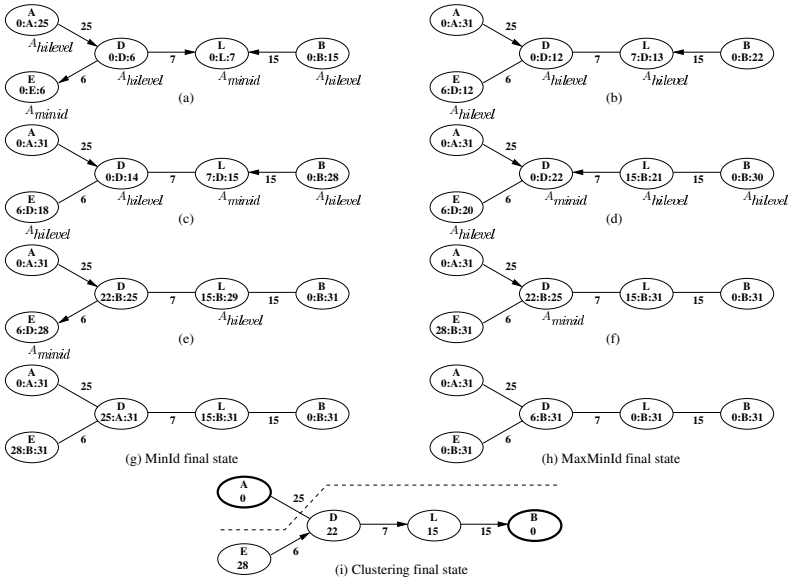


Fig. 2. Example computation of Weighted-Clustering for $k = 30$

Consider the process L. Initially it is enabled to execute Action A_{minid} (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 $A_{hilevel}$ to correct this value. The idea behind the $minhilevel$ variable, is to prevent the process from choosing a minimum ID at a distance greater than $minhilevel$ before its neighbors have a chance to copy its current value of $minid$, if necessary. 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).

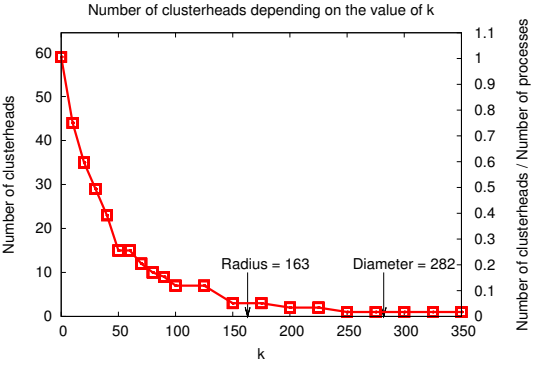
The MinId phase halts when $P.minhilevel = k + 1$ for all P (subfigure (g)). In the final step every P knows the process of minimum ID at a distance no greater than k, and $P.minlevel$ holds the distance to this process.

Sometimes, a process P can be elected clusterhead by another process Q without having elected itself clusterhead (this does 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 clusterhead to flow back to P.

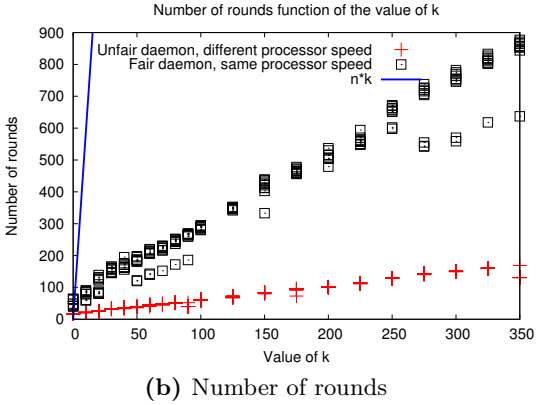
5 Simulations

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 (at least two orders of magnitude higher than the theoretical convergence time).

We ran the simulator on random weighted graphs. 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. Due to space



(a) Number of clusterheads



(b) Number of rounds

Fig. 3. Simulations results

constraints we cannot show all these results, and present results obtained for one of the random graphs containing 59 nodes, with a diameter equal to 282 (9 hops), links' weights are between 1 and 100, and the degree of the nodes are between 3 and 8 (5.4 on average).

Figure 3a 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 consist in electing each process a clusterhead. The number of clusterheads quickly decreases as k increases.

Figure 3b 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 power for all processes. The number of rounds is far lower than the theoretical bound $O(nk)$, even with an unfair daemon.

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 $O(nk)$ 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, and use the message passing model, which is more realistic. We also intend to explore the possibility of using k-clustering to design efficient deployment algorithms for applications on a grid infrastructure.

References

1. Caron, E., Desprez, F.: DIET: A scalable toolbox to build network enabled servers on the grid. Int. Jour. of HPC Applications 20(3), 335–352 (2006)
2. YarKhan, A., Dongarra, J., Seymour, K.: GridSolve: The Evolution of Network Enabled Solver. In: Patrick Gaffney, J.C.T.P. (ed.) Grid-Based Problem Solving Environments: IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments, Prescott, AZ, July 2006, pp. 215–226. Springer, Heidelberg (2007)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
5. Amis, A.D., Prakash, R., Vuong, T.H., Huynh, D.T.: Max-min d-cluster formation in wireless ad hoc networks. In: IEEE INFOCOM, pp. 32–41 (2000)
6. Spohn, M., Garcia-Luna-Aceves, J.: Bounded-distance multi-clusterhead formation in wireless ad hoc networks. Ad Hoc Networks 5, 504–530 (2004)

7. Fernandess, Y., Malkhi, D.: K-clustering in wireless ad hoc networks. In: ACM Workshop on Principles of Mobile Computing POMC 2002, pp. 31–37 (2002)
8. Datta, A.K., Larmore, L.L., Vemula, P.: A self-stabilizing O(k)-time k-clustering algorithm. Computer Journal (2008)
9. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
10. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Trans. Parallel Distrib. Syst. 8(4), 424–440 (1997)
11. Caron, E., Datta, A.K., Depardon, B., Larmore, L.L.: A self-stabilizing k-clustering algorithm using an arbitrary metric. Technical Report RR2008-31, Laboratoire de l'Informatique du Parallélisme, LIP (2008)
12. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing leader election in optimal space. In: 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Detroit, MI (November 2008)