A Self-Stabilizing K-Clustering Algorithm Using an Arbitrary Metric (Revised Version of RR2008-31)

Eddy Caron, Ajoy Datta, Benjamin Depardon, Lawrence Larmore

Laboratoire de l'Informatique du Parallélisme

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

A Self-Stabilizing K-Clustering Algorithm Using an Arbitrary Metric (Revised Version of RR2008-31)

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

[1] University of Lyon; ENS-Lyon/INRIA/CNRS/UCBL; LIP Laboratory. France.
[2] School of Computer Science, University of Nevada, Las Vegas, USA.

December 2009

Research Report N° RRLIP2009-33

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 of RR2008-31)

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

[1] University of Lyon; ENS-Lyon/INRIA/CNRS/UCBL; LIP Laboratory. France.

[2] School of Computer Science, University of Nevada, Las Vegas, USA.

December 2009

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 plates-formes de grille sont des environnements distribués et sujets à de nombreuses erreurs. Les coûts de communication au sein de ces 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 nœuds 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 distribuée au problème du k-regroupement sur des graphes pondérés.

Mots-clés: K-Partitionnement, Auto-Stabilization, Graphes pondérés.

1 Introduction

Overlay structures of distributed systems 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 function which measures proximity with respect to some relevant criterion; the clustering will result in clusters with similar 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 hops, *i.e.,* the number of links in the path between two processes, as the sole measure of distance may hide the true communication time between two nodes.

A major aspect of grid computing is the deployment of grid middleware. Hop distance is used as a metric in some applications, but it may not be relevant in some platforms, such as grids. Using an arbitrary metric (*i.e.,* a weighted metric) is a reasonable option in such heterogeneous distributed systems. Distributed grid middleware, like DIET [4] and GridSolve [17] can make use of accurate distance measurements to do efficient job scheduling.

Another important consideration 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 partitions nodes into clusters which lie within a given distance of each other, and which can dynamically adapt to any change, is valuable for many applications, including MANET and grid platforms.

Self-stabilization [8] 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. As MANET and grid platforms are dynamic and error prone infrastructures, self-stabilization is a very desirable property for the algorithms which manage those structures.

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 positive 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 positive integer k, we define a *k-cluster* of G to be a non-empty connected subgraph of G such that all processes in the the cluster are within distance k of a designated leader process, called the *clusterhead*.

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 $y \in D$

[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 $\forall v \in V - V', \exists y \in V' : (x, y) \in E$. [11]

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, or equivalently, a k-clustering with the minimum possible number of clusters, is known to be $\mathcal{NP}$-hard [1]. Our algorithm attempts to find a k-clustering which has "few" clusters.

1.2 Related Work

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 [16] give a distributed solution to a more generalized version of the k-clustering problem. In their algorithm, 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 [16] are not given. Fernandess and Malkhi [10] give an algorithm for the k-clustering problem that uses $O(\log n)$ memory per process, takes $O(n)$ steps, provided a *breadth first search* BFS tree[3] for the network is already given.

The first self-stabilizing solution to the k-clustering problem was given by Datta *et al.* in [7]; this solution takes $O(k)$ rounds and $O(k \log n)$ space. Another stabilizing solution was proposed in [5]; this algorithm needs $O(n)$ rounds and $O(\log n)$ space. Both solutions use the hop metric, and are thus unable to deal with more general weighted graphs.

Many algorithms have been proposed in the literature for constructing clusters in distributed network. Other self-stabilizing clustering algorithms deal with weighted graphs where weights are placed on the vertices, not on the edges. For example, Johnen and Nguyen give in [12] an algorithm to partition the network into 1-hop clusters, *i.e.,* the algorithm computes a *dominating set*, a set S such that ever node is a neighbor of some member of S. The article presents self-stabilizing versions of DMAC [3] and GDMAC [2]. The authors also give a robust version of both algorithms in [13], *i.e.,* after one round the network is partitioned into clusters, and stays partitioned during construction of the final clusters.

A self-stabilizing algorithm for cluster formation under a *density* criterion is presented in [15] by Mitton *et al.*. The density criterion (defined in [14]) is used to select clusterheads – a node v is elected a clusterhead if it has the highest density in its neighborhood, and the cluster headed by v contains all nodes at distance less or equal to two from v.

1.3 Contributions

Our solution, Algorithm K-CLUSTERING, given in Section 6, is partially inspired by that of Amis *et al.* [1], who use hop distance instead of arbitrary edge weights. K-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.

1.4 Outline

In Section 2, we describe the model of computation used in the paper, and give some additional needed definitions. In Section 6, we first present a broad and intuitive explanation of the algorithm K-CLUSTERING before defining it more formally, and give its time and space complexity. We give proofs of its correctness and complexity in Section 6.2. Finally, we present simulation results in Section 8, and conclude the paper in Section 9.

[3] A BFS tree has a designated *root*, and from each node, the path from that node through the BFS tree to the root is the shortest possible path in the network.

2 Preliminaries

We consider a connected undirected network of of n processes, where $n \geq 2$, and an integer $k \geq 1$. Each process P has a unique ID, $P.id$ of an ordered type, which we call ID type. 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 an algorithm, $\mathcal{A}$ 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, each protected by a *guard* . The *guard* of an action in the program of a process P is a Boolean expression involving the variables of P and of 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. In this paper, we do not use the classic representation for self-stabilizing algorithms: $< label >:: < guard > \longrightarrow < statement >$, instead we present the algorithms in pseudo-code, just like regular algorithms.

We use the composite atomicity model of computation [8, 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*, 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 [8]. 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 Best Reachable Problem

We define the *Best Reachable* problem on a network as follows. We are given a positive weight function w on edges, and we let $w(P, Q)$ be the minimum weight of any path from P to Q, as before. We are also given a number k, the *allowed distance*. Without loss of generality, the weight of any edge is at most $k + 1$.

Each process P has a *value* $P.\Theta$, of some type, and each process must calculate the *best* value of $Q.\Theta$ over all processes Q within that allowed distance of P. More specifically, each P must calculate $best\{Q.\Theta : w(P, Q) \leq k\}$.

Best means maximum under any given ordering. In our code, we will write "$\succ$" for a given order relation on values of Θ, and we say that $P.\Theta$ is *best* if $P.\Theta \succeq Q.\Theta$ for all processes Q.

Throughout the paper, we write $\mathcal{N}_P$ for the set of all neighbors of P.

3.1 Algorithm NSSBR

We now give a distributed algorithm, NSSBR, which we also call Algorithm 1, for the best reachable problem. Each process P has variables $P.best$, whose value of the best value of Θ that P has found so far, $P.dist$, the distance from P of the nearest Q for which $Q.\Theta = P.best$, and $P.span$, whose meaning is as follows: $P.best = best\{Q.\Theta : w(P, Q) < P.span\}$. That is, P has so far found the best value of Θ among all process which are closer than $P.span$, but not among those whose distance from P is greater than or equal to $P.span$.

Initially, $P.best = P.\Theta$ and $P.dist = 0$, because P only considers of its own value of Θ. The initial value of $P.span$ is the shortest distance from P to any neighbor, since P has not searched any neighbor for a better value.

As the algorithm proceeds, each process P repeatedly iterates the main loop, shown as lines 4 through 12 in the code below. The loop will iterate until $P.span > k$, which will indicate that P has searched all processes of distance at most k to find the best value of Θ.

The only way that P can become aware of values of Θ beyond its immediate neighborhood is through its neighbors. For example, if X is within k of P and $X.\Theta$ is the best value of Θ within k of P, then P must have a neighbor Y which is on the shortest path from P to X, and P will learn about $X.\Theta$ from Y. At some point in the computation, $Y.best = X.\Theta$, and P will update $P.best$ to that value.

However, there is a complication. Even though P learns about $X.\Theta$ via Y, it could be that there is some better value of Θ within k of Y, but not within k of P. This means that $Y.best$ will eventually be better than $X.\Theta$. We must make sure that P can read $Y.best$ before that happens.

Each process P has the following code.

Algorithm 1: NSSBR : A Non-Self-Stabilizing Algorithm for Best Reachable

1: $P.best \leftarrow P.\Theta$
2: $P.dist \leftarrow 0$
3: $P.span \leftarrow \min\{w(P, Q) : Q \in \mathcal{N}_P\}$
4: **while** $P.span \leq k$ **do**
5: **if** $\forall Q \in \mathcal{N}_P : ((Q.best \succeq P.best) \vee (P.dist + w(P, Q) > k)) \wedge (w(P, Q) + Q.span > P.span)$ **then**
6: **if** $\exists Q \in \mathcal{N}_P : Q.best \succ P.best$ and $Q.dist + w(P, Q) = P.span$ **then**
7: $P.best \leftarrow \max_{\succ}\{Q.best : Q \in \mathcal{N}_P$ and $Q.dist + w(P, Q) = P.span\}$
8: $P.dist \leftarrow P.span$
9: **end if**
10: $P.span \leftarrow \min \begin{cases} \min\{X.span + w(P, X) : X \in \mathcal{N}_P\} \\ \min\{X.dist + w(P, X) : X \in \mathcal{N}_P \text{ and } X.best \succ P.best\} \end{cases}$
11: **end if**
12: **end while**

In Line 7, "$\max_{\succ}$" denotes maximum with respect to the order relation "$\succ$."

In order to fit Algorithm 1 into our model of computation, we assume that each P executes lines 1 through 3 of the code immediately, *i.e.,* before any other process reads its values. Lines 4 through 12 are executed as one atomic step, so that a neighbor of P cannot, for example, read the new value of $P.best$ until the new values of $P.dist$ and $P.span$ are also computed.

The code of Algorithm 1 is not self-stabilizing. We will later show how to modify it to make it self-stabilizing.

3.2 Proof of Correctness for NSSBR

In this section, we prove that Algorithm 1 converges and that after convergence, $P.best = \max_{\succ} \{Q.\Theta : w(Q,P) \leq k\}$ for all P.

Intuition As Algorithm 1 proceeds, each process P tries to find the best value of Θ within an increasing distance. It keeps track of the *search radius*, $P.span$, as well as $P.best$, the best value of Θ within that distance of itself. It also keeps track of $P.dist$, which is the distance to the process whose Θ value is $P.best$. (In case more than on such process exists, $P.dist$ is the smallest choice of distance.)

Loop Invariant We now define the *loop invariant* of the main loop of Algorithm 1, which is the conjunction of the following invariants, each of which holds for all choices of processes P, X, and Y.

LI(i)(P): $0 \leq P.dist \leq k$ and $P.dist < P.span$

LI(ii)(P): $P.best = \max_{\succ} \{X.\Theta : w(P,X) < P.span \text{ and } w(P,X) \leq k\}$

LI(iii)(P): $P.dist = \min \{w(P,X) : X.\Theta = P.best\}$

LI(iv)(P,X): $P.span \leq X.span + w(P,X)$ if $X \in \mathcal{N}_P$

LI(v)(P,X,Y): If $Y \in \mathcal{N}_P$, $w(P,X) < P.dist$, and $w(P,X)+w(P,Y) \leq k$, then $X.\Theta \preceq Y.best$

Explanation of the Loop Invariant We now explain the intuition behind the loop invariant. Figure 1 illustrates LI(i), LI(ii), and LI(iii). For each process P, the distance from P to the nearest process Q such that $Q.\Theta = P.best$ is stored as $P.dist$, and no process closer to P has a better value of Θ. Furthermore, P has determined that better Θ exists among all processes closer than $P.span$.

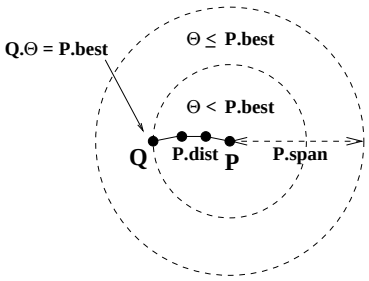


Figure 1: Invariants LI(i), LI(ii), and LI(iii).

Figure 2 illustrates LI(iv). If Q is a neighbor of P, then $Q.span + w(P,Q) \geq P.span$. The basic reason for this invariant is that P derives all information about other processes from its neighbors.

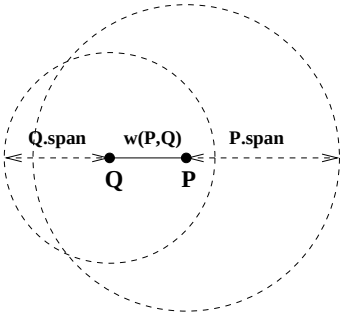


Figure 2: Invariant LI(iv).

By far the hardest invariant to explain is LI(v). Suppose $Y \in \mathcal{N}_P$, $w(P,X) < P.dist$, and $w(P,X) + w(P,Y) \leq k$. Pick processes U and V such that $U.\Theta = Y.best$ and $V.\Theta = P.best$, as illustrated in Figure 3. Suppose also that $w(Y,V) > k$. Thus, it could happen that $X.\Theta$ is the largest value of Θ within k of Y.

The only way that Y can know about $X.\Theta$ is through its neighbor, P. But $P.best = V.\Theta$, which is larger than $X.\Theta$, and thus $P.best$ will never again be equal to $X.\Theta$.

To avoid error, we must ensure that $X.\Theta$ will not be needed by Y in the remaining part of the computation. The invariant LI(v), which states that $Y.best \succeq X.\Theta$, guarantees this.

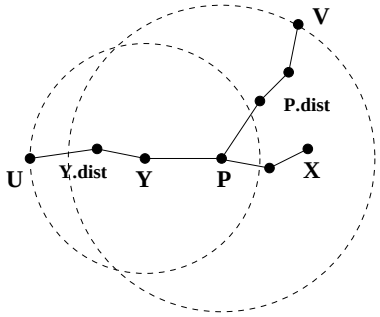


Figure 3: Invariant LI(v): $U.\Theta = Y.best$, $V.\Theta = P.best$, $w(U,Y) = Y.dist$, $w(P,V) = V.\Theta$, $w(P,X) < P.dist$, and $w(Y,X) \leq k < w(Y,V)$.

Figure 4 gives an example of how a calculation can go wrong if LI(v) is not used. In that figure, $D.best$ will be unable to achieve its correct value of $3 = B.\Theta$, since $C.best$ has already found a better value, namely $4 = A.\Theta$, for $k = 2$.

Lemma 3.1 *The loop invariant holds after each process executes Lines 1 through 3 of the code of Algorithm 1.*

Proof: Recall our assumption that no process iterates the main loop of Algorithm 1 until after all processes have *initialized*, *i.e.,* have executed Lines 1 through 3. After all processes have initialized, then $P.span = \min\{w(P,Q) : Q \in \mathcal{N}_P\} > 0$, $P.dist = 0$, and $P.best = P.\Theta$ for all P. The invariants LI(i) through LI(iv) are then trivially true, while LI(v) holds vacuously. □

Lemma 3.2 *If the loop invariant holds before a step, then it holds after that step.*

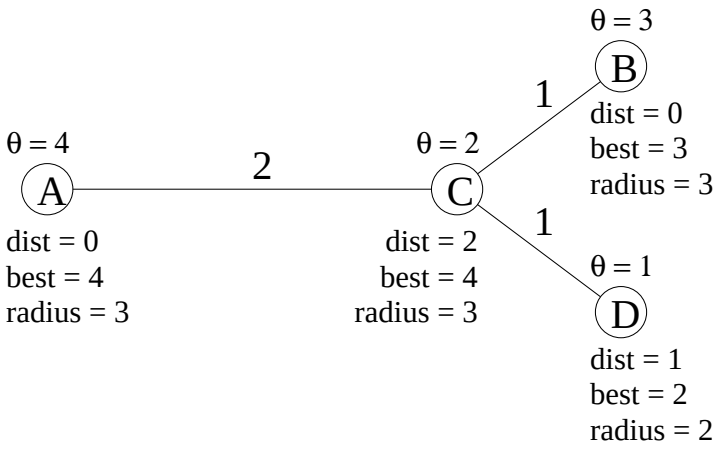


Figure 4: Example showing the necessity of LI(v). In the figure, $k = 2$, and the invariant LI(v)(C, D, B) is false, although all other parts of the loop invariant hold. It is impossible for $D.best$ to achieve its correct value of 3.

Proof: Assume that the loop invariant holds before a given step. During the step, some subset of processes executes the loop of Algorithm 1. For each process P, let $P.best$, $P.dist$, and $P.span$ be the values of P's variables before the step, and let $P.best'$, $P.dist'$, and $P.span'$ be the values after that step.

We will also write LI(i), LI(ii), *etc.* for the invariants before the step, and LI(i)′, LI(ii)′, *etc.* for the invariants after the step.

For our proof, we fix a process P, and assume that LI(i)(P), LI(ii)(P), and LI(iii)(P) hold, and that LI(iv)(P, X) and LI(v)(P, X, Y) hold for all processes X and Y. We then prove that the corresponding "primed" invariants, LI(i)′(P), LI(ii)′(P), *etc.* hold.

We will consider three cases, depending on the execution of P during the step. Case I is where the condition of the **if** statement on Line 5 is false for P. In this case, P does not change its variables during the step. Case II is where that condition is true, but the condition of the **if** statement on Line 6 is false for P. In this case, P executes Line 10, but does not execute Lines 7 or 8. Case III is where the conditions on Lines 5 and 6 are both true. In this case, P executes Lines 7, 8, and 10.

In Case III, we will choose $Q \in \mathcal{N}_P$ such that $P.span = Q.dist + w(P, Q)$, and $Q.best$ is maximum subject to that condition; thus $Q.best \succ P.best$. We will also choose a process R such that $w(Q, R) = Q.dist$ and $R.\Theta = Q.best$. In Cases I and II, Q and R are undefined.

Claim A: In Case III, $P.best' \succ P.best$ and $P.span' > P.span = P.dist' > P.dist$.

Proof (of Claim A): $P.dist' = P.span > P.dist$ by LI(i)(P). $P.best' = Q.best \succ P.best$.

We need only show that $P.span' > P.span$. Suppose not. Then, either $\exists X \in \mathcal{N}_P$ such that $X.span + w(P, X) < P.span$, which contradicts LI(iv)(P, X), or $\exists X \in \mathcal{N}_P$ such that $X.best \succ P.best$ and $X.dist + w(P, X) < P.span$. But, by the choice of Q and by LI(ii)(P), $Q.best \succeq X.best$ for all $X \in \mathcal{N}_P$ such that $X.dist + w(P, X) \leq P.span$, contradiction. The last case is when $\exists X \in \mathcal{N}_P$ such that $X.span + w(P, Q) \leq P.span$, but this case is prohibited by condition line 5. □

Claim B: If $X \in \mathcal{N}_P$ and $P.best \prec X.best$, then $P.span \leq w(P, X) + X.dist$.

Proof (of Claim B): By LI(iii), we can pick Y such that $Y.\Theta = X.best$ and $w(X, Y) = X.dist$. By LI(ii) and by the triangle inequality, $P.span \leq w(P, Y) \leq w(P, X) + w(X, Y) = w(P, X) + X.dist$. □

Claim C: For any process P, $P.best' \succeq P.best$, $P.dist' \geq P.dist$, and $P.span' \geq P.span$.

Proof (of Claim C): In Case I, there is nothing to prove. In Case III, we are done by Claim A. Consider Case II. Trivially, $P.best' = P.best$ and $P.dist' = P.dist$. $X.span + w(P,X) \geq P.span$ for all $X \in \mathcal{N}_P$, by LI(iv)(P,X), and $X.dist + w(P,X) \geq P.span$ for all $X \in \mathcal{N}_P$ such that $X.best \succ P.best = P.best'$, by Claim B. Thus, $P.span' \geq P.span$. □

In Case I, LI(i)$'(P)$, LI(ii)$'(P)$, LI(iii)$'(P)$, and LI(iv)$'(P,X)$ hold trivially, since LI(i)(P), LI(ii)(P), LI(iii)(P), LI(iv)(P,X) hold and the variables of P do not change. LI(v)$'(P,X,Y)$ holds, since LI(v)(P,X,Y) holds, and since $Y.best' \succeq Y.best$, by Claim C applied to Y. This completes the proof of the lemma in Case I, and thus henceforth, we assume that we have either Case II or Case III.

Claim D: $P.span' \leq \min \begin{cases} \min\{X.span + w(P,X) : X \in \mathcal{N}_P\} \\ \min\{X.dist + w(P,X) : X \in \mathcal{N}_P \text{ and } X.best \succ P.best\} \end{cases}$ for any process P.

Proof (of Claim D): Since P executes Line 10 during the step, that execution makes the claim true. □

Claim E: In Case III, $P.dist' = w(P,R) = w(P,Q) + w(Q,R)$.

Proof (of Claim E): By the triangle inequality and LI(iv)(P,R),
$w(P,R) \leq w(P,Q) + w(Q,R) = w(P,Q) + Q.dist = P.span \leq w(P,R)$, and $P.span = P.dist'$. □

Claim F: There is some process X such that $w(P,X) = P.dist'$.

Proof (of Claim F): In Case II, $P.dist' = P.dist$, and we are done by LI(iii)(P). In Case III, let $X = R$. We are done by Claim E. □

Claim G: For any process X:

(a) If $w(P,X) < P.dist'$, then $X.\Theta \prec P.best'$.

(b) If $w(P,X) = P.dist'$, then $X.\Theta \preceq P.best'$.

Proof (of Claim G): In Case II, $P.dist' = P.dist$ and $P.best' = P.best$, and we are done by LI(iii)(P). Consider Case III. Choose $Y \in \mathcal{N}_P$ such that $w(P,X) = w(P,Y) + w(Y,X)$. Then $Y.best \succeq P.best$ and

$$Y.span > P.span - w(P,Y) = P.dist' - w(P,Y) \geq w(P,X) - w(P,Y) = w(Y,X)$$

since the condition in Line 5 holds, and thus $X.\Theta \preceq Y.best$, by LI(ii)(Y).

Subcase (i): $Y.best = P.best$. Then $X.\Theta \leq Y.best = P.best \prec P.best'$, and thus both (a) and (b) hold.

Subcase (ii): $Y.best \succ P.best$.
(a): By LI(v)(Y,X,P), $X.\Theta \preceq P.best \prec P.best'$.

(b): $Y.dist \geq P.span - w(P,Y) = w(Y,X)$, by Claim B. If $Y.dist = w(Y,X)$, then $Y.best \preceq Q.best$ by our choice of Q, and thus $X.\Theta \preceq Y.best \preceq Q.best = P.best'$. If $Y.dist < w(Y,X)$, then $X.\Theta \preceq P.best \prec P.best'$ by LI(v)(Y,P,X). □

Claim H: For any process X, if $w(P,X) < P.span'$ and $w(P,X) = k$, then $X.\Theta \preceq P.best'$.

Proof (of Claim H): In Case II, $P.span' = P.span$ and $P.best' = P.best$, and we are done by LI(ii)(P). Consider Case III. Choose $Y \in \mathcal{N}_P$ such that $w(P,X) = w(P,Y) + w(Y,X)$. Then $Y.best \succeq X.best$ and $Y.span + w(P,Y) \geq P.span' > w(P,X)$ by Claim D.

Suppose $Y.best \preceq P.best'$. Then $w(X,Y) = w(P,X) - w(P,Y) < Y.span$, and thus $X.\Theta \preceq Y.best \preceq P.best'$, by LI(ii)$(Y)$.

On the other hand, suppose $Y.best \succ P.best$. Then $w(P,X) < P.span' \leq Y.dist + w(P,Y)$, and hence $w(Y,X) < Y.dist$. We also have that $w(Y,X) + w(Y,P) = w(P,X) \leq k$. By LI(v)$(Y,X,P)$, we have $X.\Theta \preceq P.best \prec P.best'$. □

We now finish the proof of the lemma in Cases II and III.

We first show that LI(i)$'(P)$ holds. In Case II, $P.span' \geq P.span$ by Claim C. Since LI(i)(P) holds before the step, we have $0 \leq P.dist = P.dist' \leq k$ and $P.dist' = P.dist < P.span \leq P.span'$.

In Case III, then $0 < P.dist' < P.span'$, by Claim A. Since the loop condition in Line 4 holds before the step, $k \geq P.span = P.dist'$.

We now show that LI(iv)$'(P,X)$ holds.

Assume $X \in \mathcal{N}_P$. $X.dist' \geq X.dist$ and $X.span' \geq X.span$, since Claim C holds for X. By Claim D, we are done.

LI(iii)$'(P)$ follows from Claims F and G.

LI(ii)$'(P)$ follows from Claim H, and the fact that $w(P,R) < P.span'$ and $R.\Theta = P.best'$ in Case III. □

Lemma 3.3 *If there is at least one process whose value of span is at most k, then there is at least one process P such that P can iterate the loop of Algorithm 1, and such that during that iteration, at least one variable of P changes.*

Proof: Let $\mathcal{P} = \{P : P.span \leq k\}$. Pick $P \in \mathcal{P}$ such that $P.best$ is minimum. If there is more than one choice, pick P such that $P.span$ is minimum. We will show that P changes at least one of its variables during its next iteration of the loop.

We first claim that P satisfies the condition of the **if** statement in Line 5. Suppose not. Then, there is some $Q \in \mathcal{N}_P$ such that $P.dist + w(P,Q) \leq k$, and either $Q.best \prec P.best$ or $w(P,Q) + Q.span \leq P.span$.

Suppose $Q.span + w(P,Q) \leq P.span$. By the minimality of $P.best$, we have $Q.best \succeq P.best$. If $Q.best \succ P.best$, then $w(P,Q) + Q.span \leq P.span$, by the choice of Q, which contradicts the minimality of $P.span$ in our choice of P. Thus $Q.best \succ P.best$. Pick R such that $R.\Theta = Q.best$ and $w(Q,R) = Q.dist$. By LI(i)(Q) and the triangle inequality, we have

$$w(P,R) \leq w(P,Q) + w(Q,R) \leq w(P,Q) + Q.dist < w(P,Q) + Q.span \leq P.span$$

Since $R.\Theta \succ P.best$, this contradicts LI(ii)(P).

Otherwise, $Q.best \prec P.best$. Pick a process R such that $R.\Theta = P.best$ and $w(R,P) = P.dist$. Then $w(R,Q) \leq w(R,P) + w(P,Q) = P.dist + w(P,Q) \leq k < Q.span$ and $R.\Theta = P.best \succ Q.best$, which contradicts LI(ii)(Q). This proves the claim that P satisfies the condition in Line 5.

We need to show that P changes at least one variable during the resulting iteration. There are two cases.

Case I: There is some $Q \in \mathcal{N}_P$ such that $Q.best \succ P.best$ and $Q.dist + w(P,Q) = P.span$. In case of a tie, pick that Q which has the maximum value of $Q.best$. Then P will execute Lines 7 and 8, changing $P.best$ to $Q.best$, and increasing both $P.dist$ and $P.span$.

Case II: Not case I. Then P will not execute Lines 7 and 8, but will execute Line 10. We need to show that $P.span$ will increase. Let X be any neighbor of P. If $X.best \succ P.best$, then, since Case I does not hold, and by LI(i), $P.span < X.dist + w(P,X) < X.span + w(P,X)$. Otherwise, by the choice of P, $X.span \geq P.span$, and thus $P.span < X.span + w(P,X)$. It follows that $P.span$ increases when Line 10 is executed. □

Theorem 3.4 *Algorithm 1 solves the Best Reachable Problem.*

Proof: By Lemmas 3.1, 3.2, and 3.3, the loop invariant of Algorithm 1 holds at all times, and the algorithm will continue to execute as long as the loop condition, in Line 4 of the code, remains true for at least one process.

We need only show that the algorithm cannot keep changing variables forever. Whenever a process P changes any of its variables, the values of the changed variables increase, by Claim C in the proof of Lemma 3.2. There are at most n possible values of $P.best$. Since $P.dist$ is always equal to $w(P,Q)$ for some process Q, there are at most n possible values of $P.dist$. Since $P.span$ is always either equal to $w(P,Q)$ for some $Q \neq P$, or is greater than k, it also can take on at most n different values during the execution. Thus, Algorithm 1 converges.

Upon convergence $P.span > k$ for all P, and by LI(ii)(P), the value of $P.best$ is correct. □

4 Self-Stabilizing Best Reachable

4.1 Algorithm SSBR

In this section, we give a self-stabilizing algorithm, Algorithm 2, for the best reachable problem. Algorithm 2 makes use of Algorithm 1 as a module, and also requires the construction of a rooted breadth first search (BFS) tree.

We will use Algorithm 2 as a module in Section 5. For that reason, it will be explicitly designed with input and output parameters, much like a subroutine in a program.

We assume that every process has an ID, $P.id$, which is given, and does not change, and that IDs are unique.

The inputs of Algorithm 2, SSBR, include outputs of some self-stabilizing algorithm which elects a leader and constructs a BFS tree rooted at that leader. We will refer to this algorithm as SSLEBFS. The outputs of SSLEBFS are $P.parent$, the ID of the current parent of P in SSLEBFS, $P.leader$, of ID type, but possibly not the ID of any process in the network, and $P.level \geq 0$, an integer. When SSLEBFS has converged, *i.e.,* reached a legitimate configuration, $P.leader$ is the ID of the root process, $P.level$ is the length of the shortest path from P to the root, and $P.parent$ is the parent of P in the BFS tree; the parent of the root is itself.

A process P does not execute any action of SSBR if it detects that the BFS tree is incorrect. The following conditions must hold for each P if the BFS tree is correct.

1. If $Q \in \mathcal{N}_P$, then $Q.leader = P.leader$.
2. $P.level = 0$ if and only if $P.leader = P.id$.
3. If $P.level = 0$ and $Q \in \mathcal{N}_P$, then $Q.parent = P$.
4. If $Q \in \mathcal{N}_P$, then $|Q.level - P.level| \leq 1$.
5. If $Q \in \mathcal{N}_P$ and $P.parent = Q$, then $P.level = Q.level + 1$.

We say that P is *locally correct* if the above conditions hold.

Lemma 4.1 *The BFS tree is correct if and only if P is locally correct for all P.*

In addition, we assume a function Θ on processes, whose value we refer to as $P.\Theta$ for each process P, and a specified ordering of the values of Θ. In the code for Algorithm 2, we refer to that ordering using the symbol "$\succ$."

The variables which are under the control of SSBR are as follows.

$P.status \in \{working, finished, resting, ready\}$. We will say that P is working, is finished, is resting, or is ready.

$P.stable\_best$

All the variables of NSSBR, namely *P.best*, *P.dist*, and *P.span*.

The execution of SSBR consists of two parts: *status correction* and *normal execution*. During normal execution, which presumes that the BFS tree is correct, four different *status waves* are alternately broadcast and convergecast, as shown in Figure 5. During each complete cycle of waves the values of *P.best* is recomputed, and is compared to *P.stable_best*, the output variable of SSBR. *P.stable_best* is then updated, if necessary, to agree with *P.best*. Between those updates, *P.stable_best* does not change; thus, eventually, *P.stable_best* is stable.

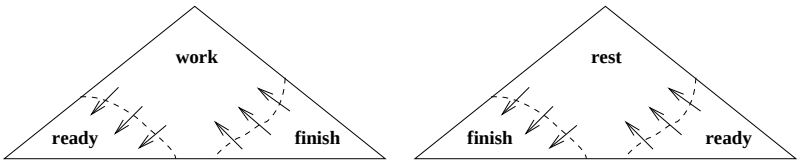


Figure 5: Broadcast waves *working* and *resting* and convergecast waves *finished* and *ready*. The *finished* wave could start before the *working* wave is completed, as shown in 5a, while the *ready* wave could start before the *resting* wave is completed, as shown in 5b.

We say that *P.status is incompatible with P.parent.status* if the current combination of status values of those two processes cannot occur during the normal part of the execution of SSBR. During status correction, *P.status* ← *P.parent.status* if the values are incompatible. Figure 6 shows the eight combinations of incompatible values.

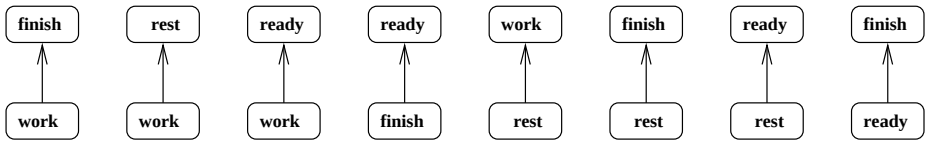


Figure 6: Corrective Status Changes. If *P.status* is incompatible with *P.parent.status*, then *P.status* ← *P.parent.status*.

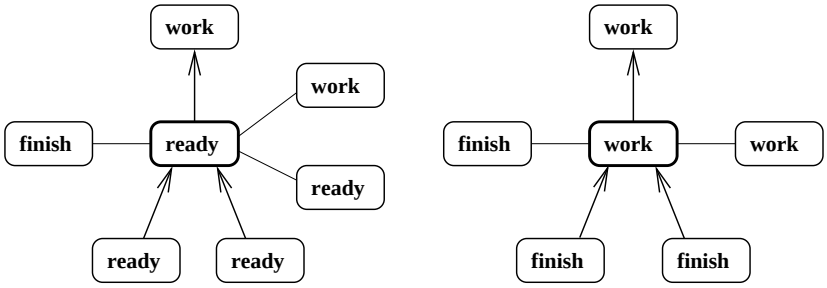
A process P can only execute normally if its status value is not incompatible with its parent's value. During normal execution of SSBR, the *P.status* can change only if the status values of the surrounding processes satisfy appropriate conditions, as shown in Figure 7.

1. If *P.status* = *ready*, then *P.status* is enabled to change to *working* if either P is the root of the BFS tree, or if *P.parent.status* = *working*, and if in addition, all children of P (in the BFS tree) have status *ready*, and no neighbor of P is resting; as shown in Figure 7a.

2. If P is working, then *P.status* is enabled to change to *finished* if all children of P are finished and all neighbors of P are either working or finished; as shown in Figure 7b.

3. If *P.status* = *finished*, then *P.status* is enabled to change to *resting* if either P is the root of the BFS tree, or if *P.parent.status* = *resting*, and if in addition, all children of P (in the BFS tree) have status *finished*, and no neighbor of P is working; as shown in Figure 7c.

4. If P is resting, then *P.status* is enabled to change to *ready* if all children of P are ready and all neighbors of P are either resting or ready; as shown in Figure 7d.



(a) P is ready and can start working.

(b) P is working and can finish.

(c) P is finished and can rest.

(d) P is resting and can get ready.

Figure 7: Normal Status Changes.

Algorithm 2 shows the code of SSBR, which is a self-stabilizing emulation of NSSBR. The algorithm takes inputs variables $P.parent$, $P.leader$, and $P.level$, which, if correct, describe a rooted breadth-first search (BFS) tree of the network, where $P.leader$ is the ID of the root process, and $P.level$ is the hop-distance from P to the root. (Note that the BFS tree is defined using hop-distance, instead of the weighted distance given as part of the specification of the best reachable problem.) SSBR also takes as inputs the function Θ which we are trying to optimize, as well as the order relation "$\succ$" on values of Θ. The sole output variable of SSBR is $P.stable\_best$. Although SSBR runs forever, the value of $P.stable\_best$ is eventually equal to the output of the best reachable problem required by the problem specification.

The local variables of SSBR are $P.status$, $P.best$, $P.dist$, and $P.span$. If the input variable of SSBR are correct, then SSBR will repeat a status wave cycle endlessly. The cycle consists of a broadcast *working* wave, a convergecast *finished* wave, a broadcast *resting* wave, and finally a convergecast *ready* wave. The *ready* wave initializes the local variable of SSBR to match the initial values of the variables of NSSBR, and while a process is working, it emulates the actions of NSSBR. When all processes have completed the emulation of NSSBR, the *finished* wave moves up the tree, followed by the *resting* wave, which then sets $P.stable\_best$ to $P.best$ for all P.

Because of arbitrary initialization, it could happen that $P.stable\_best$ is given the wrong value. But if at least one full status wave cycle has been completed, the value of $P.best$ will be correct at the time the *resting* wave reaches P. Subsequent status wave cycles will not change the value of $P.stable\_best$, although the value of $P.best$ will change endlessly.

If the input variables fail to specify a BFS tree, then the values of $P.stable\_best$ could be set to the wrong values many times. However, in that case, one of the processes will detect a *local error* in the BFS tree, and will stop executing actions of SSBR. This "freezing" of that single node will cause SSBR to eventually deadlock. If, at a future time, the input values of SSBR are correct, the deadlock be broken, and SSBR will proceed to compute its output correctly.

Algorithm 2: SSBR $(parent, leader, level, \Theta, \succ; stable\_best)$

```
1: for all P do
2:   loop {forever}
3:     if P is locally correct then {P cannot detect that the BFS tree is incorrect}
4:       if P.status is incompatible with P.parent.status then
5:         P.status ← P.parent.status
6:       else if P is ready then
7:         if P is a root or P.parent is working then
8:           if all children of P are ready and no neighbor of P is resting then
9:             P.status ← working
10:          end if
11:        end if
12:      else if P is working then
13:        if P.span > k then {P.best should now be the final value}
14:          if all children of P are finished and all neighbors of P are working
                                                    or finished then
15:            P.status ← finished
16:          end if
17:        else if P can detect that the loop invariant does not hold then
18:          P.span ← k + 1 {short-circuit the computation of P.best}
19:        else if ∀Q ∈ N_P : ((Q.best ⪰ P.best) ∨ (P.dist + w(P,Q) > k)) ∧
                 (w(P,Q) + Q.span > P.span) then {iterate the loop of NSSBR
           }
20:          if ∃Q ∈ N_P : Q.best ≻ P.best and Q.dist + w(P,Q) = P.span then
21:            P.best ← max_≻ {Q.best : Q ∈ N_P and Q.dist + w(P,Q) = P.span}
22:            P.dist ← P.span
23:          end if
24:          P.span ← min { min {X.span + w(P,X) : X ∈ N_P}
                          { min {X.dist + w(P,X) : X ∈ N_P and X.best ≻ P.best}
25:        end if
26:      else if P is finished then
27:        if P.span ≤ k then
28:          P.span ← k + 1
29:        else if P is a root or P.parent is resting then
30:          if all children of P are finished and no neighbor of P is working then
31:            P.stable_best ← P.best
32:            P.status ← resting
33:          end if
34:        end if
35:      else if P is resting then
36:        if all children of P are ready and all neighbors of P are resting
                                                    or ready then
37:          P.best ← P.Θ
38:          P.dist ← 0
39:          P.span ← min {w(P,Q) : Q ∈ N_P}
40:          P.status ← ready
41:        end if
42:      end if
43:    end if
44:  end loop
45: end for
```

4.2 Proof of Correctness of SSBR

Lemma 4.2 *Suppose e is a partial execution of* SSBR*, and suppose that during that partial execution, no input value changes. Then, during e, each process P executes a status correction action only finitely many times.*

Proof: By induction on $P.level$. If $P.level = 0$, then either P is not locally correct, in which case P cannot execute at all, or P is a root, in which case its status cannot be incompatible with its parent's status, since it is its own parent, and thus it cannot execute a status correction action.

Suppose $P.level > 0$. If P is not locally correct, then P cannot execute at all. Otherwise, let $Q = P.parent$. By the inductive hypothesis, there will be a configuration γ after which Q will not execute any status correction action. If $P.status$ is incompatible with $Q.status$, at γ, then Q is not enabled to change its status, while P is enabled to execute a status correction action, and cannot execute any other action first.

If P executes that status correction action, then no subsequent action by either P or Q can cause $P.status$ to become inconsistent with $Q.status$, and hence P will execute no further status correction action. □

Lemma 4.3 *Suppose e is a partial execution of* SSBR*, and suppose that during that partial execution, no input value changes, and there is one process P that never changes its status. Then, if $Q \in \mathcal{N}_P$, $Q.status$ changes at most three times during e.*

Proof: Without loss of generality, by Lemma 4.2, no process executes a status correction action during e. Suppose $Q.status$ changes infinitely often. Then $Q.status$ must follow the cycle $\cdots \rightarrow$ *working* $\rightarrow$ *finished* $\rightarrow$ *resting* $\rightarrow$ *ready* $\rightarrow$ *working* $\rightarrow \cdots$. Whatever the value of $P.status$, there is one value that $Q.status$ cannot change to. If P is working, then $Q.status$ cannot change to *resting*; if P is finished, then $Q.status$ cannot change to *ready*; if P is resting, then $Q.status$ cannot change to *working*; and if P is ready, then $Q.status$ cannot change to *finished*. Thus, Q cannot change its *status* more than three times, contradiction. □

Lemma 4.4 *Suppose e is a partial execution of* SSBR*, and suppose that during that partial execution, no input value changes, and there is one process that does not change its status. Then e is finite.*

Proof: Without loss of generality, by Lemma 4.2, no process executes a status correction action during e. Let P be the process that never changes its *status* during e.
Claim A: Every process P changes *status* only finitely many times during e. *Proof* (of Claim A): Let Q be the process that never changes its *status*. We prove the claim by induction on the hop distance to Q. If $P = Q$, we are done. Otherwise, P has a neighbor R which is on the minimum hop-distance path to Q. By the inductive hypothesis, $R.status$ changes finitely many times. Let γ be a configuration of e after which $R.status$ does not change. By Lemma 4.3, $Q.status$ can change at most three times after γ, and hence only finitely many times altogether during e. □

We now continue the proof of Lemma 4.4. By Claim A, after some configuration of e, no value of *status* will change. If P is not working, then P cannot execute any action. If P is working, it can execute at most finitely many actions, since either $P.dist$ or $P.span$ increases during each action. Thus, e is finite. □

Lemma 4.5 *If the BFS tree is correct, then some process is enabled to execute an action of* SSBR*.*

Proof: By Lemma 4.1, every process is locally correct. Assume that $P.status$ is not inconsistent with $P.parent.status$ for any P, since otherwise P is enabled to execute a status correction, and we are done.

Let R be the root of the BFS tree. If R is ready, then all processes are ready, and thus R is enabled to execute Line 9 of the code. If R is finished, then all processes are finished, and thus R is enabled to execute Lines 31 and 32 of the code.

If R is resting, then all processes are finished, resting, or ready. If there exist finished processes, pick a finished node P of minimum level. Then $P.parent$ is resting, and all children of P are finished; thus P is enabled to execute Lines 31 and 32 of the code. If there does not exist a finished process, pick P to be a resting process of maximum level. Then all children of P are ready, and thus P is enabled to execute Lines 37–40 of the code.

If R is working, then all processes are ready, working, or finished. If there exist ready processes, pick a ready node P of minimum level. Then $P.parent$ is working, and all children of P are ready; thus P is enabled to execute Line 9 of the code. If there does not exist a ready process and there exists a finished process P such that $P.span \leq k$, then P is enabled to execute Line 28 of the Code. If all processes have $span > k$, then pick P to be the working process of maximum level. Then all children of P are finished, and P is enabled to execute Line 15 of the code.

The remaining case is that all processes are either working or finished, all finished processes have $span > k$, and at least one working process has $span \leq k$. By Lemma 3.3, there exists some working process P which satisfies either the condition given in Line 17 or the condition given in Line 19 of the code, and is thus enabled to change at least one of its values. □

Lemma 4.6 *If e is an execution of* SSBR *during which the inputs do not change and the BFS tree is correct, then*

(a) *each process changes status infinitely often during e, and*

(b) *after finitely many steps the values of stable_best stabilize to a solution of the Best Reachable problem.*

Proof: By Lemma 4.5, e is infinite. By Lemma 4.3, each process changes *status* infinitely often during e.

Let R be the root process. By Lemma 4.2, we can pick a configuration γ_1 of e after which no process executes a status correction. By Lemma 4.3, there is a configuration γ_2 of e at which R is finished. Since status correction is not enabled, all processes are finished. Similarly, pick a configuration γ_3 of e after γ_2 at which all processes are ready, a configuration γ_4 of e after γ_3 at which all processes are finished. And finally a configuration γ_5 of e after γ_4 at which all processes are ready.

Between γ_2 and γ_3, every process executes Lines 37–39 of the code, and thus, at γ_3, $P.best = P.id$, $P.dist = 0$, and $P.span = \min\{w(P,Q) : Q \in \mathcal{N}_P\}$ for all P. Thus, the loop invariant holds at γ_3.

Between γ_3 and γ_4, SSBR emulates NSSBR, and hence at γ_4, $P.best$ is the best value of $Q.\Theta$ among all Q within distance k of P, for all P, by Theorem 3.4. Then, by the time the execution reaches γ_5, all processes will have executed Line 31 of the code, and the output variables of SSBR will be correct. □

4.3 An example computation

In Figure 8, we show an example computation of SSBR for "$\succ$" equals "$<$", "Θ" equals "*id*" and $k = 30$. 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, we show $P.dist$, followed by a colon, followed by $P.best$, followed by a colon, followed by $P.span$. In this example we consider that we start from a clean state (Figure 8(a)) and that each node is in a *ready* state (we do not deal with the other states in this example). Below each oval is shown the line of SSBR the process is enabled to execute (none if the process is disabled). An arrow from P to Q indicates that P prevents Q from executing due to conditions line 19.

In Figure 8, we show synchronous execution of SSBR. The result would have been the same with an asynchronous execution, but using synchrony makes the example easier to understand.

Consider the process L. Initially it is enabled to execute lines 21, 22 and 24 (subfigure (a)). It will, after the first execution (subfigure (b)), find the value of the smallest ID within a distance of $L.span = 7$, which is D, and will at the same time update its *dist* value to $L.span$, and $L.span$ to $D.span + w(L,D) = 6 + 7 = 13$. As during this step, L has updated its *span* value, $D.span$ is an underestimate of the real *span*, thus D is now enabled to execute line 24 to correct this value. The idea behind the *span* variable, is to prevent the process from searching a minimum ID at a

distance greater than *span*. 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 lower than *span* 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).

SSBR halts when $P.span > k$ for all P (subfigure (i)). In the final step every P knows the process of minimum ID at a distance no greater than k, and $P.dist$ 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. Hence, we need to have a second instance of SSBR that runs with "$\succ$" equal to ">" and "Θ equal to "$minid$" to corrects this; it allows the information that a process P was elected a clusterhead to flow back to P.

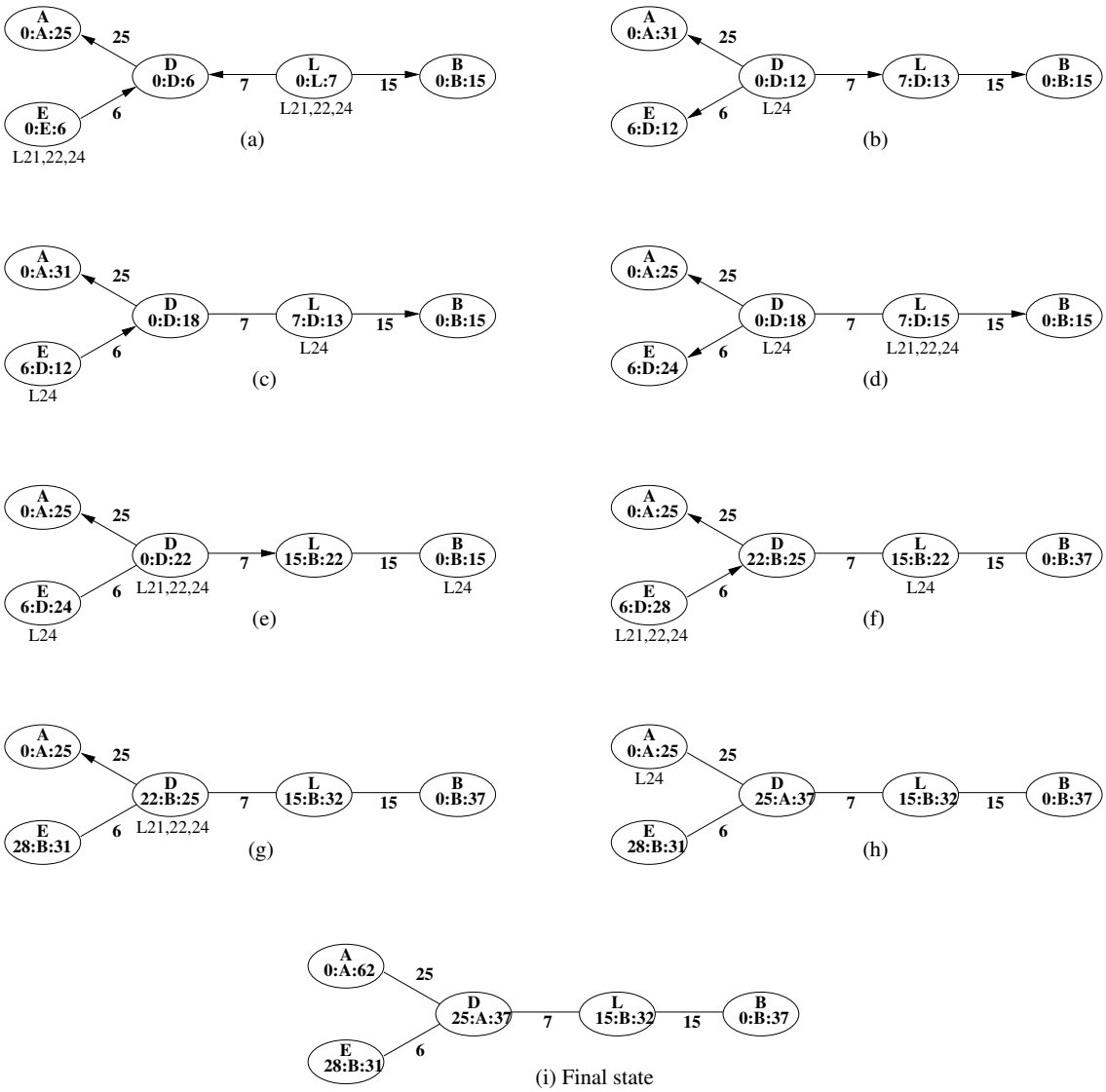


Figure 8: Example computation of SSBR for $k = 30$, "$\succ$" equals "$<$" and "Θ" equals "id".

5 Composing Self-Stabilizing Algorithms Under the Unfair Daemon

In this section, we consider the problem of combining distributed algorithms. This problem is not entirely trivial; for example, what we have discovered is that a naive combination of self-stabilizing algorithms might not be self-stabilizing.

We define a *partial execution* of a distributed algorithm A to be a sequence of configurations such that, other than the first one, each follows from its predecessor by one or more processes executing an action of A.

We define an *execution* of A to be a partial execution which is either infinite or ends at a configuration where no process is enabled.

1. We say that an execution is *unfair* if it is infinite and if there is some process that executes only finitely many times and is continuously enabled from some point on.

2. We say that an execution is *weakly fair* if it is not unfair. We say that A is *weakly fair* if every execution of A is weakly fair.

3. We say that an execution is *strongly fair* if it is either finite, or there is no process that executes only finitely many times. We say that A is *strongly fair* if every execution of A is strongly fair.

Note: a strongly fair execution is also weakly fair, and a strongly fair algorithm is also weakly fair.

Lemma 5.1 *Every distributed algorithm has a weakly fair execution.*

Proof: At each step, select the set of all enabled processes. □

Question: Is it true that every distributed algorithm has a strongly fair execution? Answer: No.

The Daemon The scheduler (daemon) chooses an execution of the algorithm A.

- We say that a daemon is *weakly fair* if it always chooses a weakly fair execution.

- We say that the daemon is *unfair* if it can choose any execution.

We say that a distributed algorithm A *works under the weakly fair daemon* if every weakly fair execution of A has whatever properties are required in the problem specification.

We say that a distributed algorithm A *works under the unfair daemon* if every execution of A has whatever properties are required in the problem specification.

Lemma 5.2 *If A is a weakly fair distributed algorithm, then A works under the unfair daemon if and only if A works under the weakly fair daemon.*

Input Variables Normally, input variables are never discussed. However, in most cases, a distributed algorithm has variables that never change their values. We can call these *input variables.* In the literature, it seems to always be assumed that the input variables are constant during the execution of an algorithm.

But what if we want to combine algorithms, so that the input variables of the second module (algorithm) are computed by the first module?

1. An *input variable* of an algorithm A is a variable that is used by A but is never changed by A. We call the vector of all input values of all nodes the *input configuration.*

2. The usual definition of an algorithm S being *self-stabilizing* is that, given that the input configuration is correct and never changes, the network will eventually be in a legitimate configuration.

   Of course, each problem specification gives a definition of what it means for an input configuration to be correct, and what it means for a configuration to be legitimate. We will assume that if the configuration is legitimate, the input configuration is correct.

   (a) S is self-stabilizing under the unfair daemon if every execution where the input configuration is correct and never changes is eventually in a legitimate configuration.

   (b) S is self-stabilizing under the weakly fair daemon if every weakly fair execution where the input configuration is correct and never changes is eventually in a legitimate configuration.

Whether an algorithm A is weakly or strongly fair depends on the definition of a configuration of the algorithm. The normal definition of configuration allows any process to have any values of its variables, but the values of its constants are uniquely specified. But what about input variables? Since this issue is not normally even discussed in the literature, we need to clarify it for our purposes.

We will adopt the definition that an algorithm A can have constants, whose values are given in the problem specification, and could also have input variables, which could take on a range of values, but whose values cannot be changed by A. (If we never combine algorithms, the distinction between these is moot.) We then define a partial execution of A to be a sequence of configurations where the input variables can be initialized to have any values in their range, and where during each step one or more processes execute an action of A. Thus, the input variables are unchanged throughout the entire sequence.

We summarize the classification of the variables of an algorithm A.

1. Constants, which have values given in the problem specification.

2. Input variables. Each input variable has a range of possible values. There is a defined set of input configurations call *correct* input configurations. Input variables cannot be changed by A.

3. Local variables. Variables which can be changed by A, and are used only for the internal computations of A.

4. Output variables. Variables which can be changed by A, and which are intended to be read by some agent or algorithm outside A.

We also require that, if the configuration is legitimate, that no output variable can change. We say that the output variables are *stable.*

For example, in the third module of K-CLUSTERING, which we fully describe in Section 6.1, k and $P.id$ are constant, $P.parent$, $P.leader$, $P.level$, and $P.minid$ are input variables, $P.dist$ and $P.span$ are local variables, and $P.maxminid$ is an output variable.

Combining Algorithms Suppose that A and B are strongly fair self-stabilizing algorithms on the same network. That means that each process P has all the variables of both A and B. We classify those variables as follows.

1. Constants.

2. Input variables of A that are not visible to B. These cannot be changed.

3. Variables that are input variables of both A and B. These cannot be changed.

4. Input variables of B which are not visible to A. These cannot be changed.

5. Local variables of A.
6. Local variables of B.
7. Output variables of A which are input variables of B. These can be changed by A but not by B.
8. Output variables of A which are not input variables of B.
9. Output variables of B.

The combination algorithm $Combine(A, B)$ is defined as follows.

1. The input variables of $Combine(A, B)$ are defined to be the variables of classes 2, 3, and 4 above.
2. The output variables of $Combine(A, B)$ are defined to be the variables of classes 8, and 9 above, possibly together with some variables of class 7.

P is enabled to execute an action of $Combine(A, B)$ if and only if P is either enabled to execute an action of A or enabled to execute an action of B.

If P executes an action of $Combine(A, B)$, then P executes both an action of A and an action of B, if both are enabled, otherwise, it executes one or the other.

Correctness and legitimacy for the three algorithms, A, B, and $Combine(A, B)$, must be defined to satisfy the following conditions.

1. If the input configuration of $Combine(A, B)$ is correct, then the input configuration of A is correct.
2. If the input configuration of $Combine(A, B)$ is correct and the configuration of A is legitimate, then the input configuration of B is correct.
3. A configuration of $Combine(A, B)$ is legitimate if and only if the configurations of both A and B are legitimate.

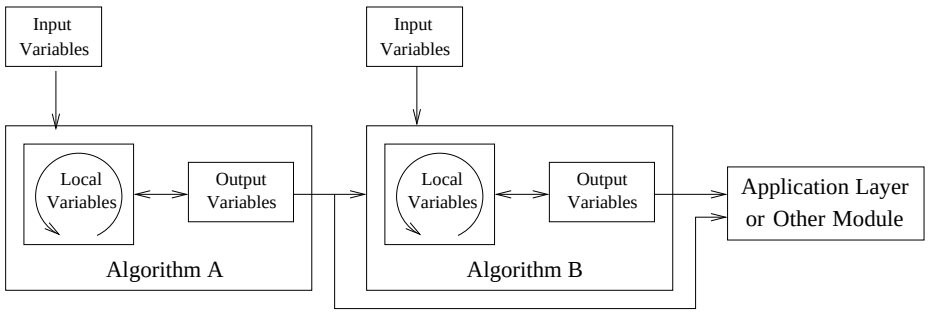


Figure 9: $Combine(A, B)$

Lemma 5.3 *If both A and B are strongly fair and self-stabilizing, then $Combine(A, B)$ is strongly fair and self-stabilizing.*

Proof: We first prove that $Combine(A, B)$ is strongly fair.

Let $E = \gamma_0, \gamma_1, \ldots$ be an execution of $Combine(A, B)$. We write $\gamma_i = (\alpha_i, \beta_i)$, where α_i is a configuration of A and β_i is a configuration of B. Let $E_A = \alpha_0, \alpha_1, \ldots$ which might not be a partial

execution of A because consecutive configurations could be equal. Let E'_A be the partial execution of A obtained from E_A by eliminating configurations which are the same as their predecessors.

Case I: E is finite. In this case, we are done.

Case II: E'_A is infinite. Then, since A is strongly fair, every process executes infinitely many actions of A in E'_A, and hence infinitely many actions of $Combine(A, B)$ in E.

Case III: E is infinite and E'_A is finite. Pick T such that A does not execute beyond the T^{th} step of E, that is, $\alpha_i = \alpha_T$ for all $i > T$. Then $E_B = \beta_T, \beta_{T+1}, \ldots$ is an execution of B. Since B is strongly fair, each process executes infinitely many actions of B during E_B, and thus infinitely many actions of $Combine(A, B)$ during E.

We now prove that $Combine(A, B)$ is self-stabilizing. We use the same notation as above.

Assume that the input configuration of $Combine(A, B)$ is correct at γ_0. Then, the input configuration of A is correct at α_0. We claim that E'_A is an execution of A.

Case I: E is finite. Then, at the last configuration of E, no process is enabled to execute an action of $Combine(A, B)$, and hence no process is enabled to execute an action of A. Thus, E'_A is an execution of A.

Case II: E'_A is infinite. Then E'_A is an execution of A.

Case III: E is infinite and E'_A is finite. Suppose that E'_A is not an execution of A. Then, at α_T, there is some process P which is enabled to execute an action of A, but that process is never selected during the remainder of the sequence E. This contradicts the fact that E contains infinitely many actions of every process.

This completes the proof of the claim that E'_A is an execution of A. It follows that α_i is eventually a legitimate configuration of A.

Continuing the proof of Lemma 5.3, we now prove that $Combine(A, B)$ is self-stabilizing.

Assume that the input configuration is correct. Since E'_A is an execution of A, there is some T such that α_T is a legitimate configuration of A. Thus, for any $i \geq T$, the input variables of B are correct and are the same as at γ_T. Let E'_B be the sequence of configurations of B, starting at β_T, with duplicates removed. Then E'_B is an execution of B, and thus will eventually be in a legitimate configuration of B at which the output variables of B are stable.

In conclusion, eventually both A and B will be in legitimate configurations, and the output variables of both will be stable, and we are done. □

6 The K-CLUSTERING Algorithm.

We now define our combined algorithm, K-CLUSTERING as the combination of four algorithms, as shown in Figure 10. Each of these algorithms is self-stabilizing and strongly fair, as defined in Section 5. These algorithms are SSLEBFS, two copies of SSBR, and SSCLUSTER, which we define below.

- A strongly fair and self-stabilizing algorithm, SSLEBFS, which elects a leader and which constructs a BFS tree rooted at that leader. This algorithm outputs variables $P.parent$, the pointer to the parent of P in the BFS tree, $P.leader$, the ID of the elected leader, and $P.level$, the distance from P to the leader. Any algorithm which meets those conditions can be used, such as the one given in [6].

- A copy of SSBR which uses the outputs of SSLEBFS as inputs, and which also uses $P.id$ for Θ and the relation "$<$" for the order relation "$\succ$." The output of this module is the variable $P.minid$, whose correct value is $\min\{Q.id : w(P, Q) \leq k\}$.

- A copy of SSBR which uses the outputs of SSLEBFS as inputs, and which uses $P.minid$ for Θ and the relation "$>$" for the order relation "$\succ$." Thus, the input variables of the third module consist of the output variables of the first two modules. The output of this module is the variable $P.maxminid$, whose correct value $\max\{Q.minid : w(P, Q) \leq k\}$, providing all values of $Q.minid$ are correct.

- The algorithm SSCLUSTER given as Algorithm 4 below, which uses *P.maxminid* as its input variable, and has output variables *P.cl_head*, *P.cl_level*, and *P.cl_parent*.

Algorithm 3 is obtained by applying the *Combine* construction, given in Section 5, three times; we first combine SSLEBFS with one copy of SSBR, we then combine that algorithm with a second copy of SSBR, and finally combine that result with the algorithm SSCLUSTER. As specified in Section 5, our construction requires that, when selected, a process is required to execute an action of each module where that is possible.

Since each of the four modules is strongly fair and self-stabilizing, then K-CLUSTERING is also strongly fair and self-stabilizing, by repeated application of Lemma 5.3. Eventually, the configuration of SSLEBFS will stabilize. After that, the configuration of the first copy of SSBR will stabilize, and after that the configuration of the second copy of SSBR will stabilize. Finally, the configuration of SSCLUSTER will stabilize.

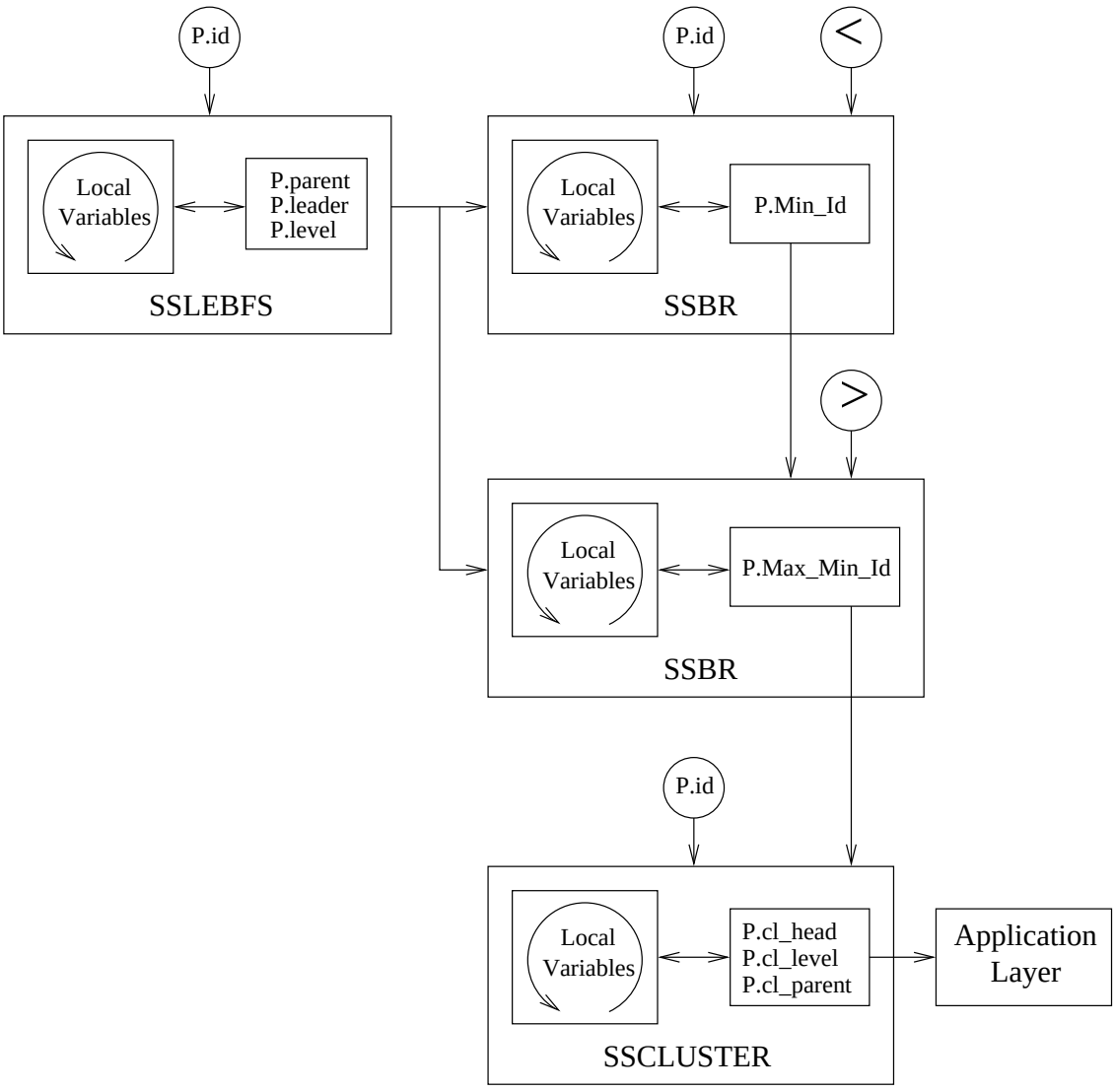


Figure 10: K-CLUSTERING is the combination of four strongly fair self-stabilizing algorithms.

Algorithm 3: K-CLUSTERING (; *cl_head*, *cl_level*, *cl_parent*)

```
1: for all P do
2:   loop {forever}
3:     if enabled to do so then
4:       Execute an action of SSLEBFS (; parent, leader, level)
5:     end if
6:     if enabled to do so then
7:       Execute an action of SSBR (parent, leader, level, id, <; minid)
8:     end if
9:     if enabled to do so then
10:      Execute an action of SSBR (parent, leader, level, minid, >; maxminid)
11:    end if
12:    if enabled to do so then
13:      Execute an action of SSCLUSTER (maxminid; cl_head, cl_level, cl_parent)
14:    end if
15:  end loop
16: end for
```

6.1 The Module SSCLUSTER

Algorithm 4: SSCLUSTER (*maxminid*; *cl_head*, *cl_level*, *cl_parent*)

```
1: for all P do
2:   loop {forever}
3:     if P.maxminid = P.id then
4:       if (P.cl_level ≠ 0 or P.cl_head ≠ P.id or P.cl_parent ≠ P) then
5:         P.cl_level ← 0
6:         P.cl_head ← P.id
7:         P.cl_parent ← P
8:       end if
9:     else
10:      if ∃Q ∈ N_P : w(P,Q) + Q.cl_level ≤ k then
11:        level ← min {w(P,Q) + Q.cl_level : Q ∈ N_P}
12:        head ← min {Q.cl_head : Q ∈ N_P
                                  and w(P,Q) + Q.cl_level = level}
13:        parent ← min {Q ∈ N_P : w(P,Q) + Q.cl_level = level
                                  and Q.cl_head = head}
14:        if (P.cl_level ≠ level or P.cl_head ≠ head or P.cl_parent ≠ parent)
           then
15:          P.cl_level ← level
16:          P.cl_head ← head
17:          P.cl_parent ← parent
18:        end if
19:      else if P.cl_level ≠ k + 1 then
20:        P.cl_level ← k + 1
21:      end if
22:    end if
23:  end loop
24: end for
```

Algorithm 4 updates variables *P.cl_level*, *P.cl_head* and *P.cl_parent*. If P is a clusterhead, then the variables get respectively values 0, *P.id* and P. Otherwise *P.cl_level* gets the weight of the shortest path from P to the closest clusterhead, *P.cl_head* receives the ID of the closest clusterhead (the lowest ID when ties need to be broken), and finally *P.cl_parent* gets the neighbor of P that is on the shortest path from P to its clusterhead.

6.2 Proof of Correctness

In order to make use of Lemma 5.3, we must first prove that SSBR and SSCLUSTER have the needed properties.

Lemma 6.1 SSBR *is strongly fair and self-stabilizing.*

Proof: We first prove that SSBR is strongly fair. Suppose that the input of SSBR does not change. We need to prove that either SSBR stops executing, or that every process executes infinitely often.

If the input is correct, then, by Lemma 4.6(a), every process executes an action of SSBR infinitely often.

If the input is incorrect, then, by Lemma 4.1, there is some process P which is not locally correct. Then P will not execute any action of SSBR. By Lemma 4.4, there can be at most finitely many remaining executions of actions of SSBR.

By Lemma 4.6(b), and the fact that the correct values of $P.stable\_best$ are unique, SSBR is self-stabilizing. □

Given an input configuration of SSCLUSTER, we define a process P to be a *clusterhead* if $P.maxminid = P.id$. We define a *correct* input configuration of SSCLUSTER to be a configuration where every process is within distance k of some clusterhead.

A legitimate configuration of SSCLUSTER is then defined to be a configuration where

1. The input configuration is correct.
2. If P is any process, then $P.cl\_level$ is the distance to the nearest clusterhead.
3. If P is any process, then $P.cl\_head$ is the ID of the nearest clusterhead. In case of a tie, $P.cl\_head$ is the smallest choice.
4. If P is a clusterhead, then $P.cl\_parent = P$. Otherwise, $P.cl\_parent$ is the neighbor of P of smallest ID that lies on a shortest path from P to $P.cl\_head$.

Note that, for any given correct input configuration, there is exactly one legitimate configuration of SSCLUSTER.

Lemma 6.2 *For any given input configuration, every execution of* SSCLUSTER *is finite.*

Proof: Let e be an execution of SSCLUSTER. During this execution, the values of the input variables of SSCLUSTER do not change, although they may not be correct. Our proof is by contradiction; suppose that e is infinite.

Let $\mathcal{B}$ be the set of process that execute actions of SSCLUSTER only finitely many times during e. Without loss of generality, each member of $\mathcal{B}$ executes no action of SSCLUSTER, since we can start e at the first configuration after all executions of the members of $\mathcal{B}$.

Case I: $\mathcal{B} = \emptyset$.

Let $L = \min\{P.cl\_level\}$, and let $\mathcal{L} = \{P : P.cl\_level = L\}$. When a process $P \in \mathcal{L}$ executes an action of SSCLUSTER, $P.cl\_level$ must increase. Thus, after each member of $\mathcal{L}$ has executed at least once, L must increase. Eventually, $L = k + 1$, which means that no process can execute, contradiction.

Case II: $\mathcal{B} \neq \emptyset$. For all P, let

$$\Lambda(P) = \begin{cases} P.cl\_level & \text{if } P \in \mathcal{B} \\ \min\{w(P,Q) + \Lambda(Q) : Q \in \mathcal{N}_P\} & \text{otherwise} \end{cases}$$

$$\overline{\Lambda(P)} = \min\{k+1, \Lambda(P)\}$$

We claim that if $P \notin \mathcal{B}$ and $Q \in \mathcal{N}_P$, and if eventually $Q.cl\_level \leq \ell$, then eventually $P.cl\_level \leq w(P,Q) + \ell$. Let γ be a configuration after which $Q.level \leq \ell$ always holds. The next step where P executes, $P.cl\_level \leq w(P,Q) + \ell$, and $P.cl\_level$ can never decrease below that value.

It follows that $P.cl\_level \leq \Lambda(P)$ for all P, by strong induction on $\Lambda(P)$. If $P \in \mathcal{B}$, the statement holds trivially. Otherwise, there is some $Q \in \mathcal{N}_P$ such that $\Lambda(P) = w(P,Q) + \Lambda(Q)$. By the inductive hypothesis, eventually $Q.level \leq \Lambda(Q)$, and thus eventually $P.cl\_level \leq \Lambda(Q) + w(P,Q) = \Lambda(P)$.

Thus, without loss of generality, $P.cl\_level \leq \Lambda(P)$ for all P and all configurations of e. We now claim that eventually $P.cl\_level \geq \overline{\Lambda(P)}$ for all P. Let $\mathcal{L} = \left\{P : P.cl\_level < \overline{\Lambda(P)}\right\}$, and let $L = \min\{P.cl\_level : P \in \mathcal{L}\}$. If $\mathcal{L} = \emptyset$, the claim holds. Otherwise, $L \leq k$. Every $P \in \mathcal{L}$ is enabled to execute, and when it does execute, $P.cl\_level$ will increase. Thus, eventually, L will increase or $\mathcal{L}$ will become empty. Since L cannot exceed k, $\mathcal{L}$ will eventually be empty, and we have proved the claim.

We can now assume that $P.cl\_level = \overline{\Lambda(P)}$ for all $P \notin \mathcal{B}$ at all configurations of e. Each $P \notin \mathcal{B}$ can then execute at most once, contradicting the infinitude of e. □

Lemma 6.3 SSCLUSTER *is strongly fair and self-stabilizing.*

Proof: SSCLUSTER is strongly fair by Lemma 6.2. We need only show that it is self-stabilizing.

Assume that the input configuration of SSCLUSTER is correct and never changes. Let $\mathcal{A}$ be the set of clusterheads, namely $\{P : P.id = P.maxminid\}$. Since $P.maxminid$ does not change, $\mathcal{A}$ is fixed.

By Lemma 6.2, we can consider only the last configuration of SSCLUSTER, *i.e.,* a configuration γ where no process is enabled to execute an action of SSCLUSTER. We need only prove that γ is legitimate.

By way of contradiction, assume that γ is not legitimate. Let $Cl\_Level(P)$, $Cl\_Head(P)$ and $Cl\_Parent(P)$ be the correct values of $P.cl\_level$, $P.cl\_head$, and $P.cl\_parent$, *i.e.,* the values those variables must have in a legitimate configuration.

Case I: There is some process P such that $P.cl\_level > Cl\_Level(P)$. Choose that P which has the smallest value of $Cl\_Level(P)$. If $P \in \mathcal{A}$, then P is enabled to execute an action, contradiction. Otherwise, let $Q = Cl\_Parent(P)$. Since $Cl\_Level(Q) < Cl\_Level(P)$, the value of $Q.cl\_level$ is correct, and hence P is enabled to execute, since $w(P,Q) + Q.cl\_level < P.cl\_level$, contradiction.

Case II: Case I does not hold, and there is some process P such that $P.cl\_level < Cl\_Level(P)$. Choose that P which has the smallest value of $P.cl\_level$, and pick $Q \in \mathcal{N}_P$ such that $w(P,Q) + Q.cl\_level$ is minimum. If $w(P,Q) + Q.cl\_level \leq P.cl\_level$, then $Q.cl\_level > Cl\_Level(Q)$ and $Q.cl\_level < P.cl\_level$, contradiction. Otherwise, P is enabled to execute, contradiction.

Case III: $P.cl\_level = Cl\_Level(P)$ for all P, and there is some P for which either $P.cl\_head \neq Cl\_Head(P)$ or $P.cl\_parent \neq Cl\_Parent(P)$. Pick such a P where $P.cl\_level$ is minimum. If P is a clusterhead, then P is enabled to execute, contradiction. Otherwise, let $\mathcal{Q} = \{Q \in \mathcal{N}_P : P.level = w(P,Q) + Q.cl\_level$ By our choice of P, we know that all variables of Q are correct for all $Q \in \mathcal{Q}$. Thus, P will be enabled to execute in order to correct its values, contradiction. □

Applying Lemma 5.3 twice, we have:

Lemma 6.4 *Eventually* $P.minid = \min\{Q.id : w(P,Q) \leq k\}$ *and* $P.maxminid = \max\{Q.minid : w(P,Q) \leq k\}$ *for all* P.

We then obtain:

Lemma 6.5 *Eventually,* $P.id = P.maxminid$ *if and only if there is some process* Q *such that* $Q.minid = P.id$.

Lemma 6.5 is given in [7]. The proof is by contradiction. If $Q.minid = P.id$ then $w(P,Q) \leq k$ and $P.maxminid \geq Q.minid$. If $P.maxminid > Q.minid$, then for some R, we have $w(P,R) \leq k$ and $R.minid > P.id$, which contradicts the required correctness condition for $R.minid$.

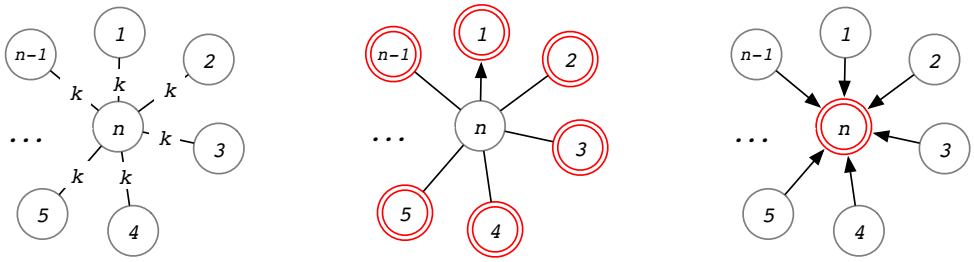
Let $\mathcal{A} = \{P : P.id = P.maxminid\}$, the set of clusterheads. By Lemma 6.5, we know that every process is within distance k of some member of $\mathcal{A}$. By the correctness of SSCLUSTER, and applying Lemma 5.3 once more, we know that K-CLUSTERING partitions the network into cluster, where each process joins the nearest clusterhead. Thus, K-CLUSTERING is correct.

Applying Lemma 5.3 thrice, we have that K-CLUSTERING is self-stabilizing, and works under the unfair daemon.

7 Theoretical bounds

7.1 Number of clusterheads

The algorithm can behave very badly compared to the optimal k-clustering, *i.e.,* the clustering with the lowest number of clusterheads. In fact, if OPT_G is the optimal number of clusterheads for a given graph G, K-CLUSTERING can return a solution with $(n-1)OPT_G$ clusterheads. An example of such a bad clustering is given on Figure 11: Figure 11a presents the initial graph, and Figures 11b and 11c show the solution returned by our algorithm and the optimal solution, processes with a doubled line are clusterheads, an arrow designates the parent.



(a) Initial graph (b) K-CLUSTERING (c) Optimal result

Figure 11: K-CLUSTERING worst case.

This problem arises because of the distribution of IDs. As our algorithm is comparison based, its solution is constrained by the distribution of the IDs among the processes. Take the same example as the one given on Figure 11a, but instead put the lowest ID, 1, on the central node. In this case our algorithm would find the optimal solution.

7.2 Number of rounds

We now present two theoretical bounds on the number of rounds required for the algorithm to return correct values.

7.2.1 The Chain Graph $G_{n,k}$

In Figure 12, 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 1 *If the algorithm runs on graph $G_{n,k}$, the convergence time is $\Theta(nk)$ rounds in the worst case.*

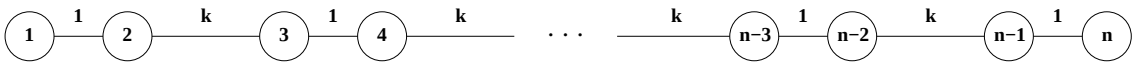


Figure 12: The Graph $G_{n,k}$

Proof: For sake of simplicity, we suppose that the processes start in a clean state, *i.e.,* all possible errors have been corrected, and $P.best = P.id$, $P.dist = 0$ and $P.span = \min\{w(P,Q) : Q \in \mathcal{N}_P\}$, note that in this graph for all Process P, $P.span = 1$. We only deal with the copy of SSBR in charge of computing $P.minid$.

Initially, only Process n is able to execute lines 21, 22 and 24 due to the condition line 9, and no other process is enabled. Thus, after one round, $n.best = n-1$, $n.dist = 1$ and $n.span = 2$; and no other process has changed its variables.

During the next $k-1$ steps, only Processes n and $n-1$ are enabled to alternatively execute line 24 to update their *span* variable. $n.span$ and $(n-1).span$ are only able to increase by 2 at each step.

Once $(n-1).span > k$, $n-1$ is enabled to execute lines 21 and 22, and set $(n-1).best = n-2$ and $(n-1).dist = k$. Then, Process $n-2$ is enabled to execute lines 21, 22 and 24, which starts a new cycle of $k-1$ rounds between $n-2$ and $n-3$ to update *span*.

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 execution of SSBR for computing $P.minid$. Hence the $\Theta(nk)$ bound. □

7.2.2 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 $\{1, \ldots n\}$.
- 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 13 presents an example of such a graph.

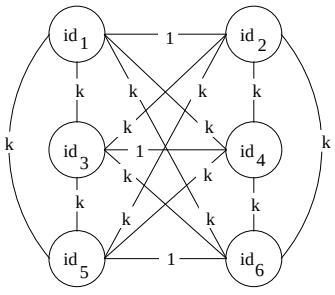


Figure 13: The Graph $R_{n,k}$.

Lemma 2 *If the algorithm runs on graph $R_{n,k}$, the convergence time is $O(n/2 \times k)$ rounds in the worst case.*

Proof: Starting from a clean configuration where:

- $i.status = READY$
- $i.best = i$
- $i.dist = 0$
- $i.span = 1$

it takes $n/2$ steps for processes to have the following values:

- $i.best = i$ if i is inferior, or $partner(i)$ if i is superior
- $i.dist = 0$ if i is inferior, or 1 if i is superior
- $i.span = 1$ if i is inferior, or 2 if i is superior

(in fact not exactly n/2 steps, but n/2 executions of lines 21,22, 24, but even if these steps do not occur consecutively, on the whole it will take n/2 steps for processes to pass through this configuration), then only 2 processes will be able to update their *span* variable, due to the condition $P.dist + w(P,Q) > k$ which is not true for most of the processes ($P.dist + w(P,Q) = k$), and $Q.best \geq P.best$. Hence, only the special pair with the highest id can update, and only one process is enabled at each step. And this until this special pair finds 1 as the best id, which takes k steps. We repeat this for the next special pair having the second highest id, and so on and so forth... Hence, as we have $n/2$ special pairs, on the whole it takes $O(n/2 \times k)$ rounds. □

8 Simulations

We designed a simulator to evaluate the performance of our algorithm[4]. 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).

We only present in this section a few simulations results: on an example graph (see Figure 14), and the results for graph $G_{n,k}$ and $R_{n,k}$.

8.1 Effect of the k value

Example Figure 14 We ran the simulator on the weighted graph illustrated in Figure 14. 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, hence the processes do not start in a clean state.

Figure 15 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 16 shows the number of rounds required to converge. This figure shows two kinds of runs: with an unfair daemon that holds a random process until no other process is able to execute, and with a fair daemon that selects every enabled process at every step. As can be seen, the number of rounds is far lower than the theoretical bound $O(nk)$, even with an unfair daemon.

[4]It can be found at http://graal.ens-lyon.fr/~bdepardo/down_files/k-clustering/k-clustering.bz2, the file also contains all the platforms and the results.

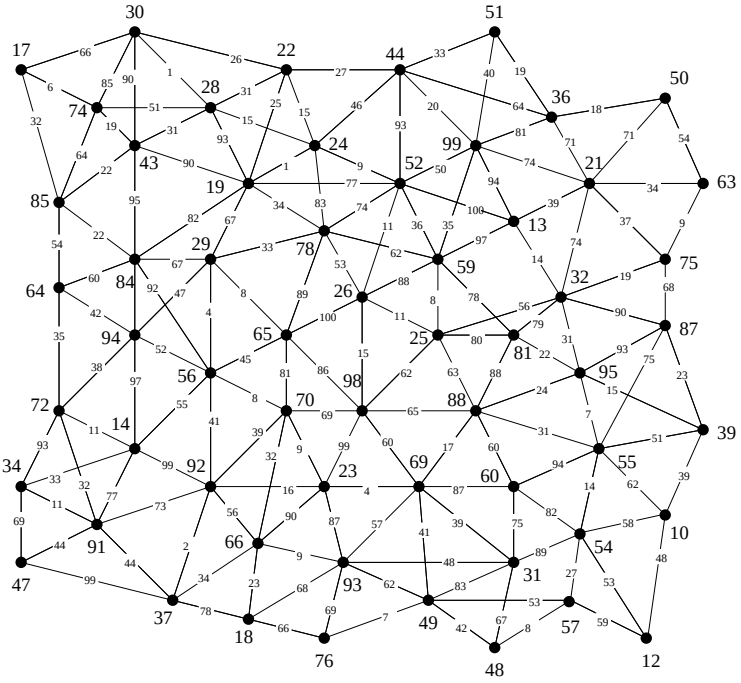


Figure 14: Example graph: *number of nodes* = 59, *diameter* = 282, *radius* = 163, weights between 1 and 100.

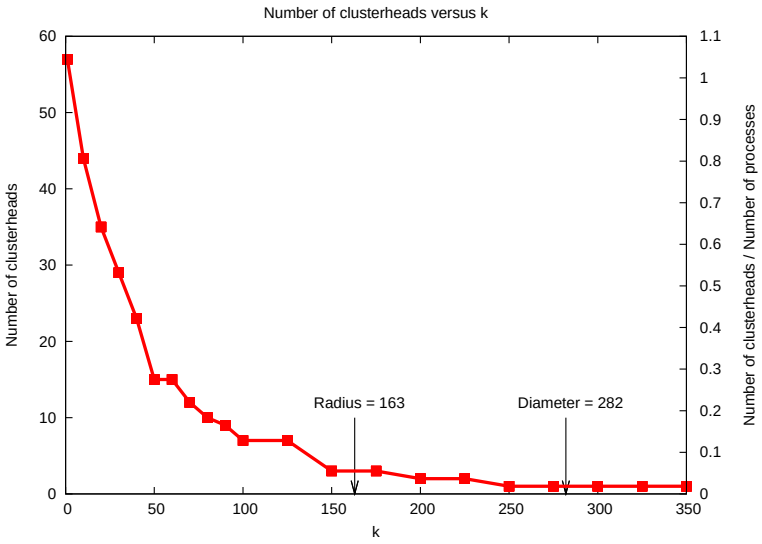


Figure 15: Number of clusterheads for graph Figure 14.

Random graphs We also generated 50 random graphs containing each 100 nodes, edges weight varied between 1 and 100. Figure 17, and 18 present respectively the number of clusterheads, and the number of rounds. As can be seen the number of rounds is far from the theoretical bound, and the number of clusterheads quickly decreases. Each type of points on the graphs represent a particular platform.

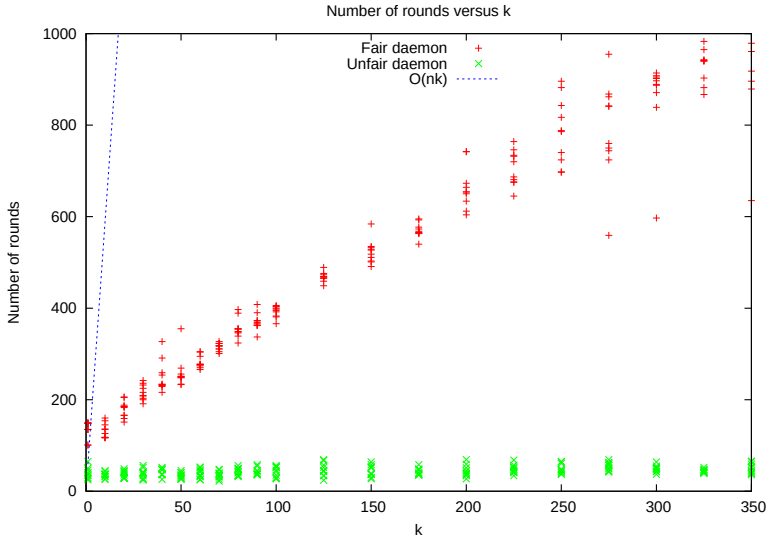


Figure 16: Number of rounds with fairness and unfairness, for the graph represented in Figure 14.

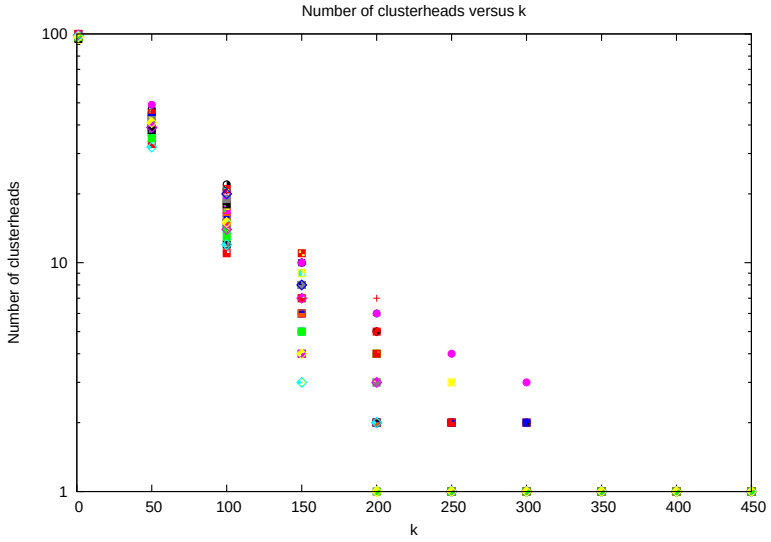


Figure 17: Random graphs: number of clusterheads.

8.2 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, apart from node n which connects itself to node $n-1$.

Figure 19 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 $O(nk)$.

Graph $R_{n,k}$ 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 20 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

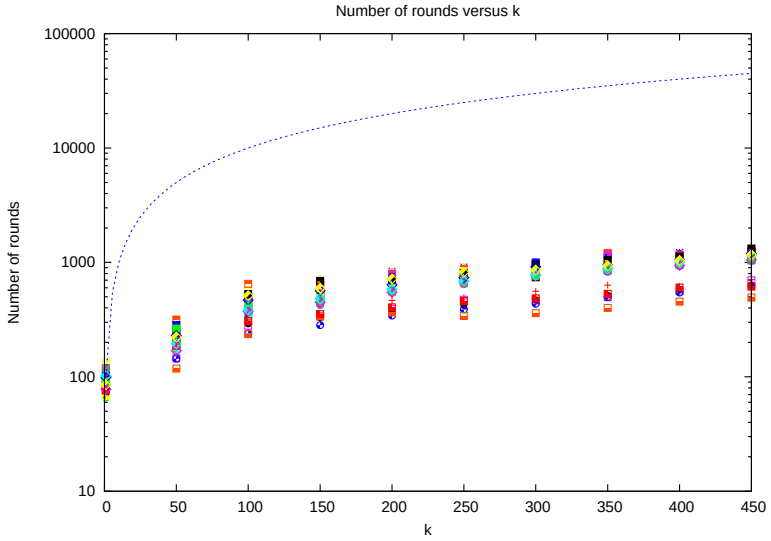


Figure 18: Random graphs: number of rounds.

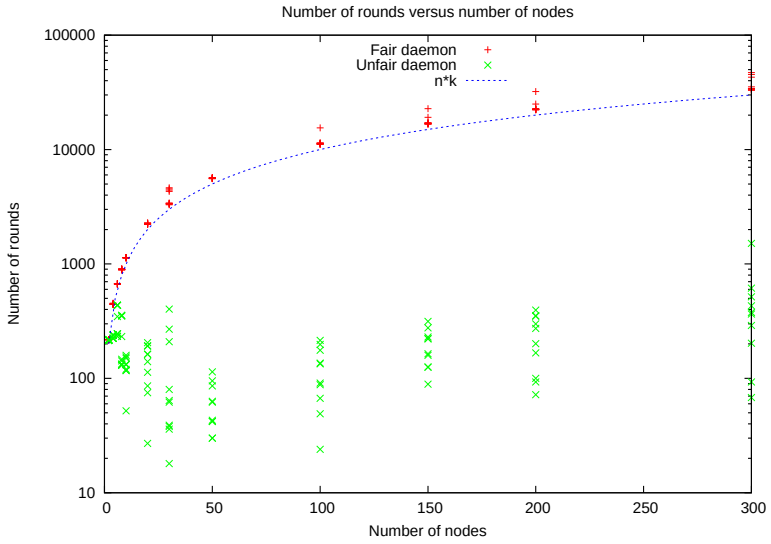


Figure 19: Number of rounds for graph $G_{n,k}$, represented in Figure 12.

bound $\Omega(\sqrt{n}k)$.

9 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 which dealt with unweighted graphs, or weights on the nodes, our algorithm deals with an arbitrary metric on the network, *i.e.,* weights on the links, and hence, is able to take into account more realistic communications' cost. K-CLUSTERING is the combination of four strongly fair self-stabilizing algorithms: SSLEBFS, SSBR and SSCLUSTER executes in $O(nk)$ rounds, and requires only $O(\log n + \log k)$ space per process. We also gave conditions under which

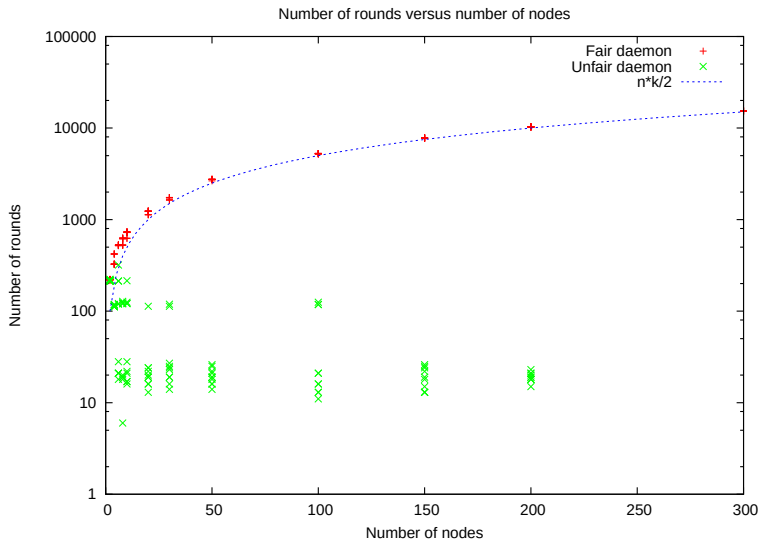


Figure 20: Number of rounds for graph $R_{n,k}$, represented in Figure 13.

the combination of self-stabilizing algorithms is also self-stabilizing.

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. Such a clustering can help for example to guaranty the latency experienced by messages in a network, *e.g.,* in each k-cluster.

10 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] S. Basagni. Distributed and mobility-adaptive clustering for multimedia support in multi-hop wireless networks. In *Vehicular Technology Conference, 1999. VTC 1999 - Fall. IEEE VTS 50th*, volume 2, pages 889–893 vol.2, 1999.

[3] S. Basagni. Distributed clustering for ad hoc networks. In *Parallel Architectures, Algorithms, and Networks, 1999. (I-SPAN '99) Proceedings. Fourth InternationalSymposium on*, pages 310–315, 1999.

[4] 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.

[5] A. K. Datta, S. Devismes, and L. L. Larmore. A self-stabilizing $o(n)$-round k-clustering algorithm. In *28th International Symposium on Reliable Distributed Systems (SRDS)*, Niagara Falls, New York, sept. 2009.

[6] A. K. Datta, L. L. Larmore, and P. Vemula. Self-stabilizing leader election in optimal space. In Saneep Kuklarni and Andre Schiper, editors, *10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 5340 of *Lecture Notes in Computer Science*, pages 109–123, Detroit, MI, nov. 2008. Springer.

[7] A. K. Datta, L. L. Larmore, and P. Vemula. A self-stabilizing O(k)-time k-clustering algorithm. *Computer Journal*, 2009.

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

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

[10] 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.

[11] 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.

[12] 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, *Algorithmic Aspects of Wireless Sensor Networks, Second International Workshop, ALGOSENSORS 2006, Venice, Italy, July 15, 2006, Revised Selected Papers*, volume 4240 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2006.

[13] C. Johnen and L. H. Nguyen. Robust self-stabilizing weight-based clustering algorithm. *Theor. Comput. Sci.*, 410(6-7):581–594, 2009.

[14] N. Mitton, A. Busson, and E. Fleury. Self-organization in large scale ad hoc networks. In *The Third Annual Mediterranean Ad Hoc Networking Workshop, MED-HOC-NET*, volume 4, June 2004.

[15] N. Mitton, E. Fleury, I. Guerin L., and S. Tixeuil. Self-stabilization in self-organized multihop wireless networks. In *ICDCSW '05: Proceedings of the Second International Workshop on Wireless Ad Hoc Networking (WWAN) ICDCSW'05*, pages 909–915, Washington, DC, USA, 2005. IEEE Computer Society.

[16] 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.

[17] 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.