J Grid Computing (2007) 5:361–378
DOI 10.1007/s10723-007-9069-5

A Local Facility Location Algorithm for Large-scale Distributed Systems

Denis Krivitski · Assaf Schuster · Ran Wolff

Received: 1 September 2005 / Accepted: 18 January 2007 / Published online: 25 April 2007

Abstract In a *facility location problem* (FLP) we are given a set of facilities and a set of clients, each of which is to be served by one facility. The goal is to decide which subset of facilities to open, such that the clients will be served at a minimal cost. In this paper we investigate the FLP in a setting where the cost depends on data known only to the clients. This setting typifies modern distributed systems: peer-to-peer file sharing networks, Grid systems, and wireless sensor networks. All of them need to perform network organization, data placement, collective power management, and other tasks of this kind. We propose a local and efficient algorithm that solves FLP in these settings. The algorithm presented here is extremely scalable, entirely decentralized, requires no routing capabilities, and is resilient to failures and changes in the data throughout its execution.

Keywords Local · Distributed · Data-mining · Large scale · Facility location · Grid · Peer-to-peer

D. Krivitski (✉) · A. Schuster
Department of Computer Science,
Technion – Israel Institute of Technology,
Technion City, Haifa 32000, Israel
e-mail: denisk@cs.technion.ac.il

A. Schuster
e-mail: assaf@cs.technion.ac.il

R. Wolff
University of Maryland, Baltimore County, Baltimore, MD, USA
e-mail: rwolff@mis.haifa.ac.il

1 Introduction

The facility location problem (FLP) deals with finding an optimal subset of facilities that will be open to serve clients. The set of open facilities should minimize the cost function—the cost incurred by serving each client and that incurred by opening each facility. This clients-facilities metaphor can be used to model many practical optimization problems occurring in large scale distributed systems. Examples of such systems are peer-to-peer networks, Grid computing, and wireless sensor networks.

Consider a scenario that may occur in large Grids. These are used for the execution of user jobs on thousands of computers. Grid systems have many resources that are shared by many, or all, of the jobs. These include the job queue, the resource collector, and possibly large reference data files that are used by many jobs. All these resources can be replicated so as to allow better scalability, reliability, and response time. However, such replication does not come for free. It requires both costly synchronization among replicas

and the allocation of valuable resources (e.g., computers with large disks that can accommodate the reference tables). Hence, the problem of selecting resources for the different replicas, so that the system achieves best overall throughput, can also be represented as a facility location problem.

A more straightforward use of facility location may be found in data-Grid systems. Data-Grids are concerned with controlled sharing and management of large amounts of distributed data. In many cases, for example in AstroGrid (which is a data-Grid designed to hold and process astronomical survey data, for more information see [26]), data stored in the system needs to be mined. In addition to being distributed among thousands of computers, the data stored in those systems is constantly updated. FLP, being a close relative to clustering, can be used to extract knowledge from this data.

A more traditional application for decentralized facility location can be found in battery operated wireless sensor networks. In these networks the major challenge is to find an efficient way by which data gathered by the sensors can be routed to a control station using as little energy a possible. In order to save energy, sensors would not transmit such data directly, but rather relay it from one sensor to its neighbors until it reaches the control station. Assume a network enhanced with a few dozen high powered relay stations capable of communicating directly with the control station. Obviously these could dramatically increase the energy efficiency of the sensors. If the routers themselves are resource limited (e.g., the number of communication channels is bounded, or these relays are battery operated as well), then it would make sense to select a handful of them to be active, and shut down the rest. The optimal selection would take into account the bandwidth requirement and the battery limitation of each sensor. Thus, it is a facility location problem where the energy cost of a solution is the sum of costs for all sensors and all active relays.

The facility location problem (FLP) has been extensively studied in the last decade. Like many other optimization problems, optimal facility location is NP-Hard [15, 17]. Thus, the problem is often subjected to a hill-climbing heuristic [6, 12, 18]. Hill-climbing is a simple and effective heuristic search technique in which it is assumed that a reasonable local optimum can be reasonably approached if on each search step the algorithm greedily chooses the direction that maximally decreases cost. The strength of the method is in its simplicity. It has been extensively tried for optimum search in exponential domains in problems such as genetic algorithms, clustering, etc. A rather surprising result by Arya et al. [1] states that, for FLP, hill climbing achieves a constant factor approximation of 3, and this factor is tight. This means that the cost of the local optimum computed by the hill-climbing heuristic is not worse than thrice the cost of the globally optimal solution. Moreover, as shown in [12], an algorithm with approximation factor better than 1.463 does not exits, unless $NP \subseteq DTIME[n^{O(loglog(n))}]$. Although, the facility location problem can also be solved by primal-dual technique [15] and linear programming relaxation [4] with better approximation factors, these techniques are less suitable for large-scale distributed systems.

Approximability of the facility location problem in distributed environments has been studied in [24]. However, their algorithm advances in globally synchronized rounds which makes it unsuitable from large-scale systems. Moreover, their algorithm uses linear programming relaxation to find the solution. To the best of our knowledge, the hill-climbing approach to FLP has never been studied specifically in a distributed setting. Nevertheless, it is easy to see how FLP can be parallelized in the shared memory model. In addition, a related problem, the k-means clustering, which is also solved by a hill-climbing heuristic, was widely addressed in the parallel settings [7, 10, 11]. We note, however, that all previous work on distributed clustering assumes tight cooperation and synchronization between the peer nodes containing the data and a central node that collects the sufficient statistics needed in each step of the hill-climbing heuristic. Such central control is not practical in large-scale networks because it imposes large bandwidth requirements and is prone to errors even in the case of single failures. Even more importantly, central control is unscalable in the presence of dynamically changing data because any such change must be reported to the center, for fear it might alter the result.

In contrast, the most important features which qualify an algorithm for a large-scale distributed system are the following: the ability to efficiently scale-up (there are peer-to-peer systems today that consist of millions of peers), the ability to perform in a router-less network (critical for wireless sensor networks), and the ability to calculate the result in-network rather than to collect all of the data to a central processor (which would quickly exhaust bandwidth in both sensor and peer-to-peer networks [13]). Most important of all, because the data in a large-scale system usually changes before the computation is complete, it is crucial that the algorithm efficiently prune redundant messages and computation, as long as the data changes do not affect the global output. All these features typify *local* algorithms.

A local algorithm is one in which the complexity of computing the result does not directly depend on the number of participants. Instead, each node usually computes the result using information gathered from just few nearby neighbors. Because communication is restricted to neighbors, a local algorithm does not require message routing, performs all computation in-network, and in many cases is able to locally overcome failures and minor changes in the input (provided that these will not change its output). Local algorithms have been studied mainly in the context of graph related problems [2, 3, 19–22, 25]. Most recently, it has been demonstrated in [27] that local algorithms can be devised for complex data analysis tasks, specifically, data mining of association in distributed transactional databases. The algorithm presented in [27] features local pruning of false propositions (candidates), in-network mining, asynchronous execution, as well as resilience to changes in the data and to partial failure.

In this work we develop a local algorithm that solves a specific version of FLP in which uncapacitated resources (i.e., those which can serve any number of clients) can be placed in any of the m possible locations. Initiating our algorithm from a fixed resource location, we show that the computation needed to agree on single hill-climbing step – shutting down an active resource or opening a new one at a free location – can be reduced to a group of majority votes. We then use a variation of the local majority voting algorithm presented in [27] to develop an algorithm which locally computes the exact same solution a hill-climbing algorithm would compute, had it been given the entire data. Our algorithm demonstrates that whenever the cost of a step is summed across the different sensors, peers, or resources, a hill-climbing heuristic can be computed using a local, in-network algorithm.

In a series of experiments employing networks of up to 1,000 simulated processors, we prove that our algorithm has good locality, incurs reasonable communication costs, and quickly converges to the correct answer whenever the input stabilizes. We further show that when faced with constant data updates, the vast majority of sensors continue to compute the optimal solution. Most importantly, the algorithm is extremely robust to sporadic changes in the data. So long as these do not change the global result, they are pruned locally by the network.

The rest of this paper is organized as follows. We first describe our notations and formally define the problem. Then, in Section 3, we give our version for the majority voting algorithm originally described in [27]. Section 4 describes a local facility location algorithm as an example of a hill climbing algorithm. In Section 5 preliminary experimental results are described. Section 6 ends the paper with some conclusions and open research problems.

2 Notations, Assumptions, and Problem Definition

The input to the *facility location problem* consists of an *input points* database, where each point represents a client, $DB = \{p_1, p_2, \ldots, p_n\}$, a set M of m possible *locations*, a cost function $d : DB \times M \to \mathbb{R}^+$, and a *configuration* (a set of open facilities) cost function $D : 2^M \to \mathbb{R}^+$. The task of a *facility location* algorithm is to find a set of *open facilities* $C \subseteq M$, such that the total cost of C and the cumulative distance of points from their nearest facility in C, $D(C) + \sum_{p_i \in DB} \min_{c \in C} d(p_i, c)$, is minimized. Note that relative difference in configuration costs vs. cumulative distance of points will influence the number of open facilities in the

optimal solution. We leave the user the flexibility to decide on this relative difference.

To relate these definitions to the sensor networks example in the introduction, consider a database that includes a list of events that occurred in the last hour. Each event has a heuristic estimate of its importance. Furthermore, each sensor evaluates its hop distance from every relay and multiplies this by the heuristic importance of each event to produce its cost. Finally, the cost of each configuration is the number of active relays. Given this input, a facility location algorithm will compute the best combination of relays such that the most important events need not travel far before they reach the nearest relay, and not too many relays are active. The less important events, we assume, will be suppressed either in the sensor which produced them, or in-network by other sensors.

Given a large number N of nodes, which can communicate with one another by sending messages, a *distributed* facility location algorithm would compute the same result even though the database is partitioned among the nodes. The database DB is partitioned into N mutually exclusive databases $\{DB^1, \ldots, DB^N\}$, each of which is stored in a separate node. A *local* facility location algorithm is a distributed algorithm whose performance does not depend on N but rather corresponds to the difficulty of the problem instance at hand.

An *anytime* facility location algorithm is one which, at any given time during its operation, outputs a set of open facilities such that the cost of this ad hoc output improves with time until the optimal solution is found.

The *hill-climbing* heuristic for facility location begins from an initial set of open facilities (henceforth, *initial configuration*). Then it selects a single facility and a single empty location such that by doing one of the following: 1) moving the selected facility to this free location, 2) closing the selected facility, or 3) opening a new facility at the empty location, the cost of the solution is reduced to the largest possible degree. If such a step exists, the algorithm changes the configuration accordingly and iterates. If every configuration which can be stepped into by closing, opening, or moving just one facility has a higher cost than the current configuration, the algorithm terminates and outputs the current configuration as the solution.

This paper presents a local, anytime algorithm which computes the hill-climbing heuristic for facility location. Note that this algorithm can easily be applied to any other hill-climbing problem. To do this, it is enough to describe the start point and the mechanism by which the next possible steps are created and their cost (or gain) evaluated. The rest of the algorithm remains the same.

3 Local Majority Voting

Our facility location algorithm reduces the problem to a large number of majority votes. In this section, we briefly describe a variation of the local majority voting algorithm from [27] which we use as the main building block for the algorithm.

We assume that communication among neighboring nodes is reliable and ordered. This assumption can be enforced by using standard numbering, ordering and retransmission mechanisms. For simplicity of explanation we assume an undirected communication tree. Yet, this assumption is not limiting. To extend the local facility location algorithm to arbitrary communication graphs, Algorithm 1 can be replaced by a majority voting algorithm described in [5]. The idea of [5] is based on variations of Bellman-Ford algorithms [9, 14] for constructing communication trees. In addition, we assume fail-stop failure and that when a node is disconnected or reconnected its neighbors are informed. Finally, the algorithm requires no more time guarantees than what is necessary for detecting failures.

Given a set of nodes V, where each $u \in V$ contains a zero-one poll with c^u votes, s^u of which are one, and given the required majority $0 < \lambda < 1$, the objective of the algorithm is to decide whether $\sum_u s^u / \sum_u c^u \geq \lambda$. Equivalently, the algorithm can compute whether $\Delta = \sum_u s^u - \lambda \sum_u c^u$ is positive or negative. We call Δ the number of excess votes.

We will now give some intuition about how the algorithm works. Observe that only the sign of Δ need to be determined, not the magnitude. The omission of magnitude determination makes

it possible to suppress many messages, and leads to a local algorithm. The suppression happens in the following manner. We look at every node as a tree root which determines its output by summing its local input with what children report about their subtrees. Each node being a root informs every child about changes in other sub-trees, but makes it selectively. The root informs only those children whose output my be altered due to new information. Since suppressed messages would not influence the output, correctness of the result is preserved.

The decision whether a message can influence the result of some subtree or not is done in the following way. Assume with out loss of generality that a neighbor of some node estimates Δ as positive. Consider two scenarios, in the first the message to the neighbor increases its estimate. The estimate becomes farther from zero and thus makes its sign harder to flip. Since the output of the neighbor is the sign of its estimate, the output can not be influenced by this message, and therefore such a message can be suppressed. In the second scenario, the message decreases the neighbor's estimate. The estimate becomes closer to zero, and thus its sign becomes easier to flip. Therefore, such a message may have influence on the output in the future and can not be suppressed. And of course, in a case where a message flips the neighbor estimate, it should not be suppressed as well.

Finally, we explain why suppression of messages carrying positive votes does not give unfair advantage to messages with negative votes. Consider a scenario when a message carrying a large amount of positive votes was suppressed and did not reach node u, whose estimate is positive. But another message with a small amount of negative votes, having a potential to change u's output, was received by u and flipped its estimate. In this case, had the positive message not been suppressed the u's output would still be positive. This apparent incorrectness is resolved in the following way. Once u's output is flipped, the suppression condition is reevaluated and the positive message is sent, because now a positive message has a potential to influence a negative estimate. After the positive message is received, u's output is flipped back to be positive.

The following local algorithm decides whether $\Delta \geq 0$. Each node $u \in V$ computes the number of excess votes in its own poll $\delta^u = s^u - \lambda c^u$. Further, in δ^{uv} it stores the number of excess votes it reported to each neighbor v, and stores the number of excess votes reported to it by v in δ^{vu}. Node u computes the total number of excess votes it knows of, Δ^u, as the sum of its own excess votes and those reported to it by the set G^u of its neighbors: $\Delta^u = \delta^u + \sum_{v \in G^u} \delta^{vu}$. It also computes the number of excess votes it negotiated with every neighbor: $v \in G^u$, $\Delta^{uv} = \delta^{uv} + \delta^{vu}$. When u chooses to inform v about a change in the number of excess votes it knows of, u sets δ^{uv} to $\Delta^u - \delta^{vu}$, which results in Δ^{uv} being equal to Δ^u. It then sends δ^{uv} to v. When u receives a message from v containing some δ, it sets δ^{vu} to δ – thus updating both Δ^{uv} and Δ^u. Finally, node u outputs that the majority is of ones if $\Delta^u \geq 0$, and of zeros otherwise.

The crux of the local majority voting algorithm is in determining when u must send a message to a neighbor v. More precisely, the question is when can sending a message be avoided, despite the fact that the local knowledge has changed. In the algorithm presented here, node u would send a message to a neighbor v in two cases: when u is initialized and when the condition $(\Delta^{uv} \geq 0 \wedge \Delta^{uv} > \Delta^u) \vee (\Delta^{uv} < 0 \wedge \Delta^{uv} < \Delta^u)$ evaluates true. Note that u must evaluate this condition upon receiving a message from a neighbor v (since this event updates Δ^u and the respective Δ^{uv}), when its input bit switches values, and when an edge connected to it fails (because then Δ^u is computed over a smaller set of edges and may change as a result). This means the algorithm is driven by local events and requires no form of global synchronization among all nodes.

The analysis in [27] reveals that the good performance of the above algorithm, in terms of message load and convergence time, stems directly from its locality. The average (as well as the worst) node would terminate after it has collected data from just a small number of nearby neighbors – its environment. The size of this environment depends on the difference, in the nearby surroundings of the node, between the average vote and majority threshold. If the two differ by as

much as 5%, then we can expect the size of the environment to be limited to a few dozen nodes. However, if the vote is close to a tie, then the environment can be as big as the entire network. In this case, the algorithm may take time proportional to the network diameter to converge. Nevertheless, we expect such global cases to be rare or not even occur.

In order to use the local majority voting algorithm for facility location we modify it slightly. We add the ability to suspend and reactivate the vote using corresponding events. A node whose voting has been suspended will continue to receive messages and to modify the corresponding local variable, but will not send any messages. When the vote is activated, the node will always check if it is required to send a message as a result of the information received while in suspended state. Furthermore, we allow a bias towards one vote, which is equivalent to starting the vote with γ additional zeros (γ can be positive or negative). This is done by changing the condition for sending messages to $(\Delta^{uv} \geq \gamma \wedge \Delta^{uv} > \Delta^u) \vee (\Delta^{uv} < \gamma \wedge \Delta^{uv} < \Delta^u)$ and outputting a majority of ones if $\Delta^u \geq \gamma$ and of zeros otherwise. The pseudo-code of the modified algorithm is given in Algorithm 1.

Algorithm 1 Local Majority Vote

Input of node u: the local poll c^u, the local support s^u, and the set of neighbors G^u
Global constants: the majority threshold λ, the bias γ
Local variables: $\forall v \in G^u : \delta^{uv}, \delta^{vu}, active^u$.
Definitions: $\delta^u = s^u - \lambda c^u$, $\Delta^u = \delta^u + \sum_{v \in G^u} \delta^{vu}$, $\Delta^{uv} = \delta^{uv} + \delta^{vu}$
Initialization: $active^u = true$
$\forall v \in G^u$: $\delta^{uv} = \delta^{vu} = 0$, SendMessage($v$)
On activate: set $active^u \leftarrow true$
On suspend: set $active^u \leftarrow false$
On receive-message δ from $v \in G^u$: $\delta^{vu} \leftarrow \delta$
On notification of failure of $v \in G^u$:
$G^u \leftarrow G^u \setminus \{v\}$
On notification of a new neighbor v:
$G^u \leftarrow G^u \cup \{v\}$
On any of the above events and on change in δ^u:
For all $v \in G^u$,
if $(\Delta^{uv} \geq \gamma \wedge \Delta^{uv} > \Delta^u) \vee (\Delta^{uv} < \gamma \wedge \Delta^{uv} < \Delta^u)$ then
– SendMessage(v)
Procedure SendMessage(v):
If $active^u = true$ then
– $\delta^{uv} \leftarrow (\Delta^u - \delta^{vu})$, Send $\langle \delta^{uv} \rangle$ to v
Output of u:
if $\Delta^u \geq \gamma$ then *positive* else *negative*

4 Majority Based Facility Location

The local facility location algorithm, which we now present, is based upon three fundamental ideas: The first is to have every node speculatively perform hill-climbing steps without waiting for a conclusive decision as to which step is globally optimal. Having performed such steps, the node continues to validate whether they agree with the globally correct ones. If there is no agreement, then these speculative steps are undone and better ones are chosen. The second idea is to choose the optimal step not by computing the cost of each step directly, but rather by voting. For each pair of possible steps, each node votes for the step it considers less costly. The third idea is a pruning technique by which many of these votes can be avoided altogether; avoiding unnecessary votes is essential because, as we further explain below, computing votes among each pair of optional steps might be arbitrarily more complicated than finding the best next step.

Key to our algorithm is the observation that the kernel problem of a hill-climbing facility location algorithm – choosing the step that reduces the cost of the solution as much as possible – is reducible to majority voting. We use this observation, together with the communication efficient local majority voting algorithm described in Section 3 to devise a local algorithm that computes the best among the set of possible configurations (ones reachable by moving just a single facility to a free location) and the current configuration. If the current configuration is the best possible one then it is a local minima and the algorithm makes no further steps. Otherwise, the algorithm steps to this best possible configuration and reiterates the computation.

4.1 Speculative Computation of an Ad Hoc Solution

Most parallel data mining algorithms use synchronization to validate that their outcome represents the global data $\bigcup_u DB^u$. We find this approach impractical for large-scale distributed systems – specifically if one assumes that the data may change with time, making the global data impossible to determine. Instead, when performing local hill climbing, we let each node proceed uphill whenever it has computed the best step according to the data it currently possesses. Then, we use local majority voting (as we describe next) to make sure that nodes which have taken erroneous steps will eventually be corrected. In the event that a node is corrected, a computation associated with configurations that were wrongly chosen is put on hold. These configurations are put aside in a designated cache in case additional data, accumulated later, will prove them correct after all.

We term the sequence of steps selected by node u at a given point in time its *path* through the space of possible configurations and denote it $R^u = \langle C_1^u, C_2^u, \ldots, C_l^u \rangle$. C_1^u is always chosen to be the first location in M. C_l^u is the ad hoc solution of node u. u refrains from developing another configuration following C_l^u when no possible successor step has lower cost.

Since the computation of all of the configurations along every node's path is concurrent, messages sent by the algorithm contain a *context* – the configuration to which they relate. Since the computation is also speculative, it may well happen that two nodes u and v intermediately have different paths R^u and R^v. Whenever u receives a message in the context of some configuration $C \notin R^u$, this message is considered *out of context*. It is not accepted by u but rather is stored in u's out-of-context message queue. Whenever a new configuration C enters R^u, u scans the out-of-context queue and accepts messages relating to C in the order in which they were received.

4.2 Locally Computing the Best Possible Step

For each configuration $C_a^u \in R^u$, node u computes the best possible step as follows. First, it generates the set of possible successor configurations $Next\left(C_a^u\right)$, such that each member of $Next\left(C_a^u\right)$ adds one more location to C_a^u, removes one of C_a^u's locations, or replaces one location in C_a^u with a location from $M \setminus C_a^u$. Next, for every $C_i, C_j \in Next(C_a^u)$, where $i < j$, node u initiates a majority vote $Majority^u_{C_a^u}\langle i, j\rangle$ which compares their costs and eventually outputs *negative* if the global cost of C_i is lower than that of C_j (as we explain below). Correctness of the majority vote process guarantees that the best configuration $C_{i_{best}} \in Next(C_a^u)$ will eventually have *negative* output for $Majority^u_{C_a^u}\langle i_{best}, j\rangle$ for all $j > i_{best}$, and *positive* output of $Majority^u_{C_a^u}\langle j, i_{best}\rangle$ for all $j < i_{best}$. Hence, the algorithm will speculatively choose C_i as the next configuration, whenever all votes indicate that C_i is better.

To determine which of two configurations has the better cost using a majority vote, we initialize $Majority^u_C\langle i, j\rangle$ with the following inputs: $s^u = \sum_{p \in DB^u} cost\left(p, C_i\right) - cost\left(p, C_j\right)$, where $cost\left(p, C\right) = \min_{f \in C}\{d\left(p, f\right)\}$, $c^u = 0$, $\lambda = 0$. Note that, as shown in [27], s^u and c^u can be set to arbitrary numbers and not just to zero or one. Further note that for every C_i, C_j the following equality holds:

$$\sum_{p \in DB} cost\left(p, C_i\right) - \sum_{p \in DB} cost\left(p, C_j\right) = \sum_u \sum_{p \in DB^u} \left[cost\left(p, C_i\right) - cost\left(p, C_j\right)\right]$$

Additionally, we set the bias of the vote to the difference in costs between the two configurations, $\gamma = D\left(C_j\right) - D\left(C_i\right)$. Hence, if the vote comparing the costs of C_i and C_j determines that $\Delta^u\langle i, j\rangle \geq \gamma$, then the cost of C_i is proven to be larger than the cost of C_j.

Note that since every majority vote is performed using the local algorithm described in Section 3, the entire computation is also local. Eventual correctness of the result and the ability to handle changes in DB^u or G^u also follow immediately from the corresponding features of the majority voting algorithm.

4.3 Pruning the Set of Comparisons

In the above subsections we have shown how it is possible to reduce facility location to a set of majority votes. However, the reduction overshoots the objective of the algorithm. While a facility location algorithm only requires that the *best* possible successor configuration be calculated given a certain configuration, the reduction above actually computes a *full order* on the possible successor configurations. This is problematic because for some inputs computing a full order may be arbitrarily more difficult (and thus less local) than computing only the best option. For instance, the algorithm may invest a lot of messages in deciding which of the two configurations is better, even though none of them is the best.

To overcome this problem we augment the algorithm with a pruning technique that limits the progress of comparisons such that only a small number of them actually take place. The technique we adopt is based on pivoting. First, each configuration is a candidate to be the least costly. We choose an arbitrary candidate configuration (without loss of generality, the first one) as a pivot and compare all of the other candidates to it. Then, we choose the configurations which are indicated to be less costly than the pivot to be the next set of candidates and select one of them (again, the one with the lowest index) as the new pivot. As soon as the next candidate set becomes empty, the development process is stopped, and the last pivot is the least costly configuration.

Formally, given a configuration C and the set of successor configurations $Next(C)$, we define for every node u: $Pivot_i^u(C) = \min S_i^u(C)$ for $1 \geq i \geq k$, and $S_1^u(C) = \{1, 2, \ldots, |Next(C)|\}$. We define $S_i^u(C) = \{j \in S_{i-1}^u(C) \mid Majority_C^u\langle j, Pivot_{i-1}^u(C)\rangle.out = negative\}$ developing pivots until $S_k^u(C) = \emptyset$. Eventual correctness of all the majority votes assure that $Pivot_{k-1}^u(C)$ is the configuration with the lowest cost.

4.4 Pseudocode of the Algorithm

The pseudocode of the algorithm is given in Algorithm 2. It relies on an underlying majority voting algorithm. The facility location algorithm tunnels

Algorithm 2 Local Facility Location

Global Constants: the set M of m possible locations

Input of processor u:
a database $DB^u = \{p_1^u, p_2^u, \ldots\}$, a set of neighbors G^u, and the distances of points from the possible locations $d : DB^u \times M \to \mathbb{R}^+$

Definitions:
$Next(C) = Swap(C) \cup Add(C) \cup Remove(C) \cup C$
$Swap(C) = \{C \setminus \{f\} \cup \{f'\} | f \in C, f' \in M \setminus C\}$
$Add(C) = \{C \cup \{f'\} \mid f' \in M \setminus C\}$
$Remove(C) = \{C \setminus \{f\} \mid f \in C\}$
For some set of configurations N, $N[i]$ is i's element of N according to lexicographic order.
For any $C_i \in R^u$ the following holds:
$C_1^u = \{M[1]\}$
let k be the minimal index for which $S_k^u(C) = \emptyset$
let l be the minimal index for which $C_l^u = C_{l-1}^u$
$C_{i+1}^u = Next(C_i^u)[Pivot_{k-1}^u(C_i^u)]$ for $1 \leq i < l$
$S_1^u(C) = \{1, 2, \ldots, |Next(C)|\}$—Set of indexes
$\forall i > 1 : S_i^u(C) = \{j \in S_{i-1}^u(C) \mid Majority_C^u\langle j, Pivot_{i-1}^u(C)\rangle.out = negative\}$,
For $1 \leq i < k$: $Pivot_i^u(C) = \min S_i^u(C)$
$ActiveSet^u = \{\langle C, i, j\rangle \mid C \in R^u, \exists q : i \in S_q^u(C), j = Pivot_q^u(C), i \neq j\}$

Local variables:
A vector $R^u = \langle C_1^u, \ldots, C_l^u\rangle$ of configurations, where $C_i^u \subseteq M$
A message queue $OutOfContext^u$
For each $C \in R^u$: a vector of pairs $MV(C) = \langle (S_1^u(C), Pivot_1^u(C)), \ldots, (S_k^u(C), Pivot_k^u(C))\rangle$ a set of majority votes referred to as $Majority_C^u\langle i, j\rangle$

Init of $Majority_C^u\langle i, j\rangle$:
if $Majority_C^u\langle i, j\rangle$ *exists* **then**
activate $Majority_C^u\langle i, j\rangle$
else
create $Majority_C^u\langle i, j\rangle$ with inputs
$s = \sum_{p \in DB^u} cost(p, N[i]) - cost(p, N[j])$,
for $N = Next(C)$,
$cost(p, C) = \min_{f \in C}\{d(p, f)\}$
$c = 0,\ \lambda = 0,\ G^u = G^u$
$\gamma = D(Next(C)[j]) - D(Next(C)[i])$
tunnel to $Majority_C^u\langle i, j\rangle$ all messages in $OutOfContext^u$ directed to it

Algorithm 2 Local Facility Location (cont.)

Initialization:
$OutOfContext^u \leftarrow \emptyset$
while R^u or $MV(C)$ for $C \in R^u$ *changes* **do**
$\forall \langle C, i, j \rangle \in ActiveSet^u$ init $Majority^u_C\langle i, j\rangle$
update R^u and $\forall C \in R^u$ update $MV(C)$
On MessageSend $\{\delta\}$ **from** $Majority^u_C\langle i, j\rangle$ **to** v**:**
send message $\{C, i, j, \delta\}$ to v
On message $\{C, i, j, \delta\}$ **from** $v \in G^u$
if $Majority^u_C\langle i, j\rangle$ exist **then**
tunnel message $\{\delta\}$ to $Majority^u_C\langle i, j\rangle$
else
enqueue $\{C, i, j, \delta\}$ in $OutOfContext^u$
On change in G^u**:**
foreach existing $Majority^u_C\langle i, j\rangle$ **do**
call on change in G^u for $Majority^u_C\langle i, j\rangle$
On change in output of $Majority^u_C\langle i, j\rangle$
repeat
$OldActive \leftarrow ActiveSet^u$
update R^u and $MV(C)$ for $C \in R^u$
foreach $\langle C, i, j\rangle \in ActiveSet^u \setminus OldActive$ **do**
init $Majority^u_C\langle i, j\rangle$
foreach $\langle C, i, j\rangle \in OldActive \setminus ActiveSet^u$ **do**
suspend $Majority^u_C\langle i, j\rangle$
until $OldActive \neq ActiveSet^u$
Output of processor u**:** C^u_l

messages to and from majority votes, removing and adding context information on the way.

4.5 An Illustration of Algorithm Execution

To provide some insight on how the algorithm works, we will illustrate an example execution of a single node in the network.

The node begins its execution from an initialization event which creates the majority votes. Immediately after creation, before receiving any messages, each majority vote outputs an ad-hoc solution. Based on the majority vote outputs, a vector R^u is created, and the node provides an ad-hoc solution. Note that this solution appears immediately after initialization and is based solely on the node's local input.

Some time later a message $\{C, i, j, \delta\}$ arrives. This message is routed to $Majority^u_C\langle i, j\rangle$ majority vote according to C, i, and j tags. Assume that this message alters the output of this majority, and assume that $C = C^u_p \in R^u$. Also assume that as a reaction $Pivot^u_{k-1}(C)$ changes it value from r to r'. In this case the successor configuration of C, $C^u_{p+1} = Next(C)[r]$ ceases to be the least cost successor configuration, and needs to be replaced with $Next(C)[r']$. During this replacement, configurations $C^u_{p+1}, C^u_{p+2}, \ldots, C^u_l$ are dropped from R^u, and a new sequence of configurations starting from $C^u_{p+1} = Next(C)[r']$ is created. All majority votes related to the dropped configurations are suspended. After replacement is finished, a new ad-hoc output appears. The new output is based on both the received message and the local input, and therefore is more accurate and closer to the optimal solution than its predecessor.

Eventually, when enough messages are received the output will converge to the exact solution.

5 Experiments

To evaluate the algorithm's performance, we ran it on simulated networks of up to 1,000 nodes using databases of varying sizes. Our experiments test for two main properties of the algorithm: its ongoing behavior when the data is constantly altered and its dependency on the different operational parameters.

We are interested in three main metrics: the percentage of nodes which compute the exact solution at any point in time, the relative costs of the solutions computed by nodes which output a wrong solution, and the communication cost for computing the solution. Ideally, most of the nodes will compute the exact solution, or else will compute a solution that is not much costlier than the exact one. Finally, all this will be done using very few messages.

The operational parameters we find most crucial are the size of the system (N), the number of data points in every local database (n/N), and the network topology. To test for the dependency of the algorithm's performance on these parame-

ters, we ran batch mode experiments in which the data did not change during the execution. This provided a controlled experiment in which each parameter could be tested on its own.

The performance (run-time, messages, etc.) of a local algorithm differs from one peer to the next. As shown in previous work [28] this performance is proportional to the distribution of the data in the near proximity of the peer. Specifically, on the largest area surrounding the peer which has data that would yield the *wrong* outcome. In our experiments, we used unbiased data by sampling the data of all peers from the same stochastic source. A uniform mixture of sources is not expected to influence performance significantly, because each small neighborhood of sources will have the same mixture of distributions. Deterministic bias, if introduced, will have an adverse effect of performance. However, this effect is difficult to quantify in a meaningful manner because of the many possible ways the data can be biased.

We used a synthetic database created using the method described in [8]. The data points (which represent clients) were generated in the 2D real domain $[0, 20] \times [0, 20]$. The source of the data points was a mixture of 10 Gaussians with a random mean and variance of one, and 20% random uniform noise. Possible facility locations were placed on an equally spaced Grid covering the points' domain. Figure 1 depicts a typical database with the large (red) dots signifying possible locations. Using these settings, each static experiment was run 10 times with different data each time. In the batch experiments, databases of between 100 and 1,000 points (depending on the experiment) were generated for each node. In the dynamic experiments, once every few simulator clock cycles (again, depending on the experiment) a node was randomly selected, and a fraction of the points in its database were replaced with new points, sampled from the same distribution.

Finally, because the behavior of distributed algorithms may depend on network topology, we repeated our experiments for two different topologies: An Internet-like topology generated by a state-of-the-art BRITE [23] simulator and a de Bruijn topology [16] that simulates a network with a fixed expansion rate.

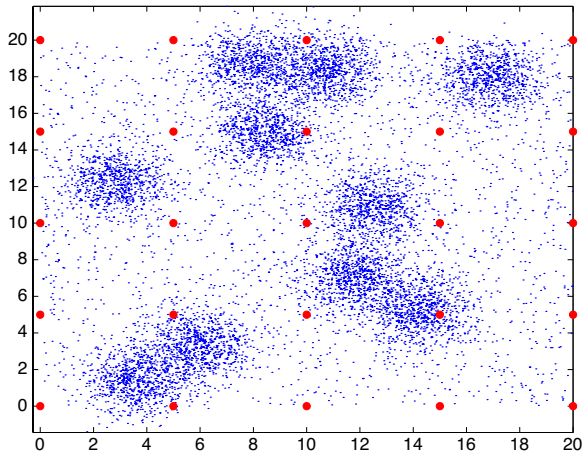


Fig. 1 A typical database

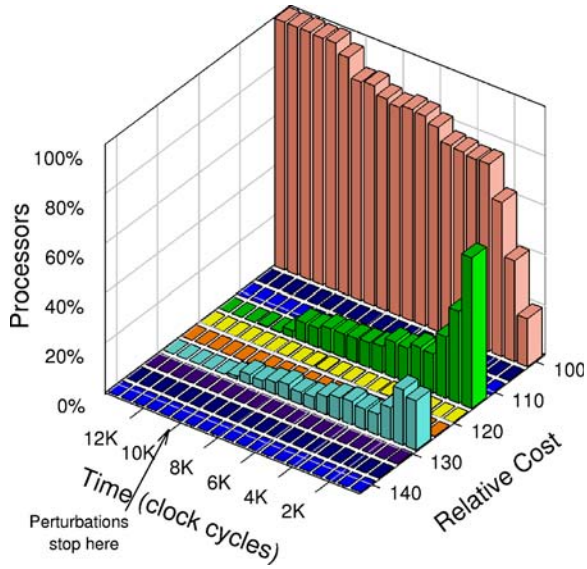


Fig. 2 Evolution of solutions costs distribution in dynamic experiment is shown. At each time, a histogram of solutions costs is plotted. The cost of the optimal solution is 100. At time interval 0–687 simulator cycles (*the first time slice in the graph*), 60% of nodes output a solution of cost 113 to 118, and 20% of nodes output a solution of cost 127 to 131. Leaving only 20% of the nodes at the optimal solution. As time passes more nodes converge to the optimal solutions. At time interval 0–10,000, node's data undergoes continuous perturbations. From time 3,000 to 10,000 the system is at steady state, where perturbations prevent from all nodes to converge to the optimal solution. As perturbations stop, the system rapidly converges to the optimum. Simulation parameters: Internet topology, $N = 512, n/N = 1,000$, and $m = 25$

5.1 Ongoing Operation

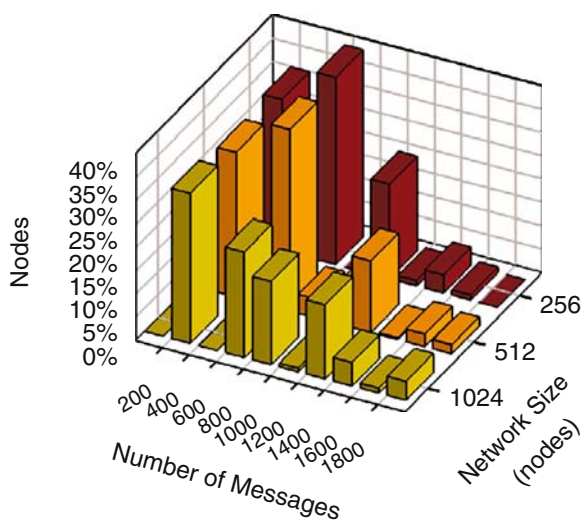
In this experiment we tried to realistically simulate a typical working scenario of the algorithm, in which the distribution of the data is stationary, but the data is continuously updated over time with new samples. To simulate dynamic data that retains a stationary distribution, we randomly select 5% of the nodes every five simulator cycles (about 35 times in an average edge delay). We replace 10% of each selected node's data points with new points selected from the same distribution. We keep changing the database this way for 10,000 simulator cycles and then stop the changes in order to let the algorithm converge to the exact result.

As the results in Fig. 2 show, during the period of data changes, more than 80% of the nodes manage to quickly compute the optimal solution. Of the nodes that compute a different result, most compute one that is about 15% more costly than the optimal (note that costs here are normalized). When at cycle 10,000 the changes stop, all of the nodes immediately converge to the correct result. Similar results were computed for different amounts of data perturbation.

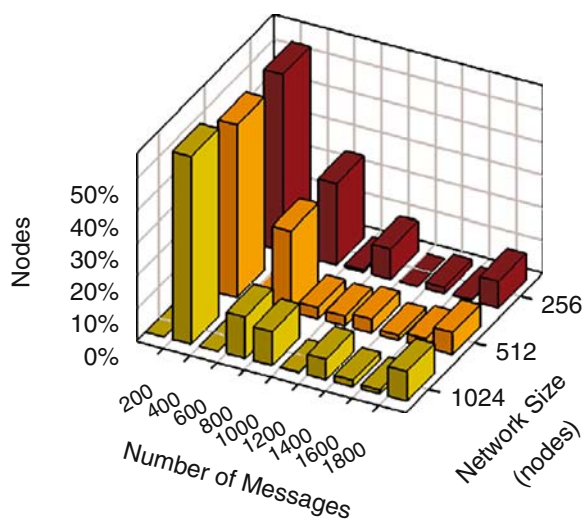
5.2 Communication Cost

We evaluated the communication cost of the algorithm by counting the number of messages sent by the average node during an entire bulk run. We were especially interested in the scale-up of the algorithm (i.e., the effect of increasing N on the message cost) and in the effect of different network topologies on the message cost. We ran experiments with 256, 512, and 1,024 nodes using both a BRITE generated Internet topology and a de Bruijn topology. For each combination we generated 10 different databases. We let the algorithm run through, counted the number of messages sent by each node, and then reported the averaged histogram for each topology and N.

The result, as reported in Fig. 3, shows some interesting trends. First, about half the nodes use 200–400 messages throughout the algorithm. This seems a very reasonable number, considering that each of the algorithm's messages takes a few dozens bytes. Being small, these messages could be buffered. Second, the algorithm scales-up well, with no real increase in costs. Last, the algorithm requires fewer messages in an Internet topology, which can be explained by its superior mixing power.

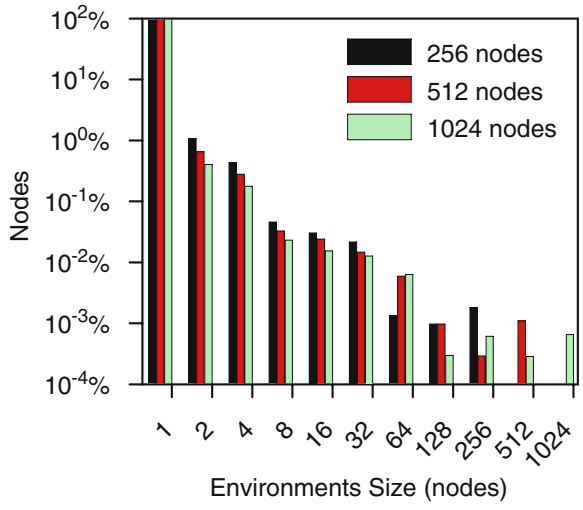


de Bruijn

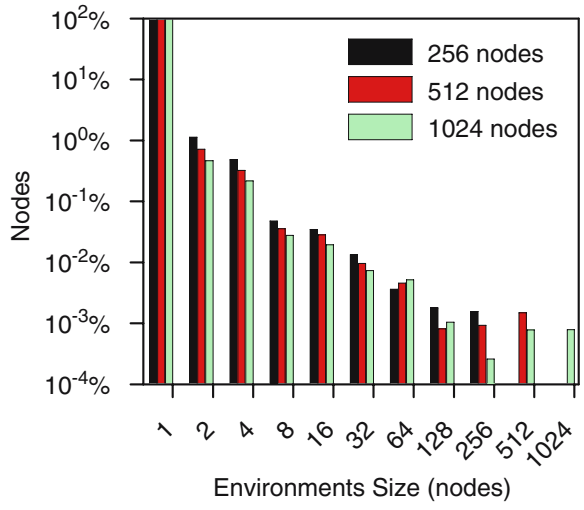


BRITE – Internet

Fig. 3 Distribution of number of messages sent by nodes for three network sizes and two topologies. *Last bars* represent all nodes which sent more than 1,800 messages. The graph shows that as the network size grows, messages distribution remains essentially the same



de Bruijn

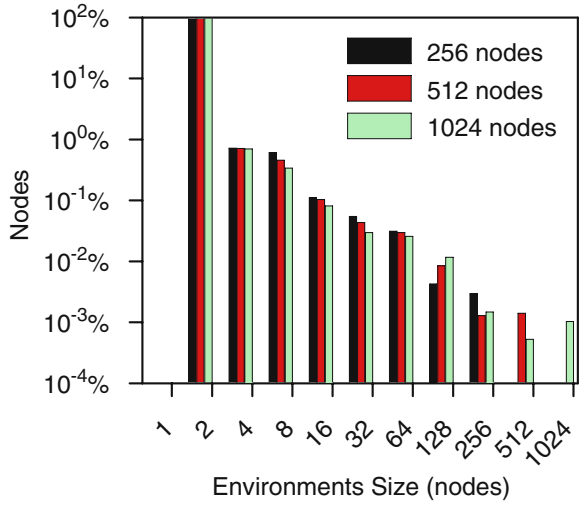


BRITE – Internet

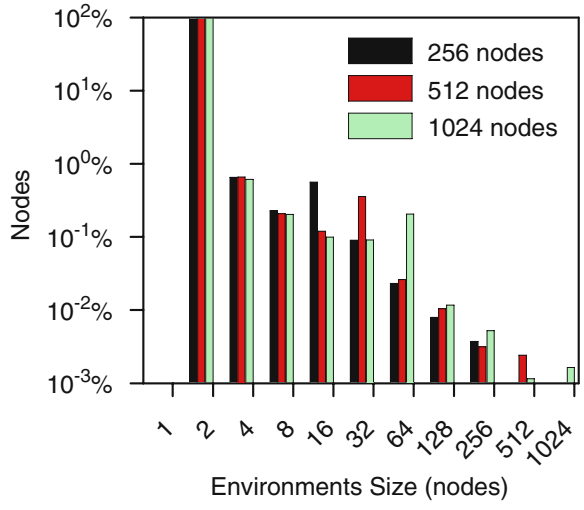
Fig. 4 A histogram of average environment sizes is depicted for three network sizes and two network topologies (de Bruijn and BRITE). *Each bar* represents the average number of nodes over 10 experiments. These graphs show that the vast majority of nodes have small environments, which supports the claim that the algorithm is local. Simulation parameters: $m = 25, n/N = 1,000$

5.3 Locality

The next set of experiments measures the size of each node's environment directly. Node u's environment is the set of neighboring nodes from which u gathers data. The size of the environment is important because the algorithm's performance strongly depends on it. For each network topology



de Bruijn



BRITE – Internet

Fig. 5 A histogram of maximum environment sizes is depicted for three network sizes and two network topologies (de Bruijn and BRITE). Each bar represents the maximum number of nodes over 10 experiments. These graphs show that even in the worst case analysis, the vast majority of nodes have small environments. Simulation parameters: $m = 25, n/N = 1,000$

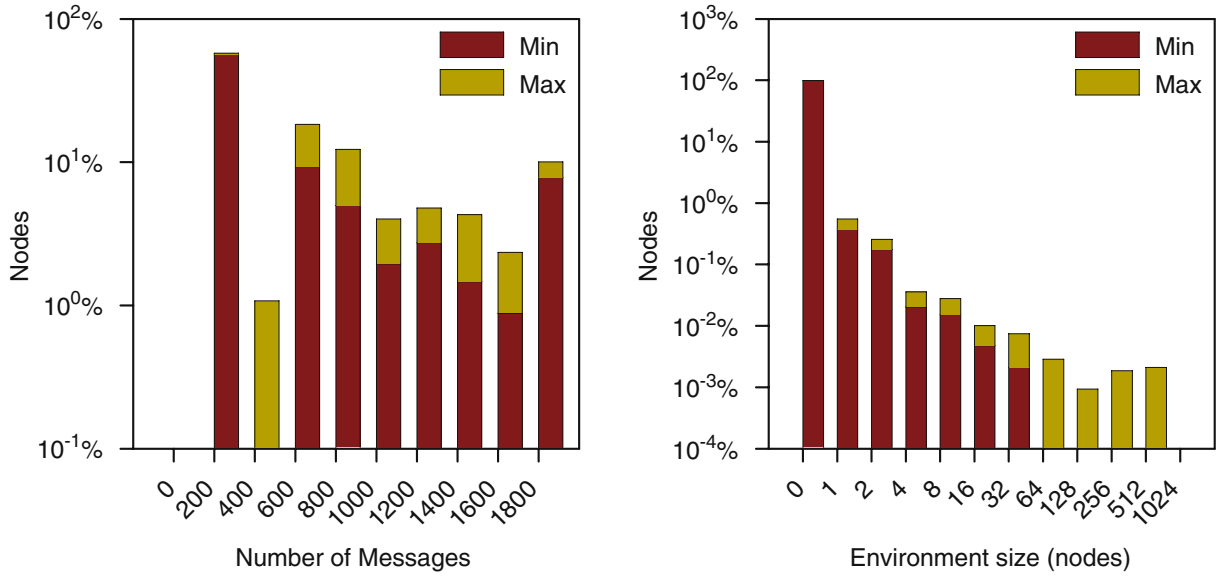


Fig. 6 Comparison of best case and worst case results when using different databases. The small differences suggest that our results are robust. Simulation parameters: $m = 25$, $n/N = 1,000$, $N = 1,024$, BRITE – Internet topology

we again ran multiple experiments with different N. In each experiment, we counted the number of majority votes which had environments with sizes 0 through N, where a majority had an environment size zero in a given node if it was not initiated at that node (we only counted, of course, majority votes that were initiated at *some* node). A majority vote had size one at a given node if it was initiated, but no messages were ever received in its context. Otherwise, the environment size of node u is the number of neighboring nodes whose data was collected to u. We report the environment sizes for both the average node and the one with the largest environment.

The results, averaged across 10 random databases, are depicted in Figs. 4 and 5. Figure 4 describes the environment size for the average node for BRITE and de Bruijn topologies. Note that both the y-axis and the x-axis here are logarithmic. The vast majority of the votes has average environment size zero, which means that they were initialized in only a few nodes. In Fig. 5 however, no majorities have zero environment size, because each has been initialized in at least one node. Looking at the bars for average environment sizes one and up, we can see that the sizes resemble a power-law distribution. One important exception to this rule is the last bar for each N. As can be seen, there are always some majorities that run into a tie and therefore have environment size N. The pattern discussed above repeats itself for both the average node and the one with the largest environment (Fig. 5). In the latter, naturally, the trend is less clear because of the fluctuating nature of worst case measurements.

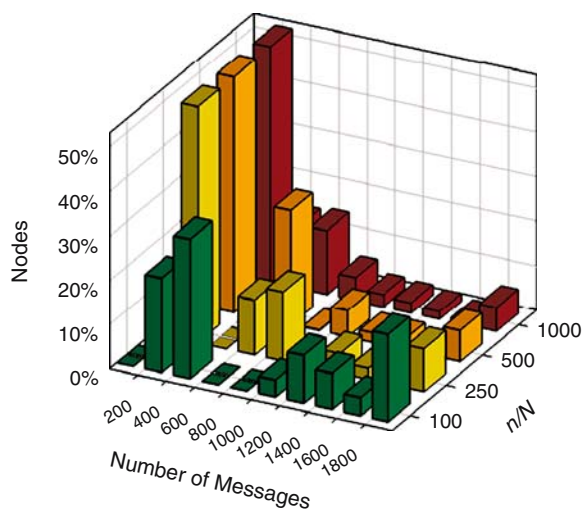


Fig. 7 Effect of n/N (the number of clients on each node) on the number of messages. The graph shows four message distributions for four different n/N values. As n/N grows the algorithm sends less messages. Simulation parameters: $m = 20$, $N = 512$, BRITE – Internet topology

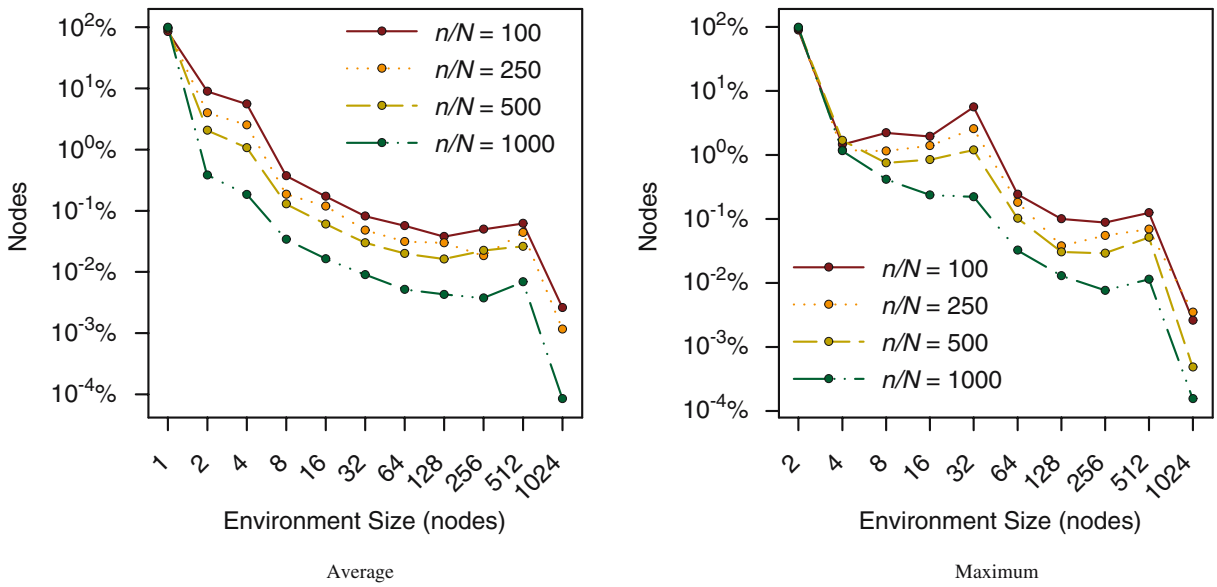


Fig. 8 Effect of n/N (the number of clients on each node) on the environment size. Two graphs show the maximum and average number of messages histograms for four different n/N values. As n/N grows, environments (both maximum and average) become smaller, and therefore, the algorithm becomes more and more local. Simulation parameters: $m = 20$, $N = 512$, BRITE – Internet topology

5.4 Robustness of our Results

One question that is often asked about average (or worst case) results is how robust they are. In Fig. 6 we depict, for one of the experiments above, the range of results that were calculated. As can be seen, the difference between the minimal and maximal number of messages is not significant. The difference between the minimal and maximal environment size for the average node can be quite large because in certain instances of the problem there happen to be no ties at all.

5.5 Size of Database

In the last experiment, we measured the effect of the size of the local database (n/N) on the locality and message load of the algorithm. n/N is dependent mostly on the characteristics of the domain, e.g., the number of files on the disk of an average e-Mule peer or the buffer size of a sensor. We expect that as this number increases, the performance of the algorithm will improve because local statistics will become more accurate and more representative of the global ones.

We varied n/N from 1,000 points down to 100. Figure 7 depicts the number of messages sent by each node, and Fig. 8 depicts the sizes of the average and largest environments. As can be seen, the same trends we discussed above persist when n/N equals 1,000, 500, and 250.[1] For n/N equals 100, the number of messages and average environment size grow significantly. However, since these experiments exhibit local behavior as well, we assume that, had we run our experiments with larger N values, the trend would have been visible for smaller n/N as well. Unfortunately, the performance of our simulator restricts us at this phase to $N = 1,024$.

6 Conclusions and Further Research

We have described a new facility location algorithm suitable for large scale distributed systems. The characteristics which qualify the algorithm for systems of this type are its message efficiency, its strong local pruning, and its ability to efficiently sustain failures and changes in the input. All these qualities stem from the algorithm's *local* nature.

[1]The slight discrepancies between the results for $n/N = 1,000$ in Fig. 7 and Figs. 3 and 4 are due to different m values used in those experiments.

Besides its immediate value, the algorithm serves to demonstrate that various data mining problems can be solved in large scale distributed settings through reduction to basic primitives like majority-vote. These primitives can later be solved by efficient local algorithms. We believe than in-network data mining may very well become one of the key techniques for accessing the output of these systems.

Appendix

Correctness Proof

We first prove that the algorithm terminates and then proceed to show that after termination the output is correct. Our proof relies on the correctness of Algorithm 1, which is proved in [28].

Global termination

Global termination is achieved if the following two conditions hold for long enough time: (1) the input of no node changes, and (2) the network topology does not change. We say that the algorithm is terminated if all messages reached their destination and no new messages are sent. Note that no node can detect the satisfaction of those global conditions, therefore from a local point of view of a single node the algorithm never terminates.

Global termination of Algorithm 2 follows directly from the global termination of Algorithm 1. Every message of Algorithm 2 is an augmented message of some majority vote (i.e. Algorithm 1's message). Therefore, when all majority votes terminate, Algorithm 2 stops sending messages and terminates.

Correctness

We will now prove the correctness of Algorithm 2. Namely, we will prove that after the algorithm terminates, the output of each node is equal to the output of the sequential facility location algorithm running on the entire database.

Note that for any node u and $\forall \langle C, i, j\rangle \in ActiveSet^u$, $Majority^u_C\langle i, j\rangle$ is active. This is implied by two events in the pseudo code: "Initialization", and "On change in output of $Majority^u_C\langle i, j\rangle$". Moreover, after termination, in every node u the structures R^u and $MV(C), \forall C \in R^u$, are updated. This is because after last majority vote output change, those structures were updated in "On change in output of $Majority^u_C\langle i, j\rangle$" event.

Lemma 1 *Every node in Algorithm 2 outputs the same result as a sequential facility location algorithm would output.*

Proof Sequential facility location algorithm begins with some initial configuration and then in each step improves it. The algorithm stops when no improvement can be done. In the improvement step the sequential algorithm moves, closes, or opens one facility, or makes no change if this is the best option. If no change was made in the last iteration, the algorithm stops.

We now prove that the R^u vector is the same in each node and that it contains the same sequence of configurations that the sequential algorithm encounters during hill climbing process. By induction on R^u's length.

Base $|R^u| = 1$, thus by definition $C^u_1 = \{M[1]\}$ which is the same on all nodes, and being the initial configuration the same as the first configuration in the sequential algorithm.

Step We assume correctness for $|R^u| = i$ and prove for $|R^u| = i + 1$. By assumption C^u_i is the same in all nodes and the same as in the sequential algorithm. By definition in pseudo code: $C^u_{i+1} = Next(C^u_i)[Pivot^u_{k-1}(C^u_i)]$. Therefore, by Corollary 1, C^u_{i+1} is the lowest cost configuration among all configurations in $Next(C^u_i)$ and is the same on all nodes. Since $Next(C^u_i)$ contains all the configurations that can be produced by moving, opening, or closing one facility of making no change, C^u_{i+1} is the same as $i + 1$'s configuration of the sequential algorithm.

Finally, since the last configuration, C^u_l, by definition equals C^u_{l-1}, it satisfies the stop condition of the sequential algorithm.

We saw that the R^u vector of any node contains the same sequence of configurations that a sequential facility location algorithm would

generate during its hill climbing process. Thus the output configurations of both algorithms are identical. □

Corollary 1 *Let $N = Next(C)$ for some $C \in R^u$ for all nodes u: $Cost(N[Pivot^u_{k-1}(C)]) \leq Cost(N[j])$ for $j \in \{1, \ldots, |N|\}$.*

Proof By Lemma 1 we have

$$S^u_k(C) = \left\{ j \in \{1, \ldots, |N|\} \;\middle|\; Cost(N[j]) < Cost(N[Pivot^u_{k-1}(C)]) \right\}$$

which by definition of k is an empty set. Therefore, there is no configuration whose cost is lower than $Cost(N[Pivot^u_{k-1}(C)])$, which proves the corollary. □

We now prove the correctness of a single step of the hill-climbing process.

Lemma 2 *Let $N = Next(C)$ for some $C \in R^u$ for all nodes u: $S^u_i(C) \left\{ j \in \{1, \ldots, |N|\} \middle| Cost(N[j]) < Cost(N[Pivot^u_{i-1}(C)]) \right\}$.*

Proof By induction on i.

Base For $i=1$ the lemma is trivially satisfied, since $S^u_1(C) = \{1, \ldots, |N|\}$ by definition and $Pivot^u_0(C)$ is undefined.

Step We assume correctness for i and prove for $i+1$. By definition:

$$S^u_{i+1}(C) = \left\{ j \in S^u_i(C) \middle| Majority^u_C\langle j, Pivot^u_i(C)\rangle.out = negative \right\}$$

by definition of $ActiveSet^u$ we have $\langle C, j, Pivot^u_i(C)\rangle \in ActiveSet^u$ and thus all those majorities are active, and we can use Lemma 1 which implies

$$S^u_{i+1}(C) = \left\{ j \in S^u_i(C) \;\middle|\; Cost(N[j]) < Cost(N[Pivot^u_i(C)]) \right\}$$

by inductive assumption

$$\forall j \notin S^u_i(C) : Cost(N[j]) \geq Cost(N[Pivot^u_{i-1}(C)])$$

and since $Pivot^u_i(C) \in S^u_i(C)$ we have

$$Cost(N[Pivot^u_{i-1}(C)]) \geq Cost(N[Pivot^u_i(C)])$$

therefore,

$$\forall j \notin S^u_i(C) : Cost(N[j]) \geq Cost(N[Pivot^u_i(C)])$$

combining this with the consequence of Lemma 1 we get

$$S^u_{i+1}(C) = \left\{ j \in \{1, \ldots, |N|\} \;\middle|\; Cost(N[j]) < Cost(N[Pivot^u_i(C)]) \right\}$$

□

We conclude with a lemma which proves that the comparison of a pair of configuration costs can be correctly done using one majority vote.

Lemma 3 *Let $N = Next(C)$ for some $C \subseteq M$. For any two configurations $N[i]$, $N[j]$ $(j < i)$ and active majority vote $Majority^u_C\langle i, j\rangle$ on every node u: $Cost(N[i]) < Cost(N[j])$ iff $Majority^u_C\langle i, j\rangle.out = negative$.*

Proof At first we break the cost into components local to each node:

$$Cost(N[i]) < Cost(N[j])$$

iff

$$Cost(N[i]) - Cost(N[j]) < 0$$

iff

$$\sum_{p \in DB} cost(p, N[i]) + D(N[i]) - \sum_{p \in DB} cost(p, N[j]) - D(N[j]) < 0$$

iff

$$\sum_u \left(\sum_{p \in DB^u} cost(p, N[i]) - \sum_{p \in DB^u} cost(p, N[j]) \right) < D(N[j] - D(N[i])$$

According to the "Init of $Majority_C^u\langle i, j\rangle$" event in Algorithm 2: $s^u = \sum_{p \in DB^u} cost(p, N[i]) - cost(p, N[j])$, and $\gamma = D(N[j]) - D(N[i])$. Therefore,

$$\sum_u s^u < \gamma$$

but since $c^u = 0, \lambda = 0$ we have

$$\sum_u (s^u - \lambda c^u) < \gamma$$

By correctness of Algorithm 1, this is true if and only if

$$Majority_C^u\langle i, j\rangle.out = negative$$

□

References

1. Arya, V., Garg, N., Khandekar, R., Munagala, K., Pandit, V.: Local search heuristic for k-median and facility location problems. In: STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing, pp. 21–29. New York, NY, USA (2001)
2. Awerbuch, B., Bar-Noy, A., Linial, N., Peleg, D.: Compact distributed data structures for adaptive routing. In: STOC '89: Proceedings of the twentyfirst annual ACM symposium on Theory of computing, pp. 479–489. New York, NY, USA (1989)
3. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by local checking and correction (extended abstract). In: IEEE Symposium on Foundations of Computer Science, pp. 268–277. Los Alamitos, CA, USA (1991)
4. Balinski., M.: On finding integer solutions to linear programs. Proc. IBM Scientific Computing Symp. on Combinatorial Problems, pp. 225–248 (1966)
5. Birk, Y., Liss, L., Schuster, A., Wolff, R.: A local algorithm for ad hoc majority voting via charge fusion. In: DISC. (2004)
6. Charikar, M., Guha, S.: Improved combinatorial algorithms for the facility location and k-median problems. In: IEEE Symposium on Foundations of Computer Science, pp. 378–388 (1999)
7. Dhillon, I.S., Modha, D.S.: A data-clustering algorithm on distributed memory multiprocessors. In: Large-scale Parallel Data Mining. pp. 245–260 (2002)
8. Ester, M., Kriegel, H., Sander, J., Wimmer, M., Xu, X.: Incremental clustering for mining in a data warehousing environment. In: VLDB., pp. 323–333 (1998)
9. Ford, L., Fulkerson, D.: Flows in Networks. Princeton University Press, Princeton, NJ (1962)
10. Forman, G., Zhang, B.: Distributed data clustering can be efficient and exact. SIGKDD Explor. Newsl. **2**(2), 34–38 (2000)
11. Foti, D., Lipari, D., Pizzuti, C., Talia, D.: Scalable parallel clustering for data mining on multicomputers. In: IPDPS 00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, pp. 390–398. London, UK (2000)
12. Guha, S., Khuller, S.: Greedy strikes back: improved facility location algorithms. In: SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)(1998)
13. Gupta, P., Kumar, P.R.: The capacity of wireless networks. IEEE Trans. Inf. Theory **46**(2), 388–404 (2000)
14. Jaffe, J., Moss, F.: A responsive routing algorithm for computer networks. IEEE Trans. Commun. pp. 1758–1762 (1982)
15. Jain, K., Vazirani, V.V.: Primal–Dual approximation algorithms for metric facility location and k-median problems. In: IEEE Symposium on Foundations of Computer Science. pp. 2–13 (1999)
16. Kaashoek, F., Karger, D.: Koorde: A simple degree-optimal distributed hash table. Peer-to-Peer Systems II: Second International Workshop (2003)
17. Kleinberg, J., Papadimitriou, C., Raghavan, P.: A microeconomic view of data mining. Data Mining and Knowledge Discovery **2**(4), 311–324 (1998)
18. Korupolu, M.R., Plaxton, C.G., Rajaraman, R.: Analysis of a local search heuristic for facility location problems. In: SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms). pp. 1–10 (1998)
19. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What cannot be computed locally!. In: PODC 04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing. New York, NY, USA,pp. 300–309 (2004)
20. Kutten, S., Patt-Shamir, B.: Time-adaptive self stabilization. In: PODC 97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing. New York, NY, USA, pp. 149–158 (1997)
21. Kutten, S., Peleg, D.: Fault-local distributed mending (extended abstract). In: PODC 95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing. New York, NY, USA, pp. 20–27 (1995)
22. Linial, N.: Locality in distributed graph algorithms. SIAM Comput. J. **21**(1), 193–201 (1992)
23. Medina, A., Lakhina, A., Matta, I., Byers, J.: BRITE: universal topology generation from a user's perspective. Technical report, Boston, MA, USA (2001)
24. Moscibroda, T., Wattenhofer, R.: Facility location: distributed approximation. In: PODC 05: Proceedings of the twenty-fourth annual ACMSIGACT-SIGOPS symposium on Principles of distributed computing. New York, NY, USA, pp. 108–117 (2005)
25. Naor, M., Stockmeyer, L.: What can be computed locally? In: STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pp. 184–193. New York, NY, USA, (1993)

26. Page, C.: Astrogrid and data mining. In: Starck, J.-L., Murtagh, F.D. (eds.) Proc. SPIE Vol. 4477, pp. 53–60, Astronomical Data Analysis, pp. 53–60 (2001)
27. Wolff, R., Schuster, A.: Association rule mining in peer-to-peer systems. In: ICDM 03: Proceedings of the Third IEEE International Conference on Data Mining. Washington, DC, USA, pp. 363 (2003)
28. Wolff, R., Schuster, A.: Association rule mining in peer-to-peer systems. IEEE Trans. Syst. Man Cybern., Part B **34**(6), 2426–2438 (2004)