A Local Facility Location Algorithm
for Sensor Networks

Denis Krivitski[1], Assaf Schuster[1], and Ran Wolff[2]

[1] Computer Science Dept., Technion – Israel Institute of Technology
{denisk,assaf}@cs.technion.ac.il
[2] Computer Science Dept., University of Maryland, Baltimore County
ranw@cs.umbc.edu

Abstract. In this paper we address a well-known facility location problem (FLP) in a sensor network environment. The problem deals with finding the optimal way to provide service to a (possibly) very large number of clients. We show that a variation of the problem can be solved using a *local* algorithm. Local algorithms are extremely useful in a sensor network scenario. This is because they allow the communication range of the sensor to be restricted to the minimum, they can operate in routerless networks, and they allow complex problems to be solved on the basis of very little information, gathered from nearby sensors. The local facility location algorithm we describe is entirely asynchronous, seamlessly supports failures and changes in the data during calculation, poses modest memory and computational requirements, and can provide an anytime solution which is guaranteed to converge to the exact same one that would be computed by a centralized algorithm given the entire data.

1 Introduction

Determining the location of facilities which provide system related services is a major issue for any large distributed system. The resource limited scenario of a sensor network makes the problem far more acute. A well-placed resource (cache server, relay, or high-powered sensor, etc.) can tremendously increase the lifespan and the productivity of dozens or even hundreds of battery operated sensors. In many cases, however, the optimal location of such resources depends on dynamic characteristics of the sensors (e.g., their remaining battery power), the environment (e.g., level of radio frequency white noise), or the phenomena they monitor (e.g., frequency of changes). Thus, optimal placement cannot be computed a priori, independently of the system's state.

One example of a facility location problem may occur in sensor networks that, in addition to regular sensors, use a few dozen relays. Regular sensors are low-power motion sensing devices, which are distributed from the air, covering the area randomly. Instead, relays are equipped with large batteries and long range transceivers and are placed at strategic points by ground transportation. The purpose of the relays is to collect data from the sensors and transmit it to a command station whenever it is requested. However, the question remains how to best utilize the relays which in themselves have but limited resources. It would make sense to shut down a relay if there was only mild

activity in its nearby surroundings and report that activity via other relays. Note, however, that the amount of activity, the available resources of the relays and the motion sensors, as well as the environmental conditions in which they all operate may influence the decision, and these factors may vary over time. The optimal solution, thus, has to be regularly adjusted.

The facility location problem (FLP) has been extensively studied in the last decade. Like many other optimization problems, optimal facility location is NP-Hard [13]. Thus, the problem is usually solved using either a hill-climbing heuristic [14, 8, 4] or linear programming [11, 19, 12]. These approaches achieve constant factor approximation of the globally optimal solution [1]. FLP also has several versions, primarily divided according to whether facilities have finite or infinite capacity (i.e., *capacitated* vs. *uncapacitated* FLP).

To the best of our knowledge FLP has never been studied specifically in a distributed setting. Nevertheless, it is easy to see how distributed formulation of related hill-climbing algorithms such as k-means and k-median clustering [5, 7, 6] can be adapted to solve distributed FLP. We note, however, that all previous work on distributed clustering assumes tight cooperation and synchronization between the processors containing the data, and a central processor that collects the sufficient statistics needed in each step of the hill-climbing heuristic. Such central control is not practical in wireless networks, because of the energy required and because it 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: any such change must be reported to the center, for fear it might alter the result.

Thus, it is clear that other features are required to qualify an algorithm for sensor networks. The most important of these are the following: the ability to perform in a routerless network (i.e., to be driven by data rather than by address), the ability to calculate the result in-network rather than collect all of the data to a central processor (which would quickly exhaust bandwidth [9]), and the ability to locally prune redundant or duplicate computations. These three 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 processor usually computes the result using information gathered from just a 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 do not change its output). Local algorithms have been mainly studied in the context of graph related problems [2, 15–18]. Most recently, [20] demonstrated that local algorithms can be devised for complex data analysis tasks, specifically, data mining of association rules in distributed transactional databases. The algorithm presented in [20] features local pruning of false propositions (candidates), in-network mining, asynchronous execution, and resilience to changes in the data and to partial failure during execution.

In this work we develop a local algorithm that solves a specific version of FLP, one in which uncapacitated resources can be placed in any k out of m possible locations. Initiating our algorithm from a fixed resource location, say, in the first k locations, we show that the computation required to reach agreement on a single hill-climbing step—

moving one resource to 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 [20] to develop an algorithm which locally computes the exact same solution a hill-climbing algorithm would compute, had it been given the entire data.

In a series of experiments employing networks of up to 10,000 simulated sensors, 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 of the majority voting algorithm originally described in [20]. Section 4 describes the local k-facility location algorithm. Finally, in Section 5, we present some of the experimental results.

2   Notations, Assumptions, and Problem Definition

We assume a large number N of processors, which can communicate with one another by sending messages. We further assume that communication among neighboring processors is reliable and ordered. This assumption can be enforced using standard numbering, ordering and retransmission mechanisms. For brevity, we assume an undirected communication tree. As shown in [3], such a tree can be efficiently constructed and maintained using variations of Bellman-Ford algorithms [10]. Finally, we assume that failure is fail-stop and that the neighbors of a processor that is disconnected or reconnected for any reason are reported.

Given a database DB containing *input points* $\{p_1, p_2, \ldots, p_n\}$, a set M of m possible *locations*, and a cost function $d : DB \times M \rightarrow \mathcal{R}^+$, the task of a *k-facility location* algorithm is to find a set of *facilities* $C \subset M$ of size k, such that the cumulative distance of points from their nearest facility $\sum_{p_i \in DB} \min_{c \in C} d(p_i, c)$ is minimized.

To relate these definitions to the example given in the introduction, consider a database that includes a list of events that occurred in the last hour. Each event would have a heuristic estimate of its importance. Furthermore, each sensor would evaluate its hop distance from every relay and multiply this by the heuristic importance of each event to produce its cost. Given this input, a facility location algorithm would compute the best combination of relays such that the most important events need not travel far before they reach the nearest relay. The less important events, we assume, would be suppressed either in the sensor that produced them, or in-network by other sensors.

An *anytime k*-facility location algorithm is one which, at any given time during its operation, outputs a placement for the location such that the cost of this ad hoc output improves with time until the optimal solution is found. A *distributed k*-facility location algorithm would compute the same result even when the DB is partitioned into N mutually exclusive databases $\{DB^1, \ldots, DB^N\}$, each of which is deposited with a separate processor, and these are then allowed to communicate by passing messages to

each other. A *local k-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.

The *hill-climbing* heuristic for k-facility location begins from an agreed upon placement of the facilities (henceforth, *configuration*). Then, it finds a single facility and a single empty location, such that by moving the facility to that free location the cost of the solution is reduced to the greatest possible degree. If such a step exists, the algorithm changes the configuration accordingly and iterates. If any configuration which can be produced by 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 that computes the hill-climbing heuristic for k-facility location.

3   Local Majority Voting

Our k-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 [20], which we use as the main building block for the algorithm. The algorithm assumes that messages sent between neighbors are reliable and ordered, and that processor failure is reported to the processor's neighbors. These assumptions can easily be enforced using standard numbering, retransmission, ordering, and heart-beat mechanisms. The algorithm makes no assumptions on the timeliness of message transfer and failure detection.

Given a set of processors 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 $\sum_u s^u / \sum_u c^u \geq \lambda$. We call Δ the number of excess votes.

The following local algorithm decides whether $\Delta \geq 0$. Each processor $u \in V$ computes the number of excess votes in its own poll, $\delta^u = s^u - \lambda c^u$. It then stores the number of excess votes it reported to each neighbor v in δ^{uv} and the number of excess votes which have been reported to it by v in δ^{vu}. Processor u computes the total number of excess votes it knows of, 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 agreed on 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 $\Delta^u - \delta^{vu}$ – thus setting Δ^{uv} to Δ^u, and 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. Processor 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 problem is to determine when sending a message can be avoided, despite the fact that the local knowledge has changed. In the algorithm presented here, there are two cases in which a processor u would send a message to a neighbor v: 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 Δ^u is then computed over a smaller set of edges and may change as a result). This means the algorithm is event driven and requires no synchronization.

We modify the local majority voting algorithm slightly in order to apply it to k-facility location. We add the ability to suspend and reactivate the vote using corresponding events. A processor whose voting has been suspended will continue to receive messages and modify the corresponding local variable, but will not send any messages. When the vote is activated, the processor will always check whether it is required to send a message as a result of the information ir received while in a suspended state.

4 Majority Based *k*-Facility Location

The local k-facility location algorithm which we now present is based upon three fundamental ideas: The first is to have every processor optimistically perform hill-climbing steps without waiting for a decision as to which is the globally optimal step. Having taken these steps, the processor continues to validate the agreement of the steps it took with the globally correct one. 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 on which pair of possible steps is more costly (i.e., more popular). 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.

4.1 Optimistic 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 — especially if one assumes that the data may change with time, and thus the global data can never be determined. Instead, when performing parallel hill-climbing, we let each processor proceed uphill whenever it computes the best step according to the data it currently possesses. Then, we use local majority voting (as we describe next) to make sure that processors which have taken erroneous steps will eventually be corrected. In the event that a processor is corrected, computations associated with configurations that were wrongly chosen are put on hold. These configurations are put aside in a designated cache in case additional data that accumulates will prove them correct after all.

We term the sequence of steps selected by processor 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 k locations in M. C_l^u is the ad hoc solution C^u. u refrains from developing another configuration following a given C_l^u when no possible

step can improve on the cost of the current configuration, or when two or more steps still compete on providing the best improvement.

Since the computation of all of the configurations along every processor's path is concurrent, messages sent by the algorithm contain a *context* – the configuration to which they relate. Since the computation is also optimistic, it may well happen that two processors u and v temporarily 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 to be *out of context*. Rather than being accepted by u, it 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 by which they were received.

4.2 Locally Computing the Best Possible Step

For each configuration $C_a^u \in R^u$, processor u computes the best possible step as follows. First, it generates the set of possible configurations $Next[C_a^u]$, such that each member of $Next[C_a^u]$ is a configuration that replaces one of the members of C_a^u with a non-member location from $M \setminus C_a^u$. Next, for each $C \in \{C_a^u\} \cup Next[C_a^u]$ and each $p \in DB^u$, the cost incurred by p in C is computed such that $cost(p, C) = \min_{x \in C} \{d(p, x)\}$. Finally, for every $C_i, C_j \in Next[C_a^u]$, where $i < j$, processor u initiates a majority vote, $Majority_{C_a^u}^u \langle i, j \rangle$, which compares their relative costs and eventually computes $\Delta_{C_a^u}^u \langle i, j \rangle \geq 0$ if the global cost of C_i is higher than that of C_j (as we explain below). Correctness of the majority vote process guarantees that the best configuration $C_{i_{best}} \in \{C_a^u\} \cup Next[C_a^u]$ will eventually have negative $\Delta_{C_a^u}^u \langle i_{best}, j \rangle$ for all $j > i_{best}$, and positive $\Delta_{C_a^u}^u \langle j, i_{best} \rangle$ for all $j < i_{best}$. Hence, the algorithm will optimistically choose C_i as the next configuration whenever C_i has the maximal number of majority votes indicating it is the better one (even if some votes indicate otherwise).

To determine, by means of majority vote, which of two configurations has the lower cost, we set for every processor u, $\delta^u \langle i, j \rangle = \sum_{p \in DB^u} cost(p, C_i) - cost(p, C_j)$. This can be done for any $\delta^u \langle i, j \rangle \in [-x, x]$ by choosing, for example, $c = 2x$, $\lambda = 1/2$ and $s = x - \lambda$. Note that, as shown in [20], 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 $\sum_{p \in DB} cost(p, C_i) - \sum_{p \in DB} cost(p, C_j) = \sum_u \sum_{p \in DB^u} [cost(p, C_i) - cost(p, C_j)]$. Hence, if the vote comparing the cost of C_i to that of C_j determines that $\Delta^u \langle i, j \rangle \geq 0$, this proves the cost of C_i is larger than that 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.
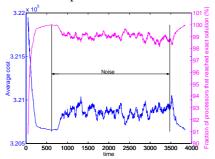
4.3 Pruning the Set of Comparisons

The subsections above show how it is possible to reduce k-facility location to a set of majority votes. However, these reductions overshoot the objective of the algorithm. This is because while a k-facility location really only requires that the *best* possible
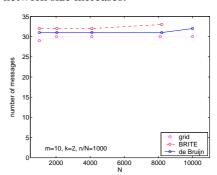
configuration be calculated given a certain configuration, the reduction above actually computes a *full order* on the possible configurations. This is problematic because, for some inputs, computing a full order may be arbitrarily more difficult (and hence, less local and more costly) than computing only the best option.

To overcome this problem, the algorithm is augmented with a pruning technique that limits the progress of comparisons such that only a small number of them actually take place. Given a configuration C, processor u sets as its *best* possible configurations the ones with the maximal number of majority votes—indicating that these configurations are less costly. It sets as *contending* those possible configurations which are indicated to be less costly than one of the best configurations. Processor u keeps track of its best and its contending configurations and the best and contending configurations of its neighbors in G^u. For this purpose u reports, with every message it sends, which configurations it currently considers best or contending. u retains in an active state those majority votes that compare a configuration to either its own or its neighbors' best and contending configurations. u suspends the rest of the majority votes, meaning that it will not send messages relating to them even if it accepts messages or data changes. It can also be shown that this pruning technique does not affect the correctness of the algorithm.

Fig. 1. Behavior in dynamic environment. The upper graph shows that more than 98% of nodes output the exact solution. The lower graph shows average solution cost. As the noise begins, the average cost deviates from the minimum but returns to it as the noise stops.

Fig. 2. Messages per processor. The graph shows the number of messages each processor sends for 3 topology types and 5 different sizes. The number of messages stays constant as the network size increases.

5 Experiments

To evaluate the algorithm's performance we ran it on simulated networks of up to ten thousand processors using up to one thousand input points in each processor. The main conclusions are as follows:

- The algorithm is local. The number of messages per processor remains constant as the network size increases (see figure 2). Moreover, the number of interlocutors of each processor don't grow with network size.
- The algorithm easily adapts to incremental data changes. In the dynamic data experiment, we swapped the databases of two random processors during a typical edge delay (we call those swaps *noise*). Throughout the experiment, not more than 2% of the processors deviated from the exact solution (see figure 1).
- The majority of processors converge rapidly. More than 90% of the processors converged to the exact solution after 4 edge delays. In addition, convergence time doesn't depend on network size.

References

1. Vijay Arya, Naveen Garg, Rohit Khandekar, Kamesh Munagala, and Vinayaka Pandit. Local search heuristic for k-median and facility location problems. In *STOC*, pages 21–29, 2001.
2. B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Compact distributed data structures for adaptive network routing. *Proc. 21st ACM STOC*, May 1989.
3. Y. Birk, L. Liss, A. Schuster, and R. Wolff. A local algorithm for ad hoc majority voting via charge fusion. In *DISC*, 2004.
4. Moses Charikar and Sudipto Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *FOCS*, pages 378–388, 1999.
5. Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining*, pages 245–260, 1999.
6. George Forman and Bin Zhang. Distributed data clustering can be efficient and exact. *SIGKDD Explor. Newsl.*, 2(2):34–38, 2000.
7. D. Foti, D. Lipari, C. Pizzuti, and D. Talia. Scalable Parallel Clustering for Data Mining on Multicomputers. In *IPDPS'00*, Cancun, Mexico, May 2000.
8. Guha and Khuller. Greedy strikes back: Improved facility location algorithms. In *SODA: ACM-SIAM)*, 1998.
9. P. Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, 46(2):388 – 404, 2000.
10. J.M. Jaffe and F.H. Moss. A responsive routing algorithm for computer networks. *IEEE Transactions on Communications*, pages 1758–1762, July 1982.
11. K. Jain, M. Mahdian, and A. Saberi. A new greedy approach for facility location problems.
12. Kamal Jain and Vijay V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. In *FOCS*, pages 2–13, 1999.
13. Jon Kleinberg, Christos Papadimitriou, and Prabhakar Raghavan. A microeconomic view of data mining. *Data Mining and Knowledge Discovery*, 1998.
14. Madhukar R. Korupolu, C. Greg Plaxton, and Rajmohan Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proc. ACM-SIAM*, pages 1–10, 1998.
15. S. Kutten and B. Patt-Shamir. Time-adaptive self-stabilization. *Proc. PODC*, pages 149–158, August 1997.
16. S. Kutten and D. Peleg. Fault-local distributed mending. *Proc. PODC*, August 1995.
17. N. Linial. Locality in distributed graph algorithms. *SIAM J. Comp.*, 21:193–201, 1992.
18. M. Naor and L. Stockmeyer. What can be computed locally? *STOC*, pages 184–193, 1993.
19. M. Sviridenko. An improved approximation algorithm for the metric uncapacitated facility location problem, 2002.
20. R. Wolff and A. Schuster. Association rule mining in peer-to-peer systems. In *Proc. ICDM*, Melbourne, Florida, 2003.