

# Cache placement in sensor networks under an update cost constraint

Bin Tang\*, Himanshu Gupta

*Department of Computer Science, Stony Brook University, Stony Brook, NY 11790, USA*

Available online 24 January 2007

---

## Abstract

In this paper, we address an optimization problem that arises in the context of cache placement in sensor networks. In particular, we consider the cache placement problem where the goal is to determine a set of nodes in the network to cache/store the given data item, such that the overall communication cost incurred in accessing the item is minimized, under the constraint that the total communication cost in updating the selected caches is less than a given constant. In our network model, there is a single server (containing the original copy of the data item) and multiple client nodes (that wish to access the data item). For various settings of the problem, we design optimal, near-optimal, heuristic-based, and distributed algorithms, and evaluate their performance through simulations on randomly generated sensor networks.

© 2006 Published by Elsevier B.V.

*Keywords:* Cache placement problem; Algorithm design; Approximation algorithm; Sensor networks

---

## 1. Introduction

Advances in embedded processing and wireless networking have made possible the creation of sensor networks [1,12]. A sensor network consists of sensor nodes with short-range radios and limited on-board processing capability, forming a multi-hop network of irregular topology. Sensor nodes must be powered by small batteries, making energy efficiency a critical design goal. There has been a significant interest in designing algorithms, applications, and network protocols to reduce energy usage of sensors. Examples include energy-aware routing [17], energy-efficient information processing [9,12], and energy-optimal topology construction [31]. We focus on designing techniques to conserve energy in the network by caching data items at selected sensor nodes in a sensor network. The techniques developed in this paper are orthogonal to some of the other mentioned approaches, and can be used in combination with them to conserve energy.

Existing sensor networks assume that the sensors are preprogrammed and send data to a sink node where the data is aggregated and stored for offline querying and analysis. Thus, sensor networks provide a simple sample-and-gather service, possibly with some in-network processing to minimize communication cost and energy consumption. However, this view of sensor network architecture is quite limited. With the rise in embedded processing technology, sensor networks are set to become a more general-purpose, heterogeneous, distributed databases that generate and process time-varying data. As energy and storage limitations always remain an issue—as much of it comes from pure physical limitations—new techniques for efficient data handling, storage, and dissemination must be developed. We take a

---

\* Corresponding author.

*E-mail addresses:* [bintang@cs.sunysb.edu](mailto:bintang@cs.sunysb.edu) (B. Tang), [hgupta@cs.sunysb.edu](mailto:hgupta@cs.sunysb.edu) (H. Gupta).

general view of the sensor network where a subset of sensor nodes (called *servers*) generate data and another subset of nodes (called *clients*) consume this data. The data generation and consumption may not be synchronous with each other, and hence, the overall communication cost can be optimized by caching generated data at appropriately selected intermediate nodes. In particular, data-centric sensor network applications which require efficient data dissemination [6,18] benefit from effective data caching strategies.

In our model of the sensor network, there is a single data item at a given server node, and many client nodes. (See Section 6 for a discussion on multiple data items and servers.) The server is essentially the data item producer and maintains the original copy of the item. All the nodes in the network cooperate to reduce the overall communication cost of accessing the data via a caching mechanism, wherein any node in the network can serve as a cache. A natural objective in this context could be to select cache nodes such that the sum of the overall access and update cost is minimized. However, such an objective does not guarantee anything about the general distribution of energy usage across the sensor network. In particular, the updates always originate from the server node, and hence, the server node and the surrounding nodes bear most of the communication cost incurred in updating. Hence, there is a need to constrain the total update cost incurred in the network, to prolong the lifetime of the server node and the nodes around it—and hence, possibly of the sensor network. Thus, we address the cache placement problem to minimize the total access cost under an update cost constraint. More formally, we address the problem of selecting nodes in the network to serve as caches in order to minimize the total access cost (communication cost incurred in accessing the data item by all the clients), under the constraint that the total update cost (communication cost incurred in updating the cache nodes using an optimal Steiner tree over the cache nodes and the server) is less than a given constant. Since we are considering only a single data item, we do not need to consider memory constraints of a node.

*Paper outline.* We start with formulating the problem and a discussion on related work in Section 2. For the cache placement problem under an update cost constraint, we consider a tree topology and a general graph topology of the sensor network. For the tree topology, we design an optimal dynamic programming algorithm in Section 3. The optimal algorithm for the tree topology can be applied to the general graph topology by extracting an appropriate tree from the given network graph. For the general graph topology, we consider a simplified multiple-unicast update cost model, and design a constant-factor approximation algorithm in Section 4.1. In Section 4.2, we present an efficient heuristic for the general cache placement problem under an update cost constraint, i.e., for a general update cost model in general graph topology. In Section 4.3, we present an efficient distributed implementation. Finally, we present simulation results in Section 5, and give concluding remarks in Section 6.

## 2. Problem formulation and related work

In this section, we formulate the problem, starting with a description of the sensor network model.

*Sensor network model.* A sensor network consists of a large number of sensor nodes distributed randomly in a geographical region. Each sensor node has a unique identifier (ID). Each sensor node has a radio interface and can communicate directly with some of the sensor nodes around it. For brevity, we sometimes just use *node* to refer to a sensor node. The sensor network can be modeled as an undirected weighted graph  $G = (V, E)$ , where  $V$  is the set of nodes, and  $E$  is the set of weighted edges in the graph. Two network nodes that can communicate directly with each other are connected by an edge in the graph. The edge weight may represent a link metric such as loss rate, delay, or transmission power. We use  $d_{ij}$  to denote the weighted distance between any two nodes  $i$  and  $j$  in  $G$ . The network has a data item, which is stored at a unique node called a *server*, and is updated at a certain update frequency. Each sensor node could be a client node. A client node  $i$  requests the data item with an *access frequency*  $a_i$ . The cost of accessing a data item (*access cost*) by a node  $i$  from a node  $j$  (the server or a cache) is  $a_i d_{ij}$ , where  $d_{ij}$  is the weighted distance between nodes  $i$  and  $j$ .

*Problem.* Informally, we address the following *cache placement problem* in sensor networks. Select a set of nodes to store copies of the data item such that the total access cost is minimized under a given update cost constraint. The total access cost is the sum of all individual access costs over all clients for accessing the data item from the nearest node (either a cache or the server) having a copy of the data item. The *update cost* incurred in updating a set of caches  $M$  is modeled as the cost of the optimal Steiner tree [13] spanning the server and the set of caches. This problem is obviously NP-hard, as even the Steiner tree problem is known to be NP-hard [4]. We look at the problem in various stages—a tree topology, a graph topology with a simplified update cost model, a graph topology with a general update cost model—and present optimal, approximation, and heuristic-based algorithms respectively.

More formally, given a sensor network graph  $G = (V, E)$ , a server  $r$  with the data item, and an update cost  $\Delta$ , select a set of cache nodes  $M \subseteq V$  ( $r \in M$ ) to store the data item such that the total access cost

$$\tau(G, M) = \sum_{i \in V} a_i \times \min_{j \in M} d_{ij}$$

is minimum, under the constraint that the total update cost  $\mu(M)$  is less than a given constant  $\Delta$ , where  $\mu(M)$  is the cost (i.e., weight) of the minimum weighted Steiner tree over the set of nodes  $M$ . We use the words cost and weight interchangeably. In the definition *all* network nodes are considered as potential clients. If some node  $i$  is not a client, the corresponding  $a_i$  would be zero.

*Related work.* The general problem of determining optimal cache placements in an arbitrary network topology has similarity to two problems widely studied in graph theory viz., the facility location problem and the  $k$ -median problem. Both problems consider only a single facility type (data item) in the network. In the facility-location problem, setting up a cache at a node incurs a certain fixed cost, and the goal is to minimize the sum of total access cost and the setting-up costs for all the caches, without any constraint. On the other hand, the  $k$ -median problem minimizes the total access cost under the number constraint, i.e., that at most  $k$  nodes can be selected as caches. Both problems are NP-hard, and a number of constant-factor approximation algorithms have been developed for each of the problems [8,10,19], under the assumption that the edge costs in the graph satisfy the triangular inequality. Without the triangular inequality assumption, either problem is as hard as approximating the set cover [19,24], and therefore cannot be approximated better than  $O(\log |V|)$  unless  $\mathbf{NP} \subseteq \tilde{\mathbf{P}}$ . Here,  $|V|$  is the size of the network.

Several papers in the literature circumvent the hardness of the facility-location and  $k$ -median problems by assuming that the network has a tree topology [23,32]. In particular, Li et al. [23] address the optimal placement of web proxies in a tree topology, essentially designing an  $O(n^3 k^2)$  time dynamic programming algorithm to solve the  $k$ -median problem optimally in a tree of  $n$  nodes. In other related work on cache placement in trees, Xu et al. [32] discuss placement of “transparent” caches to minimize the sum of reads and writes, Krishnan et al. [22] consider a cost model based on cache misses, and Kalpakis et al. [20] consider a cost model involving reads, writes, and storage. In sensor networks, which consist of a large number of energy-constrained nodes, the constraint on the number of cache nodes is of little relevance.

Cache placement has also been widely used in the web environment [3,5,28] and peer-to-peer networks [11,15,25] to alleviate problems such as server overloading, delayed response time, and inadequate bandwidth. In particular, Qiu et al. [28] have addressed effective placement of web server replicas over the Internet and evaluated several placement algorithms. Cohen and Shenkar [11] discuss the data replica placement problem in peer-to-peer networks and formulate the data replication strategies as a mapping from the query cost to the number of replicas. Relatively less work has been done on the cache placement problem in the specific context of ad hoc networks. Hara [15] addresses replica allocation methods for mobile ad hoc networks that can experience frequent disconnection. Yin and Cao [33] design and evaluate three simple cooperative caching techniques to efficiently support data access in ad hoc networks. In particular, they propose that intermediate nodes either cache data and/or nearest-cache path information to serve future requests. None of the described work offers any performance guarantee on the solutions.

Caching in sensor networks is equally important, since caching sensed information at intermediate nodes can greatly reduce overall communication cost which is the main source of energy consumption. Shenker et al. [30] propose data centric storage (DCS) as a data dissemination paradigm for sensor networks. In DCS, data is stored according to event type at corresponding sensor nodes. Data is also replicated to avoid overloading. Recently, Sheng et al. [29] study the storage node placement problem to minimize the total energy for data collection and data query. Intanagonwivat et al. [18] propose directed diffusion, a data dissemination paradigm for sensor networks, which adopts a data centric approach and enables diffusion to achieve energy savings by selecting empirically good paths and by caching/processing data in-network. Bhattacharya et al. [6] develop a distributed framework that improves energy consumption by application layer data caching and asynchronous update multicast. Prabh et al. [27] improve upon [6] by presenting and analyzing the optimality properties of Steiner tree data caching over all the cache nodes. None of the work takes into consideration the update cost incurred for the selected caches. We consider cache placement in sensor networks wherein the objective is to minimize the access cost under the constraint of maximum allowable update cost. As mentioned before, the update cost is typically mostly borne by the server and the surrounding nodes, and hence, is a critical constraint. To the best of our knowledge, we are not aware of any prior work that considers the cache placement problem under an update cost constraint.

### 3. Tree topology

In this subsection, we address the cache placement problem under the update cost constraint in a tree network. The motivation of considering a tree topology (as opposed to a general graph model which we consider in the next section) is two fold. Firstly, data dissemination or gathering in sensor networks is typically done over an appropriately constructed network tree. Secondly, for the tree topology, we can actually design polynomial time *optimal* algorithms. Thus, we can apply such optimal algorithms for the tree topology to the general graph topology by extracting an appropriate tree (e.g., shortest-path tree or near-optimal Steiner tree connecting the clients) from the general graph. In Section 5, we show through extensive simulations that such a strategy of applying an optimal tree algorithm to a general graph topology yields heuristics that deliver near-optimal cache placement solutions.

Consider an ad hoc network tree  $T$  rooted at the node  $r$ . Since the communication edges are bidirectional, any node in the network could be designated as the root; thus, we assume that the root node  $r$  is also the server for the data item. The cache placement problem under update cost constraint in a tree topology can be formally defined as follows.

Given the tree network  $T$  rooted at  $r$ , a data item whose server is  $r$ , and an update cost constraint  $\Delta$ , find a set of cache nodes  $M \subseteq T$  ( $r \in M$ ) for storing copies of the data item, such that the total access cost

$$\tau(T, M) = \sum_{i \in T} a_i \times \min_{j \in M} d_{ij}$$

is minimized under the constraint that the total update cost  $\mu(M)$  is less than  $\Delta$ , where  $\mu(M)$  is the weight of minimum-weighted Steiner tree over  $M$ . The minimum weighted Steiner tree spanning over a set of nodes  $M$  is simply the smallest subtree connecting the root  $r$  to all the nodes in  $M$ .

#### 3.1. Dynamic programming algorithm

In this subsection, we present an optimal dynamic programming algorithm for the described cache placement problem under the update cost constraint in a tree topology. We first start with some subtree notation [23] that is needed to describe our dynamic programming algorithm.

*Subtree notation.* Consider the network tree  $T$  rooted at  $r$ . We use  $T_u$  to denote the subtree rooted at  $u$  in the tree  $T$  with respect to the root  $r$ ; the tree  $T_r$  represents the entire tree  $T$ . For ease of presentation, we use  $T_u$  to also represent the set of nodes in the subtree  $T_u$ . We use  $p(i)$  to denote the parent node of a node  $i$  in the tree  $T_r$ . Let  $\pi(i, j)$  denote the unique path from node  $i$  to node  $j$  in  $T_r$ , and  $d_{k, \pi(i, j)}$  denote the distance of a node  $k$  to the closest node on  $\pi(i, j)$ .

Let us assume a left to right ordering of the children at each node. Consider two nodes  $v$  and  $u$  in the network tree, where  $v$  in an ancestor of  $u$  in  $T_r$ . See Fig. 1(a). Let  $L_{v,u}$  be the subgraph induced by the set of nodes on the left of and excluding the path  $\pi(v, u)$  in the subtree  $T_v$ , and  $R_{v,u}$  be the subgraph induced by the set of nodes on the right of

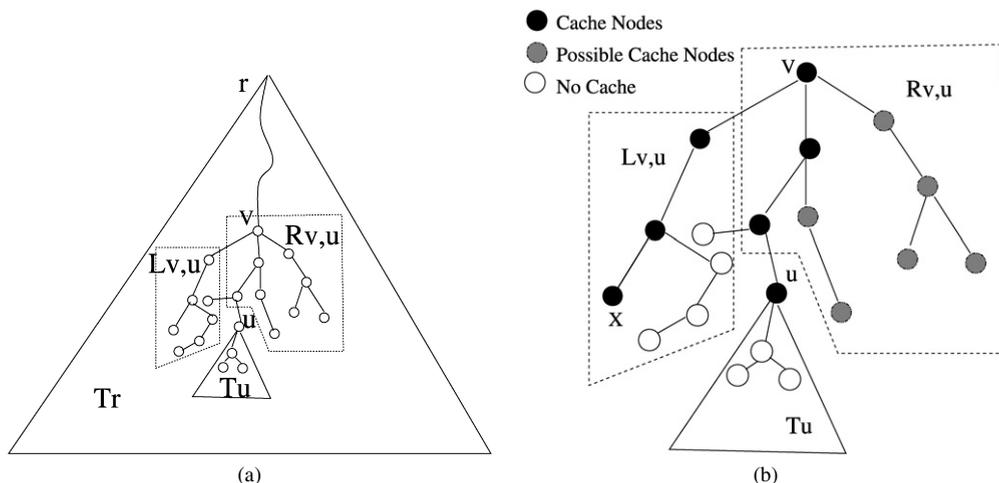


Fig. 1. Dynamic Programming algorithm for the tree topology. (a) Subtree notation. (b) Setup of the recursive equation.

and including the path  $\pi(v, u)$ , as shown in Fig. 1(a). It is easy to see  $T_v$  can be divided into three distinct subgraphs, viz.,  $L_{v,u}$ ,  $T_u$ , and  $R_{v,u}$ .  $T_u$  and  $R_{v,u}$  are trees, while  $L_{v,u}$  may not be a tree.

*DP algorithm.* Consider a subtree  $T_v$  and a node  $x$  on the leftmost branch of  $T_v$ . Let us assume that all the nodes on the path  $\pi(v, x)$  (including  $v$  and  $x$ ) have already been selected as caches. Let  $\tau(T_v, x, \delta)$  denote the optimal access cost for all the nodes in the subtree  $T_v$  under the *additional* update cost  $\delta$ , where we do *not* include the cost of updating the already selected caches on the path  $\pi(v, x)$ . We derive a recursive equation to compute  $\tau(T_v, x, \delta)$ , which would essentially yield a dynamic programming algorithm to compute  $\tau(T_r, r, \Delta)$ —the minimum value of the total access cost for the entire network tree  $T_r$  under the update cost constraint  $\Delta$ .

Let  $O_v$  be an optimal set (not including and in addition to  $\pi(v, x)$ ) of cache nodes in  $T_v$  that minimizes the total access time under the constraint of additional update cost  $\delta$ . Let  $u$  be the leftmost deepest node of  $O_v$  in  $T_v$ , i.e., the node  $u$  is such that  $L_{v,u} \cap O_v = \emptyset$  and  $T_u \cap O_v = \{u\}$ . It is easy to see that adding the nodes along the path  $\pi(v, u)$  to the optimal solution  $O_v$  does not increase the additional update cost incurred by  $O_v$ , but may reduce the total access cost. Thus, without loss of generality, we assume that the optimal solution  $O_v$  includes all the nodes along the path  $\pi(v, u)$  as cache nodes, if  $u$  is the leftmost deepest node of  $O_v$  in  $T_v$ .

*Recursive equation.* Consider an optimal solution  $O_v$  that minimizes  $\tau(T_v, x, \delta)$ , and let  $u$  be the leftmost deepest node of  $O_v$  in  $T_v$ .  $O_v$  does not include the nodes on  $\pi(v, x)$ . Based on the definition of  $u$  and possible cache placements, a node in  $L_{v,u}$  accesses the data item from either the nearest node on  $\pi(v, u)$  or the nearest node on  $\pi(v, x)$ . In addition, any node in  $T_u$  accesses the data item from the cache node  $u$ , while all other nodes (i.e., the nodes in  $R_{v,u}$ ) choose one of the cache nodes in  $R_{v,u}$  to access the data item. See Fig. 1(b). Thus, the optimal access cost  $\tau(T_v, x, \delta)$  can be recursively defined in terms of  $\tau(R_{v,u}, p(u), \delta - d_{u,\pi(v,x)})$ . The quantity  $d_{u,\pi(v,x)}$  denotes the shortest distance in  $T_v$  from  $u$  to a node on the path  $\pi(v, x)$  and hence, is the additional update cost incurred in updating the caches on the path  $\pi(v, u)$ . We first define  $S(T_v, x, \delta)$  as the set of nodes  $u$  such that the cost of updating  $u$  is less than  $\delta$ , the additional update cost. That is,

$$S(T_v, x, \delta) = \{u \mid u \in T_v \wedge (\delta > d_{u,\pi(v,x)})\}.$$

Now, the recursive equation can be defined as follows.

$$\tau(T_v, x, \delta) = \begin{cases} \sum_{i \in T_v} a_i \times d_{i,\pi(v,x)}, & \text{if } S(T_v, x, \delta) = \emptyset, \\ \min_{u \in S(T_v, x, \delta)} \left( \begin{aligned} &\sum_{i \in L_{v,u}} a_i \times \min(d_{i,\pi(v,u)}, d_{i,\pi(v,x)}) \\ &+ \sum_{i \in T_u} a_i d_{iu} \\ &+ \tau(R_{v,u}, p(u), \delta - d_{u,\pi(v,x)}) \end{aligned} \right), & \text{otherwise.} \end{cases}$$

In the recursive equation, the first case corresponds to the situation when the additional update cost  $\delta$  is not sufficient to cache the data item at any more nodes (other than already selected cache nodes on  $\pi(v, x)$ ). For the second case, we compute the total (and minimum possible) access cost for each possible value of  $u$ , the leftmost deepest additional cache node, and pick the value of  $u$  that yields the minimum total access cost. In particular, for a fixed  $u$ , the first term corresponds to the total access cost of the nodes in  $L_{v,u}$ . For a node in  $L_{v,u}$  the closest cache node is either on the path  $\pi_{v,x}$  or  $\pi_{v,u}$ . The second and third terms correspond to the total access time of nodes in  $T_u$  and  $R_{v,u}$  respectively. Since the tree  $T_u$  is devoid of any cache nodes, the cache node closest to any node in  $T_u$  is  $u$ . The minimum total access cost of all the nodes in  $R_{v,u}$  can be represented as  $\tau(R_{v,u}, p(u), \delta - d_{u,\pi(v,x)})$ , since the remaining available update cost is  $\delta - d_{u,\pi(v,x)}$  where  $d_{u,\pi(v,x)}$  is the update cost used up by the cache node  $u$ .

*Time complexity.* The recursive equation can also be used to compute the optimal *placement* of cache nodes required within  $T_v$  to attain the optimal cost  $\tau(T_v, x, \delta)$ . Also, our original problem of finding an optimal set of cache nodes in  $T_r$  under the given update constraint  $\Delta$  can be solved by evaluating  $\tau(T_r, r, \Delta)$ .

For time efficiency, we first precompute the terms  $\sum_{i \in L_{v,u}} a_i \times \min(d_{i,\pi(v,u)}, d_{i,\pi(v,x)})$  and  $\sum_{i \in T_u} a_i d_{iu}$  for all combinations of values of  $v, u$ , and  $x$ . It is easy to see that the precomputation can be done in  $O(n^4)$  time. Next, we compute  $\tau(T_v, x, \delta)$  for all values of  $v, x$  and  $\delta$ . Using the precomputed values, each such  $\tau(T_v, x, \delta)$  value takes  $O(n)$  time for computation. Since, there are a total of  $n^2 \Delta$  combinations of  $v, x$  and  $\delta$ , the overall time complexity of our dynamic programming algorithm is  $O(n^4 + n^3 \Delta)$ . In unweighted graphs, the time complexity can be simplified to  $O(n^4)$ , since  $\Delta = O(n)$ .

## 4. General graph topology

The tree topology assumption makes it possible to design a polynomial-time optimal algorithm for the cache placement problem under an update cost constraint. In this subsection, we address the cache placement problem in a general graph topology. In the general graph topology, the cache placement problem becomes NP-hard. Thus, our focus here is on designing polynomial-time algorithms with some performance guarantee on the quality of the solution.

As defined before, the total update cost incurred by a set of caches nodes is the minimum-weight of an optimal Steiner tree over the set of cache nodes and the server; we refer to this update cost model as the Steiner tree update cost model. Since the minimum-weighted Steiner tree problem is NP-hard in general graphs, we solve the cache placement problem in two steps. First, we consider a simplified multiple-unicast update cost model and present a greedy algorithm with a provable performance guarantee for the simplified model. Then, we improve our greedy algorithm based upon the more efficient Steiner tree update cost model.

### 4.1. Multiple-unicast update cost model

In this section, we consider the cache placement problem for a general network graph under a simplified update cost model. In particular, we consider the multiple-unicast update cost model, wherein we model the total update cost incurred in updating a set of caches as the sum of the individual shortest path lengths from the server to each cache node. More formally, the total update cost of a set  $M$  of cache nodes is  $\mu(M) = \sum_{i \in M} d_{si}$ , where  $s$  is the server. Using this simplified update cost model, the cache placement problem in general graphs for an update cost constraint can be formulated as follows.

*Problem under multiple-unicast model.* Given an ad hoc network graph  $G = (V, E)$ , a server  $s$  with the data item, and an update cost  $\Delta$ , select a set of cache nodes  $M \subseteq V$  ( $s \in M$ ) to store the data item such that the total access cost  $\tau(G, M) = \sum_{i \in V} a_i \times \min_{j \in M} d_{ij}$  is minimum, under the constraint that the total update cost  $\mu(M) = \sum_{i \in M} d_{si} < \Delta$ .

The cache placement problem with this simplified update cost model is still NP-hard, as can be easily shown by a reduction from the  $k$ -median problem. A number of constant-factor approximation algorithms have been proposed [8,19] for the  $k$ -median problem which can also be used to solve this cache placement problem. However, all the constant-factor approximation algorithms are based on the assumption that the edge weights in the network graph satisfy the triangular inequality. Moreover, the proposed approximation algorithms for the  $k$ -median problem cannot be easily extended to the more efficient Steiner tree update cost model. We present a greedy algorithm that returns a solution whose “access benefit” is at least 63% of the optimal benefit, where access benefit is defined as the reduction in total access cost due to cache placements.

*Greedy algorithm.* In this section, we present a greedy approximation algorithm for the cache placement problem under the multiple-unicast update cost constraint in general graphs, and show that it returns a solution with near-optimal reduction in access cost. We start with defining the concept of a benefit of a set of nodes which is important for the description of the algorithm.

**Definition 1 (Benefit of nodes).** Let  $A$  be an arbitrary set of nodes in the sensor network. The *benefit* of  $A$  with respect to an already selected set of cache nodes  $M$ , denoted as  $\beta(A, M)$ , is the decrease in total access cost resulting due to the selection of  $A$  as cache nodes. More formally,  $\beta(A, M) = \tau(G, M) - \tau(G, M \cup A)$ , where  $\tau(G, M)$ , as defined before, is the total access cost of the network graph  $G$  when the set of nodes  $M$  have been selected as caches. The *absolute benefit* of  $A$  denoted by  $\beta(A)$  is the benefit of  $A$  with respect to an empty set, i.e.,  $\beta(A) = \beta(A, \emptyset)$ .

The *benefit per unit update cost* of  $A$  with respect to  $M$  is  $\beta(A, M)/\mu(A)$ , where  $\mu(A)$  is the total update cost of the set  $A$  under the multiple-unicast update cost model.

Our proposed Greedy Algorithm works as follows. Let  $M$  be the set of caches selected at any stage. Initially,  $M$  is empty. At each stage of the Greedy Algorithm, we add to  $M$  the node  $A$  that has the highest benefit per unit update cost with respect to  $M$  at that stage. This process continues until the update cost of  $M$  reaches the allowed update cost constraint. The algorithm is now formally presented.

**Algorithm 1** (*Greedy Algorithm*).**Input:** A sensor network graph  $V = (G, E)$ .Update cost constraint  $\Delta$ .**Output:** A set of cache nodes  $M$ .**Begin** $M = \emptyset$ ;**while**  $(\mu(M) < \Delta)$ Let  $A$  be the node with maximum  $\beta(A, M)/\mu(A)$ . $M = M \cup \{A\}$ ;**end while**;**RETURN**  $M - \{A\}$  or  $\{A\}$ , whichever has the higher benefit.**END.**

The running time of the greedy algorithm is  $O(kn^2)$ , where  $n$  is the number of nodes in the network and  $k(\leq n)$  is the number of iterations.

*Performance guarantee of the Greedy Algorithm.* We now show that the Greedy Algorithm returns a solution that has a benefit at least 31% of that of the optimal solution. We start with presenting a lemma about the benefit function that leads to the final approximation result. The following lemma shows that the total benefit of a set of sets of nodes is at most the sum of the benefit of individual sets.

**Lemma 1.** Let  $O_1, O_2, \dots, O_m$  and  $M$  be arbitrary sets of nodes. Then,  $\beta(O_1 \cup O_2 \cup \dots \cup O_m, M) \leq \sum_{i=1}^m \beta(O_i, M)$ .

**Proof.** Without loss of generality, we prove the lemma for  $m = 2$ . By definition of the benefit function, we have

$$\beta(O_1 \cup O_2, M) = \beta(O_1, M) + \beta(O_2, M \cup O_1).$$

In the next paragraph, we show that

$$\beta(O_2, M \cup O_1) \leq \beta(O_2, M).$$

Thus, we get  $\beta(O_1 \cup O_2, M) \leq \beta(O_1, M) + \beta(O_2, M)$ .

To complete the proof, we now show that  $\beta(O_2, M) \geq \beta(O_2, M \cup O_1)$  for arbitrary sets of nodes  $M, O_1$ , and  $O_2$ . Let  $V$  be the set of all nodes in the given network graph, and let  $d(i, M)$  denote the distance (number of hops) from a node  $i$  to the closest node in the set  $M$ . For an arbitrary node  $i$  and arbitrary sets of nodes  $M, O_1$ , and  $O_2$ , we have

$$d(i, M) - d(i, M \cup O_2) \geq d(i, M \cup O_1) - d(i, M \cup O_1 \cup O_2).$$

To see this, consider the following three cases viz. the closest node to  $i$  in the set  $M \cup O_1 \cup O_2$  is in  $M$ , or  $O_1$  or  $O_2$ . In the first case, both sides of the equation are zero. For the second case, the right hand side is zero while the left hand side is positive. For the third case,  $d(i, M \cup O_1 \cup O_2) = d(i, M \cup O_2) = d(i, O_2)$  and  $d(i, M) \geq d(i, M \cup O_1)$ .

Now, by the definition of the benefit function, we have

$$\begin{aligned} \beta(O_2, M) &= \sum_{i \in V} a_i \times (d(i, M) - d(i, M \cup O_2)) \\ &\geq \sum_{i \in V} a_i \times (d(i, M \cup O_1) - d(i, M \cup O_1 \cup O_2)) \\ &= \beta(O_2, M \cup O_1). \quad \square \end{aligned}$$

Now, we show that the Greedy Algorithm returns a solution with near-optimal benefit. The proof technique used here is similar to that used in [14] for the closely related problem of selection of views in a data warehouse.

**Theorem 1.** Greedy Algorithm (Algorithm 1) returns a solution whose absolute benefit is of at least  $(1 - 1/e)/2$  times the absolute benefit of an optimal solution.

**Proof.** Let  $M$  be the set of cache nodes selected by the Greedy Algorithm at the end of the while loop, i.e., before the very last step. We show that the benefit of  $M$  is at least  $(1 - 1/e)$  times the absolute benefit of an optimal solution (we actually allow the optimal solution to use  $\mu(M)$  update cost). Since the Greedy Algorithm partitions  $M$  into two feasible solutions and returns the better of them, we get the desired approximation result.

Let  $\mu(M)$ , the total multiple-unicast update cost of  $M$ , be equal to  $k$ . Let the optimal solution using at most  $k$  units of multiple-unicast update cost be  $O$ .

Consider a stage when the greedy algorithm has already chosen a set  $M = G_l$  occupying  $l$  units of update cost with “incremental” benefits  $b_1, b_2, \dots, b_l$ . Incremental benefit  $b_i$  is defined as the increase in benefit when the node with the  $i$ th unit of update cost is added into the set of cache nodes. So, the absolute benefit of  $G_l$ ,  $\beta(G_l) = \sum_{i=1}^l b_i$ . Since, the absolute benefit of  $O \cup G_l$  is at least that of  $O$ , we have  $\beta(O, G_l) \geq \beta(O) - \sum_{i=1}^l b_i$ .

Let  $O = \{o_1, o_2, \dots, o_m\}$ . By Lemma 1 for the sets  $\{o_i\}$ 's, we have  $\beta(O, G_l) \leq \sum_{i=1}^m \beta(\{o_i\}, G_l)$ . Now, we show by contradiction that there exists a node  $o_h$  in  $O$  such that  $\beta(\{o_h\}, G_l)/\mu(o_h) \geq \beta(O, G_l)/k$ .  $O$  and  $G_l$  may not be disjoint. Let us assume that there is no such node  $o_h$  in  $O$ . Then,  $\beta(\{o_i\}, G_l) < (\beta(O, G_l)/k)\mu(o_i)$  for every node  $o_i \in O$ . Thus,  $\sum_{i=1}^m \beta(\{o_i\}, G_l) < (\beta(O, G_l)/k) \sum_{i=1}^m \mu(o_i) = \beta(O, G_l)$ , which violates Lemma 1. Therefore, there exists a node  $o_h$  in  $O$  such that

$$\beta(\{o_h\}, G_l)/\mu(o_h) \geq \beta(O, G_l)/k \geq \left( \beta(O) - \sum_{i=1}^l b_i \right) / k.$$

Now, the benefit per unit update cost with respect to  $G_l$  of the node  $C$  selected by the algorithm is at least that of  $o_h$ , which is at least  $(\beta(O) - \sum_{i=1}^l b_i)/k$ . Distributing the benefit of  $C$  over each of its unit update costs equally (for the purpose of analysis), we get

$$b_{l+j} \geq \left( \beta(O) - \sum_{i=1}^l b_i \right) / k \quad \text{for } 0 < j \leq \mu(C),$$

where  $\mu(C)$  is the update cost for  $C$ .

As the analysis is true for each node  $C$  selected at any stage, we have for  $0 < j \leq k$ :

$$b_j \geq \left( \beta(O) - \sum_{i=1}^{j-1} b_i \right) / k.$$

Rearranging a few terms, we get:

$$\left( \beta(O) - \sum_{i=1}^j b_i \right) \leq \left( \beta(O) - \sum_{i=1}^{j-1} b_i \right) (k - 1) / k.$$

Application of the equation  $j$  times, we get  $(\beta(O) - \sum_{i=1}^j b_i) \leq ((k - 1)/k)^j \beta(O)$ , which yields

$$\left( \beta(O) - \sum_{i=1}^k b_i \right) \leq ((k - 1)/k)^k \beta(O) \quad \text{when } j = k.$$

Thus, we get  $(\sum_{i=1}^k b_i)/\beta(O) \geq 1 - ((k-1)/k)^k \geq 1 - 1/e$ . Since, the absolute benefit of  $M$  is  $\beta(M) = \sum_{i=1}^k b_i$ , we have  $\beta(M)/\beta(O) \geq 1 - 1/e$ .  $\square$

#### 4.2. Steiner tree update cost model

Recall that the constant factor performance guarantee of the Greedy Algorithm described in the previous section is based on the multiple-unicast update cost model, wherein whenever the data item in a cache node needs to be updated, the updated information is transmitted along the individual shortest path between the server and the cache node. However, the more efficient method of updating a set of caches from the server is by using the optimal (minimum-weighted) Steiner tree over the selected cache nodes and the server. In this section, we improve the performance of our

Greedy Algorithm by using the more efficient Steiner tree update cost model, wherein the total update cost incurred for a set of cache nodes is the cost of the optimal Steiner tree over the set of nodes  $M$  and the server of the data item.

Since the minimum-weighted Steiner tree problem is NP-hard, we adopt the simple 2-approximation algorithm [13] for the Steiner tree construction, which constructs a Steiner tree over a set of nodes  $L$  by first computing a minimum spanning tree in the “distance graph” of the set of nodes  $L$ . We use the term 2-approximate Steiner tree to refer to the solution returned by the 2-approximation Steiner tree approximation algorithm. Based on the notion of 2-approximate Steiner tree, we define the following update cost terms.

**Definition 2** (*Steiner update cost*). The *Steiner update cost* for a set  $M$  of cache nodes, denoted by  $\mu'(M)$ , is defined as the cost of a 2-approximate Steiner tree over the set of nodes  $M$  and the server  $s$ .

The *incremental Steiner update cost* for a set  $A$  of nodes with respect to a set of nodes  $M$  is denoted by  $\mu'(A, M)$  and is defined as the increase in the cost of the 2-approximate Steiner tree due to addition of  $A$  to  $M$ , i.e.,  $\mu'(A, M) = \mu'(A \cup M) - \mu'(M)$ .

Based on these definitions, we describe the Greedy-Steiner Algorithm which uses the more efficient Steiner tree update cost model as follows.

**Algorithm 2** (*Greedy-Steiner Algorithm*).

**Input:** A network graph  $V = (G, E)$ .

Update cost constraint  $\Delta$ .

**Output:** The set of cache nodes  $M$ .

**Begin**

$M = \emptyset$ ;

**while**  $(\mu'(M) < \Delta)$

Let  $A$  be the node with maximum  $\beta(A, M)/\mu'(A, M)$ .

$M = M \cup \{A\}$ ;

**end while**;

**RETURN**  $M - \{A\}$  or  $\{A\}$ , whichever has the higher benefit.

**END.**

Unfortunately, there is no performance guarantee of the solution delivered by the Greedy-Steiner Algorithm. However, as we show in Section 5, the Greedy-Steiner Algorithm performs the best among all our designed algorithms for the cache placement problem under an update cost constraint.

### 4.3. Distributed implementation

In this subsection, we design a distributed version of the centralized Greedy-Steiner Algorithm (Algorithm 2). Using similar ideas as presented in this section, we can also design a distributed version of the centralized Greedy Algorithm (Algorithm 1). However, since the centralized Greedy-Steiner Algorithm outperformed the centralized Greedy Algorithm for all ranges of parameter values in our simulations, we present only the distributed version of Greedy-Steiner Algorithm. As in the case of the centralized Greedy-Steiner Algorithm, we cannot prove any performance guarantee for the presented distributed version. However, we observe in our simulations that the solution delivered by the distributed version is very close to that delivered by the centralized Greedy-Steiner Algorithm. Here, we assume the presence of an underlying routing protocol in the sensor network. Due to limited memory resources at each sensor node, a proactive routing protocol [26] that builds routing tables at each node is unlikely to be feasible. In such a case, a location-aided routing protocol such as GPSR [21] is sufficient for our purposes, if each node is aware of its location (either through GPS [16] or other localization techniques [2,7]).

*Distributed Greedy-Steiner Algorithm.* The distributed version of the centralized Greedy-Steiner Algorithm consists of rounds. During a round, each non-cache node  $A$  estimates its benefit per unit update cost, i.e.,  $\beta(A, M)/\mu'(A, M)$ . If the estimate at a node  $A$  is maximum among all its communication neighbors, then  $A$  decides to cache itself, and sends the estimated incurred update cost  $\mu'(A, M)$  to the server. During each round, a number of

sensor nodes may decide to cache the data item according to this criteria. At the end of each round, the server node sums the update cost incurred by newly added cache nodes, and calculates the remaining update cost by deducting it from the given update cost constraint. Then the remaining update cost is broadcast by the server to the entire network and a new round is *initiated*. To avoid many cache nodes being selected in the first round, we can have a node selecting itself as a cache node only if its estimate of benefit per unit cost estimate is the maximum among all its  $k$ -hop neighbors, where  $k > 1$ . The constant  $k$  may be chosen iteratively, until the number of nodes selecting themselves is small enough. We do not need to assume a synchronized mode, since each round is initiated by the server using a message. If there is no remaining update cost, then the server decides to discard some of the recently added caches (to keep the total update cost under the given update cost constraint), and the algorithm terminates. In this case, the server can deal with it by order. The algorithm is now formally presented.

**Algorithm 3** (*Distributed Greedy-Steiner Algorithm*).

**Input:** A network graph  $V = (G, E)$ .

Update cost constraint  $\Delta$ .

**Output:** The set of cache nodes  $M$ .

**Begin**

$M = \emptyset$ ;

**while**  $(\mu'(M) < \Delta)$

Let  $\mathcal{A}$  be the set of nodes each of which (denoted as  $A$ )  
has the maximum  $\beta(A, M)/\mu'(A, M)$  among its  
non-cache neighbors.

$M = M \cup \mathcal{A}$ ;

**end while;**

**RETURN**  $M$ ;

**END.**

*Estimation of  $\mu'(A, M)$ .* Let  $A$  be a non-cache node, and  $T_A^S$  be the shortest path tree from the server to the set of communication neighbors of  $A$ . Let  $C \in M$  be the cache node in  $T_A^S$  that is closest to  $A$ , and let  $d$  be the distance from  $A$  to  $C$ . In the Distributed Greedy-Steiner Algorithm, we estimate the incremental Steiner update cost  $\mu'(A, M)$  to be  $d \times u$ , where  $u$  is the update frequency of the server. The value  $d$  can be computed in a distributed manner at the start of each round as follows. As mentioned before, the server initiates a new round by broadcasting a packet containing the remaining update cost to the entire network. If we append to this packet all the cache nodes encountered on the way, then each node should get the set of cache nodes on the shortest path from the server to itself. Now, to compute  $d$ , each node only needs to exchange the information with all its immediate neighbors.

*Estimation of  $\beta(A, M)$ .* A non-cache node  $A$  considers only its “local” traffic to estimate  $\beta(A, M)$ , the benefit with respect to an already selected set of cache nodes  $M$ . The local traffic of  $A$  is defined as the data access requests that use  $A$  as an intermediate/origin node. Thus, the local traffic of a node includes its own data requests. We estimate the benefit of caching the data item at  $A$  as  $\beta(A, M) = d \times t$ , where  $t$  is the frequency of the local traffic observed at  $A$  and  $d$  is the distance to the nearest cache from  $A$ . The local traffic  $t$  can be computed if we let the normal network traffic (using only the already selected caches in previous rounds) run for some time between successive rounds. The data access requests of a node  $A$  during normal network traffic between rounds can be directed to the nearest cache in the tree  $T_A^S$  as defined in the previous paragraph.

*Dynamic topologies.* The sensor network topology may be very dynamic due to node/link failures, mobility of sensor nodes, new sensor nodes entering the network, etc. The Distributed Greedy-Steiner Algorithm can be adapted to handle node failures if the active cache nodes periodically send a probe to the server node, and the server initiates a new round if the current update cost is sufficiently less than the update cost constraint. If the server node is static, then mobility of cache nodes can be handled in a similar way. However, in this case, the server node may need to discard cache nodes that have moved too far away. The situation is more challenging if the server node itself is mobile. In the most general scenario of mobile server and client nodes, the server node may need to gather latest location of active cache nodes by periodically flooding the network (in absence of a proactive routing scheme that adapts to mobility

of nodes). New nodes entering the network automatically become part of the network and play a useful role in later rounds of the algorithm.

## 5. Performance evaluation

We empirically evaluate the relative performance of the cache placement algorithms for randomly generated sensor networks of various densities. As the focus of our work is to optimize access cost, this metric is evaluated for a wide range of parameters: (i) network-related—such as the number of nodes and network density, (ii) application-related—such as the number of clients accessing each data item.

We study various caching schemes on a randomly generated sensor network of 2000 to 5000 nodes in a square region of  $30 \times 30$ . The distances are in terms of arbitrary units. We assume all the nodes have the same transmission radius ( $T_r$ ), and all edges in the network graph have unit weight. We have varied the number of clients over a wide range. For clarity, we first present the data for the case where number of clients is 50% of the number of nodes, and then present a specific case with varying numbers of clients. All the data presented here are representative of a very large number of experiments we have run. Each point in a plot represents an average of five runs, in each of which the server is randomly chosen. The access costs are plotted against number of nodes and transmission radius and several caching schemes are evaluated:

- *No Caching*—serves as a baseline case.
- *Greedy Algorithm*—greedy algorithm using the multiple-unicast update cost model (Algorithm 1).
- *Centralized Greedy-Steiner Algorithm*—greedy algorithm using the Steiner tree-based update cost model (Algorithm 2).
- *Distributed Greedy-Steiner Algorithm*—distributed implementation of the Greedy-Steiner Algorithm (Algorithm 3).
- *DP on Shortest Path Tree of Clients*—Dynamic Programming algorithm (Section 3.1) on the tree formed by the shortest paths between the clients and the server.
- *DP on Steiner Tree of Clients*—Dynamic Programming algorithm (Section 3.1) on the 2-approximate Steiner tree over the clients and the server.

*Varying network size for multiple update constraints.* We first compare the performance of the six algorithms under different update cost constraints with varying number of nodes (see Fig. 2). The transmission radius ( $T_r$ ) is fixed at 2. Instead of using absolute cost values to describe the update cost constraint, we represent it in terms of a fraction of the cost of the near-optimal (2-approximate [4]) Steiner tree over all clients and the server node. Clearly, this cost represents a measure of the *maximum* possible update cost. The update cost constraint is set to 25% and 75% of the

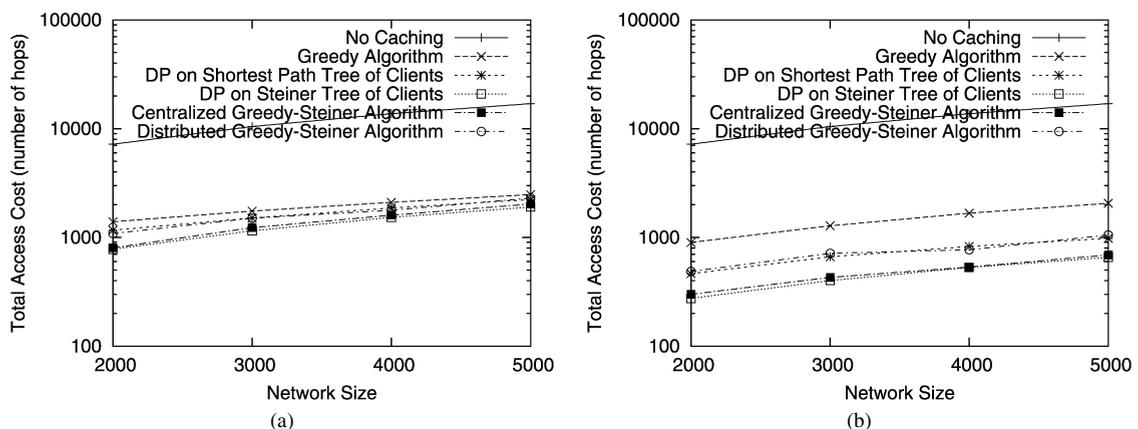


Fig. 2. Access cost with varying number of nodes in the network for different update cost constraints. Transmission radius ( $T_r$ ) = 2. Number of clients = 50% of the number of nodes, and hence increases with the network size. (a) Update cost = 25% of the near-optimal Steiner tree cost. (b) Update cost = 75% of the near-optimal Steiner tree cost.

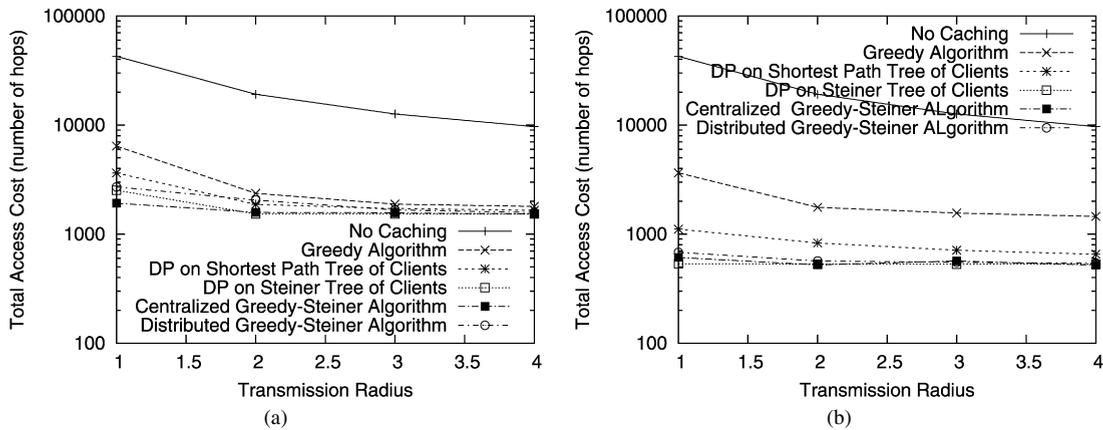


Fig. 3. Access cost with varying transmission radius ( $T_r$ ) for different update cost constraints. Number of nodes = 4000, and number of clients = 2000 (50% of number of nodes). (a) Update cost = 25% of the near-optimal Steiner tree cost. (b) Update cost = 75% of the near-optimal Steiner tree cost.

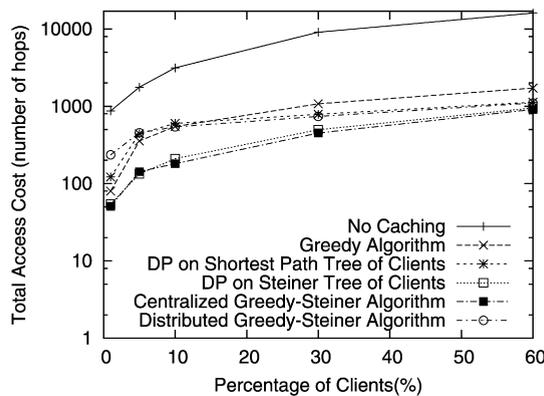


Fig. 4. Effect of the number of clients on the access cost.  $T_r = 2$ . Update cost = 50% of the minimum Steiner tree cost. Number of nodes = 3000.

cost of the near-optimal Steiner tree. Fig. 2 shows that the proposed algorithms perform significantly better (up to an order of magnitude) than the no-caching case (the vertical axis uses a logarithmic scale). Fig. 2(a) shows that when the update cost constraint is small, all our proposed algorithms perform very similarly, especially for large network size. However, a closer look shows that Greedy Algorithm using the multiple-unicast update cost model performs the worst among our five designed algorithms. The performance differences can be seen more clearly in Fig. 2(b), where the update cost constraint is larger. In particular, the best performing algorithms are the Steiner tree based centralized algorithms viz. DP on a Steiner tree of clients and the Centralized Greedy-Steiner Algorithm. Finally, we observe that the Distributed Greedy-Steiner Algorithm performs quite closely to its centralized version.

*Varying transmission radius.* Fig. 3 shows the effect of the transmission radius ( $T_r$ ) on access cost. A network of 4000 nodes is chosen for these experiments. The transmission radius  $T_r$  is varied from 1 to 4. This range is sufficient for evaluation.  $T_r$  smaller than 1 disconnects the network with high probability. A convergence of behavior of our caching algorithms is seen near  $T_r = 4$ , as the network is already dense enough. So,  $T_r$  is not increased any further. The total access cost of all the algorithms decreases with the increase in  $T_r$ , since clients come closer to the server in terms of number of hops as the network density increases. However, when the update cost is large (75% of the near-optimal Steiner tree) as shown in Fig. 3(b), the performance of the two Steiner-tree based centralized algorithms is almost the same for all values of  $T_r$ . Moreover, we again observe that the Distributed Greedy-Steiner Algorithm performs very close to its centralized version.

*Summary.* The general trend in these two sets of plots (Figs. 2 and 3) is similar. Aside from the fact that our algorithms offer much smaller total access cost than the no-caching case, the plots show that (i) the two Steiner tree-based algorithms (DP on the Steiner Tree of Clients and the Centralized Greedy-Steiner Algorithm) perform equally

well and the best among all algorithms except on very sparse graphs; (ii) the Greedy-Steiner Algorithm provides the best overall behavior; (iii) the Distributed Greedy-Steiner Algorithm performs very closely to its centralized version. Fig. 4 shows the total access cost as a function of number of clients for a network with 3000 nodes. The general behavior is no different from before.

## 6. Conclusions

We have developed a suite of data caching techniques to support effective data dissemination in sensor networks. In particular, we have considered an update cost constraint and developed efficient algorithms to determine optimal or near-optimal cache placements to minimize overall access cost. Minimization of access cost leads to communication cost savings and hence, energy efficiency. The choice of update constraint also indirectly contributes to resource efficiency. Two models have been considered—one for a tree topology, where an optimal algorithm based on dynamic programming has been developed, and the other for the general graph topology, which presents an NP-hard problem where a polynomial-time approximation algorithm has been developed. We also designed efficient distributed implementations of our centralized algorithms, and empirically showed that they perform well for random sensor networks.

Cache placement of multiple data items at different servers can be solved as independent single data item cache placement problems, since the update cost constraint at different servers would presumably be independent. The cache placement problem of multiple data items at a single server is challenging, but we can use a heuristic of allocating update costs for each item in proportion to the sum of access frequencies. Each of the scenarios assumes no memory constraints at network nodes. Since, sensor nodes are characterized by limited memory capacity and limited battery energy, we are currently addressing the more general cache placement problem in sensor networks under memory and update constraints for multiple data items.

## References

- [1] B. Badrinath, M. Srivastava, K. Mills, J. Scholtz, K. Sollins, (Eds.), Special Issue on Smart Spaces and Environments, IEEE Personal Communications, 2000.
- [2] P. Bahl, V.N. Padmanabhan, Radar: An in-building RF-based user-location and tracking system, in: Proceedings of the IEEE INFOCOM, 2000.
- [3] G. Barish, K. Obraczka, World wide web caching: Trends and technologies, in: IEEE Communications Magazine, Internet Technology Series, May 2000, pp. 178–184.
- [4] P. Berman, V. Ramaiyer, Improved approximation algorithms for the steiner tree problem, Journal of Algorithms 17 (1994) 381–408.
- [5] T. Berners-Lee, H.F. Nielsen, Propagation, caching and replication on the web, <http://www.w3.org/Propagation>.
- [6] S. Bhattacharya, H. Kim, S. Prabh, T. Abdelzaher, Energy-conserving data placement and asynchronous multicast in wireless sensor networks, in: Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys), 2003.
- [7] N. Bulusu, J. Heidemann, D. Estrin, GPS-less low cost outdoor localization for very small devices, IEEE Personal Communications Magazine 7 (5) (2000).
- [8] M. Charikar, S. Guha, Improved combinatorial algorithms for the facility location and  $k$ -median problems, in: International Conference on Foundations of Computer Science (FOCS), 1999.
- [9] M. Chu, H. Haussecker, F. Zhao, Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks, IEEE Journal of High Performance Computing Applications (2002).
- [10] F.A. Chudak, D. Shmoys, Improved approximation algorithms for a capacitated facility location problem, in: Lecture Notes in Computer Science, vol. 1610, Springer, 1999, pp. 99–131.
- [11] E. Cohen, S. Shenkar, Replication strategies in unstructured peer-to-peer networks, in: Proceedings of the ACM SIGCOMM, 2002.
- [12] D. Estrin, R. Govindan, J. Heidemann (Eds.), Special Issue on Embedding the Internet, Communications of the ACM, vol. 43, 2000.
- [13] E.N. Gilbert, H.O. Pollak, Steiner minimal trees, SIAM Journal of Applied Mathematics (1968).
- [14] H. Gupta, Selection and maintenance of views in a data warehouse, PhD thesis, Computer Science Department, Stanford University, 1999.
- [15] T. Hara, Effective replica allocation in ad hoc networks for improving data accessibility, in: Proceedings of the IEEE INFOCOM, 2001.
- [16] B. Hofmann-Wellenhof, H. Lichtenegger, J. Collins, Global Positioning System: Theory and Practice, Springer-Verlag Telos, 1997.
- [17] C. Intanagonwiwat, R. Govindan, D. Estrin, Directed diffusion: a scalable and robust communication paradigm for sensor networks, in: Proceedings of the International Conference on Mobile Computing and Networking (MobiCom), 2000.
- [18] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, Directed diffusion for wireless sensor networks, IEEE Transactions on Networkings (TON) 11 (1) (2003) 2–16.
- [19] K. Jain, V.V. Vazirani, Approximation algorithms for metric facility location and  $k$ -median problems using the primal-dual schema and Lagrangian relaxation, Journal of ACM 48 (2) (2001) 274–296.
- [20] K. Kalpakis, K. Dasgupta, O. Wolfson, Steiner-optimal data replication in tree networks with storage costs, in: International Database Engineering and Applications Symposium (IDEAS), 2001.

- [21] B. Karp, H.T. Kung, GPSR: Greedy perimeter stateless routing for wireless networks, in: Proceedings of the International Conference on Mobile Computing and Networking (MobiCom), 2000.
- [22] P. Krishnan, D. Raz, Y. Shavitt, The cache location problem, *IEEE Transactions on Networkings (TON)* 8 (2000) 568–582.
- [23] B. Li, M.J. Golin, G.F. Italiano, X. Deng, On the optimal placement of web proxies in the internet, in: Proceedings of the IEEE INFOCOM, 1999.
- [24] J.-H. Lin, J. Vitter, Approximation algorithms for geometric median problems, *Information Processing Letters* 44 (5) (1992).
- [25] P. Nuggehalli, V. Srinivasan, C. Chiasserini, Energy-efficient caching strategies in ad hoc wireless networks, in: Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc), 2003.
- [26] C.E. Perkins, P. Bhagwat, Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers, in: Proceedings of the ACM SIGCOMM, 1994.
- [27] S. Prabh, T. Abdelzaher, Energy-conserving data cache placement in sensor networks, *ACM Transactions on Sensor Networks (TOSN)* 1 (2) (2005) 178–203.
- [28] L. Qiu, V.N. Padmanabhan, G.M. Voelker, On the placement of web server replicas, in: Proceedings of the IEEE INFOCOM, 2001.
- [29] B. Sheng, Q. Li, W. Mao, Data storage placement in sensor networks, in: Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc), 2006.
- [30] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, D. Estrin, Data-centric storage in sensornets, in: Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HOTNETS), 2002.
- [31] R. Wattenhofer, L. Li, P. Bahl, Y.-M. Wang, Distributed topology control for wireless multihop ad-hoc networks, in: Proceedings of the IEEE INFOCOM, 2001.
- [32] J. Xu, B. Li, D.L. Lee, Placement problems for transparent data replication proxy services, *IEEE Journal on Selected Areas in Communications* 20 (7) (2002) 1383–1397.
- [33] L. Yin, G. Cao, Supporting cooperative caching in ad hoc networks, in: Proceedings of the IEEE INFOCOM, 2004.