Many-to-Many Aggregation for Sensor Networks*

Adam Silberstein Jun Yang
Department of Computer Science, Duke University, Durham, NC 27708, USA
{adam,junyang}@cs.duke.edu

Abstract

Wireless sensor networks have enormous potential to aid data collection in a number of areas, such as environmental and wildlife research. In this paper, we address the challenges of supporting many-to-many aggregation in a sensor network. An application of many-to-many aggregation is in-network control of sensors. For expensive sensing tasks such as sap flux measurements and camera repositioning, we use low-cost information obtained at multiple other nodes in the network to control such tasks, e.g., decreasing sampling rates when readings are predictable or unimportant, while increasing sampling rates when there are interesting activities. In general, there is a many-to-many relationship between sources (nodes providing control inputs) and destinations (nodes requiring control outputs). We present a method for implementing many-to-many aggregation in a sensor network that minimizes the communication cost by optimally balancing a combination of multicast and in-network aggregation. Our optimization technique is efficient in finding the initial solution and handling dynamic updates.

1 Introduction

Wireless sensor networks are playing an increasingly important role in monitoring applications. A network consists of battery-powered nodes, equipped with sensors for taking readings (e.g. temperature, light, or even pictures), and a radio for communicating over short distances with neighboring nodes. Nodes are deployed in an area of interest, such as a forest, and instructed to take readings. They forward data through the network, relaying through other nodes, to a *base station* for collection. Because of limited battery, energy efficiency is of primary concern in wireless sensor networks.

In this paper we investigate the problem of supporting *many-to-many aggregation* in an energy-efficient manner within a sensor network. In many-to-many aggregation, there are multiple *destination* nodes; to each destination, we need compute and deliver an aggregate over the readings at some *source* nodes. The relationship between sources and destinations is many-to-many, i.e., one source may be needed in computing aggregates for multiple destinations, and one destination's aggregate is computed over multiple sources.

Motivation Many-to-many aggregation is a general abstraction for implementing *in-network control* of sensors, where we control the activities of one node (e.g., sensor sampling rate) using information collected at other nodes. The control signal for destination node k can be generated by an aggregation function $f_k(v_{k,1}, v_{k,2}, \ldots, v_{k,n_k})$, evaluated over the readings at the source nodes of k, where $v_{k,j}$ denotes the j-th source reading.

In-network control is beginning to play an increasingly important role as the applications of sensor networks continue to expand. Many types of advanced sensors consume a great deal of energy for sensing, even more than for communication. Being able to intelligently control sampling rates of such sensors is key to energy conservation. One such example is a type of sap flux sensor used by our ecologist collaborators to study forest growth. This sensor utilizes two prongs inserted into a tree. One prong heats the tree's sap at the insertion point. At the other prong, the sensor measures the time until it detects higher temperature sap, giving a measurement of sap flow speed. Since this sensor needs to generate heat, it consumes a great deal more energy than a passive one that simply measures the environment. Because sap flux measurements are so expensive, it is not prudent to operate these sensors at high sampling rates unless conditions suggest there may be significant changes in the readings. For example, sap flow is negligible at night, increases at the beginning of the day, and decreases at the end of the day. Furthermore, lack of moisture also prevents sap flow from rising. These factors, which direct when high-frequency sampling will be beneficial, can be measured at low cost with light and soil moisture sensors, and used to control the sap flux sensors.

As another example, consider a sensor network deployed in a wildlife habitat with the goal of taking pictures of animals living there. The network contains sparsely placed camera sensors that are too expensive to use at high frequency. Instead, the network also utilizes motion and vibration sensors. They are sampled frequently and used to control the orientations and sampling rates of the cameras. As the cameras can shoot from a distance, the motion and vibration readings may be located many hops away, so the control may require a fair amount of communication. The communication cost of computing and transmitting control signals is the price we pay for optimizing high-cost sensing tasks. Therefore, minimizing this communi-

*This work is supported by the NSF CAREER and DDDAS programs (under awards IIS-0238386 and CNS-0540347), and by an IBM Faculty Award.

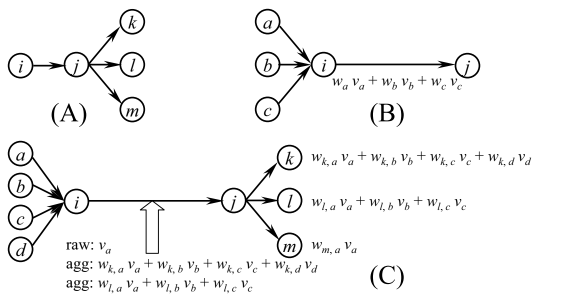
i
j
k
l
m
(A)
a
b
c
i
j
w_a v_a + w_b v_b + w_c v_c
(B)
a
b
c
d
i
j
k
l
m
w_{k,a} v_a + w_{k,b} v_b + w_{k,c} v_c + w_{k,d} v_d
w_{l,a} v_a + w_{l,b} v_b + w_{l,c} v_c
w_{m,a} v_a
raw: v_a
agg: w_{k,a} v_a + w_{k,b} v_b + w_{k,c} v_c + w_{k,d} v_d
agg: w_{l,a} v_a + w_{l,b} v_b + w_{l,c} v_c
(C)

Figure 1. (A) Multicast. (B) In-network aggregation. (C) A more complex example.

cation cost is critical to the effectiveness of this approach.

One possible approach is *out-of-network control* of sensors: All sources to send data to the base station, where all control signals are computed and sent to destinations. In contrast, we argue for in-network control without the base station's participation, which is an instance of the many-to-many aggregation problem. This approach has several advantages. First, we avoid round trips to the base station, for which the number of hops (and therefore the communication cost) increases with network size. Second, we avoid creating bottlenecks at nodes near the base station, which would otherwise be overburdened with message traffic and deplete their energy earlier than other nodes. Finally, even for applications that need to collect all readings at the base station anyway, real-time in-network control is beneficial since it allows *batching*. With batching, data is grouped together for transmission, which lowers message overhead and potentially increases compression ratio. Batching is not an option for real-time out-for-network control.

Challenge in Many-to-Many Aggregation We identify two opportunities in optimizing many-to-many aggregation. First, because each source reading may be needed at multiple destinations, it is natural to use multicast, a well-studied technique in networking. The simplest way to exploit multicast is to build a multicast tree rooted at each source spanning all destinations to which the source contributes. For example, in Figure 1(A), source i's reading is needed as input for controlling destinations k, l, and m. Instead of sending v_i from i in three separate messages, which would result in three copies to j, i only need send one copy of v_i to j, which then forwards v_i to k, l, m following the multicast protocol.

Another optimization is to push aggregate computation into the network, a technique commonly used in sensor networks pioneered by [11]. Consider the example in Figure 1(B). Suppose the aggregation function for destination j is $w_a v_a + w_b v_b + w_c v_c$, the weighted sum of three source readings. As values v_a, v_b, and v_c converge at the intermediate node i on their way to j, i can evaluate the function and simply pass on a single result instead of three individual values. Message routes from sources to the same destination essentially form an aggregation tree, and we can store additional state (e.g., weights in the weighted sum) at the intermediate nodes to enable in-network aggregation.

While both multicast and in-network aggregation have been studied separately before, the interaction of the two techniques involves interesting trade-offs, which we investigate in depth in this paper. In general, destinations have different control functions; for example, weights in weighted sums may vary depending on distances between sources and destinations. Hence, once a source value has been aggregated with others for a destination, the result becomes specific to that destination, and cannot benefit other destinations. In contrast, a *raw* source value, i.e., one that has not been aggregated, is useful to any of its destinations. Intuitively, in the initial hops from a source, we prefer to leave the value raw, since nodes near the source likely still have many destinations as their descendants in the multicast tree, which can benefit from shared transmission of this raw value. On the other hand, closer to the leaves of the multicast tree, we prefer to aggregate, because these nodes have fewer descendants, but likely have more values converging from other multicast routes headed to the same destination. These two cases are exemplified in Figures 1(A) and (B). In (A), it is better for source i to leave v_i raw and transmit a single message to node j, instead of creating three separate partial aggregate values intended for k, l, and m. In (B), it is better for node i to aggregate v_a, v_b, and v_c because leaving them raw would not benefit any node other than the destination j.

The preceding intuition and examples are helpful, but also simple. Consider now the example in Figure 1(C). Nodes a through d are sources, while k through m are destinations with aggregation functions to their right. The arrows depict the routes used to deliver data. The plan illustrated has edge $i \rightarrow j$ transmit one raw value and two aggregated values, for a total message size of three. This choice involves carefully balancing multicast and aggregation—decisions of whether to multicast or aggregate are different for different values, and depend on the actual demand for the values downstream. The problem of finding a network-wide optimal plan becomes more difficult as the number of edges, and the traffic at each, increase. We must also consider consistency of the plan across edges. A plan that aggregates a value at an upstream edge and then requires it to be raw at a downstream edge is infeasible. Once a value is aggregated, we cannot in general recover the raw value. Therefore, it would seem that we cannot solve the problem by looking at each edge in isolation. A surprising result of this paper, however, is that we can indeed do so efficiently without sacrificing consistency or optimality (under reasonable assumption on the multicast routes). This result is not only interesting theoretically, but also has important practical implications: It enables an efficient divide-and-conquer approach to optimization, and makes it easier for a plan to adapt to dynamic changes.

Contributions In this paper we address the problem of efficiently implementing aggregation functions with multiple sources and destinations, which requires combining and balancing the techniques of multicast and in-network aggregation. Given the multicast trees (under minor restrictions), we prove the problem can be solved independently for each edge while still guaranteeing global optimality and consistency of the plan. We show the single-edge problem reduces to an instance of weighted bipartite vertex cover, which can be solved efficiently in polynomial time. We discuss how to implement the plan inside a sensor network with low overhead, demonstrating that the amount of in-network state required by our approach is no more than a constant factor of the basic multicast approach. We also briefly describe techniques for handling dynamic route adjustments and incorporating temporal suppression for continuous computation of aggregation functions. Finally, we demonstrate the effectiveness of our optimal solution to the multiple-aggregation problem through experiments.

2 Optimizing Many-to-Many Aggregation

2.1 Preliminaries

The sensor network consists of a set of fixed-location nodes. At each time step, each node takes a low-cost numerical reading, such as temperature or light. For simplicity of presentation, we assume each node produces exactly one reading, and we use v_i to denote the value of the reading produced by node i. It is straightforward to generalize our results to the case where a node is responsible for any number of readings.

Problem Definition Our task is to evaluate, at each time step, a set of *aggregation functions* within the network, and make each result available at a *destination* node (for controlling its behavior). Again, for simplicity, we assume each node can be the destination of at most one aggregation function, though this assumption is simple to lift. The aggregation function destined for node d, denoted f_d, is defined over the readings from the set of *source* nodes of d. Let S denote the set of all sources and D denote the set of all destinations in the entire network. Let $\sim$ denote the many-to-many producer-consumer relationship between sources and destinations: $s \sim d$ means $s \in S$ is a source of $d \in D$.

In this paper, we consider aggregation functions that generalize *algebraic aggregates* [7, 11]. More precisely, for destination d, suppose its sources are $s_1, \ldots, s_n$. We require there exist an *evaluator function* e_d, a *merging function* m_d, and a set of *pre-aggregation functions* w_{d,s_i} (one for each source s_i), such that $f_d(v_{s_1}, \ldots, v_{s_n}) = e_d(m_d(\{w_{d,s_1}(v_{s_1}), \ldots, w_{d,s_n}(v_{s_n})\}))$. Each pre-aggregation function transforms the corresponding source reading into, in general, a constant-size *partial aggregate record*. The merging function combines partial aggregates records together, and satisfies the property that $m_d(R_1 \cup R_2) = m_d(\{m_d(R_1), m_d(R_2)\})$ for any sets R_1 and R_2 of partial aggregate records. Finally, the evaluator function takes a partial aggregate record and computes the actual result of the aggregation function. Our aggregation functions generalize algebraic aggregates by allowing each input to be transformed differently. We have found this feature necessary in our ecological monitoring application, as it allows us to express weighted versions of aggregate functions. As an example, consider a weighted-average function $f(v_1, \ldots v_n) = \frac{1}{n}\sum_{i=1}^{n} \alpha_i v_i$. The pre-aggregation function for each input is $w_i(x) = \langle \alpha_i x, 1\rangle$. The merging function is $m(\{\langle x, a\rangle, \langle y, b\rangle\}) = \langle x + y, a + b\rangle$. Finally, the evaluator function is $e(\langle x, a\rangle) = x/a$.

In general, aggregation functions running in a network can be quite different from each other. One may be a weighted sum, while another may be a weighted standard deviation; they may have overlapping but different sets of sources, and even for common sources, they may have different weights. We assume once a pre-aggregation function $w_{d,s}$ is applied to raw value v_s, the result becomes specific to destination d, and cannot be used for computing aggregation functions for other destinations.

Routing Ideally, we would like to jointly optimize many-to-many aggregation *and* the choice of routes. The resulting optimization problem, however, would likely become intractable. Therefore, in this paper, we restrict ourselves to the problem of optimizing many-to-many aggregation given a set of multicast trees. Each multicast tree is rooted at a source and spans all destinations of this source; the edges are always directed from a node to its children. We impose the following restrictions on these multicast trees. (1) *Minimality*: Removing any edge from a multicast tree rooted at source s will cause the tree to no longer span all destinations of s. (2) *Sharing*: If node i can reach node j via directed paths in two multicast trees, then the two paths are identical. Intuitively, the second restriction encourages path sharing across multicast trees because it creates more opportunities for aggregation: When two raw values head to a common destination in their respective multicast trees, as soon as their paths converge, they can be aggregated and the partial aggregate record can travel on a unique path to the destination.

For simplicity of presentation, we assume stable multicast routes in the remainder of this section. In Section 3, we discuss how to lift this assumption.

2.2 Single-Edge Optimization

We begin by tackling the problem of finding the optimal plan for many-to-many aggregation at each directed edge, independent of solutions at other edges. Note it is conceivable that two independently obtained single-edge solutions may turn out to be inconsistent with each other. Specifically, if an upstream edge decides to aggregate a raw value (with others), then it would be infeasible for a downstream edge to still transmit this raw value. In Section 2.3 we show how to combine single-edge solutions into a consistent global plan without sacrificing optimality.

For each directed edge $e : i \to j$, we must determine how to transmit as little data as possible through e to be able to compute the aggregation functions downstream. If e is on the multicast path from source node s to destination d, we say that $s \sim_e d$. Let $S_e = \{s \in S \mid \exists d \in D : s \sim_e d\}$ and $D_e = \{d \in D \mid \exists s \in S : s \sim_e d\}$; that is, S_e and D_e are the sets of sources and destinations (respectively) connected through e. Obviously, we can just focus on how to deliver the information from S_e to D_e; there is no need to consider source-destination connections that do not go through e. As a concrete example, consider edge $i \to j$ in Figure 1(C). In Figure 2(A) we show $S_{i\to j}$, $D_{i\to j}$, and the producer-consumer relationship $\sim_{i\to j}$ among them (as a binary matrix).

As a side note, it is possible for the reverse edge $j \to i$ to appear in some other multicast trees, and we would optimize that edge independently from $i \to j$. Also, a node can be in both S_e and D_e; our algorithm handles this case perfectly well.

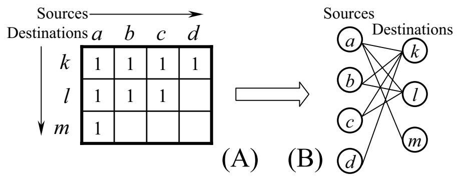

Figure 2. Single-edge optimization. (A) Inputs. (B) Reduction to vertex cover.

Our options for what to transmit on edge e are: (1) *Transmit-raw*: for a source $s \in S_e$, transmit its reading raw. (2) *Transmit-aggregate*: for a destination $d \in D_e$, aggregate readings from all of d's sources in S_e and transmit the partial aggregate record. Either option results in a value being transmitted, and we seek to minimize the total amount of transmission required to evaluate all aggregation functions. Note it is not helpful to have the additional option of aggregating some, but not all, of d's sources available at e. The reason is that the resulting partial aggregate record is specific to d and cannot benefit other destinations, so we may as well replace it with one that aggregates all available sources of d; doing so would not increase the total amount of transmission, and in fact might decrease it as we no longer need to transmit to d raw values already aggregated.

Weighted Bipartite Vertex Cover This problem elegantly reduces to an instance of weighted bipartite vertex cover. The reduction, illustrated in Figure 2(B), works as follows. Source nodes S_e form one set of bipartite graph vertices U while destination nodes D_e form the other set of bipartite graph vertices V. (If a node is in both S_e and D_e, there will be two bipartite graph vertices, one in U and one in V, corresponding to the roles of source and destination, respectively.) If $s \sim_e d$, an undirected edge connects their corresponding vertices in the bipartite graph.

A vertex cover of a bipartite graph (U, V, E) is a subset of vertices $C \subseteq U \cup V$ such that for each bipartite graph edge $(u, v) \in E$, at least one of u and v belongs to C. Each vertex cover C corresponds to a possible solution to our single-edge optimization problem: Choosing a vertex $u \in U$ in C corresponds to transmitting the raw value for source u, while choosing a vertex $v \in V$ corresponds to aggregating all source values for destination v and transmitting the partial aggregate record. The fact that C is a vertex cover guarantees the sufficiency of the information transmitted along the edge: If a destination d is chosen in C, the partial aggregate record we transmit already includes contributions from all upstream sources of d, so we need not transmit anything else for d along this edge. If, on the other hand, d is not chosen in C, then C must have chosen every source s of d (otherwise some bipartite graph edge $s \sim_e d$ would not be covered); in that case, all upstream source values of d are transmitted raw, still allowing the aggregation function to be evaluated at d.

We further assign weights to bipartite graph vertices according to the cost of the corresponding transmission options: A vertex in U is weighted by the size of the corresponding raw value; a vertex in V is weighted by the size of the corresponding partial aggregate record. For example, for weighted sum, source and destination weights would be equal (because both raw values and partial aggregate records are floating-point numbers), but for weighted average, destinations would weigh more (because each partial aggregate record includes an additional integer count). To minimize the total amount of transmission, we look for a vertex cover with minimum total weight.

The minimum weighted vertex cover problem for bipartite graphs can be solved efficiently in polynomial time [1] using standard network flow techniques. For completeness, we present a simple mixed-integer programming formulation below. In the following, constant c_v denotes the weight of vertex v, and boolean variable x_v indicates whether vertex v is included in the cover (1 if included; 0 otherwise).

$$\text{Minimize: } \sum_{v \in U \cup V} c_v x_v \text{ subject to:}$$
$$\forall (u, v) \in E : x_u + x_v \geq 1.$$

As an example, a solution to the single-edge optimization problem in Figure 2 for edge $i \to j$ in Figure 1(C) includes source a and destinations k and l, which corresponds to the plan shown in Figure 1(C).

2.3 Global Plan

The single-edge optimization problem in the preceding section has been solved without regard to decisions made at other edges. As mentioned several times before, independently obtained single-edge solutions may not work consistently together. Specifically, aggregating a value at an upstream edge makes it impossible to discern the raw value at a downstream edge. Therefore, downstream edges can only consider solutions that also aggregate the value. Interestingly, we show that independently obtained single-edge solutions are indeed consistent with each other. In other words, selecting the optimal solution for any particular edge does not eliminate the optimal solution at any other edge as a choice. Therefore, we can independently optimize for each edge, and assemble the single-edge solutions together into a consistent and optimal global plan.

The only additional requirement we impose is that every single-edge optimization problem has a unique solution (i.e., there is only one vertex cover with the minimum weight). We can easily satisfy this requirement by adding minuscule weights to each vertex to consistently create tiebreakers among choices. The weight we assign to each source and each destination is consistent for all instances of weighted bipartite vertex cover problems across all edges. We are now ready to state the main result of this section.

Theorem 1. *Optimal solutions to the individual weighted bipartite vertex cover problems at each edge in the multicast trees form, in combination, a consistent global plan for the entire network.*

The proof is far from trivial, and we present it in Appendix A. It is difficult to overstate the importance of this result to our approach. First, Theorem 1 allows us to simplify optimization greatly. Instead of solving a complex problem over the entire network, we are able to divide the problem up into tractable pieces and solve them independently. Potentially, this optimization can be carried out by the individual nodes themselves inside the network. Second, a direct corollary of Theorem 1 is that small changes to the input have fairly localized impact on the global plan. For example, adding one new source to an aggregation function would only change single-edge solutions at nodes lying on the multicast tree path from the source to the destination; for other nodes, inputs to their single-edge optimization problems remain unchanged, so their solutions remain unchanged and still part of the optimal global plan according to Theorem 1. As we will see in Section 3, this property makes our approach easily adaptable to dynamic changes.

3 Implementing the Plan in Network

3.1 Ordering and Merging Messages

We call each raw value or partial aggregate record transmitted along an edge a *message unit*. For example, in Figure 1(C), edge $i \to j$ transmits three message units. There are timing dependencies among message units. We say that message unit u' *waits for* message unit u if u is immediately upstream of u' (i.e., the node sending u' is the recipient of u), and contains data required in computing or sending u'. For example, the third message unit $w_{l,a}v_a + w_{l,b}v_b + w_{l,c}v_c$ at edge $i \to j$ waits for message units v_a, v_b, and v_c from edges $a \to i$, $b \to i$, and $c \to i$ respectively. The following theorem guarantees that there are no cycles of dependencies.

Theorem 2. *There are no wait-for cycles among the message units in the optimal global many-to-many aggregation plan; that is, there exist no message units $u_1, u_2, \ldots, u_n$ such that u_n waits for u_1, and u_i waits for u_{i+1} for $i = 1, \ldots, n-1$.*

Proof. First, note it is impossible by definition for a raw-value message unit to wait for a partial aggregate record. Therefore, any dependency cycle must consists of (1) only raw-value message units, or (2) only partial aggregate records. In the following, let T_s denote the multicast tree rooted at source s.

In case (1), all message units in the cycle must contain the same raw value, say from source s. By construction of the single-edge optimization problems, a raw value from s can be transmitted only through an edge in T_s. Therefore, a wait-for cycle of raw-value message units would imply a cycle of edges in T_s, which is impossible.

In case (2), all message units in the cycle would be partial aggregate records intended for the same destination, say d. In this case, if message unit u' waits for message unit u, then the set of raw values participating in the partial aggregate record of u' must be a superset of that of u. Hence, a wait-for cycle would imply that all partial aggregate records in the cycle are identical; i.e., they involve the same set of raw values. Let s denote the source of one such raw value. By construction of the single-edge optimization problems, a partial aggregate record destined for d involving a raw value from s must be transmitted along an edge on the path to d in T_s. Therefore, a wait-for cycle of such partial aggregate records would imply a cycle of edges in T_s, which is impossible. $\square$

Theorem 2 immediately leads to a straightforward, though suboptimal, method for scheduling transmissions, which transmits each message unit as an individual message. Each node, upon receiving an incoming message unit, produces and transmits all outgoing message units that are no longer waiting for any additional message units.

We can do substantially better by combining single-unit messages into multiple-unit messages, thereby saving the per-message transmission overhead. Messages to be sent out along the same edge are eligible for merging. Before merging two messages into one, we need to check that doing so does not create wait-for cycles. The new message inherits all wait-for relationship involving the two original messages, and we check to see if the new message is part of any cycle; if not, we proceed with the merge. In general, it may be possible an edge has several messages that cannot be merged. For example, edge e_1's message unit x_1 may be waiting for e_2's message unit y_2, e_2's x_2 may be waiting for e_3's y_3, and e_3's x_3 may in turn be waiting for e_1's y_1; in this case, one e_i must transmit x_i and y_i separately to break the cycle. Such situations seem to be quite rare. For all our experiments in Section 4, we simply use a greedy algorithm that keeps finding two messages to merge as long as they do not create cycles, and this algorithm is able to merge all messages along each edge into one. In other words, for all our experiments, our approach only sends one message per multicast tree edge, regardless of the number of trees sharing this edge. This result is the best one can hope for, as every edge must transmit some information.

A number of additional optimizations are possible, but we do not explore them further in this paper. One optimization is to construct a detailed transmission schedule from the global plan, aimed at avoiding collisions and reducing node listening time. Another optimization is to use broadcast to transmit message units shared by multiple edges.

3.2 Implementing Node Behavior

A message unit can be either a raw value, tagged by the source node identifier, or a partial aggregate record, tagged by the destination node identifier. Based on the single-edge solutions for its incident edges, each node n maintains state for many-to-many aggregation in four tables.

- *Raw table* specifies how to forward raw values. If n needs to transmit the value of source s raw to an outgoing edge e, then this table contains an entry $\langle s, g \rangle$, where g identifies the outgoing message in which to send this raw value.
- *Pre-aggregation table* specifies how to pre-aggregate raw values. If the value of source s reaches n raw but n needs to aggregate it for destination d (including the case when $d = n$), then this table contains an entry $\langle s, d, w_{d,s} \rangle$, where $w_{d,s}$ is the pre-aggregation function (or more precisely, information needed to evaluate this function). For example, if the aggregation function for d is a weighted average of its sources, then we store the weight associated with s.

- *Partial aggregate table* specifies how to merge and/or forward partial aggregate records. If n needs to compute and/or forward a partial aggregate record for destination d (including the case when $d = n$), then this table contains an entry $\langle d, c, m_d, g\rangle$, where c is the total number of partial aggregate records to be combined at n for d, including both those received by n and those generated by n by pre-aggregating raw values; m_d is the merging function (omitted if $c = 1$); g identifies the outgoing message (omitted if $d = n$) in which to send the result partial aggregate record.
- *Outgoing message table* specifies how to combine outgoing message units into messages and where to send them. For each outgoing message, the table contains an entry $\langle g, c, n'\rangle$, where g identifies the message, c is the total number of message units in g, and n' is the recipient of the message.

Finally, each destination d stores the evaluation function e_d for its aggregate function. The contents of above tables are computed out-of-network according to the optimal many-to-many aggregation plan, and disseminated into the network.

Algorithm 1 specifies how to control the runtime execution of the plan at each node using the above information. Each node calls BEGINTIMESTEP at the beginning of a timestep; sources produce raw values, which set the execution in motion. Upon receiving a message, the node breaks it down into individual message units and processes each using PROCESSINCOMINGMESSAGEUNIT. Raw-value units are forwarded according to the raw table, and/or pre-aggregated according to the pre-aggregation table. The resulting partial aggregate records, as well as those received by the node in incoming message units, are processed one by one as they become available, using ADDPARTIALAGGREGATE. ADDPARTIALAGGREGATE merges these partial aggregate records into a single result record; when the number of records merged reaches that specified in the partial aggregate table, the result record can be put in a message unit and sent to its destination, or, if the node is the destination, used to compute the final result of the aggregation function. Finally, to implement message merging, we use ADDOUTGOINGMESSAGEUNIT to add a unit to the temporary buffer of a message; the entire message is transmitted only when its buffer has collected the number of message units as specified in the outgoing message table.

It is worthwhile noting, as shown by the following theorem, the amount of state required inside the network is quite low—in fact on the same order as what is required to implement just the pure multicast approach (which aggregates only at destinations) or the pure in-network aggregation approach (which aggregates at the earliest opportunity), whichever is less.

Algorithm 1: Controlling runtime execution at node n.

```
Variables: A counter $c_d$ and a temporary partial aggr.
           record $r_d$ for each destination $d$ in $n$'s partial
           aggr. table; a counter $c_g$ and a temporary
           buffer $b_g$ for each message $g$ in $n$'s outgoing
           msg. table.
1  BEGINTIMESTEP() begin
2      foreach $d$ in partial aggr. table do
3          $c_d \leftarrow 0$; $r_d \leftarrow \perp$;
4      foreach $g$ in outgoing msg. table do
5          $c_g \leftarrow 0$; clear $b_g$;
6      if $n$ is a source then
7          $u \leftarrow$ new msg. unit with $n$'s value tagged by $n$;
8          PROCESSINCOMINGMESSAGEUNIT($u$);
9  end
10 PROCESSINCOMINGMESSAGEUNIT($u$) begin
11     if $u$ is a raw value $v$ from source $s$ then
12         if $\exists\langle s, g\rangle \in$ raw table then
13             ADDOUTGOINGMESSAGEUNIT($g, u$);
14         foreach $\langle s, d, w_{d,s}\rangle \in$ pre-aggr. table do
15             ADDPARTIALAGGREGATE($d, w_{d,s}(v)$);
16     if $u$ is a partial aggr. record $r$ for destination $d$ then
17         ADDPARTIALAGGREGATE($d, r$);
18 end
19 ADDPARTIALAGGREGATE($d, r$) begin
20     $\langle d, c, m_d, g\rangle \leftarrow$ partial aggr. table entry keyed by $d$;
21     if $r_d = \perp$ then $r_d \leftarrow r$;
22     else $r_d \leftarrow m_d(r_d, r)$;
23     $c_d \leftarrow c_d + 1$;
24     if $c_d = c$ then
25         if $d = n$ then compute final result $e_d(r_d)$;
26         else
27             $u \leftarrow$ new msg. unit with $r_d$ tagged by $d$;
28             ADDOUTGOINGMESSAGEUNIT($g, u$);
29 end
30 ADDOUTGOINGMESSAGEUNIT($g, u$) begin
31     append $u$ to $b_g$; $c_g \leftarrow c_g + 1$;
32     if $c_g = c$ then
33         $\langle g, c, n'\rangle \leftarrow$ outgoing msg. table entry keyed by $g$;
34         SENDMESSAGE($n', b_g$);
35 end
```

Theorem 3. *Let $|T|$ denote the size of a tree T in the number of nodes. Let A_d denote the* aggregation tree *rooted at destination d formed by multicast paths from d's sources to d. Recall that T_s denotes the multicast tree rooted at source s. The total amount of state required by our optimal many-to-many aggregation plan is $O(\min\{\sum_s |T_s|, \sum_d |A_d|\})$, assuming that pre-aggregation, merge, and evaluation functions each take constant space, and that the size of each partial aggregate record is no more than a constant multiple of the raw value size.*

Proof. We account for the amount of state maintained by each type of tables below.

- Pre-aggregation tables: There is exactly one entry for each source/destination pair (s, d) where $s \sim d$. Note that this state is required for any approach, because each $w_{d,s}$ is unique and must be stored somewhere. Let $I(s, d)$ be an indicator function such that $I(s, d) = 1$ if $s \sim d$, or 0 otherwise. We have $\sum_{s,d} I(s, d) = \sum_s(\sum_d I(s, d)) \le \sum_s |T_s|$

and also $\sum_{s,d} I(s,d) = \sum_d (\sum_s I(s,d)) \leq \sum_d |A_d|$. Therefore the overall total number of pre-aggregation table entries $\sum_{s,d} I(s,d) = O(\min\{\sum_s |T_s|, \sum_d |A_d|\})$.

- Raw and partial aggregate tables: Consider the single-edge optimization problem at each edge $e : i \to j$. Recall that S_e and D_e denote sources and destinations connected through e. The pure multicast plan corresponds to transmitting raw for every source in S_e, which requires node i to store $|S_e|$ forwarding entries, for a total of $O(\sum_s |T_s|)$ entries over the entire network. The pure in-network aggregation plan corresponds to transmitting aggregate for every destination in D_e, which requires node i to store $|D_e|$ entries, for a total of $O(\sum_d |A_d|)$ entries over the entire network. Suppose the minimum vertex cover for e includes n_s vertices from S_e and n_d vertices from D_e, for a total weight of no less than $m(n_s + n_d)$, where m is the minimum weight of any single vertex. Because this vertex cover is minimum, its weight should be no more than the total weight of all S_e nodes (which form a vertex cover), which is $O(m|S_e|)$ since the maximum weight of any single vertex is bounded by a constant multiple of the minimum weight m. By a similar argument (that the minimum vertex cover weighs no more than the vertex cover containing all D_e nodes), $m(n_s + n_d) = O(m|D_e|)$. Therefore, we have $n_s + n_d = O(\min\{|S_e|, |D_e|\})$, so the total number of raw and partial aggregate table entries across the entire network is $O(\min\{\sum_s |T_s|, \sum_d |A_d|\})$.
- Outgoing message tables: At each node, the number of entries in this table is bounded by the total number of entries in the raw and partial aggregate tables at this node. As shown above, the total size of raw and partial aggregate tables over all nodes is $O(\min\{\sum_s |T_s|, \sum_d |A_d|\})$; therefore, the total size of outgoing message tables over all nodes has the same bound. $\square$

3.3 Flexibility Trade-Off in Routing using Milestones

So far, our optimization has been based on multicast trees fully specified down to individual hops. While knowing every hop allows maximal control by our many-to-many aggregation algorithm and creates the most optimization opportunities, it constrains the flexibility of the communication layer at runtime. For example, consider a fully specified path $a \to b \to c \to d \to e$. The communication layer must guarantee reliable message delivery on every hop (using acknowledgments and retransmissions, for example). Routing around an intermediate node, say b, is not an option, because b would be waiting indefinitely for the message from a while possibly holding other messages destined for c. If the route between a and c is unstable (e.g., susceptible to transient failures), then requiring it to always pass through b (or any other particular node for that matter) is suboptimal.

Motivated by the need for more routing flexibility, we propose the *milestone* approach to handle routing for many-to-many aggregation. A route from a source to a destination may go through a number of intermediate nodes. We select a subset of these intermediate nodes as *milestones*. Optimization in Section 2 would be done on sources, destinations, milestones, and "virtual" edges among them, instead of physical one-hop edges. Specifying milestones guarantees communication from the source to the destination must go through each milestone. This guarantee provides the basis for our compile-time optimization, which depends on multiple routes converging at a common node at runtime. On the other hand, how the message is actually delivered between milestones is completely up to the communication layer, which is free to choose any route as it sees fit at runtime. We can choose any number of milestones per route, based on the expected route stability. If most parts of a route are very unstable, we should choose few milestones, because it may be much more expensive for the communication layer to route through pre-selected milestones than simply to the destination. On the other hand, if a route is very stable, every intermediate node can be chosen as a milestone, which would provide additional opportunities for aggregation. For example, in the path $a \to b \to c \to d \to e$, if the portion between a and c is unstable but the rest is dependable, we can choose c and d to be milestones.

3.4 Adapting to Dynamic Situations

Although the milestones can prevent transient routing changes from affecting the plan, we still need to deal with other types of changes. Changes to multicast trees and choice of milestones may happen if stability of certain routes have changed significantly. Changes to aggregation functions may happen when old nodes die or new nodes are deployed. We briefly sketch our approach here. The following corollary, which follows directly from Theorem 1, provides the theoretical underpinning of our approach.

Corollary 1. *The globally optimal many-to-many aggregation plan remains unchanged at edges for which inputs to their single-edge optimization problems remain unchanged.*

Based on this corollary, when changes occur, we simply need to identify edges whose workloads have changed, re-optimize the corresponding single-edge optimization problems, and install the new local plans at the incident nodes. In many cases, updates to the global plan will be limited to a small part of the network. For example, when a source is removed from an aggregation function, only the edges along the multicast path from the source to the destination will be affected; nodes not on this path need not be updated. Even under a simple multicast approach with no aggregation, most of the nodes contacted by our approach would have to be contacted to have their routing states adjusted.

Therefore, Theorem 1 is significant not only for reducing the complexity of optimization, but also, and arguably more importantly, for reducing the cost of transmitting updated edge plans into the network. If a small update were to force us to re-optimize and transmit new plans to all edges, the cost would perhaps be prohibitively high. Theorem 1 demonstrates the feasibility of adapting our plan to dynamic situations.

3.5 Continuous Control with Suppression

While all aggregation functions for in-network control can be recomputed periodically at every time step, some types of aggregation functions can be continuously maintained (up to desired precision) using a variant of temporal suppression. With temporal suppression, each source transmits the difference between its latest value and its value at the time of its last transmission, if this difference exceeds a certain threshold. For example, for weighted sum $\alpha v_i + \beta v_j + \gamma v_k + \cdots$, if v_i and v_j have changed by $\hat{v}_i$ and $\hat{v}_j$ respectively, then the weighted sum will change by $\alpha \hat{v}_i + \beta \hat{v}_j$, computed by aggregating the changes themselves. This approach may work for certain aggregation functions better than periodic recomputation.

Temporal suppression can impact the optimality of a plan. The current solution, which we call the "default" solution, optimizes the case where all sources transmit values. Consider the solution for edge $i \rightarrow j$ in Figure 1(C), which transmits one raw value and two partial aggregate records. Suppose in a particular timestep, $\hat{v}_a$ and $\hat{v}_b$ arrive at i, while v_c and v_d have not changed and are suppressed. Using the default solution requires transmitting three message units; the two partial aggregate records contain $w_{k,a}\hat{v}_a + w_{k,b}\hat{v}_b$ and $w_{l,a}\hat{v}_a + w_{l,b}\hat{v}_b$ respectively. The optimal solution, though, is to transmit both $\hat{v}_a$ and $\hat{v}_b$ raw, using only two message units.

It may be possible to solve for the optimal solution at runtime at each node within the network, based on the actual set of unsuppressed source values in every timestep. Doing so incurs a higher computational cost. Also, we would need to store pre-aggregation functions redundantly at multiple nodes (specifically, $w_{d,s}$ must be stored at all milestones on the path from s to d), so that they are available wherever the dynamic solution dictates aggregation. Because of these significant expenses, we offer below a compromise approach that incurs less computation and storage overhead, but no longer guarantees the optimality of the global solution.

Dynamic Override We still install the default plan (optimized for the case where all sources transmit) in the network. The basic idea is to allow a node to override the default plan at runtime to improve efficiency, based on the actual input the node receives each timestep. Instead of aggregating a raw value v for multiple destinations $d_1, d_2, \ldots$, a node n may decide to continue transmitting v raw. This override decision has the consequence that v will be transmitted raw all the way to $d_1, d_2, \ldots$ (using multicast), because only n stores v's pre-aggregation functions for $d_1, d_2, \ldots$; downstream nodes do not. Hence, the resulting plan may be suboptimal because of the potential of missing some opportunities for aggregating v downstream, though such opportunities will be rare if temporal suppression is highly effective.

We now describe how dynamic override works in more detail. There are three types of message units: (1) a raw value (or more precisely, the change in raw value since the last report—we omit this technicality in the following because it does not affect discussion) tagged by its source; (2) a partial aggregate record tagged by its destination; (3) an override raw value tagged by its source. The third type is new and will be used to override the default plan.

In addition to the state required to implement the default plan, we introduce a new *override table* at each node n, which specifies how to forward override message units. Each entry $\langle s, d, w_{d,s} \rangle$ in n's pre-aggregation table corresponds to a decision by the default plan for n to aggregate the raw value from s for d. Since this decision can be dynamically overridden, we store $\langle s, g \rangle$ in n's override table, where g refers to the same message in the partial aggregate table entry for destination d. Furthermore, for each descendant node m of n on the multicast path from s to d, we store $\langle s, g \rangle$ in m's override table, for each message g in m's partial aggregate table entry for d. Clearly, this additional state does not alter the asymptotic space complexity given in Theorem 3.

In each timestep, a node n operates as follows. To keep the discussion simple, let us assume for now that n receives all incoming message units before making any decision on what to transmit. We show how to lift this assumption after describing the basic algorithm below.

1. If the incoming message unit is an override raw value from source s, n simply forwards it to all neighbors leading to destinations requiring this raw value. These recipients are identified by joining the override table entries for s with the outgoing message table.
2. If the incoming message unit is a partial aggregate record destined for d, n aggregates for d as in the default plan. Intuitively, since n must send one partial aggregate record that is specific to d, incorporating more inputs into this record would not incur extra cost. Thus, we include in this aggregation any partial aggregate records received by n destined for d, and any raw values from sources listed with d in n's pre-aggregation table.
3. If the incoming message unit is a non-override raw value from source s, then for each raw table entry at n containing s (if any), n simply forwards the raw value along the corresponding outgoing edge, as in the default plan.
4. Finally, consider incoming message units containing non-override raw values whose sources are found in n's pre-aggregation table. For these message units, the default plan decides to aggregate, but here we have the option to override that decision. We formulate a bipartite vertex cover problem for each outgoing edge e of n. We start from the bipartite graph for the single-edge optimization problem in Section 2.2 (without considering suppression) and construct a reduced graph as follows:
    - We remove source s and its incident edges if n does not receive a non-override raw value from source s.
    - We remove source s and its incident edges if there exists a raw table entry with s that joins with an outgoing message table entry with e (i.e., the default plan already transmits the value raw along e, so there is nothing to override). Recall from case 3 above that we follow the default plan for such sources.
    - We remove destination d and its incident edges if n re-

ceives a partial aggregate record destined for d. Recall from case 2 above that we choose to aggregate for such destinations.

We find the minimum vertex cover of this reduced bipartite graph. If the cover includes a destination d', then n aggregates for d' as in the default plan. If the cover includes a source s', then n sends the raw value from s' as an override raw value along e.

We now discuss how to lift the assumption that each node receives all incoming message units before making transmission decisions. This assumption is unrealistic for two reasons. First, with suppression, we do not know in advance how many sources will change values, so it is difficult for a node to determine when it has received all incoming message units for a timestep. Second, the assumption ignores possibly complex timing dependencies among message units. For example, it is possible that node n is an ancestor of node n' in the multicast tree rooted at s, while n' is an ancestor of n in the multicast tree rooted at s'; therefore, at least one of n and n' must transmit something before receiving all incoming message units. To cope with these issues, we use the following approach. Instead of relying on the count c in the partial aggregate table (or the outgoing message table) to determine when a message unit (or an outgoing message, respectively) is ready, we use a timer for each outgoing message. We set the timer durations based on an execution of the default plan (when all sources transmit): The duration of g's timer is set to the time elapsed between the beginning of the timestep and the transmission of g, multiplied by a slack factor that accounts for any additional time needed for runtime optimization. With suppression and dynamic override, at the beginning of each timestep, node n resets all its timers. When message g's timer expires, n uses the basic algorithm above to process all message units received up to this point (and not yet processed by a timer expiration earlier in the same timestep). Then, n transmits message g (and only g) if it is not empty. By following the timing of message transmissions in the default plan, we expect the node to have received most of the relevant incoming message units when a timer expires, because suppression usually results in fewer of them than the default plan; waiting until timer expiration ensures that the node does not make a decision prematurely. Furthermore, transmitting only one message for each timer expiration prevents generation of an excessive number of messages. Finally, to guard against possibly imperfect timing, after all timers have expired across the network, nodes reset the timers and start a "mop-up" phase to transmit and processing any remaining message units not yet transmitted.

Dynamic override is a heuristic. As discussed earlier, an override decision made at n, although locally optimal, precludes the override raw value to be aggregated downstream. Intuitively, if source values are volatile, fewer sources will be suppressed, and there will be a greater chance for an overridden raw value to meet with another one downstream with the same destination, which would favor aggregation had it been an option. To alleviate this problem, we introduce into the bipartite vertex cover problem an extra weight for each source, which is adjusted according to the degree of volatility in source values. A larger extra weight discourages the minimum vertex cover from including a source (i.e., override). We investigate the effect of this extra weight via experiments in Section 4.

3.6 Handling Failures

The milestone routing approach, discussed earlier in this section, handles transient failures that can be resolved by temporary route changes. Permanent link failures may in general require redesigning the multicast trees and hence re-optimizing the many-to-many aggregation plan; permanent node failures may additionally necessitate changes in aggregation functions themselves. Techniques described earlier for adapting to dynamic situations can be applied here.

The case where a particular node fails and cannot be reached is fundamentally problematic, since its tables stores critical information, namely the pre-aggregation functions. If such a failure is temporary and does not mandate re-planning, we can use redundant state inserted into the network to still execute the plan in the current round. The general idea is for each milestone node to both replicate the tables of its milestone "parents" and store a list of its milestone "grandchildren." For example, in the path $a \rightarrow b \rightarrow c \rightarrow d$, c replicates b's tables, and a lists c as a grandchild. At runtime, if b cannot be reached, a re-transmits directly to c. c processes a's message as though it were b, in turn forwarding messages to itself. Additionally, to reduce replicated state, a node need not store all of the state of its parents, but primarily the pre-aggregation table containing the crucial pre-aggregation functions. The raw and partial aggregate tables are essentially forwarding tables. In skipping over a failed node, we skip the need for its forwarding.

4 Experimental Evaluation

In this section we present experimental results drawn from a simulation of a network of Mica2 motes [4]. We assume a generic MAC-layer protocol and measure the energy spent on both sending and receiving. Each transmitted message includes a header of fixed size, followed by the body. Radio range is set at 50 meters. For node locations, we use the coordinates of the 2003 deployment on Great Duck Island [12], with some modification to filter out multiple nodes at identical coordinates. The resulting configuration has 68 nodes in a $106 \times 203\,\mathrm{m}^2$ area. For all experiments except on suppression, all node readings change at every timestep, and we periodically recompute all aggregate functions. For routing, we build a multicast tree from each source to all destinations requiring it.

We have implemented four algorithms. *Multicast* simply multicasts raw values to destinations. *Aggregation* aggregates raw values as soon as their routes to the same destination converge. *Optimal* implements our optimal many-to-many aggregation plan, which combines and balances multicast and in-network aggregation. *Flood* is a simple algorithm where sources flood the entire network using broadcasts; unlike the other algorithms, it needs no additional state in the network. To reduces the per-message overhead for *flood*, we build in a delay

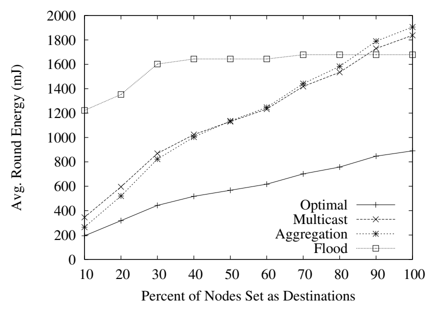

Figure 3. Varying the number of aggregation functions.

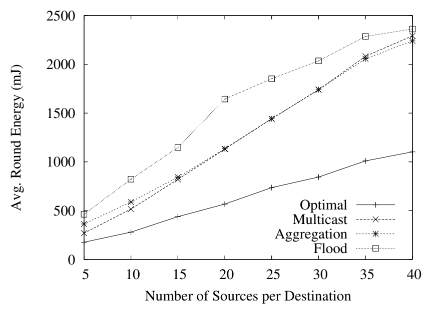

Figure 4. Varying the number of sources per function.

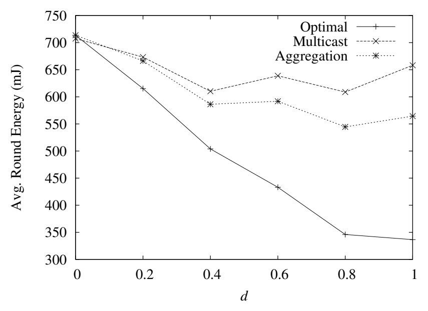

Figure 5. Varying the dispersion factor.

at each node so that it can collect a large number of values and combine them into one message before sending.

Adjusting Aggregation Workload The first set of experiments exposes the impact of the number, size, and "shape" of aggregation functions in the workload. We control these factors by adjusting the number of destinations, the number of sources per destination, and a *dispersion factor* d between 0 and 1 that dictates what proportion of sources are at each hop distance from the destination. The relative contribution from each hop distance h is given by $d^{h-1}/\sum_{h=1}^{H} d^{h-1}$, where H is the distance limit for which nodes may be chosen as sources. This formula captures a typical situation where a destination is influenced more by close neighbors, but might also require some measurements available only at distant nodes. A larger d means the sources are more "dispersed."

In Figure 3 we vary the number of aggregation functions, or destinations. Each aggregation function involves 20 sources, and $d = 0.9$. For most workloads, *flood* is much more expensive than all others. As long as a node is a source, *Flood* sends its value to the entire network; it does not take a large workload to make most nodes sources. For very heavy workloads, *flood* is slightly better than *multicast* and *aggregation* due to its efficient use of broadcasts and the simplicity of its protocol.[1] When the number of destinations is small, *multicast* does not work well because each source value is needed at a few destinations; *aggregation* works better because each destination has 20 sources to aggregate. As the number of destinations increases, costs generally rise for *multicast*, *aggregation*, and *optimal*. At the same time, transmitting raw values begins to have a higher benefit as each source has more destinations, so *multicast* begins to outperform *aggregation*. Note that *optimal* significantly outperforms all other algorithms, and its advantage over *multicast* and *aggregation* continues to grow throughout.

In Figure 4 we vary the size of the aggregation functions, i.e., their number of sources. *Multicast* outperforms *aggregation* at the lowest sizes, where there are fewer opportunities to aggregate converging values headed for the same destination. As the number of sources per destination increases, more aggregation opportunities arise, so *aggregation* beats *multicast*. Again, by jointly exploiting both multicast and in-network aggregation, *optimal* outperforms and scales much better than using either *multicast* or *aggregation* exclusively.

The final experiment, shown in Figure 5, is designed to observe the impact of aggregation "shape" as controlled by the dispersion factor d. 20% of all nodes are destinations, each aggregating 20 sources from 1–4 hops away. We range from $d = 0$, where all sources are within one hop, to $d = 1$, where sources one to four hops away are equally likely. Again, we see *optimal* significantly outperforms the other algorithms, and its advantage grows as more optimization opportunities arise when the multicast trees become deeper.

As a side note, it may be somewhat counterintuitive that the costs decrease as we disperse the sources more. The explanation lies in the standard algorithm we used for constructing single-source multicast trees, which tends to create many edges that are not shared across trees, especially when shallow trees are possible. For example, when all sources are one hop from their destinations ($d = 0$), all multicast tree edges are distinct; therefore, *optimal*, *multicast* and *aggregation* have identical costs. When destinations are farther away, a source contributing to several destinations may only need to transmit its value one hop before it can be incorporated into messages already headed to its various destinations. Because of this sharing, the total number of distinct edges in all multicast trees may actually be lower than in the case of $d = 0$. *Optimal* exploits this sharing most effectively; *aggregation* also performs better, but not as dramatically; *multicast* does not appear to follow any pattern because, as d increase, it trades off the advantage of having fewer distinct edges (meaning that more message units can be combined into messages to reduce overhead) with the disadvantage of not being able to aggregate (meaning that each raw value needs to travel through more hops to reach its destination). This experiment also indicates that simple criteria for constructing multicast trees, which work well for single trees (such as minimizing the total distances to destinations), may not be the best option when there are multiple sources and multiple destinations, and when aggregation is possible. This observation illustrates the importance of jointly designing routing and data processing in

[1]The other algorithms, especially *multicast* and *optimal*, can also benefit from broadcasts (cf. Section 3) if we implement selective listening. While we have not implemented this optimization, we note that it would further increase the advantage of the other algorithms over *flood*.

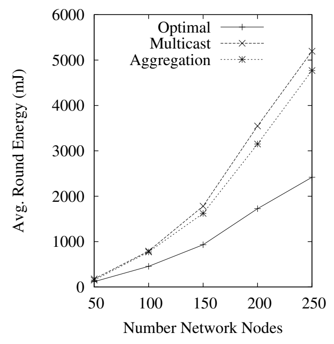

Figure 6. Increasing network size.

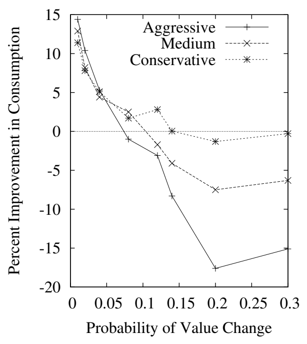

Figure 7. Override policies.

sensor networks, a topic worthy of future investigation but beyond the scope of this paper.

Increasing Network Size We next evaluate how our many-to-many aggregation approach scales with increased network size. We create a series of five simulated networks with increasing area and number of nodes. In each case, 25% of all nodes are destinations, each aggregating using 15% of all nodes as sources. The result is shown in Figure 6. We omit *flood*; it is over an order of magnitude more costly than *optimal* in all but the smallest network. Compared with *multicast* and *aggregation*, *optimal* exerts its usual flexibility to obtain substantial energy savings. The larger the network, the more edges there are at which *optimal* makes the best decision, while *multicast* and *aggregation* are forced to make suboptimal ones.

Suppression and Override We next experimentally investigate how our many-to-many aggregation plan works with temporal suppression. In Figure 7, we control the probability with which node values change in each timestep, and compare the dynamic override approach (Section 3) with the approach of simply recomputing all aggregation function in each timestep. Recall override is heuristic, and may backfire as it destroys aggregation opportunities downstream. We have implemented three versions of the heuristic. (1) *Aggressive* overrides the decision to aggregate and instead transmits the raw value as long as doing so is locally optimal; source weight is set equal to destination weight (i.e., there is no extra weight on sources). (2) *Conservative* only overrides if transmitting raw is substantially cheaper locally; source weight is set 3.3 times as high as destination weight. (3) *Medium* falls in between the two; source weight is set 2 times as high as destination weight. As expected, the higher the change probability, the less effective is temporal suppression, and the more energy is consumed each round for all approaches. To better illustrate the trade-off among the three override policies, we plot the improvement they gain over the solution given by full recomputation. The results are averaged over 10 timesteps in 3 random networks (each with 30% of nodes as destinations with 25 sources each). When change probability is low, override policies earn savings of 10–15%; *aggressive* is slightly better, though *conservative* also achieves most of the savings. When probability is higher, however, more values are likely to change, creating more aggregation opportunities that override might miss downstream. As a result, all override policies suffer, but *conservative*, being more judicious about override, does not degrade much past the recomputation approach (which would be optimal when change probability is 1). In practice, the choice of override policy should depend on the volatility of source values. It may also be possible to add additional hints, such as the number of multicast descendants, to suggest the potential penalty of overriding.

Summary There are three main points to glean from this section. First, flooding is quite expensive, and especially when the number and the size of aggregation functions are small; it simply transmits much more data than necessary. Second, multicast and in-network aggregation, when optimally combined, is very powerful. Experiments confirm significant reduction in energy consumption, and support our initial intuition that values should be initially transmitted raw from their sources, but aggregated closer to their destinations. Finally, for continuous control using multiple aggregation functions, our dynamic override approach provides an effective mechanism for altering the default plan to match the degree of volatility in source values.

5 Related Work

In-Network Aggregation Aggregating data en route to a destination is a feature in the well-known systems, TAG [11] and Cougar [17]. TAG addresses the problem of determining time to wait for child nodes to return their partial aggregates with *epoch duration*. Yao and Gehrke [18] suggest communication between child and parent to trade estimates of waiting time. TAG and TiNA [15] discuss a group-by clause that allows for aggregation among sources in the same group; each source, however, can only participate in one group. TiNA has nodes favor a parent in the same group, which reduces the number of partial aggregate records listed in the parent's outgoing message. Hellerstein and Wang [8] give the example of computing a Haar wavelet over the network, and the problem of using the communication tree as the support tree, which can be used to decode values at varying resolutions. In all of these examples, data from multiple sources are aggregated en route to a single destination (the base station). In this paper, we too consider multiple sources, but then also multiple destinations.

In-Network Control of Sensing Several examples exist of in-network control of sensing. Deshpande *et al.* [5] construct query plans that sample low-cost sensors first in hope of answering queries without sampling expensive ones. Cardell-Oliver *et al.* [2] use a single precipitation sensor to assign a sampling rate to all soil moisture sensors in the network, setting the frequency high when it rains. Our many-to-many aggregation framework significantly generalizes control. Readings from an arbitrary subset of source nodes can be used at runtime to control sampling at an arbitrary subset of destination nodes.

One-to-Many and Many-to-Many Communication Beside the many-to-one communication pattern of in-network aggregation, other patterns include one-to-many and many-to-many. One-to-many communication disseminates data from a single source to multiple destinations. The main technique is multicast, which has been well studied in sensor networks (see [13] for a survey). Many-to-many communication delivers data from multiple sources to multiple respective destinations. Our many-to-many aggregation problem is one example. Other sensor network work also addresses this pattern. *Data funneling* [14] allows various controller nodes to request data from particular network regions. In each such region, nodes send data to a border node for compression. While there can be many controller nodes, each one's request is optimized in isolation as an instance of many-to-one. In contrast, we optimize the entire many-to-many problem as a whole. In *directed diffusion* [9], nodes transmit interests for data, and nodes providing that data diffuse it in the direction of the interest. In this case, since requests are made ad-hoc, our style of optimization is difficult.

Models The pervasive use of models [6, 16, 3] in sensor networks for improving energy efficiency provides potential applications for our many-to-many aggregation. Although initially designed for out-of-network use [6], models are now not only being used in-network [16, 3], but also becoming spatial, with each model taking multiple nodes' readings as input. Maintaining multiple such models in-network requires many-to-many communication. If the associated computation can be expressed as aggregation functions, then our approach may be appropriate for supporting these in-network models.

Shared Aggregation The problem of computing multiple aggregations also arises in stream processing, where many users have different, but often similar, queries to be run against the stream. Krishnamurthy *et al.* [10] develop techniques for finding commonalities among queries and sharing work between them. Similar approaches may be beneficial in our case, especially when multiple destinations have very similar aggregations. The bipartite vertex cover reduction, as depicted in Figure 2, does not capture the possibility of using the same partial aggregate for different destinations. An interesting direction for future work would be to reconsider the optimization problem to accommodate this possibility.

6 Conclusion

We have described the many-to-many aggregation problem for sensor networks, where we need to compute and deliver to each destination node a different aggregate over a subset of source nodes. We optimize many-to-many aggregation by combining and balancing the techniques of multicast and in-network aggregation. Interestingly, we show that the optimal multiple-aggregation plan can be obtained by independently solving the optimization problem at each edge, and combining the solutions into a globally optimal and consistent plan. This result implies that dynamic adjustments to routes and aggregation workload can be handled efficiently, by only re-optimizing the affected edges. We also show how to extend the multiple-aggregation plan for continuous evaluation with temporal suppression. We believe this work will become more important as networks grow larger, as it enables in-network control of sensor nodes, allowing them react to changes without expensive intervention by the base station.

References

[1] R.K. Ahuja, T.L Magnanti, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993.

[2] R. Cardell-Oliver, K. Smetten, M. Kranz, and K. Mayer. A Reactive Soil Moisture Sensor Network: Design and Field Evaluation. *Intl. Journal of Distributed Sensor Networks*, 1(2), 2005.

[3] D. Chu, A. Deshpande, J. Hellerstein, and W. Hong. Approximate Data Collection in Sensor Networks using Probabilistic Models. In *Proc. of the 2006 Intl. Conf. on Data Engineering*, Atlanta, Georgia, USA, April 2006.

[4] Crossbow Inc. *"MPR-Mote Processor Radio Board User's Manual"*.

[5] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting Correlated Attributes in Acquisitional Query Processing. In *Proc. of the 2005 Intl. Conf. on Data Engineering*, Tokyo, Japan, April 2005.

[6] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-Driven Data Acquisition in Sensor Networks. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, Toronto, Canada, August 2004.

[7] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proc. of the 1996 Intl. Conf. on Data Engineering*, New Orleans, Louisiana, USA, February 1996.

[8] J. Hellerstein and W. Wang. Optimization of In-Network Data Reduction. In *Proc. of the 2004 Workshop on Data Management for Sensor Networks*, Toronto, Canada, August 2004.

[9] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed Diffusion for Wireless Sensor Networking. *ACM/IEEE Trans. on Networking*, 11(1):2–16, 2002.

[10] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proc. of the 2006 ACM SIGMOD Intl. Conf. on Management of Data*, Chicago, Illinois, USA, June 2006.

[11] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. of the 2002 USENIX Symp. on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, December 2002.

[12] A. Mainwaring, R. Szewczyk, J. Anderson, and J. Polastre. Great Duck Island. http://www.greatduckisland.net.

[13] P. Mohapatra and S. Krishnamurthy. *Ad Hoc Networks Technologies and Protocols*, chapter Multicasting in Ad Hoc Networks. Springer, 2005.

[14] D. Petrovic, R. Shah, K. Ramchandran, and J. Rabaey. Data Funneling: Routing with Aggregation and Compression for Wireless Sensor Networks. In *Proc. of the 2003 IEEE Sensor Network Protocols and Applications*, Anchorage, Alaska, USA, May 2003.

[15] M. Sharaf, J. Beaver, A. Labrinidis, and P. Chrysanthis.

Balancing Energy Efficiency and Quality of Aggregate Data in Sensor Networks. *The VLDB Journal*, 13(4):384–403, 2004.

[16] D. Tulone and S. Madden. PAQ: Time Series Forecasting for Approximate Query Answering in Sensor Networks. In *Proc. of the 2006 European Workshop on Sensor Networks*, Zurich, Switzerland, February 2006.

[17] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3), 2002.

[18] Y. Yao and J. Gehrke. Query Processing for Sensor Networks. In *Proc. of the 2003 Conf. on Innovative Data Systems Research*, Ansilomar, California, USA, January 2003.

A Proof of Theorem 1

This section is devoted to the non-trivial proof of Theorem 1. Let $\mathit{mvc}(U,V,E)$ denote the minimum vertex cover (unique by our assumption) of bipartite graph (U,V,E). We start with a lemma:

Lemma 1. *Consider any* (U,V,E).

A. For any set of destination vertices Y disjoint from V, and any set of edges F between U and Y: if source vertex $u \in \mathit{mvc}(U,V,E)$, then $u \in \mathit{mvc}(U,V\cup Y,E\cup F)$.

B. For any set of source vertices X disjoint from U, and any set of edges F between X and V: if source vertex $u \in \mathit{mvc}(U\cup X,V,E\cup F)$, then $u \in \mathit{mvc}(U,V,E)$.

Proof. We simultaneously prove (A) and (B) by induction on the size of (U,V,E) (as measured by its total number of vertices). The base case of $(\varnothing,\varnothing,\varnothing)$ is trivial: (A) is vacuously true because (U,V,E) has no source vertex; (B) is also vacuously true because the edge set $E\cup F$ is empty (F must be empty because V is empty).

Now, suppose that (A) and (B) hold for any (U,V,E) with size less than k (inductive hypothesis). We now show that (A) and (B) also hold for any (U,V,E) of size k. We begin with (A).

1. Consider first the graph $(U,V\cup Y,E\cup F)$. Let $Y_1 = Y\cap \mathit{mvc}(U,V\cup Y,E\cup F)$ be the subset of Y included in $\mathit{mvc}(U,V\cup Y,E\cup F)$, and $Y_2 = Y - Y_1$ be the subset that is not included. Let F_1 and F_2 be the subsets of F incident to Y_1 and Y_2, respectively. $F = F_1\cup F_2$.
2. Consider the graph $(U,V\cup Y_2,E\cup F_2)$ (i.e., Y_1 and incident edges are removed). We argue that $\mathit{mvc}(U,V\cup Y_2,E\cup F_2) = \mathit{mvc}(U,V\cup Y,E\cup F) - Y_1$, i.e., the two minimum covers include the same vertices, except those removed. The reason is as follows. Note that $\mathit{mvc}(U,V\cup Y,E\cup F) - Y_1$ covers $(U,V\cup Y_2,E\cup F_2)$, while $\mathit{mvc}(U,V\cup Y_2,E\cup F_2)\cup Y_1$ covers $(U,V\cup Y,E\cup F)$. If $\mathit{mvc}(U,V\cup Y,E\cup F) - Y_1$ is different from, and hence weighs more than (by our assumption of unique solution), $\mathit{mvc}(U,V\cup Y_2,E\cup F_2)$, then $\mathit{mvc}(U,V\cup Y_2,E\cup F_2)\cup Y_1$ would weigh less than $\mathit{mvc}(U,V\cup Y,E\cup F)$, a contradiction.

   Now, because $\mathit{mvc}(U,V\cup Y,E\cup F)$ does not include Y_2 by definition, and we have just seen that $\mathit{mvc}(U,V\cup Y_2,E\cup F_2)$ and $\mathit{mvc}(U,V\cup Y,E\cup F)$ include the same vertices in U and $V\cup Y_2$, we know that $\mathit{mvc}(U,V\cup Y_2,E\cup F_2)$ includes none of Y_2. Therefore, $\mathit{mvc}(U,V\cup Y_2,E\cup F_2)$ must include all of X_2, the source vertices in U that are connected to Y_2 via F_2 (otherwise we cannot cover F_2).
3. Consider the graph $(U-X_2,V,E-E_2)$, where E_2 is the subset of E incident to X_2 (i.e., Y_2, X_2 and incident edges are further removed). We argue that $\mathit{mvc}(U-X_2,V,E-E_2) = \mathit{mvc}(U,V\cup Y_2,E\cup F_2) - X_2$, i.e., the two minimum covers include the same vertices, except those removed. The reasoning is similar to that in the second step above. Note that $\mathit{mvc}(U,V\cup Y_2,E\cup F_2) - X_2$ covers $(U-X_2,V,E-E_2)$, while $\mathit{mvc}(U-X_2,V,E-E_2)\cup X_2$ covers $(U,V\cup Y_2,E\cup F_2)$. If $\mathit{mvc}(U,V\cup Y_2,E\cup F_2) - X_2$ is different from, and hence weighs more than (by our assumption of unique solution), $\mathit{mvc}(U-X_2,V,E-E_2)$, then $\mathit{mvc}(U-X_2,V,E-E_2)\cup X_2$ would weigh less than $\mathit{mvc}(U,V\cup Y_2,E\cup F_2)$, a contradiction.
4. From the above steps, we see that if source vertex $u \in \mathit{mvc}(U-X_2,V,E-E_2)$, then $u \in \mathit{mvc}(U,V\cup Y,E\cup F)$. At this point, it suffices to prove that if $u \in \mathit{mvc}(U,V,E)$, then $u \in \mathit{mvc}(U-X_2,V,E-E_2)$. If $X_2 = \varnothing$, then $E_2 = \varnothing$, and the claim is obviously true. Otherwise, $|(U-X_2,V,E-E_2)| < k$. By the inductive hypothesis for (B), if $u \in \mathit{mvc}(U,V,E)$, then $u \in \mathit{mvc}(U-X_2,V,E-E_2)$. The proof for (A) completes.

The proof of (B) is almost the mirror image of the proof for (A) and follows the same logical steps. Therefore, we only present a sketch below.

1. Consider first the graph $(U\cup X,V,E\cup F)$. Let X_1 be the subset of X included in $\mathit{mvc}(U\cup X,V,E\cup F)$, and X_2 be the subset that is not included. Let F_1 and F_2 be the subsets of F incident to X_1 and X_2, respectively.
2. Consider the graph $(U\cup X_2,V,E\cup F_2)$ (i.e., X_1 and incident edges are removed). We can show that $\mathit{mvc}(U\cup X_2,V,E\cup F_2) = \mathit{mvc}(U\cup X,V,E\cup F) - X_1$, i.e., the two minimum covers include the same vertices, except those removed. We can then show that $\mathit{mvc}(U\cup X_2,V,E\cup F_2)$ includes none of X_2 and therefore all of Y_2, the vertices in V that are connected to X_2 via edges $E_2 \subseteq E$.
3. Consider the graph $(U,V-Y_2,E-E_2)$ (i.e., X_2, Y_2 and incident edges are further removed). We can show that $\mathit{mvc}(U,V-Y_2,E-E_2) = \mathit{mvc}(U\cup X_2,V,E\cup F_2) - Y_2$, i.e., the two minimum covers include the same vertices, except for those removed.
4. From the above steps, we it suffices to prove that if source vertex $u \in \mathit{mvc}(U,V-Y_2,E-E_2)$, then $u \in \mathit{mvc}(U,V,E)$. The case when $Y_2 = \varnothing$ is trivial, and the case when $Y_2 \neq \varnothing$ follows from the inductive hypothesis for (A). □

Proof of Theorem 1. Consider two multicast edges e and e'. Suppose their single-edge optimization problems are solved on (U,V,E) and (U',V',E'), respectively. Let $U_0 = U\cap U'$,

$U^- = U - U'$, $U^+ = U' - U$, $V_0 = V \cap V'$, $V^- = V - V'$, $V^+ = V' - V$, $E_0 = E \cap E'$, $E^- = E - E'$, and $E^+ = E' - E$. Clearly, U_0, U^-, U^+ are disjoint; V_0, V^-, V^+ are disjoint; E_0, E^-, E^+ are disjoint. These defintions are illustrated in Figure 8(0).

The lone threat to a consistent global plan is the following case: e is immediately upstream of e' (i.e., the arrowhead of e coincides with the tail of e') in the multicast tree rooted at a source s, and s is included in the solution of e' but not in the solution of e. To this end, we will show that $s \in mvc(U', V', E')$ implies $s \in mvc(U, V, E)$. The intermediate steps required to draw this implication, discussed below, are illustrated in Figure 8.

First, $s \in mvc(U', V', E')$ implies $s \in mvc(U' \cup U^-, V', E') = mvc(U_0 \cup U^+ \cup U^-, V', E_0 \cup E^+)$. The reason is that all E' edges are between U' and V', so U^- has no incident edges at all in $(U' \cup U^-, V', E')$, and should not be chosen in $mvc(U' \cup U^-, V', E')$.

Next, by Lemma 1(B), $s \in mvc(U_0 \cup U^+ \cup U^-, V', E_0 \cup E^+)$ implies that $s \in mvc(U_0 \cup U^-, V', E_0) = mvc(U, V_0 \cup V^+, E_0)$. Lemma 1(B) is applicable here because E^+ can connect only U^+ and V'. To prove E^+ connects only U^+ and V', it suffices to show that any connection between U and V' must be in E. In other words, we need to show that if $u \sim_{e'} v'$ for some $u \in U$ and $v' \in V'$, then $u \sim_e v'$. To this end, note that $u \in U$ means there exists v such that $u \sim_e v$. Thus, the multicast path from u to v goes through e, and therefore also the tail of e'. At the same time, $u \sim_{e'} v'$ means that the multicast path from u to v' goes through e', and therefore also the tail of e'. However, in the multicast tree rooted at u, there can be only one unique path from u to the tail of e'. Therefore, the path from u to v and the path from u to v' are identical from u to the tail of e', so e must be on the path from u to v', i.e., $u \sim_e v'$.

Then, by Lemma 1(A), $s \in mvc(U, V_0 \cup V^+, E_0)$ in turn implies $s \in mvc(U, V_0 \cup V^+ \cup V^-, E_0 \cup E^-) = mvc(U, V \cup V^+, E)$. Lemma 1(A) is applicable here because E^- can connect only U and V^-. To prove E^- connects only U and V^-, it suffices to show that any connection between U and V' must be in E'. In other words, we need to show that if $u \sim_e v'$ for some $u \in U$ and $v' \in V'$, then $u \sim_{e'} v'$. To this end, note that $v' \in V'$ means there exists u' such that $u' \sim_{e'} v'$. Thus, the multicast path from u' to v' goes through e', and therefore also the arrowhead of e. At the same time, $u \sim_e v'$ means that the multicast path from u to v' goes through e, and therefore also the arrowhead of e. However, according to the multicast path sharing assumption (Section 2.1), the two paths from the arrowhead of e to v' in two multicast trees (one rooted at u' and the other rooted at u) must be identical. Therefore, e' must be on the path from u to v', i.e., $u \sim_{e'} v'$.

Finally, $s \in mvc(U, V \cup V^+, E)$ implies $s \in mvc(U, V, E)$, completing the proof. The reason is that all E edges are between U and V, so V^+ has no incident edges at all in $(U, V \cup V^+, E)$, and should not be chosen in $mvc(U, V \cup V^+, E)$. □

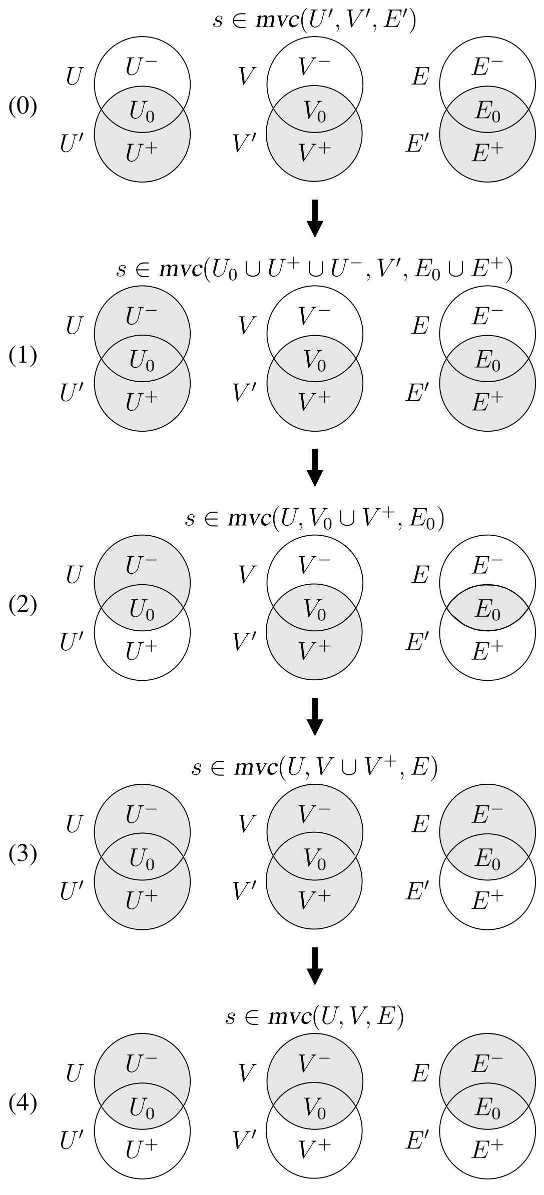

Figure 8. Stepwise proof of Theorem 1.